# Table of Contents

1. Preamble .................................................................................................................. 1
2. Introduction .............................................................................................................. 3  
   2.1. Document Conventions ..................................................................................... 3
3. Fundamentals ............................................................................................................ 6  
   3.1. Host and Device Environment ........................................................................ 6  
   3.2. Execution Model .............................................................................................. 6  
   3.3. Object Model .................................................................................................... 8
   3.4. Application Binary Interface .......................................................................... 12  
   3.5. Command Syntax and Duration ...................................................................... 12  
   3.6. Threading Behavior ......................................................................................... 14
   3.7. Valid Usage ....................................................................................................... 22  
   3.8. *vkResult* Return Codes ................................................................................ 28  
   3.9. Numeric Representation and Computation ..................................................... 30  
   3.10. Fixed-Point Data Conversions ...................................................................... 32
   3.11. Common Object Types ................................................................................... 34  
   3.12. API Name Aliases .......................................................................................... 44
4. Initialization ................................................................................................................ 45  
   4.1. Command Function Pointers ........................................................................... 45  
   4.2. Instances ......................................................................................................... 48
5. Devices and Queues .................................................................................................... 56  
   5.1. Physical Devices .............................................................................................. 56
   5.2. Devices ............................................................................................................. 87
   5.3. Queues ............................................................................................................. 99
6. Command Buffers ....................................................................................................... 107  
   6.1. Command Buffer Lifecycle ........................................................................... 107  
   6.2. Command Pools .............................................................................................. 109  
   6.3. Command Buffer Allocation and Management ............................................ 116  
   6.4. Command Buffer Recording ......................................................................... 120  
   6.5. Command Buffer Submission ....................................................................... 129  
   6.6. Queue Forward Progress .............................................................................. 146  
   6.7. Secondary Command Buffer Execution ....................................................... 146
   6.8. Command Buffer Device Mask ...................................................................... 152
7. Synchronization and Cache Control .......................................................................... 155  
   7.1. Execution and Memory Dependencies ............................................................ 155  
   7.2. Implicit Synchronization Guarantees ............................................................... 176  
   7.3. Fences .............................................................................................................. 178
   7.4. Semaphores ..................................................................................................... 188
   7.5. Events ............................................................................................................. 203
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>32.1. Feature Requirements</td>
<td>1129</td>
</tr>
<tr>
<td>32.2. Profile Features</td>
<td>1131</td>
</tr>
<tr>
<td>33. Limits</td>
<td>1133</td>
</tr>
<tr>
<td>33.1. Limit Requirements</td>
<td>1161</td>
</tr>
<tr>
<td>33.2. Profile Limits</td>
<td>1172</td>
</tr>
<tr>
<td>34. Formats</td>
<td>1174</td>
</tr>
<tr>
<td>34.1. Format Definition</td>
<td>1174</td>
</tr>
<tr>
<td>34.2. Format Properties</td>
<td>1221</td>
</tr>
<tr>
<td>34.3. Required Format Support</td>
<td>1232</td>
</tr>
<tr>
<td>35. Additional Capabilities</td>
<td>1251</td>
</tr>
<tr>
<td>35.1. Additional Image Capabilities</td>
<td>1251</td>
</tr>
<tr>
<td>35.2. Additional Buffer Capabilities</td>
<td>1264</td>
</tr>
<tr>
<td>35.3. Optional Semaphore Capabilities</td>
<td>1266</td>
</tr>
<tr>
<td>35.4. Optional Fence Capabilities</td>
<td>1270</td>
</tr>
<tr>
<td>36. Debugging</td>
<td>1275</td>
</tr>
<tr>
<td>36.1. Active Tooling Information</td>
<td>1276</td>
</tr>
<tr>
<td>Appendix A: Vulkan Environment for SPIR-V</td>
<td>1280</td>
</tr>
<tr>
<td>Versions and Formats</td>
<td>1280</td>
</tr>
<tr>
<td>Validation Rules within a Module</td>
<td>1286</td>
</tr>
<tr>
<td>Precision and Operation of SPIR-V Instructions</td>
<td>1301</td>
</tr>
<tr>
<td>Signedness of SPIR-V Image Accesses</td>
<td>1306</td>
</tr>
<tr>
<td>Image Format and Type Matching</td>
<td>1307</td>
</tr>
<tr>
<td>Compatibility Between SPIR-V Image Formats And Vulkan Formats</td>
<td>1308</td>
</tr>
<tr>
<td>Appendix B: Memory Model</td>
<td>1310</td>
</tr>
<tr>
<td>Agent</td>
<td>1310</td>
</tr>
<tr>
<td>Memory Location</td>
<td>1310</td>
</tr>
<tr>
<td>Allocation</td>
<td>1310</td>
</tr>
<tr>
<td>Memory Operation</td>
<td>1310</td>
</tr>
<tr>
<td>Reference</td>
<td>1311</td>
</tr>
<tr>
<td>Program-Order</td>
<td>1311</td>
</tr>
<tr>
<td>Scope</td>
<td>1312</td>
</tr>
<tr>
<td>Atomic Operation</td>
<td>1312</td>
</tr>
<tr>
<td>Scoped Modification Order</td>
<td>1313</td>
</tr>
<tr>
<td>Memory Semantics</td>
<td>1313</td>
</tr>
<tr>
<td>Release Sequence</td>
<td>1315</td>
</tr>
<tr>
<td>Synchronizes-With</td>
<td>1315</td>
</tr>
<tr>
<td>System-Synchronizes-With</td>
<td>1317</td>
</tr>
<tr>
<td>Private vs. Non-Private</td>
<td>1317</td>
</tr>
<tr>
<td>Inter-Thread-Happens-Before</td>
<td>1317</td>
</tr>
<tr>
<td>Happens-Before</td>
<td>1318</td>
</tr>
</tbody>
</table>
Chapter 1. Preamble

Copyright 2014-2022 The Khronos Group Inc.

This Specification is protected by copyright laws and contains material proprietary to Khronos. Except as described by these terms, it or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of Khronos.

Khronos grants a conditional copyright license to use and reproduce the unmodified Specification for any purpose, without fee or royalty, EXCEPT no licenses to any patent, trademark or other intellectual property rights are granted under these terms. Parties desiring to implement the Specification and make use of Khronos trademarks in relation to that implementation, and receive reciprocal patent license protection under the Khronos IP Policy must become Adopters and confirm the implementation as conformant under the process defined by Khronos for this Specification; see https://www.khronos.org/adopters.

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

This Specification has been created under the Khronos Intellectual Property Rights Policy, which is Attachment A of the Khronos Group Membership Agreement available at https://www.khronos.org/files/member_agreement.pdf.

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

The Khronos Intellectual Property Rights defines the terms 'Scope', 'Compliant Portion', and 'Necessary Patent Claims'.

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

Where this Specification uses technical terminology, defined in the Glossary or otherwise, that refer to enabling technologies that are not expressly set forth in this Specification, those enabling technologies are EXCLUDED from the Scope of this Specification. For clarity, enabling technologies not disclosed with particularity in this Specification (e.g. semiconductor manufacturing technology, hardware architecture, processor architecture or microarchitecture, memory architecture, compiler technology, object oriented technology, basic operating system technology, compression technology, algorithms, and so on) are NOT to be considered expressly set forth; only those application program interfaces and data structures disclosed with particularity are included in the Scope of this Specification.
For purposes of the Khronos Intellectual Property Rights Policy as it relates to the definition of Necessary Patent Claims, all recommended or optional features, behaviors and functionality set forth in this Specification, if implemented, are considered to be included as Compliant Portions.

Where this Specification identifies specific sections of external references, only those specifically identified sections define normative functionality. The Khronos Intellectual Property Rights Policy excludes external references to materials and associated enabling technology not created by Khronos from the Scope of this specification, and any licenses that may be required to implement such referenced materials and associated technologies must be obtained separately and may involve royalty payments.

Khronos and Vulkan are registered trademarks of The Khronos Group Inc. OpenCL is a trademark of Apple Inc.; and OpenGL and OpenGL ES are registered trademarks of Hewlett Packard Enterprise, all used under license by Khronos. ASTC is a trademark of ARM Holdings PLC. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.
Chapter 2. Introduction

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

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

2.1. Document Conventions

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

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

2.1.1. Informative Language

Some language in the specification is purely informative, intended to give background or suggestions to implementors or developers.

If an entire chapter or section contains only informative language, its title will be suffixed with “(Informative)".

All NOTEs are implicitly informative.

2.1.2. Normative Terminology

Within this specification, the key words must, required, should, recommended, may, and optional are to be interpreted as described in RFC 2119 - Key words for use in RFCs to Indicate Requirement Levels (https://www.ietf.org/rfc/rfc2119.txt). The additional key word optionally is an alternate form of optional, for use where grammatically appropriate.

These key words are highlighted in the specification for clarity. In text addressing application developers, their use expresses requirements that apply to application behavior. In text addressing implementors, their use expresses requirements that apply to implementations.
In text addressing application developers, the additional key words can and cannot are to be interpreted as describing the capabilities of an application, as follows:

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

**cannot**
This word means that the API and/or the execution environment provide no mechanism through which the application can express or accomplish the action described.

These key words are never used in text addressing implementors.

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

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

### 2.1.3. Technical Terminology

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

### 2.1.4. Normative References

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

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


Chapter 3. Fundamentals

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

3.1. Host and Device Environment

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

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

**Note**

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

3.2. Execution Model

This section outlines the execution model of a Vulkan system.

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

**Note**

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

Device memory is explicitly managed by the application. Each device **may** advertise one or more heaps, representing different areas of memory. Memory heaps are either device-local or host-local,
but are always visible to the device. Further detail about memory heaps is exposed via memory types available on that heap. Examples of memory areas that may be available on an implementation include:

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

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

### 3.2.1. Queue Operation

Vulkan queues provide an interface to the execution engines of a device. Commands for these execution engines are recorded into command buffers ahead of execution time, and then submitted to a queue for execution. Once submitted to a queue, command buffers will begin and complete execution without further application intervention, though the order of this execution is dependent on a number of implicit and explicit ordering constraints.

Work is submitted to queues using *queue submission commands* that typically take the form `vkQueue*` (e.g. `vkQueueSubmit`, `vkQueueBindSparse`), and **can** take a list of semaphores upon which to wait before work begins and a list of semaphores to signal once work has completed. The work itself, as well as signaling and waiting on the semaphores are all *queue operations*. Queue submission commands return control to the application once queue operations have been submitted - they do not wait for completion.

There are no implicit ordering constraints between queue operations on different queues, or between queues and the host, so these **may** operate in any order with respect to each other. Explicit ordering constraints between different queues or with the host **can** be expressed with *semaphores* and *fences*.

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

Before a fence or semaphore is signaled, it is guaranteed that any previously submitted queue operations have completed execution, and that memory writes from those queue operations are available to future queue operations. Waiting on a signaled semaphore or fence guarantees that previous writes that are available are also visible to subsequent commands.

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

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

Commands recorded in command buffers can perform actions, set state that persists across commands, synchronize other commands, or indirectly launch other commands, with some commands fulfilling several of these roles. The “Command Properties” section for each such command lists which of these roles the command takes. State setting commands update the current state of the command buffer. Some commands that perform actions (e.g. draw/dispatch) do so based on the current state set cumulatively since the start of the command buffer. The work involved in performing action commands is often allowed to overlap or to be reordered, but doing so must not alter the state to be used by each action command. In general, action commands are those commands that alter framebuffer attachments, read/write buffer or image memory, or write to query pools.

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

3.3. Object Model

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

Non-dispatchable handle types are a 64-bit integer type whose meaning is implementation-dependent. If the privateData feature is enabled for a VkDevice, each object of a non-dispatchable type created on that device must have a handle value that is unique among objects created on that device, for the duration of the object’s lifetime. Otherwise, non-dispatchable handles may encode object information directly in the handle rather than acting as a reference to an underlying object, and thus may not have unique handle values. If handle values are not unique, then destroying one such handle must not cause identical handles of other types to become invalid, and must not cause identical handles of the same type to become invalid if that handle value has been created more times than it has been destroyed.
All objects created or allocated from a `VkDevice` (i.e. with a `VkDevice` as the first parameter) are private to that device, and **must** not be used on other devices.

### 3.3.1. Object Lifetime

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

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

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

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

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

- `VkShaderModule`
- `VkPipelineCache`

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

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

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

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

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

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

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

• VkFence
• VkSemaphore
• VkCommandBuffer
• VkCommandPool

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

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

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

• **VkQueue** objects **cannot** be explicitly destroyed. Instead, they are implicitly destroyed when the
• Destroying a pool object implicitly frees all objects allocated from that pool. Specifically, destroying \texttt{VkCommandPool} frees all \texttt{VkCommandBuffer} objects that were allocated from it, and destroying \texttt{VkDescriptorPool} frees all \texttt{VkDescriptorSet} objects that were allocated from it.

• \texttt{VkDevice} objects \textbf{can} be destroyed when all \texttt{VkQueue} objects retrieved from them are idle, and all objects created from them have been destroyed. This includes the following objects:
  ◦ \texttt{VkFence}
  ◦ \texttt{VkSemaphore}
  ◦ \texttt{VkEvent}
  ◦ \texttt{VkQueryPool}
  ◦ \texttt{VkBuffer}
  ◦ \texttt{VkBufferView}
  ◦ \texttt{VkImage}
  ◦ \texttt{VkImageView}
  ◦ \texttt{VkShaderModule}
  ◦ \texttt{VkPipelineCache}
  ◦ \texttt{VkPipeline}
  ◦ \texttt{VkPipelineLayout}
  ◦ \texttt{VkSampler}
  ◦ \texttt{VkSamplerYcbcrConversion}
  ◦ \texttt{VkDescriptorSetLayout}
  ◦ \texttt{VkDescriptorPool}
  ◦ \texttt{VkFramebuffer}
  ◦ \texttt{VkRenderPass}
  ◦ \texttt{VkCommandPool}
  ◦ \texttt{VkCommandBuffer}
  ◦ \texttt{VkDeviceMemory}

• \texttt{VkPhysicalDevice} objects \textbf{cannot} be explicitly destroyed. Instead, they are implicitly destroyed when the \texttt{VkInstance} object they are retrieved from is destroyed.

• \texttt{VkInstance} objects \textbf{can} be destroyed once all \texttt{VkDevice} objects created from any of its \texttt{VkPhysicalDevice} objects have been destroyed.

### 3.3.2. External Object Handles

As defined above, the scope of object handles created or allocated from a \texttt{VkDevice} is limited to that logical device. Objects which are not in scope are said to be external. To bring an external object into scope, an external handle \textbf{must} be exported from the object in the source scope and imported into the destination scope.
3.4. Application Binary Interface

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

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

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

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

Note

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

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

3.5. Command Syntax and Duration

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

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

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

```c
// Provided by VK_VERSION_1_0
typedef uint32_t VkBool32;
```

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

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

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

**VK_TRUE** is a constant representing a **VkBool32** **True** value.

```c
#define VK_TRUE                           1U
```

**VK_FALSE** is a constant representing a **VkBool32** **False** value.

```c
#define VK_FALSE                          0U
```

**VkDeviceSize** represents device memory size and offset values:

```c
// Provided by VK_VERSION_1_0
typedef uint64_t VkDeviceSize;
```

**VkDeviceAddress** represents device buffer address values:

```c
// Provided by VK_VERSION_1_0
typedef uint64_t VkDeviceAddress;
```

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

The last in-parameter to each command that creates or destroys a Vulkan object is **pAllocator**. The **pAllocator** parameter can be set to a non-**NULL** value such that allocations for the given object are delegated to an application provided callback; refer to the Memory Allocation chapter for further
Commands that allocate Vulkan objects owned by pool objects are of the form `vkAllocate*`, and take `Vk*AllocateInfo` structures. These Vulkan objects are freed with commands of the form `vkFree*`. These objects do not take allocators; if host memory is needed, they will use the allocator that was specified when their parent pool was created.

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

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

### 3.5.1. Lifetime of Retrieved Results

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

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

### 3.6. Threading Behavior

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

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

**Note**

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

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

Many object types are immutable, meaning the objects cannot change once they have been created. These types of objects never need external synchronization, except that they must not be destroyed while they are in use on another thread. In certain special cases mutable object parameters are internally synchronized, making external synchronization unnecessary. Any command parameters that are not labeled as externally synchronized are either not mutated by the command or are internally synchronized. Additionally, certain objects related to a command’s parameters (e.g. command pools and descriptor pools) may be affected by a command, and must also be externally synchronized. These implicit parameters are documented as described below.

Parameters of commands that are externally synchronized are listed below.

### Externally Synchronized Parameters

- The `instance` parameter in `vkDestroyInstance`
- The `device` parameter in `vkDestroyDevice`
- The `queue` parameter in `vkQueueSubmit`
- The `fence` parameter in `vkQueueSubmit`
- The `queue` parameter in `vkQueueWaitIdle`
- The `memory` parameter in `vkFreeMemory`
- The `memory` parameter in `vkMapMemory`
- The `memory` parameter in `vkUnmapMemory`
- The `buffer` parameter in `vkBindBufferMemory`
- The `image` parameter in `vkBindImageMemory`
- The `queue` parameter in `vkQueueBindSparse`
- The `fence` parameter in `vkQueueBindSparse`
- The `fence` parameter in `vkDestroyFence`
- The `semaphore` parameter in `vkDestroySemaphore`
- The `event` parameter in `vkDestroyEvent`
- The `event` parameter in `vkSetEvent`
- The `event` parameter in `vkResetEvent`
- The `queryPool` parameter in `vkDestroyQueryPool`
- The `buffer` parameter in `vkDestroyBuffer`
- The `bufferView` parameter in `vkDestroyBufferView`
- The `image` parameter in `vkDestroyImage`
- The `imageView` parameter in `vkDestroyImageView`
- The `shaderModule` parameter in `vkDestroyShaderModule`
- The `pipelineCache` parameter in `vkDestroyPipelineCache`
- The `dstCache` parameter in `vkMergePipelineCaches`
- The `pipeline` parameter in `vkDestroyPipeline`
- The `pipelineLayout` parameter in `vkDestroyPipelineLayout`
- The `sampler` parameter in `vkDestroySampler`
- The `descriptorSetLayout` parameter in `vkDestroyDescriptorSetLayout`
- The `descriptorPool` parameter in `vkDestroyDescriptorPool`
- The `descriptorPool` parameter in `vkResetDescriptorPool`
- The `descriptorPool` member of the `pAllocateInfo` parameter in `vkAllocateDescriptorSets`
- The `descriptorPool` parameter in `vkFreeDescriptorSets`
- The `framebuffer` parameter in `vkDestroyFramebuffer`
- The `renderPass` parameter in `vkDestroyRenderPass`
- The `commandPool` parameter in `vkDestroyCommandPool`
- The `commandPool` parameter in `vkResetCommandPool`
- The `commandPool` member of the `pAllocateInfo` parameter in `vkAllocateCommandBuffers`
- The `commandPool` parameter in `vkFreeCommandBuffers`
- The `commandBuffer` parameter in `vkBeginCommandBuffer`
- The `commandBuffer` parameter in `vkEndCommandBuffer`
- The `commandBuffer` parameter in `vkResetCommandBuffer`
- The `commandBuffer` parameter in `vkCmdBindPipeline`
- The `commandBuffer` parameter in `vkCmdSetViewport`
- The `commandBuffer` parameter in `vkCmdSetScissor`
- The `commandBuffer` parameter in `vkCmdSetLineWidth`
- The `commandBuffer` parameter in `vkCmdSetDepthBias`
- The `commandBuffer` parameter in `vkCmdSetBlendConstants`
- The `commandBuffer` parameter in `vkCmdSetDepthBounds`
- The `commandBuffer` parameter in `vkCmdSetStencilCompareMask`
- The `commandBuffer` parameter in `vkCmdSetStencilWriteMask`
- The `commandBuffer` parameter in `vkCmdSetStencilReference`
- The `commandBuffer` parameter in `vkCmdBindDescriptorSets`
- The `commandBuffer` parameter in `vkCmdBindIndexBuffer`
- The `commandBuffer` parameter in `vkCmdBindVertexBuffers`
- The `commandBuffer` parameter in `vkCmdDraw`
The `commandBuffer` parameter in `vkCmdDrawIndexed`
- The `commandBuffer` parameter in `vkCmdDrawIndirectCount`
- The `commandBuffer` parameter in `vkCmdDrawIndexedIndirectCount`
- The `commandBuffer` parameter in `vkCmdBeginRenderPass2`
- The `commandBuffer` parameter in `vkCmdNextSubpass2`
- The `commandBuffer` parameter in `vkCmdEndRenderPass2`
- The `privateDataSlot` parameter in `vkDestroyPrivateDataSlot`
- The `commandBuffer` parameter in `vkCmdSetEvent2`
- The `commandBuffer` parameter in `vkCmdResetEvent2`
- The `commandBuffer` parameter in `vkCmdWaitEvents2`
- The `commandBuffer` parameter in `vkCmdPipelineBarrier2`
- The `commandBuffer` parameter in `vkCmdWriteTimestamp2`
- The `queue` parameter in `vkQueueSubmit2`
- The `fence` parameter in `vkQueueSubmit2`
- The `commandBuffer` parameter in `vkCmdCopyBuffer2`
- The `commandBuffer` parameter in `vkCmdCopyImage2`
- The `commandBuffer` parameter in `vkCmdCopyBufferToImage2`
- The `commandBuffer` parameter in `vkCmdCopyImageToBuffer2`
- The `commandBuffer` parameter in `vkCmdBlitImage2`
- The `commandBuffer` parameter in `vkCmdResolveImage2`
- The `commandBuffer` parameter in `vkCmdBeginRendering`
- The `commandBuffer` parameter in `vkCmdEndRendering`
- The `commandBuffer` parameter in `vkCmdSetCullMode`
- The `commandBuffer` parameter in `vkCmdSetFrontFace`
- The `commandBuffer` parameter in `vkCmdSetPrimitiveTopology`
- The `commandBuffer` parameter in `vkCmdSetViewportWithCount`
- The `commandBuffer` parameter in `vkCmdSetScissorWithCount`
- The `commandBuffer` parameter in `vkCmdBindVertexBuffers2`
- The `commandBuffer` parameter in `vkCmdSetDepthTestEnable`
- The `commandBuffer` parameter in `vkCmdSetDepthWriteEnable`
- The `commandBuffer` parameter in `vkCmdSetDepthCompareOp`
- The `commandBuffer` parameter in `vkCmdSetDepthBoundsTestEnable`
- The `commandBuffer` parameter in `vkCmdSetStencilTestEnable`
- The `commandBuffer` parameter in `vkCmdSetStencilOp`
- The `commandBuffer` parameter in `vkCmdSetRasterizerDiscardEnable`
- The `commandBuffer` parameter in `vkCmdSetDepthBiasEnable`
The commandBuffer parameter in `vkCmdSetPrimitiveRestartEnable`.

For `VkPipelineCache` objects created with `flags` containing `VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT`, the above table is extended with the `pipelineCache` parameter to `vkCreate*Pipelines` being externally synchronized.

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

**Externally Synchronized Parameter Lists**

- Each element of the `pFences` parameter in `vkResetFences`
- Each element of the `pDescriptorSets` parameter in `vkFreeDescriptorSets`
- Each element of the `pCommandBuffers` parameter in `vkFreeCommandBuffers`

In addition, there are some implicit parameters that need to be externally synchronized. For example, when a commandBuffer parameter needs to be externally synchronized, it implies that the commandPool from which that command buffer was allocated also needs to be externally synchronized. The implicit parameters and their associated object are listed below.

**Implicit Externally Synchronized Parameters**

- All `VkPhysicalDevice` objects enumerated from `instance` in `vkDestroyInstance`
- All `VkQueue` objects created from `device` in `vkDestroyDevice`
- All `VkQueue` objects created from `device` in `vkDeviceWaitIdle`
- Any `VkDescriptorSet` objects allocated from `descriptorPool` in `vkResetDescriptorPool`
- The `VkCommandPool` that `commandBuffer` was allocated from in `vkBeginCommandBuffer`
- The `VkCommandPool` that `commandBuffer` was allocated from in `vkEndCommandBuffer`
- The `VkCommandPool` that `commandBuffer` was allocated from in `vkResetCommandBuffer`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBindPipeline`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetViewport`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetScissor`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetLineWidth`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetDepthBias`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetBlendConstants`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetDepthBounds`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetStencilCompareMask`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetStencilWriteMask`
• The VkCommandPool that commandBuffer was allocated from, in vkCmdSetStencilReference
• The VkCommandPool that commandBuffer was allocated from, in vkCmdBindDescriptorSets
• The VkCommandPool that commandBuffer was allocated from, in vkCmdBindIndexBuffer
• The VkCommandPool that commandBuffer was allocated from, in vkCmdBindVertexBuffer
• The VkCommandPool that commandBuffer was allocated from, in vkCmdDraw
• The VkCommandPool that commandBuffer was allocated from, in vkCmdDrawIndexed
• The VkCommandPool that commandBuffer was allocated from, in vkCmdDrawIndirect
• The VkCommandPool that commandBuffer was allocated from, in vkCmdDispatch
• The VkCommandPool that commandBuffer was allocated from, in vkCmdDispatchIndirect
• The VkCommandPool that commandBuffer was allocated from, in vkCmdCopyBuffer
• The VkCommandPool that commandBuffer was allocated from, in vkCmdCopyImage
• The VkCommandPool that commandBuffer was allocated from, in vkCmdBlitImage
• The VkCommandPool that commandBuffer was allocated from, in vkCmdCopyBufferToImage
• The VkCommandPool that commandBuffer was allocated from, in vkCmdCopyImageToBuffer
• The VkCommandPool that commandBuffer was allocated from, in vkCmdUpdateBuffer
• The VkCommandPool that commandBuffer was allocated from, in vkCmdFillBuffer
• The VkCommandPool that commandBuffer was allocated from, in vkCmdClearColorImage
• The VkCommandPool that commandBuffer was allocated from, in vkCmdClearDepthStencilImage
• The VkCommandPool that commandBuffer was allocated from, in vkCmdClearAttachments
• The VkCommandPool that commandBuffer was allocated from, in vkCmdResolveImage
• The VkCommandPool that commandBuffer was allocated from, in vkCmdSetEvent
• The VkCommandPool that commandBuffer was allocated from, in vkCmdResetEvent
• The VkCommandPool that commandBuffer was allocated from, in vkCmdWaitEvents
• The VkCommandPool that commandBuffer was allocated from, in vkCmdPipelineBarrier
• The VkCommandPool that commandBuffer was allocated from, in vkCmdBeginQuery
• The VkCommandPool that commandBuffer was allocated from, in vkCmdEndQuery
• The VkCommandPool that commandBuffer was allocated from, in vkCmdResetQueryPool
• The VkCommandPool that commandBuffer was allocated from, in vkCmdWriteTimestamp
• The VkCommandPool that commandBuffer was allocated from, in vkCmdCopyQueryPoolResults
• The VkCommandPool that commandBuffer was allocated from, in vkCmdPushConstants
• The VkCommandPool that commandBuffer was allocated from, in vkCmdBeginRenderPass
• The VkCommandPool that commandBuffer was allocated from, in vkCmdNextSubpass
• The VkCommandPool that commandBuffer was allocated from, in vkCmdEndRenderPass
• The VkCommandPool that commandBuffer was allocated from, in vkCmdExecuteCommands
• The VkCommandPool that commandBuffer was allocated from, in vkCmdSetDeviceMask
• The VkCommandPool that commandBuffer was allocated from, in vkCmdDispatchBase
• The VkCommandPool that commandBuffer was allocated from, in vkCmdDrawIndirectCount
• The VkCommandPool that commandBuffer was allocated from, in vkCmdDrawIndexedIndirectCount
• The VkCommandPool that commandBuffer was allocated from, in vkCmdBeginRenderPass2
• The VkCommandPool that commandBuffer was allocated from, in vkCmdNextSubpass2
• The VkCommandPool that commandBuffer was allocated from, in vkCmdEndRenderPass2
• The VkCommandPool that commandBuffer was allocated from, in vkCmdSetEvent2
• The VkCommandPool that commandBuffer was allocated from, in vkCmdResetEvent2
• The VkCommandPool that commandBuffer was allocated from, in vkCmdWaitEvents2
• The VkCommandPool that commandBuffer was allocated from, in vkCmdPipelineBarrier2
• The VkCommandPool that commandBuffer was allocated from, in vkCmdWriteTimestamp2
• The VkCommandPool that commandBuffer was allocated from, in vkCmdCopyBuffer2
• The VkCommandPool that commandBuffer was allocated from, in vkCmdCopyImage2
• The VkCommandPool that commandBuffer was allocated from, in vkCmdCopyBufferToImage2
• The VkCommandPool that commandBuffer was allocated from, in vkCmdCopyImageToBuffer2
• The VkCommandPool that commandBuffer was allocated from, in vkCmdBlitImage2
• The VkCommandPool that commandBuffer was allocated from, in vkCmdResolveImage2
• The VkCommandPool that commandBuffer was allocated from, in vkCmdBeginRendering
• The VkCommandPool that commandBuffer was allocated from, in vkCmdEndRendering
• The VkCommandPool that commandBuffer was allocated from, in vkCmdSetCullMode
• The VkCommandPool that commandBuffer was allocated from, in vkCmdSetFrontFace
• The VkCommandPool that commandBuffer was allocated from, in vkCmdSetPrimitiveTopology
• The VkCommandPool that commandBuffer was allocated from, in vkCmdSetViewportWithCount
• The VkCommandPool that commandBuffer was allocated from, in vkCmdSetScissorWithCount
• The VkCommandPool that commandBuffer was allocated from, in vkCmdBindVertexBuffers2
• The VkCommandPool that commandBuffer was allocated from, in vkCmdSetDepthTestEnable
• The VkCommandPool that commandBuffer was allocated from, in vkCmdSetDepthWriteEnable
• The VkCommandPool that commandBuffer was allocated from, in vkCmdSetDepthCompareOp
• The VkCommandPool that commandBuffer was allocated from, in vkCmdSetDepthBoundsTestEnable
• The VkCommandPool that commandBuffer was allocated from, in vkCmdSetStencilTestEnable
• The VkCommandPool that commandBuffer was allocated from, in vkCmdSetStencilOp

21
3.7. Valid Usage

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

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

Note

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

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

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

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

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

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

### 3.7.1. Usage Validation

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

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

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

### 3.7.2. Implicit Valid Usage

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

#### Valid Usage for Object Handles

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

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

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

**Valid Usage for Pointers**

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

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

**Valid Usage for Strings**

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

**Valid Usage for Enumerated Types**

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

- The enumerant is defined as part of the enumerated type.
- The enumerant is not the special value (suffixed with `_MAX_ENUM`) defined for the enumerated type.

1

This special value exists only to ensure that C `enum` types are 32 bits in size. It is not part of the API, and should not be used by applications.

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

**Note**

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

**Note**

Application developers are encouraged to be careful when using `switch` statements with Vulkan API enums. This is because new extensions can add new values to existing enums. Using a `default:` statement within a `switch` may avoid future compilation issues.

This is particularly true for enums such as `VkDriverId`, which may have values added that do not belong to a corresponding new extension.
Valid Usage for Flags

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

```c
// Provided by VK_VERSION_1_0
typedef uint32_t VkFlags;
```

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

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

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

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

Only the low-order 31 bits (bit positions zero through 30) are available for use as flag bits.

*Note*

This restriction is due to poorly defined behavior by C compilers given a C enumerant value of 0x80000000. In some cases adding this enumerant value may increase the size of the underlying Vk*FlagBits type, breaking the ABI.

A collection of 64-bit flags is represented by a bitmask using the type VkFlags64:

```c
// Provided by VK_VERSION_1_3
typedef uint64_t VkFlags64;
```

When the 31 bits available in VkFlags are insufficient, the VkFlags64 type can be passed to commands and structures to represent up to 64 options. VkFlags64 is not used directly in the API. Instead, a Vk*Flags2 type which is an alias of VkFlags64, and whose name matches the corresponding Vk*FlagBits2 that are valid for that type, is used.

Any Vk*Flags2 member or parameter used in the API as an input must be a valid combination of bit flags. A valid combination is either zero or the bitwise OR of valid bit flags. A bit flag is valid if:

- The bit flag is defined as part of the Vk*FlagBits2 type, where the bits type is obtained by taking the flag type and replacing the trailing Flags2 with FlagBits2. For example, a flag value of type VkAccessFlags2KHR must contain only bit flags defined by VkAccessFlagBits2KHR.
The flag is allowed in the context in which it is being used. For example, in some cases, certain bit flags or combinations of bit flags are mutually exclusive.

Any Vk*Flags2 member or parameter returned from a query command or otherwise output from Vulkan to the application may contain bit flags undefined in its corresponding Vk*FlagBits2 type. An application cannot rely on the state of these unspecified bits.

Note
Both the Vk*FlagBits2 type, and the individual bits defined for that type, are defined as uint64_t integers in the C API. This is in contrast to the 32-bit types, where the Vk*FlagBits type is defined as a C enum and the individual bits as enumerants belonging to that enum. As a result, there is less compile-time type checking possible for the 64-bit types. This is unavoidable since there is no sufficiently portable way to define a 64-bit enum type in C99.

Valid Usage for Structure Types
Any parameter that is a structure containing a sType member must have a value of sType which is a valid VkStructureType value matching the type of the structure.

Valid Usage for Structure Pointer Chains
Any parameter that is a structure containing a void* pNext member must have a value of pNext that is either NULL, or is a pointer to a valid extending structure, containing sType and pNext members as described in the Vulkan Documentation and Extensions document in the section “Extending Structures”. The set of structures connected by pNext pointers is referred to as a pNext chain.

Each structure included in the pNext chain must be defined at runtime by either:

- a core version which is supported
- an extension which is enabled
- a supported device extension in the case of physical-device-level functionality added by the device extension

Each type of extending structure must not appear more than once in a pNext chain, including any aliases. This general rule may be explicitly overridden for specific structures.

Any component of the implementation (the loader, any enabled layers, and drivers) must skip over, without processing (other than reading the sType and pNext members) any extending structures in the chain not defined by core versions or extensions supported by that component.

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

The VkBaseInStructure structure is defined as:

```c
// Provided by VK_VERSION_1_0
```
typedef struct VkBaseInStructure {
  VkStructureType sType;
  const struct VkBaseInStructure* pNext;
} VkBaseInStructure;

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

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

The `VkBaseOutStructure` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkBaseOutStructure {
  VkStructureType sType;
  struct VkBaseOutStructure* pNext;
} VkBaseOutStructure;
```

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

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

**Valid Usage for Nested Structures**

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

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

**Valid Usage for Extensions**

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

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

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

Device functionality or behavior added by a device extension to the API **must** not be used unless that extension is supported by the device as determined by `vkEnumerateDeviceExtensionProperties`, and that extension is enabled in `VkDeviceCreateInfo`.
Valid Usage for Newer Core Versions

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

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

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

3.8. VkResult Return Codes

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

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

All return codes in Vulkan are reported via `VkResult` return values. The possible codes are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkResult {
    VK_SUCCESS = 0,
    VK_NOT_READY = 1,
    VK_TIMEOUT = 2,
    VK_EVENT_SET = 3,
    VK_EVENT_RESET = 4,
    VK_INCOMPLETE = 5,
    VK_ERROR_OUT_OF_HOST_MEMORY = -1,
    VK_ERROR_OUT_OF_DEVICE_MEMORY = -2,
    VK_ERROR_INITIALIZATION_FAILED = -3,
    VK_ERROR_DEVICE_LOST = -4,
    VK_ERROR_MEMORY_MAP_FAILED = -5,
    VK_ERROR_LAYER_NOT_PRESENT = -6,
    VK_ERROR_EXTENSION_NOT_PRESENT = -7,
    VK_ERROR_FEATURE_NOT_PRESENT = -8,
    VK_ERROR_INCOMPATIBLE_DRIVER = -9,
    VK_ERROR_TOO_MANY_OBJECTS = -10,
    VK_ERROR_FORMAT_NOT_SUPPORTED = -11,
    VK_ERROR_FRAGMENTED_POOL = -12,
    VK_ERROR_UNKNOWN = -13,
} VkResult;
// Provided by VK_VERSION_1_1
```
VK_ERROR_OUT_OF_POOL_MEMORY = -1000069000,
// Provided by VK_VERSION_1_1
VK_ERROR_INVALID_EXTERNAL_HANDLE = -1000072003,
// Provided by VK_VERSION_1_2
VK_ERROR_FRAGMENTATION = -1000161000,
// Provided by VK_VERSION_1_2
VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS = -1000257000,
// Provided by VK_VERSION_1_3
VK_PIPELINE_COMPILE_REQUIRED = 1000297000,
} VkResult;

Success Codes

- **VK_SUCCESS** Command successfully completed
- **VK_NOT_READY** A fence or query has not yet completed
- **VK_TIMEOUT** A wait operation has not completed in the specified time
- **VK_EVENT_SET** An event is signaled
- **VK_EVENT_RESET** An event is unsignaled
- **VK_INCOMPLETE** A return array was too small for the result
- **VK_PIPELINE_COMPILE_REQUIRED** A requested pipeline creation would have required compilation, but the application requested compilation to not be performed.

Error codes

- **VK_ERROR_OUT_OF_HOST_MEMORY** A host memory allocation has failed.
- **VK_ERROR_OUT_OF_DEVICE_MEMORY** A device memory allocation has failed.
- **VK_ERROR_INITIALIZATION_FAILED** Initialization of an object could not be completed for implementation-specific reasons.
- **VK_ERROR_DEVICE_LOST** The logical or physical device has been lost. See [Lost Device](#)
- **VK_ERROR_MEMORY_MAP_FAILED** Mapping of a memory object has failed.
- **VK_ERROR_LAYER_NOT_PRESENT** A requested layer is not present or could not be loaded.
- **VK_ERROR_EXTENSION_NOT_PRESENT** A requested extension is not supported.
- **VK_ERROR_FEATURE_NOT_PRESENT** A requested feature is not supported.
- **VK_ERROR_INCOMPATIBLE_DRIVER** The requested version of Vulkan is not supported by the driver or is otherwise incompatible for implementation-specific reasons.
- **VK_ERROR_TOO_MANY_OBJECTS** Too many objects of the type have already been created.
- **VK_ERROR_FORMAT_NOT_SUPPORTED** A requested format is not supported on this device.
- **VK_ERROR_FRAGMENTED_POOL** A pool allocation has failed due to fragmentation of the pool’s memory. This **must** only be returned if no attempt to allocate host or device memory was made to accommodate the new allocation. This **should** be returned in preference to VK_ERROR_OUT_OF_POOL_MEMORY, but only if the implementation is certain that the pool allocation failure was due to fragmentation.
- **VK_ERROR_OUT_OF_POOL_MEMORY** A pool memory allocation has failed. This **must** only be returned if
no attempt to allocate host or device memory was made to accommodate the new allocation. If the failure was definitely due to fragmentation of the pool, `VK_ERROR_FRAGMENTED_POOL` should be returned instead.

- `VK_ERROR_INVALID_EXTERNAL_HANDLE`: An external handle is not a valid handle of the specified type.
- `VK_ERROR_FRAGMENTATION`: A descriptor pool creation has failed due to fragmentation.
- `VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS`: A buffer creation or memory allocation failed because the requested address is not available.
- `VK_ERROR_UNKNOWN`: An unknown error has occurred; either the application has provided invalid input, or an implementation failure has occurred.

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

`VK_ERROR_OUT_OF_*_MEMORY` errors do not modify any currently existing Vulkan objects. Objects that have already been successfully created can still be used by the application.

Note

As a general rule, Free, Release, and Reset commands do not return `VK_ERROR_OUT_OF_HOST_MEMORY`, while any other command with a return code may return it. Any exceptions from this rule are described for those commands.

`VK_ERROR_UNKNOWN` will be returned by an implementation when an unexpected error occurs that cannot be attributed to valid behavior of the application and implementation. Under these conditions, it may be returned from any command returning a `VkResult`.

Note

`VK_ERROR_UNKNOWN` is not expected to ever be returned if the application behavior is valid, and if the implementation is bug-free. If `VK_ERROR_UNKNOWN` is received, the application should be checked against the latest validation layers to verify correct behavior as much as possible. If no issues are identified it could be an implementation issue, and the implementor should be contacted for support.

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

### 3.9. Numeric Representation and Computation

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

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

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

### 3.9.1. Floating-Point Computation

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

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

We require simply that numbers’ floating-point parts contain enough bits and that their exponent fields are large enough so that individual results of floating-point operations are accurate to about 1 part in $10^5$. The maximum representable magnitude for all floating-point values must be at least $2^{32}$.

\[
x \times 0 = 0 \times x = 0 \text{ for any non-infinite and non-NaN } x.
\]

\[
1 \times x = x \times 1 = x.
\]

\[
x + 0 = 0 + x = x.
\]

\[
0^0 = 1.
\]

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

The special values Inf and -Inf encode values with magnitudes too large to be represented; the special value NaN encodes “Not A Number” values resulting from undefined arithmetic operations such as $0 / 0$. Implementations may support Inf and NaN in their floating-point computations. Any computation which does not support either Inf or NaN, for which that value is an input or output will yield an undefined value.

### 3.9.2. Floating-Point Format Conversions

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

3.9.3. 16-Bit Floating-Point Numbers

16-bit floating point numbers are defined in the “16-bit floating point numbers” section of the Khronos Data Format Specification.

3.9.4. Unsigned 11-Bit Floating-Point Numbers

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

3.9.5. Unsigned 10-Bit Floating-Point Numbers

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

3.9.6. General Requirements

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

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

3.10. Fixed-Point Data Conversions

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

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

The signed and unsigned fixed-point representations are assumed to be b-bit binary two’s-complement integers and binary unsigned integers, respectively.
3.10.1. Conversion from Normalized Fixed-Point to Floating-Point

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

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

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

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

Only the range \([-2^{b-1} + 1, 2^{b-1} - 1]\) is used to represent signed fixed-point values in the range \([-1, 1]\). For example, if \(b = 8\), then the integer value -127 corresponds to -1.0 and the value 127 corresponds to 1.0. This equation is used everywhere that signed normalized fixed-point values are converted to floating-point.

Note that while zero is exactly expressible in this representation, one value (-128 in the example) is outside the representable range, and implementations must clamp it to -1.0. Where the value is subject to further processing by the implementation, e.g. during texture filtering, values less than -1.0 may be used but the result must be clamped before the value is returned to shaders.

3.10.2. Conversion from Floating-Point to Normalized Fixed-Point

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

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

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

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

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

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

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

3.11. Common Object Types

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

3.11.1. Offsets

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

A two-dimensional offset is defined by the structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkOffset2D {
    int32_t  x;
    int32_t  y;
} VkOffset2D;
```

- \(x\) is the \(x\) offset.
- \(y\) is the \(y\) offset.

A three-dimensional offset is defined by the structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkOffset3D {
    int32_t  x;
    int32_t  y;
    int32_t  z;
} VkOffset3D;
```

- \(x\) is the \(x\) offset.
- \(y\) is the \(y\) offset.
- \(z\) is the \(z\) offset.

3.11.2. Extents

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

A two-dimensional extent is defined by the structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkExtent2D {
    int32_t  width;
    int32_t  height;
} VkExtent2D;
```
• *width* is the width of the extent.
• *height* is the height of the extent.

A three-dimensional extent is defined by the structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkExtent3D {
    uint32_t width;
    uint32_t height;
    uint32_t depth;
} VkExtent3D;
```

• *width* is the width of the extent.
• *height* is the height of the extent.
• *depth* is the depth of the extent.

### 3.11.3. Rectangles

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

```c
// Provided by VK_VERSION_1_0
typedef struct VkRect2D {
    VkOffset2D offset;
    VkExtent2D extent;
} VkRect2D;
```

• *offset* is a *VkOffset2D* specifying the rectangle offset.
• *extent* is a *VkExtent2D* specifying the rectangle extent.

### 3.11.4. Structure Types

Each value corresponds to a particular structure with a *sType* member with a matching name. As a general rule, the name of each *VkStructureType* value is obtained by taking the name of the structure, stripping the leading *Vk*, prefixing each capital letter with _, converting the entire resulting string to upper case, and prefixing it with VK_STRUCTURE_TYPE_. For example, structures of type *VkImageCreateInfo* correspond to a *VkStructureType* value of VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO, and thus a structure of this type must have its *sType* member set to this value before it is passed to the API.
The values `VK_STRUCTURE_TYPE_LOADER_INSTANCE_CREATE_INFO` and `VK_STRUCTURE_TYPE_LOADER_DEVICE_CREATE_INFO` are reserved for internal use by the loader, and do not have corresponding Vulkan structures in this Specification.

Structure types supported by the Vulkan API include:

```c
// Provided by VK_VERSION_1_0
typedef enum VkStructureType {
    VK_STRUCTURE_TYPE_APPLICATION_INFO = 0,
    VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO = 1,
    VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO = 2,
    VK_STRUCTURE_TYPEDEVICE_CREATE_INFO = 3,
    VK_STRUCTURE_TYPE_SUBMIT_INFO = 4,
    VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO = 5,
    VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE = 6,
    VK_STRUCTURE_TYPE_BIND_SPARSE_INFO = 7,
    VK_STRUCTURE_TYPEFENCE_CREATE_INFO = 8,
    VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO = 9,
    VK_STRUCTURE_TYPE_EVENT_CREATE_INFO = 10,
    VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO = 11,
    VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO = 12,
    VK_STRUCTURE_TYPE_BUFFER_VIEW_CREATE_INFO = 13,
    VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO = 14,
    VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO = 15,
    VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO = 16,
    VK_STRUCTURE_TYPE_PIPELINE_CACHE_CREATE_INFO = 17,
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO = 18,
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO = 19,
    VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO = 20,
    VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO = 21,
    VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO = 22,
    VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO = 23,
    VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO = 24,
    VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO = 25,
    VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO = 26,
    VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO = 27,
    VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO = 28,
    VK_STRUCTURE_TYPECOMPUTE_PIPELINE_CREATE_INFO = 29,
    VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO = 30,
    VK_STRUCTURE_TYPESAALAR_CREATE_INFO = 31,
    VK_STRUCTURE_TYPEDESCRIPOR_SET_LAYOUT_CREATE_INFO = 32,
    VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO = 33,
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO = 34,
    VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET = 35,
    VK_STRUCTURE_TYPE_COPY_DESCRIPTOR_SET = 36,
    VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO = 37,
    VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO = 38,
    VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO = 39,
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO = 40,
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_INFO = 41,
};
```
VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO = 42,
VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO = 43,
VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER = 44,
VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER = 45,
VK_STRUCTURE_TYPE_MEMORY_BARRIER = 46,
VK_STRUCTURE_TYPE_LOADER_INSTANCE_CREATE_INFO = 47,
VK_STRUCTURE_TYPE_LOADER_DEVICE_CREATE_INFO = 48,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_PROPERTIES = 1000094000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO = 1000157000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_INFO = 1000157001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_16BIT_STORAGE_FEATURES = 1000083000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_MEMORY_DEDICATED_REQUIREMENTS = 1000127000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_MEMORY_DEDICATE_ALLOCATE_INFO = 1000127001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_FLAGS_INFO = 1000060000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DEVICE_GROUP_RENDER_PASS_BEGIN_INFO = 1000060003,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DEVICE_GROUP_COMMAND_BUFFER_BEGIN_INFO = 1000060004,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DEVICE_GROUP_SUBMIT_INFO = 1000060005,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DEVICE_GROUP_BIND_SPARSE_INFO = 1000060006,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_DEVICE_GROUP_INFO = 1000060013,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_DEVICE_GROUP_INFO = 1000060014,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GROUP_PROPERTIES = 1000070000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DEVICE_GROUP_DEVICE_CREATE_INFO = 1000070001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_BUFFER_MEMORY_REQUIREMENTS_INFO_2 = 1000146000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_IMAGE_MEMORY_REQUIREMENTS_INFO_2 = 1000146001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_IMAGE_SPARSE_MEMORY_REQUIREMENTS_INFO_2 = 1000146002,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_MEMORY_REQUIREMENTS_2 = 1000146003,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_SPARSE_IMAGE_MEMORY_REQUIREMENTS_2 = 1000146004,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2 = 1000059000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2 = 1000059001;
VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_2 = 1000059002,
VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2 = 1000059003,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_FORMAT_INFO_2 = 1000059004,
VK_STRUCTURE_TYPE_QUEUE_FAMILY_PROPERTIES_2 = 1000059005,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PROPERTIES_2 = 1000059006,
VK_STRUCTURE_TYPE_SPARSE_IMAGE_FORMAT_PROPERTIES_2 = 1000059007,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SPARSE_IMAGE_FORMAT_INFO_2 = 1000059008,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_POINT_CLIPPING_PROPERTIES = 1000117000,
VK_STRUCTURE_TYPE_RENDER_PASS_INPUT_ATTACHMENT_ASPECT_CREATE_INFO = 1000117001,
VK_STRUCTURE_TYPE_IMAGE_VIEW_USAGE_CREATE_INFO = 1000117002,
VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_DOMAIN_ORIGIN_STATE_CREATE_INFO = 1000117003,
VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO = 1000053000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_FEATURES = 1000053001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PROPERTIES = 1000053002,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTERS_FEATURES = 1000120000,
VK_STRUCTURE_TYPE_PROTECTED_SUBMIT_INFO = 1000145000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_FEATURES = 1000145001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_PROPERTIES = 1000145002,
VK_STRUCTURE_TYPE_DEVICE_QUEUE_INFO_2 = 1000145003,
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_CREATE_INFO = 1000156000,
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_INFO = 1000156001,
VK_STRUCTURE_TYPE_BIND_IMAGE_PLANE_MEMORY_INFO = 1000156002,
VK_STRUCTURE_TYPE_IMAGE_PLANE_MEMORY_REQUIREMENTS_INFO = 1000156003,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_YCBCR_CONVERSION_FEATURES = 1000156004,
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_IMAGE_FORMAT_PROPERTIES = 1000156005,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO = 1000085000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_IMAGE_FORMAT_INFO = 1000071000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXTERNAL_IMAGE_FORMAT_PROPERTIES = 1000071001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_BUFFER_INFO = 1000071002,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXTERNAL_BUFFER_PROPERTIES = 1000071003,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_PROPERTIES = 1000071004,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO = 1000072000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO = 1000072001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO = 1000072002,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_FENCE_INFO = 1000112000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXTERNAL_FENCE_PROPERTIES = 1000112001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXPORT_FENCE_CREATE_INFO = 1000113000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO = 1000077000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SEMAPHORE_INFO = 1000076000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES = 1000076001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_3_PROPERTIES = 1000168000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_SUPPORT = 1000168001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICALDEVICE_SHADER_DRAW_PARAMETERS_FEATURES = 1000063000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICALDEVICE_VULKAN_1_1_FEATURES = 49,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICALDEVICE_VULKAN_1_1_PROPERTIES = 50,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICALDEVICE_VULKAN_1_2_FEATURES = 51,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICALDEVICE_VULKAN_1_2_PROPERTIES = 52,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_IMAGE_FORMAT_LIST_CREATE_INFO = 1000147000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_2 = 1000109000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_2 = 1000109001,
VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_2 = 1000109002,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_SUBPASS_DEPENDENCY_2 = 1000109003,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO_2 = 1000109004,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_SUBPASS_BEGIN_INFO = 1000109005,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_SUBPASS_END_INFO = 1000109006,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_8BIT_STORAGE_FEATURES = 1000177000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DRIVER_PROPERTIES = 1000196000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_INT64_FEATURES = 1000180000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_FLOAT16_INT8_FEATURES = 1000082000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FLOAT_CONTROLS_PROPERTIES = 1000197000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_BINDING_FLAGS_CREATE_INFO = 1000161000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_FEATURES = 1000161001,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_PROPERTIES = 1000161002,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_ALLOCATE_INFO = 1000161003,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_LAYOUT_SUPPORT = 1000161004,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEPTH_STENCIL_RESOLVE_PROPERTIES = 1000199000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_DEPTH_STENCIL_RESOLVE = 1000199001,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SCALAR_BLOCK_LAYOUT_FEATURES = 1000221000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_IMAGE_STENCIL_USAGE_CREATE_INFO = 1000246000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_FILTER_MINMAX_PROPERTIES = 1000130000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_SAMPLER_REDUCTION_MODE_CREATE_INFO = 1000130001,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_MEMORY_MODEL_FEATURES = 1000211000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGELESS_FRAMEBUFFER_FEATURES = 1000108000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENTS_CREATE_INFO = 1000108001,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENT_IMAGE_INFO = 1000108002,
VK_STRUCTURE_TYPE_RENDER_PASS_ATTACHMENT_BEGIN_INFO = 1000108003,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_UNIFORM_BUFFER_STANDARD_LAYOUT_FEATURES = 1000253000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SUBGROUP_EXTENDED_TYPES_FEATURES = 1000175000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SEPARATE_DEPTH_STENCIL_LAYOUTS_FEATURES = 1000241000,
VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_STENCIL_LAYOUT = 1000241001,
VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_STENCIL_LAYOUT = 1000241002,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_HOST_QUERY_RESET_FEATURES = 1000261000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_FEATURES = 1000207000,
VK_STRUCTURE_TYPE_PHYSICALDEVICE_TIMELINE_SEMAPHORE_PROPERTIES = 1000207001,
VK_STRUCTURE_TYPE_SEMAPHORE_TYPE_CREATE_INFO = 1000207002,
VK_STRUCTURE_TYPE_TIMELINE_SEMAPHORE_SUBMIT_INFO = 1000207003,
VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO = 1000207004,
VK_STRUCTURE_TYPE_SEMAPHORE_SIGNAL_INFO = 1000207005,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES = 1000257000,
VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO = 1000244001,
VK_STRUCTURE_TYPE_BUFFER_OPAQUE_CAPTURE_ADDRESS_CREATE_INFO = 1000257002,
VK_STRUCTURE_TYPE_MEMORY_OPAQUE_CAPTURE_ADDRESS_ALLOCATE_INFO = 1000257003,
VK_STRUCTURE_TYPE_DEVICE_MEMORY_OPAQUE_CAPTURE_ADDRESS_INFO = 1000257004,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_3_FEATURES = 53,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_3_PROPERTIES = 54,
VK_STRUCTURE_TYPE_PIPELINE_CREATION_FEEDBACK_CREATE_INFO = 1000192000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_TERMINATE_INVOCATION_FEATURES = 1000215000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TOOL_PROPERTIES = 1000245000.
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DEMOTE_TO_HELPER_INVOCATION_FEATURES = 1000276000,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PRIVATE_DATA_FEATURES = 1000295000,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_DEVICE_PRIVATE_DATA_CREATE_INFO = 1000295001,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_PRIVATE_DATA_SLOT_CREATE_INFO = 1000295002,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PIPELINE_CREATION_CACHE_CONTROL_FEATURES = 1000297000,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_MEMORY_BARRIER_2 = 1000314000,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER_2 = 1000314001,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER_2 = 1000314002,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_DEPENDENCY_INFO = 1000314003,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_SUBMIT_INFO_2 = 1000314004,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_SEMAPHORE_SUBMIT_INFO = 1000314005,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_SUBMIT_INFO = 1000314006,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SYNCHRONIZATION_2_FEATURES = 1000314007,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ZERO_INITIALIZE_WORKGROUP_MEMORY_FEATURES = 1000325000,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_ROBUSTNESS_FEATURES = 1000335000,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_COPY_BUFFER_INFO_2 = 1000337000,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_COPY_IMAGE_INFO_2 = 1000337001,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_COPY_BUFFER_TO_IMAGE_INFO_2 = 1000337002,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_COPY_IMAGE_TO_BUFFER_INFO_2 = 1000337003,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_BLIT_IMAGE_INFO_2 = 1000337004,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_RESOLVE_IMAGE_INFO_2 = 1000337005,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_BUFFER_COPY_2 = 1000337006,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_IMAGE_COPY_2 = 1000337007,
    // Provided by VK_VERSION_1_3
    VK_STRUCTURE_TYPE_IMAGE_BLIT_2 = 1000337008,
    // Provided by VK_VERSION_1_3
VK_STRUCTURE_TYPE_BUFFER_IMAGE_COPY_2 = 0x000337009,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_IMAGE_RESOLVE_2 = 0x000337010,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_PROPERTIES = 0x000225000,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPEPIPELINE_SHADER_STAGE_REQUIRED_SUBGROUP_SIZE_CREATE_INFO = 0x000225001,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_FEATURES = 0x000225002,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INLINE_UNIFORM_BLOCK_FEATURES = 0x000138000,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INLINE_UNIFORM_BLOCK_PROPERTIES = 0x000138001,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_INLINE_UNIFORM_BLOCK = 0x000138002,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_INLINE_UNIFORM_BLOCK_CREATE_INFO = 0x000138003,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXTURE_COMPRESSION_ASTC_HDR_FEATURES = 0x000066000,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_RENDERING_INFO = 0x000044000,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_RENDERING_ATTACHMENT_INFO = 0x000044001,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_PIPELINE_RENDERING_CREATE_INFO = 0x000044002,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DYNAMIC_RENDERING_FEATURES = 0x000044003,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_RENDERING_INFO = 0x000044004,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_INTEGER_DOT_PRODUCT_FEATURES = 0x000280000,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_INTEGER_DOT_PRODUCT_PROPERTIES = 0x000280001,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_PROPERTIES = 0x000281001,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_3 = 0x000360000,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_4_FEATURES = 0x000413000,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_4_PROPERTIES = 0x000413001,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_DEVICE_BUFFER_MEMORY_REQUIREMENTS = 0x000413002,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_DEVICE_IMAGE_MEMORY_REQUIREMENTS = 0x000413003,
// Provided by VK_VERSION_1.3
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTER_FEATURES = 0x000413004,
// Provided by VK_VERSION_1.3
3.12. API Name Aliases

A small number of APIs did not follow the naming conventions when initially defined. For consistency, when we discover an API name that violates the naming conventions, we rename it in the Specification, XML, and header files. For backwards compatibility, the original (incorrect) name is retained as a “typo alias”. The alias is deprecated and should not be used, but will be retained indefinitely.

*Note*

VK_STENCIL_FRONT_AND_BACK is an example of a typo alias. It was initially defined as part of VkStencilFaceFlagBits. Once the naming inconsistency was noticed, it was renamed to VK_STENCIL_FACE_FRONT_AND_BACK, and the old name was aliased to the correct name.
Chapter 4. Initialization

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

4.1. Command Function Pointers

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

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

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

```c
// Provided by VK_VERSION_1_0
PFN_vkVoidFunction vkGetInstanceProcAddr(
    VkInstance instance, const char* pName);
```

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

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

The table below defines the various use cases for vkGetInstanceProcAddr and expected return value (“fp” is “function pointer”) for each case. A valid returned function pointer (“fp”) must not be NULL.

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

<table>
<thead>
<tr>
<th>instance</th>
<th>pName</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>*1 NULL</td>
<td>NULL</td>
<td>undefined</td>
</tr>
<tr>
<td>invalid non-NULL</td>
<td>*1</td>
<td>undefined</td>
</tr>
<tr>
<td>NULL</td>
<td>global command</td>
<td>fp</td>
</tr>
<tr>
<td>instance</td>
<td>pName</td>
<td>return value</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>NULL</td>
<td>vkGetInstanceProcAddr</td>
<td>fp&lt;sup&gt;5&lt;/sup&gt;</td>
</tr>
<tr>
<td>instance</td>
<td>vkGetInstanceProcAddr</td>
<td>fp</td>
</tr>
<tr>
<td>instance</td>
<td>core dispatchable command</td>
<td>fp&lt;sup&gt;3&lt;/sup&gt;</td>
</tr>
<tr>
<td>instance</td>
<td>enabled instance extension dispatchable command for instance</td>
<td>fp&lt;sup&gt;3&lt;/sup&gt;</td>
</tr>
<tr>
<td>instance</td>
<td>available device extension dispatchable command for instance</td>
<td>fp&lt;sup&gt;3&lt;/sup&gt;</td>
</tr>
<tr>
<td>any other case, not covered above</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

1

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

2

The global commands are: vkEnumerateInstanceVersion, vkEnumerateInstanceExtensionProperties, vkEnumerateInstanceLayerProperties, and vkCreateInstance. Dispatchable commands are all other commands which are not global.

3

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

4

An “available device extension” is a device extension supported by any physical device enumerated by instance.

5

Starting with Vulkan 1.2, vkGetInstanceProcAddr can resolve itself with a NULL instance pointer.

### Valid Usage (Implicit)

- VUID-vkGetInstanceProcAddr-instance-parameter
  If instance is not NULL, instance must be a valid VkInstance handle
- VUID-vkGetInstanceProcAddr-pName-parameter
  pName must be a null-terminated UTF-8 string

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

```c
// Provided by VK_VERSION_1_0
PFN_vkVoidFunction vkGetDeviceProcAddr(
    VkDevice device,
    const char* pName);
```

The table below defines the various use cases for `vkGetDeviceProcAddr` and expected return value (“fp” is “function pointer”) for each case. A valid returned function pointer (“fp”) must not be `NULL`.

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

<table>
<thead>
<tr>
<th>device</th>
<th>pName</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>*1</td>
<td>undefined</td>
</tr>
<tr>
<td>invalid device</td>
<td>*1</td>
<td>undefined</td>
</tr>
<tr>
<td>device</td>
<td>NULL</td>
<td>undefined</td>
</tr>
<tr>
<td>device</td>
<td>requested core version</td>
<td>fp^4</td>
</tr>
<tr>
<td></td>
<td>device-level dispatchable command</td>
<td></td>
</tr>
<tr>
<td>device</td>
<td>enabled extension</td>
<td>fp^4</td>
</tr>
<tr>
<td></td>
<td>device-level dispatchable command</td>
<td></td>
</tr>
<tr>
<td>any other case, not covered above</td>
<td></td>
<td>NULL</td>
</tr>
</tbody>
</table>

1

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

2

Device-level commands which are part of the core version specified by `VkApplicationInfo::apiVersion` when creating the instance will always return a valid function pointer. Core commands beyond that version which are supported by the implementation may either return `NULL` or a function pointer, though the function pointer must not be called.

3

In this function, device-level excludes all physical-device-level commands.

4

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

- VUID-vkGetDeviceProcAddr-device-parameter
  - `device` must be a valid `VkDevice` handle
- VUID-vkGetDeviceProcAddr-pName-parameter
  - `pName` must be a null-terminated UTF-8 string

The definition of `PFN_vkVoidFunction` is:

```c
// Provided by VK_VERSION_1_0
typedef void (VKAPI_PTR *PFN_vkVoidFunction)(void);
```

This type is returned from command function pointer queries, and **must** be cast to an actual command function pointer before use.

### 4.1.1. Extending Physical Device Core Functionality

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

### 4.1.2. Extending Physical Device From Device Extensions

When the `VK_KHR_get_physical_device_properties2` extension is enabled, or when both the instance and the physical-device versions are at least 1.1, physical-device-level functionality of a device extension **can** be used with a physical device if the corresponding extension is enumerated by `vkEnumerateDeviceExtensionProperties` for that physical device, even before a logical device has been created.

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

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

### 4.2. Instances

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

Instances are represented by `VkInstance` handles:
To query the version of instance-level functionality supported by the implementation, call:

```c
// Provided by VK_VERSION_1_1
VkResult vkEnumerateInstanceVersion(
    uint32_t* pApiVersion);
```

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

**Note**

The intended behaviour of `vkEnumerateInstanceVersion` is that an implementation should not need to perform memory allocations and should unconditionally return `VK_SUCCESS`. The loader, and any enabled layers, may return `VK_ERROR_OUT_OF_HOST_MEMORY` in the case of a failed memory allocation.

### Valid Usage (Implicit)

- `VUID-vkEnumerateInstanceVersion-pApiVersion-parameter`
  
  `pApiVersion` must be a valid pointer to a `uint32_t` value

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`

To create an instance object, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateInstance(
    const VkInstanceCreateInfo* pCreateInfo, 
    const VkAllocationCallbacks* pAllocator, 
    VkInstance* pInstance);
```

- `pCreateInfo` is a pointer to a `VkInstanceCreateInfo` structure controlling creation of the instance.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pInstance` points a `VkInstance` handle in which the resulting instance is returned.
vkCreateInstance verifies that the requested layers exist. If not, vkCreateInstance will return VK_ERROR_LAYER_NOT_PRESENT. Next vkCreateInstance verifies that the requested extensions are supported (e.g. in the implementation or in any enabled instance layer) and if any requested extension is not supported, vkCreateInstance must return VK_ERROR_EXTENSION_NOT_PRESENT. After verifying and enabling the instance layers and extensions the VkInstance object is created and returned to the application. If a requested extension is only supported by a layer, both the layer and the extension need to be specified at vkCreateInstance time for the creation to succeed.

### Valid Usage

- VUID-vkCreateInstance-ppEnabledExtensionNames-01388
  All required extensions for each extension in the VkInstanceCreateInfo::ppEnabledExtensionNames list must also be present in that list

### Valid Usage (Implicit)

- VUID-vkCreateInstance-pCreateInfo-parameter
  pCreateInfo must be a valid pointer to a valid VkInstanceCreateInfo structure

- VUID-vkCreateInstance-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkCreateInstance-pInstance-parameter
  pInstance must be a valid pointer to a VkInstance handle

### Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_INITIALIZATION_FAILED
- VK_ERROR_LAYER_NOT_PRESENT
- VK_ERROR_EXTENSION_NOT_PRESENT
- VK_ERROR_INCOMPATIBLE_DRIVER

The VkInstanceCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkInstanceCreateInfo {
    VkStructureType sType;
    // Other fields
} VkInstanceCreateInfo;
```
const void* pNext;
VkInstanceCreateFlags flags;
const VkApplicationInfo* pApplicationInfo;
uint32_t enabledLayerCount;
const char* const* ppEnabledLayerNames;
const char* const* ppEnabledExtensionNames;

VkInstanceCreateInfo

• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **flags** is a bitmask of **VkInstanceCreateFlagBits** indicating the behavior of the instance.
• **pApplicationInfo** is NULL or a pointer to a **VkApplicationInfo** structure. If not NULL, this information helps implementations recognize behavior inherent to classes of applications. **VkApplicationInfo** is defined in detail below.
• **enabledLayerCount** is the number of global layers to enable.
• **ppEnabledLayerNames** is a pointer to an array of **enabledLayerCount** null-terminated UTF-8 strings containing the names of layers to enable for the created instance. The layers are loaded in the order they are listed in this array, with the first array element being the closest to the application, and the last array element being the closest to the driver. See the **Layers** section for further details.
• **enabledExtensionCount** is the number of global extensions to enable.
• **ppEnabledExtensionNames** is a pointer to an array of **enabledExtensionCount** null-terminated UTF-8 strings containing the names of extensions to enable.

---

**Valid Usage (Implicit)**

• **VUID-VkInstanceCreateInfo-sType-sType**
  * **sType** must be **VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO**

• **VUID-VkInstanceCreateInfo-pNext-pNext**
  * **pNext** must be NULL

• **VUID-VkInstanceCreateInfo-flags-zerobitmask**
  * **flags** must be 0

• **VUID-VkInstanceCreateInfo-pApplicationInfo-parameter**
  If **pApplicationInfo** is not NULL, **pApplicationInfo** must be a valid pointer to a valid **VkApplicationInfo** structure

• **VUID-VkInstanceCreateInfo-ppEnabledLayerNames-parameter**
  If **enabledLayerCount** is not 0, **ppEnabledLayerNames** must be a valid pointer to an array of **enabledLayerCount** null-terminated UTF-8 strings

• **VUID-VkInstanceCreateInfo-ppEnabledExtensionNames-parameter**
  If **enabledExtensionCount** is not 0, **ppEnabledExtensionNames** must be a valid pointer to an array of **enabledExtensionCount** null-terminated UTF-8 strings
typedef enum VkInstanceCreateFlagBits {
} VkInstanceCreateFlagBits;

Note
All bits for this type are defined by extensions, and none of those extensions are enabled in this build of the specification.

typedef VkFlags VkInstanceCreateFlags;

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

The VkApplicationInfo structure is defined as:

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

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

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

**Note**

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

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

**Note**

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

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

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

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

**Note**

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

### Valid Usage

- **VUID-VkApplicationInfo-apiVersion-04010**
  
  If apiVersion is not 0, then it must be greater than or equal to VK_API_VERSION_1_0
Valid Usage (Implicit)

- VUID-VkApplicationInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_APPLICATION_INFO`

- VUID-VkApplicationInfo-pNext-pNext
  `pNext` must be `NULL`

- VUID-VkApplicationInfo-pApplicationName-parameter
  If `pApplicationName` is not `NULL`, `pApplicationName` must be a null-terminated UTF-8 string

- VUID-VkApplicationInfo-pEngineName-parameter
  If `pEngineName` is not `NULL`, `pEngineName` must be a null-terminated UTF-8 string

To destroy an instance, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyInstance(
    VkInstance instance,                
    const VkAllocationCallbacks* pAllocator);
```

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

Valid Usage

- VUID-vkDestroyInstance-instance-00629
  All child objects created using `instance` must have been destroyed prior to destroying `instance`

- VUID-vkDestroyInstance-instance-00630
  If `VkAllocationCallbacks` were provided when `instance` was created, a compatible set of callbacks must be provided here

- VUID-vkDestroyInstance-instance-00631
  If no `VkAllocationCallbacks` were provided when `instance` was created, `pAllocator` must be `NULL`

Valid Usage (Implicit)

- VUID-vkDestroyInstance-instance-parameter
  If `instance` is not `NULL`, `instance` must be a valid `VkInstance` handle

- VUID-vkDestroyInstance-pAllocator-parameter
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
Host Synchronization

- Host access to instance must be externally synchronized
- Host access to all VkPhysicalDevice objects enumerated from instance must be externally synchronized
Chapter 5. Devices and Queues

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

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

Physical devices are represented by VkPhysicalDevice handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_HANDLE(VkPhysicalDevice)
```

## 5.1. Physical Devices

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

```c
// Provided by VK_VERSION_1_0
VkResult vkEnumeratePhysicalDevices(VkInstance instance, uint32_t* pPhysicalDeviceCount, VkPhysicalDevice* pPhysicalDevices);
```

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

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

### Valid Usage (Implicit)

- VUID-vkEnumeratePhysicalDevices-instance-parameter
  - `instance` must be a valid `VkInstance` handle
- VUID-vkEnumeratePhysicalDevices-pPhysicalDeviceCount-parameter
pPhysicalDeviceCount must be a valid pointer to a uint32_t value

- VUID-vkEnumeratePhysicalDevices-pPhysicalDevices-parameter
  If the value referenced by pPhysicalDeviceCount is not 0, and pPhysicalDevices is not NULL, pPhysicalDevices must be a valid pointer to an array of pPhysicalDeviceCount VkPhysicalDevice handles

### Return Codes

**Success**
- VK_SUCCESS
- VK_INCOMPLETE

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_INITIALIZATION_FAILED

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

```c
// Provided by VK_VERSION_1_0
void vkGetPhysicalDeviceProperties(
  VkPhysicalDevice physicalDevice,
  VkPhysicalDeviceProperties* pProperties);
```

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

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceProperties-physicalDevice-parameter
  `physicalDevice` must be a valid VkPhysicalDevice handle
- VUID-vkGetPhysicalDeviceProperties-pProperties-parameter
  `pProperties` must be a valid pointer to a VkPhysicalDeviceProperties structure

The `VkPhysicalDeviceProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPhysicalDeviceProperties {
  uint32_t apiVersion;
  uint32_t driverVersion;
  uint32_t vendorID;
} VkPhysicalDeviceProperties;
```
uint32_t deviceID;
VkPhysicalDeviceType deviceType;
char deviceName[VK_MAX_PHYSICAL_DEVICE_NAME_SIZE];
pipelineCacheUUID[VK_UUID_SIZE];
limits;
sparseProperties;

VkPhysicalDeviceProperties {
    apiVersion is the version of Vulkan supported by the device, encoded as described in Version Numbers.
    driverVersion is the vendor-specified version of the driver.
    vendorID is a unique identifier for the vendor (see below) of the physical device.
    deviceID is a unique identifier for the physical device among devices available from the vendor.
    deviceType is a VkPhysicalDeviceType specifying the type of device.
    deviceName is an array of VK_MAX_PHYSICAL_DEVICE_NAME_SIZE char containing a null-terminated UTF-8 string which is the name of the device.
    pipelineCacheUUID is an array of VK_UUID_SIZE uint8_t values representing a universally unique identifier for the device.
    limits is the VkPhysicalDeviceLimits structure specifying device-specific limits of the physical device. See Limits for details.
    sparseProperties is the VkPhysicalDeviceSparseProperties structure specifying various sparse related properties of the physical device. See Sparse Properties for details.

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

Note
The encoding of driverVersion is implementation-defined. It may not use the same encoding as apiVersion. Applications should follow information from the vendor on how to extract the version information from driverVersion.

On implementations that claim support for the Roadmap 2022 profile, the major and minor version expressed by apiVersion must be at least Vulkan 1.3.

The vendorID and deviceID fields are provided to allow applications to adapt to device characteristics that are not adequately exposed by other Vulkan queries.
Note
These may include performance profiles, hardware errata, or other characteristics.

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

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

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

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

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

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

```c
// Provided by VK_VERSION_1_0
typedef enum VkVendorId {
    VK_VENDOR_ID_VIV = 0x10001,
    VK_VENDOR_ID_VSI = 0x10002,
    VK_VENDOR_ID_KAZAN = 0x10003,
    VK_VENDOR_ID_CODEPLAY = 0x10004,
    VK_VENDOR_ID_MESA = 0x10005,
    VK_VENDOR_ID_POCL = 0x10006,
} VkVendorId;
```

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

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

`VK_MAX_PHYSICAL_DEVICE_NAME_SIZE` is the length in `char` values of an array containing a physical device name string, as returned in `VkPhysicalDeviceProperties::deviceName`.

```c
#define VK_MAX_PHYSICAL DEVICE_NAME_SIZE 256U
```

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

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

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

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

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

```c
// Provided by VK_VERSION_1_1
void vkGetPhysicalDeviceProperties2(
    VkPhysicalDevice physicalDevice,
    VkPhysicalDeviceProperties2* pProperties);
```

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

Each structure in **pProperties** and its **pNext** chain contains members corresponding to implementation-dependent properties, behaviors, or limits. **vkGetPhysicalDeviceProperties2** fills in each member to specify the corresponding value for the implementation.

## Valid Usage (Implicit)

- **VUID-vkGetPhysicalDeviceProperties2-physicalDevice-parameter**
  - **physicalDevice** must be a valid `VkPhysicalDevice` handle

- **VUID-vkGetPhysicalDeviceProperties2-pProperties-parameter**
  - **pProperties** must be a valid pointer to a `VkPhysicalDeviceProperties2` structure

The `VkPhysicalDeviceProperties2` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceProperties2 {
    VkStructureType sType;
    void* pNext;
    VkPhysicalDeviceProperties properties;
} VkPhysicalDeviceProperties2;
```

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

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

## Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceProperties2-sType-sType**
  - **sType** must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2`

- **VUID-VkPhysicalDeviceProperties2-pNext-pNext**
  - **pNext** chain must be either **NULL** or a pointer to a valid instance of `VkPhysicalDeviceDepthStencilResolveProperties`, `VkPhysicalDeviceDescriptorIndexingProperties`, `VkPhysicalDeviceDriverProperties`, `VkPhysicalDeviceFloatControlsProperties`, `VkPhysicalDeviceIDProperties`, `VkPhysicalDeviceInlineUniformBlockProperties`, `VkPhysicalDeviceMaintenance3Properties`, `VkPhysicalDeviceMaintenance4Properties`, `VkPhysicalDeviceMultiviewProperties`, `VkPhysicalDevicePointClippingProperties`, `VkPhysicalDeviceProtectedMemoryProperties`, etc.
The `VkPhysicalDeviceVulkan11Properties` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceVulkan11Properties {
    VkStructureType sType;
    void* pNext;
    uint8_t deviceUUID[VK_UUID_SIZE];
    uint8_t driverUUID[VK_UUID_SIZE];
    uint8_t deviceLUID[VK_LUID_SIZE];
    VkBool32 deviceLUIDValid;
    uint32_t subgroupSize;
    VkShaderStageFlags subgroupSupportedStages;
    VkSubgroupFeatureFlags subgroupSupportedOperations;
    VkBool32 subgroupQuadOperationsInAllStages;
    VkPointClippingBehavior pointClippingBehavior;
    uint32_t maxMultiviewViewCount;
    uint32_t maxMultiviewInstanceIndex;
    VkBool32 protectedNoFault;
    uint32_t maxPerSetDescriptors;
    VkDeviceSize maxMemoryAllocationSize;
} VkPhysicalDeviceVulkan11Properties;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **deviceUUID** is an array of `VK_UUID_SIZE` `uint8_t` values representing a universally unique identifier for the device.
- **driverUUID** is an array of `VK_UUID_SIZE` `uint8_t` values representing a universally unique identifier for the driver build in use by the device.
- **deviceLUID** is an array of `VK_LUID_SIZE` `uint8_t` values representing a locally unique identifier for the device.
- **deviceNodeMask** is a `uint32_t` bitfield identifying the node within a linked device adapter corresponding to the device.
- **deviceLUIDValid** is a boolean value that will be `VK_TRUE` if `deviceLUID` contains a valid LUID and `deviceNodeMask` contains a valid node mask, and `VK_FALSE` if they do not.
- **subgroupSize** is the default number of invocations in each subgroup. *subgroupSize* is at least 1 if any of the physical device's queues support **VK_QUEUE_GRAPHICS_BIT** or **VK_QUEUE_COMPUTE_BIT**. *subgroupSize* is a power-of-two.

- **subgroupSupportedStages** is a bitfield of **VkShaderStageFlagBits** describing the shader stages that group operations with subgroup scope are supported in. *subgroupSupportedStages* will have the **VK_SHADER_STAGE_COMPUTE_BIT** bit set if any of the physical device's queues support **VK_QUEUE_COMPUTE_BIT**.

- **subgroupSupportedOperations** is a bitmask of **VkSubgroupFeatureFlagBits** specifying the sets of group operations with subgroup scope supported on this device. *subgroupSupportedOperations* will have the **VK_SUBGROUP_FEATURE_BASIC_BIT** bit set if any of the physical device's queues support **VK_QUEUE_GRAPHICS_BIT** or **VK_QUEUE_COMPUTE_BIT**.

- **subgroupQuadOperationsInAllStages** is a boolean specifying whether quad group operations are available in all stages, or are restricted to fragment and compute stages.

- **pointClippingBehavior** is a **VkPointClippingBehavior** value specifying the point clipping behavior supported by the implementation.

- **maxMultiviewViewCount** is one greater than the maximum view index that can be used in a subpass.

- **maxMultiviewInstanceIndex** is the maximum valid value of instance index allowed to be generated by a drawing command recorded within a subpass of a multiview render pass instance.

- **protectedNoFault** specifies how an implementation behaves when an application attempts to write to unprotected memory in a protected queue operation, read from protected memory in an unprotected queue operation, or perform a query in a protected queue operation. If this limit is **VK_TRUE**, such writes will be discarded or have undefined values written, reads and queries will return undefined values. If this limit is **VK_FALSE**, applications must not perform these operations. See Protected Memory Access Rules for more information.

- **maxPerSetDescriptors** is a maximum number of descriptors (summed over all descriptor types) in a single descriptor set that is guaranteed to satisfy any implementation-dependent constraints on the size of a descriptor set itself. Applications can query whether a descriptor set that goes beyond this limit is supported using **vkGetDescriptorSetLayoutSupport**.

- **maxMemoryAllocationSize** is the maximum size of a memory allocation that can be created, even if there is more space available in the heap.

If the **VkPhysicalDeviceVulkan11Properties** structure is included in the **pNext** chain of the **VkPhysicalDeviceProperties2** structure passed to **vkGetPhysicalDeviceProperties2**, it is filled in with each corresponding implementation-dependent property.

These properties correspond to Vulkan 1.1 functionality.

The members of **VkPhysicalDeviceVulkan11Properties** have the same values as the corresponding members of **VkPhysicalDeviceIDProperties**, **VkPhysicalDeviceSubgroupProperties**, **VkPhysicalDevicePointClippingProperties**, **VkPhysicalDeviceMultiviewProperties**, **VkPhysicalDeviceProtectedMemoryProperties**, and **VkPhysicalDeviceMaintenance3Properties**.
Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceVulkan11Properties-sType-sType**
  - *sType must be* **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_1_PROPERTIES**

The `VkPhysicalDeviceVulkan12Properties` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceVulkan12Properties {
    VkStructureType sType;
    void* pNext;
    VkDriverId driverID;
    char driverName[VK_MAX_DRIVER_NAME_SIZE];
    char driverInfo[VK_MAX_DRIVER_INFO_SIZE];
    VkConformanceVersion conformanceVersion;
    VkShaderFloatControlsIndependence denormBehaviorIndependence;
    VkShaderFloatControlsIndependence roundingModeIndependence;
    VkBool32 shaderSignedZeroInfNanPreserveFloat16;
    VkBool32 shaderSignedZeroInfNanPreserveFloat32;
    VkBool32 shaderSignedZeroInfNanPreserveFloat64;
    VkBool32 shaderDenormPreserveFloat16;
    VkBool32 shaderDenormPreserveFloat32;
    VkBool32 shaderDenormPreserveFloat64;
    VkBool32 shaderDenormFlushToZeroFloat16;
    VkBool32 shaderDenormFlushToZeroFloat32;
    VkBool32 shaderDenormFlushToZeroFloat64;
    VkBool32 shaderRoundingModeRTEFloat16;
    VkBool32 shaderRoundingModeRTEFloat32;
    VkBool32 shaderRoundingModeRTEFloat64;
    VkBool32 shaderRoundingModeRTZFloat16;
    VkBool32 shaderRoundingModeRTZFloat32;
    VkBool32 shaderRoundingModeRTZFloat64;
    VkBool32 maxUpdateAfterBindDescriptorsInAllPools;
    VkBool32 shaderUniformBufferArrayNonUniformIndexingNative;
    VkBool32 shaderSampledImageArrayNonUniformIndexingNative;
    VkBool32 shaderStorageBufferArrayNonUniformIndexingNative;
    VkBool32 shaderStorageImageArrayNonUniformIndexingNative;
    VkBool32 shaderInputAttachmentArrayNonUniformIndexingNative;
    VkBool32 robustBufferAccessUpdateAfterBind;
    VkBool32 quadDivergentImplicitLod;
    uint32_t maxPerStageDescriptorUpdateAfterBindSamplers;
    uint32_t maxPerStageDescriptorUpdateAfterBindUniformBuffers;
} VkPhysicalDeviceVulkan12Properties;
```
maxPerStageDescriptorUpdateAfterBindStorageBuffers;
    uint32_t
maxPerStageDescriptorUpdateAfterBindSampledImages;
    uint32_t
maxPerStageDescriptorUpdateAfterBindStorageImages;
    uint32_t
maxPerStageDescriptorUpdateAfterBindInputAttachments;
    uint32_t
maxPerStageUpdateAfterBindResources;
    uint32_t
maxDescriptorSetUpdateAfterBindSamplers;
    uint32_t
maxDescriptorSetUpdateAfterBindUniformBuffers;
    uint32_t
maxDescriptorSetUpdateAfterBindUniformBuffersDynamic;
    uint32_t
maxDescriptorSetUpdateAfterBindStorageBuffers;
    uint32_t
maxDescriptorSetUpdateAfterBindStorageBuffersDynamic;
    uint32_t
maxDescriptorSetUpdateAfterBindStorageBuffersDynamic;
    uint32_t
maxDescriptorSetUpdateAfterBindInputAttachments;
    VkResolveModeFlags
supportedDepthResolveModes;
    VkResolveModeFlags
supportedStencilResolveModes;
    VkBool32
independentResolveNone;
    VkBool32
independentResolve;
    VkBool32
filterMinmaxSingleComponentFormats;
    VkBool32
filterMinmaxImageComponentMapping;
    uint64_t
maxTimelineSemaphoreValueDifference;
    VkSampleCountFlags
framebufferIntegerColorSampleCounts;
} VkPhysicalDeviceVulkan12Properties;

• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **driverID** is a unique identifier for the driver of the physical device.
• **driverName** is an array of **VK_MAX_DRIVER_NAME_SIZE char** containing a null-terminated UTF-8 string which is the name of the driver.
• **driverInfo** is an array of **VK_MAX_DRIVER_INFO_SIZE char** containing a null-terminated UTF-8 string with additional information about the driver.
• **conformanceVersion** is the version of the Vulkan conformance test this driver is conformant against (see **VkConformanceVersion**).
• **denormBehaviorIndependence** is a **VkShaderFloatControlsIndependence** value indicating whether, and how, denorm behavior can be set independently for different bit widths.
• **roundingModeIndependence** is a **VkShaderFloatControlsIndependence** value indicating whether, and how, rounding modes can be set independently for different bit widths.
• **shaderSignedZeroInfNanPreserveFloat16** is a boolean value indicating whether sign of a zero, Nans and ±∞ can be preserved in 16-bit floating-point computations. It also indicates whether
the SignedZeroInfNanPreserve execution mode can be used for 16-bit floating-point types.

- `shaderSignedZeroInfNanPreserveFloat32` is a boolean value indicating whether sign of a zero, Nans and ±∞ can be preserved in 32-bit floating-point computations. It also indicates whether the SignedZeroInfNanPreserve execution mode can be used for 32-bit floating-point types.

- `shaderSignedZeroInfNanPreserveFloat64` is a boolean value indicating whether sign of a zero, Nans and ±∞ can be preserved in 64-bit floating-point computations. It also indicates whether the SignedZeroInfNanPreserve execution mode can be used for 64-bit floating-point types.

- `shaderDenormPreserveFloat16` is a boolean value indicating whether denormals can be preserved in 16-bit floating-point computations. It also indicates whether the DenormPreserve execution mode can be used for 16-bit floating-point types.

- `shaderDenormPreserveFloat32` is a boolean value indicating whether denormals can be preserved in 32-bit floating-point computations. It also indicates whether the DenormPreserve execution mode can be used for 32-bit floating-point types.

- `shaderDenormPreserveFloat64` is a boolean value indicating whether denormals can be preserved in 64-bit floating-point computations. It also indicates whether the DenormPreserve execution mode can be used for 64-bit floating-point types.

- `shaderDenormFlushToZeroFloat16` is a boolean value indicating whether denormals can be flushed to zero in 16-bit floating-point computations. It also indicates whether the DenormFlushToZero execution mode can be used for 16-bit floating-point types.

- `shaderDenormFlushToZeroFloat32` is a boolean value indicating whether denormals can be flushed to zero in 32-bit floating-point computations. It also indicates whether the DenormFlushToZero execution mode can be used for 32-bit floating-point types.

- `shaderDenormFlushToZeroFloat64` is a boolean value indicating whether denormals can be flushed to zero in 64-bit floating-point computations. It also indicates whether the DenormFlushToZero execution mode can be used for 64-bit floating-point types.

- `shaderRoundingModeRTEFloat16` is a boolean value indicating whether an implementation supports the round-to-nearest-even rounding mode for 16-bit floating-point arithmetic and conversion instructions. It also indicates whether the RoundingModeRTE execution mode can be used for 16-bit floating-point types.

- `shaderRoundingModeRTEFloat32` is a boolean value indicating whether an implementation supports the round-to-nearest-even rounding mode for 32-bit floating-point arithmetic and conversion instructions. It also indicates whether the RoundingModeRTE execution mode can be used for 32-bit floating-point types.

- `shaderRoundingModeRTEFloat64` is a boolean value indicating whether an implementation supports the round-to-nearest-even rounding mode for 64-bit floating-point arithmetic and conversion instructions. It also indicates whether the RoundingModeRTE execution mode can be used for 64-bit floating-point types.

- `shaderRoundingModeRTZFloat16` is a boolean value indicating whether an implementation supports the round-towards-zero rounding mode for 16-bit floating-point arithmetic and conversion instructions. It also indicates whether the RoundingModeRTZ execution mode can be used for 16-bit floating-point types.

- `shaderRoundingModeRTZFloat32` is a boolean value indicating whether an implementation supports the round-towards-zero rounding mode for 32-bit floating-point arithmetic and conversion instructions. It also indicates whether the RoundingModeRTZ execution mode can be used for 32-bit floating-point types.
conversion instructions. It also indicates whether the `RoundingModeRTZ` execution mode can be used for 32-bit floating-point types.

- **shaderRoundingModeRTZFloat64** is a boolean value indicating whether an implementation supports the round-towards-zero rounding mode for 64-bit floating-point arithmetic and conversion instructions. It also indicates whether the `RoundingModeRTZ` execution mode can be used for 64-bit floating-point types.

- **maxUpdateAfterBindDescriptorsInAllPools** is the maximum number of descriptors (summed over all descriptor types) that can be created across all pools that are created with the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT` bit set. Pool creation may fail when this limit is exceeded, or when the space this limit represents is unable to satisfy a pool creation due to fragmentation.

- **shaderUniformBufferArrayNonUniformIndexingNative** is a boolean value indicating whether uniform buffer descriptors natively support nonuniform indexing. If this is `VK_FALSE`, then a single dynamic instance of an instruction that nonuniformly indexes an array of uniform buffers may execute multiple times in order to access all the descriptors.

- **shaderSampledImageArrayNonUniformIndexingNative** is a boolean value indicating whether sampler and image descriptors natively support nonuniform indexing. If this is `VK_FALSE`, then a single dynamic instance of an instruction that nonuniformly indexes an array of samplers or images may execute multiple times in order to access all the descriptors.

- **shaderStorageBufferArrayNonUniformIndexingNative** is a boolean value indicating whether storage buffer descriptors natively support nonuniform indexing. If this is `VK_FALSE`, then a single dynamic instance of an instruction that nonuniformly indexes an array of storage buffers may execute multiple times in order to access all the descriptors.

- **shaderStorageImageArrayNonUniformIndexingNative** is a boolean value indicating whether storage image descriptors natively support nonuniform indexing. If this is `VK_FALSE`, then a single dynamic instance of an instruction that nonuniformly indexes an array of storage images may execute multiple times in order to access all the descriptors.

- **shaderInputAttachmentArrayNonUniformIndexingNative** is a boolean value indicating whether input attachment descriptors natively support nonuniform indexing. If this is `VK_FALSE`, then a single dynamic instance of an instruction that nonuniformly indexes an array of input attachments may execute multiple times in order to access all the descriptors.

- **robustBufferAccessUpdateAfterBind** can be enabled on a device simultaneously with `descriptorBindingUniformBufferUpdateAfterBind`, `descriptorBindingStorageBufferUpdateAfterBind`, `descriptorBindingUniformTexelBufferUpdateAfterBind`, and/or `descriptorBindingStorageTexelBufferUpdateAfterBind`. If this is `VK_FALSE`, then either `robustBufferAccess` must be disabled or all of these update-after-bind features must be disabled.

- **quadDivergentImplicitLod** is a boolean value indicating whether implicit level of detail calculations for image operations have well-defined results when the image and/or sampler objects used for the instruction are not uniform within a quad. See Derivative Image Operations.

- **maxPerStageDescriptorUpdateAfterBindSamplers** is similar to `maxPerStageDescriptorSamplers` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set.
• maxPerStageDescriptorUpdateAfterBindUniformBuffers is similar to maxPerStageDescriptorUniformBuffers but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

• maxPerStageDescriptorUpdateAfterBindStorageBuffers is similar to maxPerStageDescriptorStorageBuffers but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

• maxPerStageDescriptorUpdateAfterBindSampledImages is similar to maxPerStageDescriptorSampledImages but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

• maxPerStageDescriptorUpdateAfterBindStorageImages is similar to maxPerStageDescriptorStorageImages but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

• maxPerStageDescriptorUpdateAfterBindInputAttachments is similar to maxPerStageDescriptorInputAttachments but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

• maxPerStageUpdateAfterBindResources is similar to maxPerStageResources but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

• maxDescriptorSetUpdateAfterBindSamplers is similar to maxDescriptorSetSamplers but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

• maxDescriptorSetUpdateAfterBindUniformBuffers is similar to maxDescriptorSetUniformBuffers but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

• maxDescriptorSetUpdateAfterBindUniformBuffersDynamic is similar to maxDescriptorSetUniformBuffersDynamic but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set. While an application can allocate dynamic uniform buffer descriptors from a pool created with the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT, bindings for these descriptors must not be present in any descriptor set layout that includes bindings created with VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT.

• maxDescriptorSetUpdateAfterBindStorageBuffers is similar to maxDescriptorSetStorageBuffers but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

• maxDescriptorSetUpdateAfterBindStorageBuffersDynamic is similar to maxDescriptorSetStorageBuffersDynamic but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set. While an application can allocate dynamic storage buffer descriptors from a pool created with the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT, bindings for these descriptors must not be present in any descriptor set layout that includes bindings created with VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT.

• maxDescriptorSetUpdateAfterBindSampledImages is similar to maxDescriptorSetSampledImages but counts descriptors from descriptor sets created with or without the
VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

- `maxDescriptorSetUpdateAfterBindStorageImages` is similar to `maxDescriptorSetStorageImages` but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

- `maxDescriptorSetUpdateAfterBindInputAttachments` is similar to `maxDescriptorSetInputAttachments` but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

- `supportedDepthResolveModes` is a bitmask of `VkResolveModeFlagBits` indicating the set of supported depth resolve modes. `VK_RESOLVE_MODE_SAMPLE_ZERO_BIT` must be included in the set but implementations **may** support additional modes.

- `supportedStencilResolveModes` is a bitmask of `VkResolveModeFlagBits` indicating the set of supported stencil resolve modes. `VK_RESOLVE_MODE_SAMPLE_ZERO_BIT` must be included in the set but implementations **may** support additional modes. `VK_RESOLVE_MODE_AVERAGE_BIT` must not be included in the set.

- `independentResolveNone` is `VK_TRUE` if the implementation supports setting the depth and stencil resolve modes to different values when one of those modes is `VK_RESOLVE_MODE_NONE`. Otherwise the implementation only supports setting both modes to the same value.

- `independentResolve` is `VK_TRUE` if the implementation supports all combinations of the supported depth and stencil resolve modes, including setting either depth or stencil resolve mode to `VK_RESOLVE_MODE_NONE`. An implementation that supports `independentResolve` must also support `independentResolveNone`.

- `filterMinmaxSingleComponentFormats` is a boolean value indicating whether a minimum set of required formats support min/max filtering.

- `filterMinmaxImageComponentMapping` is a boolean value indicating whether the implementation supports non-identity component mapping of the image when doing min/max filtering.

- `maxTimelineSemaphoreValueDifference` indicates the maximum difference allowed by the implementation between the current value of a timeline semaphore and any pending signal or wait operations.

- `framebufferIntegerColorSampleCounts` is a bitmask of `VkSampleCountFlagBits` indicating the color sample counts that are supported for all framebuffer color attachments with integer formats.

If the `VkPhysicalDeviceVulkan12Properties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

These properties correspond to Vulkan 1.2 functionality.

The members of `VkPhysicalDeviceVulkan12Properties` **must** have the same values as the corresponding members of `VkPhysicalDeviceDriverProperties`, `VkPhysicalDeviceFloatControlsProperties`, `VkPhysicalDeviceDescriptorIndexingProperties`, `VkPhysicalDeviceDepthStencilResolveProperties`, `VkPhysicalDeviceSamplerFilterMinmaxProperties`, and `VkPhysicalDeviceTimelineSemaphoreProperties`. 
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceVulkan12Properties-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_2_PROPERTIES

The VkPhysicalDeviceVulkan13Properties structure is defined as:

```c
// Defined by VK_VERSION_1_3
typedef struct VkPhysicalDeviceVulkan13Properties {
    VkStructureType sType;
    void* pNext;
    uint32_t minSubgroupSize;
    uint32_t maxSubgroupSize;
    uint32_t maxComputeWorkgroupSubgroups;
    VkShaderStageFlags requiredSubgroupSizeStages;
    uint32_t maxInlineUniformBlockSize;
    uint32_t maxPerStageDescriptorInlineUniform Blocks;
    uint32_t maxPerStageDescriptorUpdateAfterBindInlineUniformBlocks;
    uint32_t maxDescriptorSetInlineUniform Blocks;
    uint32_t maxDescriptorSetUpdateAfterBindInlineUniformBlocks;
    uint32_t maxInlineUniformTotalSize;
    VkBool32 integerDotProduct8BitUnsignedAccelerated;
    VkBool32 integerDotProduct8BitSignedAccelerated;
    VkBool32 integerDotProduct8BitMixedSignednessAccelerated;
    VkBool32 integerDotProduct4x8BitPackedUnsignedAccelerated;
    VkBool32 integerDotProduct4x8BitPackedSignedAccelerated;
    VkBool32 integerDotProduct4x8BitPackedMixedSignednessAccelerated;
    VkBool32 integerDotProduct16BitUnsignedAccelerated;
    VkBool32 integerDotProduct16BitSignedAccelerated;
    VkBool32 integerDotProduct16BitMixedSignednessAccelerated;
    VkBool32 integerDotProduct32BitUnsignedAccelerated;
    VkBool32 integerDotProduct32BitSignedAccelerated;
    VkBool32 integerDotProduct32BitMixedSignednessAccelerated;
    VkBool32 integerDotProduct64BitUnsignedAccelerated;
    VkBool32 integerDotProduct64BitSignedAccelerated;
    VkBool32 integerDotProduct64BitMixedSignednessAccelerated;
    VkBool32 integerDotProductAccumulatingSaturating8BitUnsignedAccelerated;
    VkBool32 integerDotProductAccumulatingSaturating8BitSignedAccelerated;
    VkBool32 integerDotProductAccumulatingSaturating8BitMixedSignednessAccelerated;
    VkBool32 integerDotProductAccumulatingSaturating4x8BitPackedUnsignedAccelerated;
    VkBool32 integerDotProductAccumulatingSaturating4x8BitPackedSignedAccelerated;
    VkBool32 integerDotProductAccumulatingSaturating4x8BitPackedMixedSignednessAccelerated;
} VkPhysicalDeviceVulkan13Properties;
```
integerDotProductAccumulatingSaturating16BitUnsignedAccelerated;
VkBool32
integerDotProductAccumulatingSaturating16BitSignedAccelerated;
VkBool32
integerDotProductAccumulatingSaturating16BitMixedSignednessAccelerated;
VkBool32
integerDotProductAccumulatingSaturating32BitUnsignedAccelerated;
VkBool32
integerDotProductAccumulatingSaturating32BitSignedAccelerated;
VkBool32
integerDotProductAccumulatingSaturating32BitMixedSignednessAccelerated;
VkBool32
integerDotProductAccumulatingSaturating64BitUnsignedAccelerated;
VkBool32
integerDotProductAccumulatingSaturating64BitSignedAccelerated;
VkBool32
integerDotProductAccumulatingSaturating64BitMixedSignednessAccelerated;
VkDeviceSize storageTexelBufferOffsetAlignmentBytes;
VkBool32 storageTexelBufferOffsetSingleTexelAlignment;
VkDeviceSize uniformTexelBufferOffsetAlignmentBytes;
VkBool32 uniformTexelBufferOffsetSingleTexelAlignment;
VkDeviceSize maxBufferSize;
}

VkPhysicalDeviceVulkan13Properties;

• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.

• **minSubgroupSize** is the minimum subgroup size supported by this device. minSubgroupSize is at least one if any of the physical device’s queues support VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT. minSubgroupSize is a power-of-two. minSubgroupSize is less than or equal to maxSubgroupSize. minSubgroupSize is less than or equal to subgroupSize.

• **maxSubgroupSize** is the maximum subgroup size supported by this device. maxSubgroupSize is at least one if any of the physical device’s queues support VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT. maxSubgroupSize is a power-of-two. maxSubgroupSize is greater than or equal to minSubgroupSize. maxSubgroupSize is greater than or equal to subgroupSize.

• **maxComputeWorkgroupSubgroups** is the maximum number of subgroups supported by the implementation within a workgroup.

• **requiredSubgroupSizeStages** is a bitfield of what shader stages support having a required subgroup size specified.

• **maxInlineUniformBlockSize** is the maximum size in bytes of an inline uniform block binding.

• **maxPerStageDescriptorInlineUniformBlock** is the maximum number of inline uniform block bindings that can be accessible to a single shader stage in a pipeline layout. Descriptor bindings with a descriptor type of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK count against this limit. Only descriptor bindings in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit.

• **maxPerStageDescriptorUpdateAfterBindInlineUniformBlocks** is similar to
- `maxPerStageDescriptorInlineUniformBlocks` but counts descriptor bindings from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set.

- `maxDescriptorSetInlineUniformBlocks` is the maximum number of inline uniform block bindings that can be included in descriptor bindings in a pipeline layout across all pipeline shader stages and descriptor set numbers. Descriptor bindings with a descriptor type of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK` count against this limit. Only descriptor bindings in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set count against this limit.

- `maxDescriptorSetUpdateAfterBindInlineUniformBlocks` is similar to `maxDescriptorSetInlineUniformBlocks` but counts descriptor bindings from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set.

- `maxInlineUniformTotalSize` is the maximum total size in bytes of all inline uniform block bindings, across all pipeline shader stages and descriptor set numbers, that can be included in a pipeline layout. Descriptor bindings with a descriptor type of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK` count against this limit.

- `integerDotProduct8BitUnsignedAccelerated` is a boolean that will be `VK_TRUE` if the support for 8-bit unsigned dot product operations using the `OpUDotKHR` SPIR-V instruction is accelerated as defined below.

- `integerDotProduct8BitSignedAccelerated` is a boolean that will be `VK_TRUE` if the support for 8-bit signed dot product operations using the `OpSDotKHR` SPIR-V instruction is accelerated as defined below.

- `integerDotProduct8BitMixedSignednessAccelerated` is a boolean that will be `VK_TRUE` if the support for 8-bit mixed signedness dot product operations using the `OpSUDotKHR` SPIR-V instruction is accelerated as defined below.

- `integerDotProduct4x8BitPackedUnsignedAccelerated` is a boolean that will be `VK_TRUE` if the support for 8-bit unsigned dot product operations from operands packed into 32-bit integers using the `OpUDotKHR` SPIR-V instruction is accelerated as defined below.

- `integerDotProduct4x8BitPackedSignedAccelerated` is a boolean that will be `VK_TRUE` if the support for 8-bit signed dot product operations from operands packed into 32-bit integers using the `OpSDotKHR` SPIR-V instruction is accelerated as defined below.

- `integerDotProduct4x8BitPackedMixedSignednessAccelerated` is a boolean that will be `VK_TRUE` if the support for 8-bit mixed signedness dot product operations from operands packed into 32-bit integers using the `OpSUDotKHR` SPIR-V instruction is accelerated as defined below.

- `integerDotProduct16BitUnsignedAccelerated` is a boolean that will be `VK_TRUE` if the support for 16-bit unsigned dot product operations using the `OpUDotKHR` SPIR-V instruction is accelerated as defined below.

- `integerDotProduct16BitSignedAccelerated` is a boolean that will be `VK_TRUE` if the support for 16-bit signed dot product operations using the `OpSDotKHR` SPIR-V instruction is accelerated as defined below.

- `integerDotProduct16BitMixedSignednessAccelerated` is a boolean that will be `VK_TRUE` if the support for 16-bit mixed signedness dot product operations using the `OpSUDotKHR` SPIR-V instruction is accelerated as defined below.

- `integerDotProduct32BitUnsignedAccelerated` is a boolean that will be `VK_TRUE` if the support for
32-bit unsigned dot product operations using the OpUDotKHR SPIR-V instruction is accelerated as defined below.

- integerDotProduct32BitSignedAccelerated is a boolean that will be VK_TRUE if the support for 32-bit signed dot product operations using the OpSDotKHR SPIR-V instruction is accelerated as defined below.

- integerDotProduct32BitMixedSignednessAccelerated is a boolean that will be VK_TRUE if the support for 32-bit mixed signedness dot product operations using the OpSUDotKHR SPIR-V instruction is accelerated as defined below.

- integerDotProduct64BitUnsignedAccelerated is a boolean that will be VK_TRUE if the support for 64-bit unsigned dot product operations using the OpUDotKHR SPIR-V instruction is accelerated as defined below.

- integerDotProduct64BitSignedAccelerated is a boolean that will be VK_TRUE if the support for 64-bit signed dot product operations using the OpSDotKHR SPIR-V instruction is accelerated as defined below.

- integerDotProduct64BitMixedSignednessAccelerated is a boolean that will be VK_TRUE if the support for 64-bit mixed signedness dot product operations using the OpSUDotKHR SPIR-V instruction is accelerated as defined below.

- integerDotProductAccumulatingSaturating8BitUnsignedAccelerated is a boolean that will be VK_TRUE if the support for 8-bit unsigned accumulating saturating dot product operations using the OpUDotAccSatKHR SPIR-V instruction is accelerated as defined below.

- integerDotProductAccumulatingSaturating8BitSignedAccelerated is a boolean that will be VK_TRUE if the support for 8-bit signed accumulating saturating dot product operations using the OpSDotAccSatKHR SPIR-V instruction is accelerated as defined below.

- integerDotProductAccumulatingSaturating8BitMixedSignednessAccelerated is a boolean that will be VK_TRUE if the support for 8-bit mixed signedness accumulating saturating dot product operations using the OpSUDotAccSatKHR SPIR-V instruction is accelerated as defined below.

- integerDotProductAccumulatingSaturating4x8BitPackedUnsignedAccelerated is a boolean that will be VK_TRUE if the support for 8-bit unsigned accumulating saturating dot product operations from operands packed into 32-bit integers using the OpUDotAccSatKHR SPIR-V instruction is accelerated as defined below.

- integerDotProductAccumulatingSaturating4x8BitPackedSignedAccelerated is a boolean that will be VK_TRUE if the support for 8-bit signed accumulating saturating dot product operations from operands packed into 32-bit integers using the OpSDotAccSatKHR SPIR-V instruction is accelerated as defined below.

- integerDotProductAccumulatingSaturating4x8BitPackedMixedSignednessAccelerated is a boolean that will be VK_TRUE if the support for 8-bit mixed signedness accumulating saturating dot product operations from operands packed into 32-bit integers using the OpSUDotAccSatKHR SPIR-V instruction is accelerated as defined below.

- integerDotProductAccumulatingSaturating16BitUnsignedAccelerated is a boolean that will be VK_TRUE if the support for 16-bit unsigned accumulating saturating dot product operations using the OpUDotAccSatKHR SPIR-V instruction is accelerated as defined below.

- integerDotProductAccumulatingSaturating16BitSignedAccelerated is a boolean that will be VK_TRUE if the support for 16-bit signed accumulating saturating dot product operations using
the `OpSDotAccSatKHR` SPIR-V instruction is accelerated as defined below.

- `integerDotProductAccumulatingSaturating16BitMixedSignednessAccelerated` is a boolean that will be `VK_TRUE` if the support for 16-bit mixed signedness accumulating saturating dot product operations using the `OpSUDDotAccSatKHR` SPIR-V instruction is accelerated as defined below.

- `integerDotProductAccumulatingSaturating32BitUnsignedAccelerated` is a boolean that will be `VK_TRUE` if the support for 32-bit unsigned accumulating saturating dot product operations using the `OpUDDotAccSatKHR` SPIR-V instruction is accelerated as defined below.

- `integerDotProductAccumulatingSaturating32BitSignedAccelerated` is a boolean that will be `VK_TRUE` if the support for 32-bit signed accumulating saturating dot product operations using the `OpSDDotAccSatKHR` SPIR-V instruction is accelerated as defined below.

- `integerDotProductAccumulatingSaturating32BitMixedSignednessAccelerated` is a boolean that will be `VK_TRUE` if the support for 32-bit mixed signedness accumulating saturating dot product operations using the `OpSUDDotAccSatKHR` SPIR-V instruction is accelerated as defined below.

- `integerDotProductAccumulatingSaturating64BitUnsignedAccelerated` is a boolean that will be `VK_TRUE` if the support for 64-bit unsigned accumulating saturating dot product operations using the `OpUDDotAccSatKHR` SPIR-V instruction is accelerated as defined below.

- `integerDotProductAccumulatingSaturating64BitSignedAccelerated` is a boolean that will be `VK_TRUE` if the support for 64-bit signed accumulating saturating dot product operations using the `OpSDDotAccSatKHR` SPIR-V instruction is accelerated as defined below.

- `integerDotProductAccumulatingSaturating64BitMixedSignednessAccelerated` is a boolean that will be `VK_TRUE` if the support for 64-bit mixed signedness accumulating saturating dot product operations using the `OpSUDDotAccSatKHR` SPIR-V instruction is accelerated as defined below.

- `storageTexelBufferOffsetAlignmentBytes` is a byte alignment that is sufficient for a storage texel buffer of any format. The value must be a power of two.

- `storageTexelBufferOffsetSingleTexelAlignment` indicates whether single texel alignment is sufficient for a storage texel buffer of any format.

- `uniformTexelBufferOffsetAlignmentBytes` is a byte alignment that is sufficient for a uniform texel buffer of any format. The value must be a power of two.

- `uniformTexelBufferOffsetSingleTexelAlignment` indicates whether single texel alignment is sufficient for a uniform texel buffer of any format.

- `maxBufferSize` is the maximum size `VkBuffer` that can be created.

If the `VkPhysicalDeviceVulkan13Properties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

These properties correspond to Vulkan 1.3 functionality.

The members of `VkPhysicalDeviceVulkan13Properties` must have the same values as the corresponding members of `VkPhysicalDeviceInlineUniformBlockProperties` and `VkPhysicalDeviceSubgroupSizeControlProperties`.
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceVulkan13Properties-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_3_PROPERTIES

The **VkPhysicalDeviceIDProperties** structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceIDProperties {
    VkStructureType sType;
    void* pNext;
    uint8_t deviceUUID[VK_UUID_SIZE];
    uint8_t driverUUID[VK_UUID_SIZE];
    uint8_t deviceLUID[VK_LUID_SIZE];
    uint32_t deviceNodeMask;
    VkBool32 deviceLUIDValid;
} VkPhysicalDeviceIDProperties;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.

- **deviceUUID** is an array of VK_UUID_SIZE uint8_t values representing a universally unique identifier for the device.
- **driverUUID** is an array of VK_UUID_SIZE uint8_t values representing a universally unique identifier for the driver build in use by the device.
- **deviceLUID** is an array of VK_LUID_SIZE uint8_t values representing a locally unique identifier for the device.
- **deviceNodeMask** is a uint32_t bitfield identifying the node within a linked device adapter corresponding to the device.
- **deviceLUIDValid** is a boolean value that will be VK_TRUE if deviceLUID contains a valid LUID and deviceNodeMask contains a valid node mask, and VK_FALSE if they do not.

If the **VkPhysicalDeviceIDProperties** structure is included in the pNext chain of the **VkPhysicalDeviceProperties2** structure passed to vkGetPhysicalDeviceProperties2, it is filled in with each corresponding implementation-dependent property.

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

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

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

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

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

Implementations should return deviceUUID values which are likely to be unique even in the presence of multiple Vulkan implementations (such as a GPU driver and a software renderer; two drivers for different GPUs; or the same Vulkan driver running on two logically different devices).

Khronos' conformance testing is unable to guarantee that deviceUUID values are actually unique, so implementors should make their own best efforts to ensure this. In particular, hard-coded deviceUUID values, especially all-0 bits, should never be used.

A combination of values unique to the vendor, the driver, and the hardware environment can be used to provide a deviceUUID which is unique to a high degree of certainty. Some possible inputs to such a computation are:

• Information reported by vkGetPhysicalDeviceProperties
• PCI device ID (if defined)
• PCI bus ID, or similar system configuration information.
• Driver binary checksums.

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

Valid Usage (Implicit)

• VUID-VkPhysicalDeviceIDProperties-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_PROPERTIES

`VK_UUID_SIZE` is the length in `uint8_t` values of an array containing a universally unique device or
driver build identifier, as returned in `VkPhysicalDeviceIDProperties::deviceUUID` and
`VkPhysicalDeviceIDProperties::driverUUID`.

```cpp
#define VK_UUID_SIZE                      16U
```

`VK_LUID_SIZE` is the length in `uint8_t` values of an array containing a locally unique device identifier,
as returned in `VkPhysicalDeviceIDProperties::deviceLUID`.

```cpp
#define VK_LUID_SIZE                      8U
```

The `VkPhysicalDeviceDriverProperties` structure is defined as:

```cpp
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceDriverProperties {
    VkStructureType sType;
    void* pNext;
    VkDriverId driverID;
    char driverName[VK_MAX_DRIVER_NAME_SIZE];
    char driverInfo[VK_MAX_DRIVER_INFO_SIZE];
    VkConformanceVersion conformanceVersion;
} VkPhysicalDeviceDriverProperties;
```

• `sType` is the type of this structure.
• `pNext` is NULL or a pointer to a structure extending this structure.
• `driverID` is a unique identifier for the driver of the physical device.
• `driverName` is an array of `VK_MAX_DRIVER_NAME_SIZE` `char` containing a null-terminated UTF-8 string
  which is the name of the driver.
• `driverInfo` is an array of `VK_MAX_DRIVER_INFO_SIZE` `char` containing a null-terminated UTF-8 string
  with additional information about the driver.
• `conformanceVersion` is the version of the Vulkan conformance test this driver is conformant
  against (see `VkConformanceVersion`).
If the `VkPhysicalDeviceDriverProperties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

These are properties of the driver corresponding to a physical device.

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

---

Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceDriverProperties-sType-sType` 
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DRIVER_PROPERTIES`

Khronos driver IDs which may be returned in `VkPhysicalDeviceDriverProperties::driverID` are:

```c
// Provided by VK_VERSION_1_2
typedef enum VkDriverId {
    VK_DRIVER_ID_AMD_PROPRIETARY = 1,
    VK_DRIVER_ID_AMD_OPEN_SOURCE = 2,
    VK_DRIVER_ID_MESA_RADV = 3,
    VK_DRIVER_ID_NVIDIA_PROPRIETARY = 4,
    VK_DRIVER_ID_INTEL_PROPRIETARY_WINDOWS = 5,
    VK_DRIVER_ID_INTEL_OPEN_SOURCE_MESA = 6,
    VK_DRIVER_ID_IMAGINATION_PROPRIETARY = 7,
    VK_DRIVER_ID_QUALCOMM_PROPRIETARY = 8,
    VK_DRIVER_ID_ARM_PROPRIETARY = 9,
    VK_DRIVER_ID_GOOGLE_SWIFTSHADER = 10,
    VK_DRIVER_ID_GGP_PROPRIETARY = 11,
    VK_DRIVER_ID_BROADCOM_PROPRIETARY = 12,
    VK_DRIVER_ID_MESA_LLVMPIPE = 13,
    VK_DRIVER_ID_MOLTENVK = 14,
    VK_DRIVER_ID_COREAVI_PROPRIETARY = 15,
    VK_DRIVER_ID_JUICE_PROPRIETARY = 16,
    VK_DRIVER_ID_VERISILICON_PROPRIETARY = 17,
    VK_DRIVER_ID_MESA_TURNIP = 18,
    VK_DRIVER_ID_MESA_V3DV = 19,
    VK_DRIVER_ID_MESA_PANVK = 20,
    VK_DRIVER_ID_SAMSUNG_PROPRIETARY = 21,
    VK_DRIVER_ID_MESA_VENUS = 22,
    VK_DRIVER_ID_MESA_DOZEN = 23,
    VK_DRIVER_ID_MESA_NVK = 24,
} VkDriverId;
```

---

Note

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

Only driver IDs registered with Khronos are given symbolic names. There \textbf{may be} unregistered driver IDs returned.

\texttt{VK\_MAX\_DRIVER\_NAME\_SIZE} is the length in \texttt{char} values of an array containing a driver name string, as returned in \texttt{VkPhysicalDeviceDriverProperties::driverName}.

```c
#define VK_MAX_DRIVER_NAME_SIZE 256U
```

\texttt{VK\_MAX\_DRIVER\_INFO\_SIZE} is the length in \texttt{char} values of an array containing a driver information string, as returned in \texttt{VkPhysicalDeviceDriverProperties::driverInfo}.

```c
#define VK_MAX_DRIVER_INFO_SIZE 256U
```

The conformance test suite version an implementation is compliant with is described with the \texttt{VkConformanceVersion} structure:

```c
// Provided by VK\_VERSION\_1\_2
typedef struct VkConformanceVersion {
    uint8_t major;
    uint8_t minor;
    uint8_t subminor;
    uint8_t patch;
} VkConformanceVersion;
```

- \texttt{major} is the major version number of the conformance test suite.
- \texttt{minor} is the minor version number of the conformance test suite.
- \texttt{subminor} is the subminor version number of the conformance test suite.
- \texttt{patch} is the patch version number of the conformance test suite.

The \texttt{VkPhysicalDeviceShaderIntegerDotProductProperties} structure is defined as:

```c
// Provided by VK\_VERSION\_1\_3
typedef struct VkPhysicalDeviceShaderIntegerDotProductProperties {
    VkStructureType sType;
    void* pNext;
    VkBool32 integerDotProduct8BitUnsignedAccelerated;
    VkBool32 integerDotProduct8BitSignedAccelerated;
    VkBool32 integerDotProduct8BitMixedSignednessAccelerated;
    VkBool32 integerDotProduct4x8BitPackedUnsignedAccelerated;
    VkBool32 integerDotProduct4x8BitPackedSignedAccelerated;
    VkBool32 integerDotProduct4x8BitPackedMixedSignednessAccelerated;
} VkPhysicalDeviceShaderIntegerDotProductProperties;
```
 VkBool32 integerDotProduct16BitUnsignedAccelerated;
 VkBool32 integerDotProduct16BitSignedAccelerated;
 VkBool32 integerDotProduct16BitMixedSignednessAccelerated;
 VkBool32 integerDotProduct32BitUnsignedAccelerated;
 VkBool32 integerDotProduct32BitSignedAccelerated;
 VkBool32 integerDotProduct32BitMixedSignednessAccelerated;
 VkBool32 integerDotProduct64BitUnsignedAccelerated;
 VkBool32 integerDotProduct64BitSignedAccelerated;
 VkBool32 integerDotProduct64BitMixedSignednessAccelerated;
 VkBool32 integerDotProductAccumulatingSaturating8BitUnsignedAccelerated;
 VkBool32 integerDotProductAccumulatingSaturating8BitSignedAccelerated;
 VkBool32 integerDotProductAccumulatingSaturating8BitMixedSignednessAccelerated;
 VkBool32 integerDotProductAccumulatingSaturating4x8BitPackedUnsignedAccelerated;
 VkBool32 integerDotProductAccumulatingSaturating4x8BitPackedSignedAccelerated;
 VkBool32 integerDotProductAccumulatingSaturating4x8BitPackedMixedSignednessAccelerated;
 VkBool32 integerDotProductAccumulatingSaturating16BitUnsignedAccelerated;
 VkBool32 integerDotProductAccumulatingSaturating16BitSignedAccelerated;
 VkBool32 integerDotProductAccumulatingSaturating16BitMixedSignednessAccelerated;
 VkBool32 integerDotProductAccumulatingSaturating32BitUnsignedAccelerated;
 VkBool32 integerDotProductAccumulatingSaturating32BitSignedAccelerated;
 VkBool32 integerDotProductAccumulatingSaturating32BitMixedSignednessAccelerated;
 VkBool32 integerDotProductAccumulatingSaturating64BitUnsignedAccelerated;
 VkBool32 integerDotProductAccumulatingSaturating64BitSignedAccelerated;
 VkBool32 integerDotProductAccumulatingSaturating64BitMixedSignednessAccelerated;
 }

VsType is the type of this structure.

pNext is NULL or a pointer to a structure extending this structure.

integerDotProduct8BitUnsignedAccelerated is a boolean that will be VK_TRUE if the support for 8-bit unsigned dot product operations using the OpUDotKHR SPIR-V instruction is accelerated as defined below.

integerDotProduct8BitSignedAccelerated is a boolean that will be VK_TRUE if the support for 8-bit signed dot product operations using the OpSDotKHR SPIR-V instruction is accelerated as defined below.

integerDotProduct8BitMixedSignednessAccelerated is a boolean that will be VK_TRUE if the support for 8-bit mixed signedness dot product operations using the OpSUDotKHR SPIR-V instruction is accelerated as defined below.
• **integerDotProduct4x8BitPackedUnsignedAccelerated** is a boolean that will be **VK_TRUE** if the support for 8-bit unsigned dot product operations from operands packed into 32-bit integers using the **OpUDotKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProduct4x8BitPackedSignedAccelerated** is a boolean that will be **VK_TRUE** if the support for 8-bit signed dot product operations from operands packed into 32-bit integers using the **OpSDotKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProduct4x8BitPackedMixedSignednessAccelerated** is a boolean that will be **VK_TRUE** if the support for 8-bit mixed signedness dot product operations from operands packed into 32-bit integers using the **OpSUDotKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProduct16BitUnsignedAccelerated** is a boolean that will be **VK_TRUE** if the support for 16-bit unsigned dot product operations using the **OpUDotKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProduct16BitSignedAccelerated** is a boolean that will be **VK_TRUE** if the support for 16-bit signed dot product operations using the **OpSDotKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProduct16BitMixedSignednessAccelerated** is a boolean that will be **VK_TRUE** if the support for 16-bit mixed signedness dot product operations using the **OpSUDot_KHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProduct32BitUnsignedAccelerated** is a boolean that will be **VK_TRUE** if the support for 32-bit unsigned dot product operations using the **OpUDotKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProduct32BitSignedAccelerated** is a boolean that will be **VK_TRUE** if the support for 32-bit signed dot product operations using the **OpSDotKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProduct32BitMixedSignednessAccelerated** is a boolean that will be **VK_TRUE** if the support for 32-bit mixed signedness dot product operations using the **OpSUDotKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProduct64BitUnsignedAccelerated** is a boolean that will be **VK_TRUE** if the support for 64-bit unsigned dot product operations using the **OpUDotKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProduct64BitSignedAccelerated** is a boolean that will be **VK_TRUE** if the support for 64-bit signed dot product operations using the **OpSDotKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProduct64BitMixedSignednessAccelerated** is a boolean that will be **VK_TRUE** if the support for 64-bit mixed signedness dot product operations using the **OpSUDotKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProductAccumulatingSaturating8BitUnsignedAccelerated** is a boolean that will be **VK_TRUE** if the support for 8-bit unsigned accumulating saturating dot product operations using the **OpUDotAccSatKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProductAccumulatingSaturating8BitSignedAccelerated** is a boolean that will be **VK_TRUE** if the support for 8-bit signed accumulating saturating dot product operations using the **OpSDotAccSatKHR** SPIR-V instruction is accelerated as defined below.
• **integerDotProductAccumulatingSaturating8BitMixedSignednessAccelerated** is a boolean that will be **VK_TRUE** if the support for 8-bit mixed signedness accumulating saturating dot product operations using the **OpSUDotAccSatKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProductAccumulatingSaturating4x8BitPackedUnsignedAccelerated** is a boolean that will be **VK_TRUE** if the support for 8-bit unsigned accumulating saturating dot product operations from operands packed into 32-bit integers using the **OpUDotAccSatKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProductAccumulatingSaturating4x8BitPackedSignedAccelerated** is a boolean that will be **VK_TRUE** if the support for 8-bit signed accumulating saturating dot product operations from operands packed into 32-bit integers using the **OpSDotAccSatKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProductAccumulatingSaturating4x8BitPackedMixedSignednessAccelerated** is a boolean that will be **VK_TRUE** if the support for 8-bit mixed signedness accumulating saturating dot product operations from operands packed into 32-bit integers using the **OpSUDotAccSatKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProductAccumulatingSaturating16BitUnsignedAccelerated** is a boolean that will be **VK_TRUE** if the support for 16-bit unsigned accumulating saturating dot product operations using the **OpUDotAccSatKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProductAccumulatingSaturating16BitSignedAccelerated** is a boolean that will be **VK_TRUE** if the support for 16-bit signed accumulating saturating dot product operations using the **OpSDotAccSatKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProductAccumulatingSaturating16BitMixedSignednessAccelerated** is a boolean that will be **VK_TRUE** if the support for 16-bit mixed signedness accumulating saturating dot product operations using the **OpSUDotAccSatKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProductAccumulatingSaturating32BitUnsignedAccelerated** is a boolean that will be **VK_TRUE** if the support for 32-bit unsigned accumulating saturating dot product operations using the **OpUDotAccSatKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProductAccumulatingSaturating32BitSignedAccelerated** is a boolean that will be **VK_TRUE** if the support for 32-bit signed accumulating saturating dot product operations using the **OpSDotAccSatKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProductAccumulatingSaturating32BitMixedSignednessAccelerated** is a boolean that will be **VK_TRUE** if the support for 32-bit mixed signedness accumulating saturating dot product operations using the **OpSUDotAccSatKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProductAccumulatingSaturating64BitUnsignedAccelerated** is a boolean that will be **VK_TRUE** if the support for 64-bit unsigned accumulating saturating dot product operations using the **OpUDotAccSatKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProductAccumulatingSaturating64BitSignedAccelerated** is a boolean that will be **VK_TRUE** if the support for 64-bit signed accumulating saturating dot product operations using the **OpSDotAccSatKHR** SPIR-V instruction is accelerated as defined below.

• **integerDotProductAccumulatingSaturating64BitMixedSignednessAccelerated** is a boolean that will be **VK_TRUE** if the support for 64-bit mixed signedness accumulating saturating dot product operations using the **OpSUDotAccSatKHR** SPIR-V instruction is accelerated as defined below.
If the `VkPhysicalDeviceShaderIntegerDotProductProperties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

These are properties of the integer dot product acceleration information of a physical device.

**Note**
A dot product operation is deemed accelerated if its implementation provides a performance advantage over application-provided code composed from elementary instructions and/or other dot product instructions, either because the implementation uses optimized machine code sequences whose generation from application-provided code cannot be guaranteed or because it uses hardware features that cannot otherwise be targeted from application-provided code.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceShaderIntegerDotProductProperties-sType-sType
  
sType must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_INTEGER_DOT_PRODUCT_PROPERTIES`

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

```c
// Provided by VK_VERSION_1_0
void vkGetPhysicalDeviceQueueFamilyProperties(
    VkPhysicalDevice physicalDevice,
    uint32_t* pQueueFamilyPropertyCount,
    VkQueueFamilyProperties* pQueueFamilyProperties);
```

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

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

**Valid Usage (Implicit)**

- VUID-vkGetPhysicalDeviceQueueFamilyProperties-physicalDevice-parameter
  
  `physicalDevice` must be a valid `VkPhysicalDevice` handle

- VUID-vkGetPhysicalDeviceQueueFamilyProperties-pQueueFamilyPropertyCount-
parameter
pQueueFamilyPropertyCount must be a valid pointer to a uint32_t value

- VUID-vkGetPhysicalDeviceQueueFamilyProperties-pQueueFamilyPropertyCount-parameter
If the value referenced by pQueueFamilyPropertyCount is not 0, and pQueueFamilyProperties is not NULL, pQueueFamilyProperties must be a valid pointer to an array of pQueueFamilyPropertyCount VkQueueFamilyProperties structures

The VkQueueFamilyProperties structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkQueueFamilyProperties {
    VkQueueFlags queueFlags;
    uint32_t queueCount;
    uint32_t timestampValidBits;
    VkExtent3D minImageTransferGranularity;
} VkQueueFamilyProperties;
```

- `queueFlags` is a bitmask of VkQueueFlagBits indicating capabilities of the queues in this queue family.

- `queueCount` is the unsigned integer count of queues in this queue family. Each queue family must support at least one queue.

- `timestampValidBits` is the unsigned integer count of meaningful bits in the timestamps written via vkCmdWriteTimestamp2 or vkCmdWriteTimestamp. The valid range for the count is 36 to 64 bits, or a value of 0, indicating no support for timestamps. Bits outside the valid range are guaranteed to be zeros.

- `minImageTransferGranularity` is the minimum granularity supported for image transfer operations on the queues in this queue family.

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

Possible values of `minImageTransferGranularity` are:

- (0,0,0) specifies that only whole mip levels must be transferred using the image transfer operations on the corresponding queues. In this case, the following restrictions apply to all offset and extent parameters of image transfer operations:
  - The x, y, and z members of a VkOffset3D parameter must always be zero.
  - The width, height, and depth members of a VkExtent3D parameter must always match the width, height, and depth of the image subresource corresponding to the parameter, respectively.

- (A_x, A_y, A_z) where A_x, A_y, and A_z are all integer powers of two. In this case the following restrictions apply to all image transfer operations:
  - x, y, and z of a VkOffset3D parameter must be integer multiples of A_x, A_y, and A_z, respectively.
width of a VkExtent3D parameter must be an integer multiple of \( A_x \), or else \( x + \text{width} \) must equal the width of the image subresource corresponding to the parameter.

height of a VkExtent3D parameter must be an integer multiple of \( A_y \), or else \( y + \text{height} \) must equal the height of the image subresource corresponding to the parameter.

depth of a VkExtent3D parameter must be an integer multiple of \( A_z \), or else \( z + \text{depth} \) must equal the depth of the image subresource corresponding to the parameter.

If the format of the image corresponding to the parameters is one of the block-compressed formats then for the purposes of the above calculations the granularity must be scaled up by the compressed texel block dimensions.

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

The Device Memory section describes memory properties queried from the physical device.

For physical device feature queries see the Features chapter.

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

```c
// Provided by VK_VERSION_1_0
typedef enum VkQueueFlagBits {
    VK_QUEUE_GRAPHICS_BIT = 0x00000001,
    VK_QUEUE_COMPUTE_BIT = 0x00000002,
    VK_QUEUE_TRANSFER_BIT = 0x00000004,
    VK_QUEUE_SPARSE_BINDING_BIT = 0x00000008,
// Provided by VK_VERSION_1_1
    VK_QUEUE_PROTECTED_BIT = 0x00000010,
} VkQueueFlagBits;
```

• VK_QUEUE_GRAPHICS_BIT specifies that queues in this queue family support graphics operations.

• VK_QUEUE_COMPUTE_BIT specifies that queues in this queue family support compute operations.

• VK_QUEUE_TRANSFER_BIT specifies that queues in this queue family support transfer operations.

• VK_QUEUE_SPARSE_BINDING_BIT specifies that queues in this queue family support sparse memory management operations (see Sparse Resources). If any of the sparse resource features are enabled, then at least one queue family must support this bit.

• VK_QUEUE_PROTECTED_BIT specifies that queues in this queue family support the VKDEVICE_QUEUE_CREATE_PROTECTED_BIT bit. (see Protected Memory). If the physical device supports the protectedMemory feature, at least one of its queue families must support this bit.

If an implementation exposes any queue family that supports graphics operations, at least one queue family of at least one physical device exposed by the implementation must support both graphics and compute operations.
Furthermore, if the `protectedMemory` physical device feature is supported, then at least one queue family of at least one physical device exposed by the implementation **must** support graphics operations, compute operations, and protected memory operations.

**Note**

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

For further details see [Queues](#).

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkQueueFlags;
```

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

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

```c
// Provided by VK_VERSION_1_1
void vkGetPhysicalDeviceQueueFamilyProperties2(
    VkPhysicalDevice physicalDevice,
    uint32_t* pQueueFamilyPropertyCount,
    VkQueueFamilyProperties2* pQueueFamilyProperties);
```

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

`vkGetPhysicalDeviceQueueFamilyProperties2` behaves similarly to `vkGetPhysicalDeviceQueueFamilyProperties`, with the ability to return extended information in a `pNext` chain of output structures.

**Valid Usage (Implicit)**

- VUID-vkGetPhysicalDeviceQueueFamilyProperties2-physicalDevice-parameter `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceQueueFamilyProperties2-pQueueFamilyPropertyCount-parameter `pQueueFamilyPropertyCount` **must** be a valid pointer to a `uint32_t` value
- VUID-vkGetPhysicalDeviceQueueFamilyProperties2-pQueueFamilyProperties-parameter If the value referenced by `pQueueFamilyPropertyCount` is not 0, and `pQueueFamilyProperties`
The `VkQueueFamilyProperties2` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkQueueFamilyProperties2 {
    VkStructureType sType;
    void* pNext;
    VkQueueFamilyProperties queueFamilyProperties;
} VkQueueFamilyProperties2;
```

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

### Valid Usage (Implicit)

- `VUID-VkQueueFamilyProperties2-sType-sType`  
  `sType` must be `VK_STRUCTURE_TYPE_QUEUE_FAMILY_PROPERTIES_2`

- `VUID-VkQueueFamilyProperties2-pNext-pNext`  
  `pNext` must be `NULL`

## 5.2. Devices

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

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

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

A single logical device can be created from multiple physical devices, if those physical devices belong to the same device group. A device group is a set of physical devices that support accessing each other's memory and recording a single command buffer that can be executed on all the physical devices. Device groups are enumerated by calling `vkEnumeratePhysicalDeviceGroups`, and a logical device is created from a subset of the physical devices in a device group by passing the physical devices through `VkDeviceGroupDeviceCreateInfo`. For two physical devices to be in the
same device group, they must support identical extensions, features, and properties.

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

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

```c
// Provided by VK_VERSION_1_1
VkResult vkEnumeratePhysicalDeviceGroups(
    VkInstance instance,           // Provided by VK_VERSION_1_1
    uint32_t* pPhysicalDeviceGroupCount,   // Provided by VK_VERSION_1_1
    VkPhysicalDeviceGroupProperties* pPhysicalDeviceGroupProperties);
```

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

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

Every physical device must be in exactly one device group.

**Valid Usage (Implicit)**

- VUID-vkEnumeratePhysicalDeviceGroups-instance-parameter
  `instance` must be a valid `VkInstance` handle

- VUID-vkEnumeratePhysicalDeviceGroups-pPhysicalDeviceGroupCount-parameter
  `pPhysicalDeviceGroupCount` must be a valid pointer to a `uint32_t` value

- VUID-vkEnumeratePhysicalDeviceGroups-pPhysicalDeviceGroupProperties-parameter
  If the value referenced by `pPhysicalDeviceGroupCount` is not 0, and `pPhysicalDeviceGroupProperties` is not `NULL`, `pPhysicalDeviceGroupProperties` must be a
valid pointer to an array of `pPhysicalDeviceGroupCount` `VkPhysicalDeviceGroupProperties` structures

Return Codes

**Success**
- `VK_SUCCESS`
- `VK_INCOMPLETE`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INITIALIZATION_FAILED`

The `VkPhysicalDeviceGroupProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceGroupProperties {
    VkStructureType sType;
    void* pNext;
    uint32_t physicalDeviceCount;
    VkPhysicalDevice physicalDevices[VK_MAX_DEVICE_GROUP_SIZE];
    VkBool32 subsetAllocation;
} VkPhysicalDeviceGroupProperties;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `physicalDeviceCount` is the number of physical devices in the group.
- `physicalDevices` is an array of `VK_MAX_DEVICE_GROUP_SIZE` `VkPhysicalDevice` handles representing all physical devices in the group. The first `physicalDeviceCount` elements of the array will be valid.
- `subsetAllocation` specifies whether logical devices created from the group support allocating device memory on a subset of devices, via the `deviceMask` member of the `VkMemoryAllocateFlagsInfo`. If this is `VK_FALSE`, then all device memory allocations are made across all physical devices in the group. If `physicalDeviceCount` is 1, then `subsetAllocation must be `VK_FALSE`.

Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceGroupProperties-sType-sType` `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GROUP_PROPERTIES`
- `VUID-VkPhysicalDeviceGroupProperties-pNext-pNext`
**5.2.1. Device Creation**

Logical devices are represented by VkDevice handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_HANDLE(VkDevice)
```

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

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateDevice(
    VkPhysicalDevice physicalDevice,
    const VkDeviceCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkDevice* pDevice);
```

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

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

After verifying and enabling the extensions the VkDevice object is created and returned to the application.

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

Valid Usage

- VUID-vkCreateDevice-ppEnabledExtensionNames-01387
  All required device extensions for each extension in the VkDeviceCreateInfo ::ppEnabledExtensionNames list must also be present in that list

Valid Usage (Implicit)

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

Return Codes

Success
  - VK_SUCCESS

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

The VkDeviceCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDeviceCreateInfo {
    VkStructureType       sType;
    const void*            pNext;
    VkDeviceCreateFlags   flags;
} VkDeviceCreateInfo;
```
uint32_t queueCreateInfoCount;
const VkDeviceQueueCreateInfo* pQueueCreateInfos;
uint32_t enabledLayerCount;
const char* const* ppEnabledLayerNames;
uint32_t enabledExtensionCount;
const char* const* ppEnabledExtensionNames;
const VkPhysicalDeviceFeatures* pEnabledFeatures;
}

VkDeviceCreateInfo

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

Valid Usage

• VUID-VkDeviceCreateInfo-queueFamilyIndex-02802
  The queueFamilyIndex member of each element of pQueueCreateInfos must be unique within pQueueCreateInfos, except that two members can share the same queueFamilyIndex if one describes protected-capable queues and one describes queues that are not protected-capable.

• VUID-VkDeviceCreateInfo-pQueueCreateInfos-06755
  If multiple elements of pQueueCreateInfos share the same queueFamilyIndex, the sum of their queueCount members must be less than or equal to the queueCount member of the VkQueueFamilyProperties structure, as returned by vkGetPhysicalDeviceQueueFamilyProperties in the pQueueFamilyProperties[queueFamilyIndex].

• VUID-VkDeviceCreateInfo-pNext-00373
  If the pNext chain includes a VkPhysicalDeviceFeatures2 structure, then pEnabledFeatures must be NULL.
• VUID-VkDeviceCreateInfo-pNext-02829
If the pNext chain includes a VkPhysicalDeviceVulkan11Features structure, then it must not include a VkPhysicalDevice16BitStorageFeatures, VkPhysicalDeviceMultiviewFeatures, VkPhysicalDeviceVariablePointersFeatures, VkPhysicalDeviceProtectedMemoryFeatures, VkPhysicalDeviceSamplerYcbcrConversionFeatures, or VkPhysicalDeviceShaderDrawParametersFeatures structure

• VUID-VkDeviceCreateInfo-pNext-02830
If the pNext chain includes a VkPhysicalDeviceVulkan12Features structure, then it must not include a VkPhysicalDevice8BitStorageFeatures, VkPhysicalDeviceShaderAtomicInt64Features, VkPhysicalDeviceDescriptorIndexingFeatures, VkPhysicalDeviceScalarBlockLayoutFeatures, VkPhysicalDeviceImagelessFramebufferFeatures, VkPhysicalDeviceUniformBufferStandardLayoutFeatures, VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures, VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures, VkPhysicalDeviceHostQueryResetFeatures, VkPhysicalDeviceTimelineSemaphoreFeatures, VkPhysicalDeviceBufferDeviceAddressFeatures, or VkPhysicalDeviceVulkanMemoryModelFeatures structure

• VUID-VkDeviceCreateInfo-pNext-06532
If the pNext chain includes a VkPhysicalDeviceVulkan13Features structure, then it must not include a VkPhysicalDeviceDynamicRenderingFeatures, VkPhysicalDeviceImageRobustnessFeatures, VkPhysicalDeviceInlineUniformBlockFeatures, VkPhysicalDeviceMaintenance4Features, VkPhysicalDevicePipelineCreationCacheControlFeatures, VkPhysicalDevicePrivateDataFeatures, VkPhysicalDeviceShaderDemoteToHelperInvocationFeatures, VkPhysicalDeviceShaderIntegerDotProductFeatures, VkPhysicalDeviceShaderTerminateInvocationFeatures, VkPhysicalDeviceSubgroupSizeControlFeatures, VkPhysicalDeviceSynchronization2Features, VkPhysicalDeviceTextureCompressionASTCHDRFeatures, or VkPhysicalDeviceZeroInitializeWorkgroupMemoryFeatures structure

Valid Usage (Implicit)

• VUID-VkDeviceCreateInfo-sType-sType
sType must be VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO

• VUID-VkDeviceCreateInfo-pNext-pNext
Each pNext member of any structure (including this one) in the pNext chain must be either NULL or a pointer to a valid instance of VkDeviceGroupDeviceCreateInfo, VkDevicePrivateDataCreateInfo, VkPhysicalDevice16BitStorageFeatures,

• **VUID-VkDeviceCreateInfo-sType-unique**
The `sType` value of each struct in the `pNext` chain **must** be unique, with the exception of structures of type `VkDevicePrivateDataCreateInfo`

• **VUID-VkDeviceCreateInfo-flags-zerobitmask**
  `flags` **must** be 0

• **VUID-VkDeviceCreateInfo-pQueueCreateInfos-parameter**
  `pQueueCreateInfos` **must** be a valid pointer to an array of `queueCreateInfoCount` valid `VkDeviceQueueCreateInfo` structures

• **VUID-VkDeviceCreateInfo-ppEnabledLayerNames-parameter**
  If `enabledLayerCount` is not 0, `ppEnabledLayerNames` **must** be a valid pointer to an array of `enabledLayerCount` null-terminated UTF-8 strings

• **VUID-VkDeviceCreateInfo-ppEnabledExtensionNames-parameter**
  If `enabledExtensionCount` is not 0, `ppEnabledExtensionNames` **must** be a valid pointer to an array of `enabledExtensionCount` null-terminated UTF-8 strings

• **VUID-VkDeviceCreateInfo-pEnabledFeatures-parameter**
  If `pEnabledFeatures` is not NULL, `pEnabledFeatures` **must** be a valid pointer to a valid `VkPhysicalDeviceFeatures` structure
• VUID-VkDeviceCreateInfo-queueCreateInfoCount-arraylength
queueCreateInfoCount must be greater than 0

// Provided by VK_VERSION_1_0
typedef VkFlags VkDeviceCreateFlags;

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

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

// Provided by VK_VERSION_1_1
typedef struct VkDeviceGroupDeviceCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t physicalDeviceCount;
    const VkPhysicalDevice* pPhysicalDevices;
} VkDeviceGroupDeviceCreateInfo;

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• physicalDeviceCount is the number of elements in the pPhysicalDevices array.
• pPhysicalDevices is a pointer to an array of physical device handles belonging to the same
device group.

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

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

Valid Usage

• VUID-VkDeviceGroupDeviceCreateInfo-pPhysicalDevices-00375
Each element of pPhysicalDevices must be unique

• VUID-VkDeviceGroupDeviceCreateInfo-pPhysicalDevices-00376
All elements of pPhysicalDevices must be in the same device group as enumerated by
If physicalDeviceCount is not 0, the physicalDevice parameter of vkCreateDevice must be an element of pPhysicalDevices

Valid Usage (Implicit)

- VUID-VkDeviceGroupDeviceCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_DEVICE_GROUP_DEVICE_CREATE_INFO

- VUID-VkDeviceGroupDeviceCreateInfo-pPhysicalDevices-parameter
  If physicalDeviceCount is not 0, pPhysicalDevices must be a valid pointer to an array of
  physicalDeviceCount valid VkPhysicalDevice handles

To reserve private data storage slots, add a VkDevicePrivateDataCreateInfo structure to the pNext chain of the VkDeviceCreateInfo structure. Reserving slots in this manner is not strictly necessary, but doing so may improve performance.

```
// Provided by VK_VERSION_1_3
typedef struct VkDevicePrivateDataCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t privateDataSlotRequestCount;
} VkDevicePrivateDataCreateInfo;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- privateDataSlotRequestCount is the amount of slots to reserve.

Valid Usage (Implicit)

- VUID-VkDevicePrivateDataCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_DEVICE_PRIVATE_DATA_CREATE_INFO

5.2.2. Device Use

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

- Creation of queues. See the Queues section below for further details.
- Creation and tracking of various synchronization constructs. See Synchronization and Cache Control for further details.
- Allocating, freeing, and managing memory. See Memory Allocation and Resource Creation for
further details.

• Creation and destruction of command buffers and command buffer pools. See Command Buffers for further details.

• Creation, destruction, and management of graphics state. See Pipelines and Resource Descriptors, among others, for further details.

5.2.3. Lost Device

A logical device may become lost for a number of implementation-specific reasons, indicating that pending and future command execution may fail and cause resources and backing memory to become undefined.

Note

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

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

When this happens, certain commands will return VK_ERROR_DEVICE_LOST. After any such event, the logical device is considered lost. It is not possible to reset the logical device to a non-lost state, however the lost state is specific to a logical device (VkDevice), and the corresponding physical device (VkPhysicalDevice) may be otherwise unaffected.

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

Note

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

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

Once a device is lost, command execution may fail, and certain commands that return a VkResult may return VK_ERROR_DEVICE_LOST. These commands can be identified by the inclusion of VK_ERROR_DEVICE_LOST in the Return Codes section for each command. Commands that do not allow runtime errors must still operate correctly for valid usage and, if applicable, return valid data.

Commands that wait indefinitely for device execution (namely vkDeviceWaitIdle, vkQueueWaitIdle, vkWaitForFences with a maximum timeout, and vkGetQueryPoolResults with the VK_QUERY_RESULT_WAIT_BIT bit set in flags) must return in finite time even in the case of a lost device, and return either VK_SUCCESS or VK_ERROR_DEVICE_LOST. For any command that may return VK_ERROR_DEVICE_LOST, for the purpose of determining whether a command buffer is in the pending state, or whether resources are considered in-use by the device, a return value of VK_ERROR_DEVICE_LOST is equivalent to VK_SUCCESS.

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

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

5.2.4. Device Destruction

To destroy a device, call:

```c
// Provided by VK_VERSION_1_0
do
vkDestroyDevice(
    VkDevice device,
    const VkAllocationCallbacks* pAllocator);
```

- device is the logical device to destroy.
- pAllocator controls host memory allocation as described in the Memory Allocation chapter.

To ensure that no work is active on the device, vkDeviceWaitIdle can be used to gate the destruction of the device. Prior to destroying a device, an application is responsible for destroying/freeing any Vulkan objects that were created using that device as the first parameter of the corresponding vkCreate* or vkAllocate* command.
Note
The lifetime of each of these objects is bound by the lifetime of the VkDevice object. Therefore, to avoid resource leaks, it is critical that an application explicitly free all of these resources prior to calling vkDestroyDevice.

Valid Usage

• VUID-vkDestroyDevice-device-00378
  All child objects created on device must have been destroyed prior to destroying device

• VUID-vkDestroyDevice-device-00379
  If VkAllocationCallbacks were provided when device was created, a compatible set of callbacks must be provided here

• VUID-vkDestroyDevice-device-00380
  If no VkAllocationCallbacks were provided when device was created, pAllocator must be NULL

Valid Usage (Implicit)

• VUID-vkDestroyDevice-device-parameter
  If device is not NULL, device must be a valid VkDevice handle

• VUID-vkDestroyDevice-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

Host Synchronization

• Host access to device must be externally synchronized

• Host access to all VkQueue objects created from device must be externally synchronized

5.3. Queues

5.3.1. Queue Family Properties

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

Each index in the pQueueFamilyProperties array returned by vkGetPhysicalDeviceQueueFamilyProperties describes a unique queue family on that physical device. These indices are used when creating queues, and they correspond directly with the queueFamilyIndex that is passed to the vkCreateDevice command via the VkDeviceQueueCreateInfo structure as described in the Queue Creation section below.
Grouping of queue families within a physical device is implementation-dependent.

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

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

5.3.2. Queue Creation

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

Queues are represented by VkQueue handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_HANDLE(VkQueue)
```

The VkDeviceQueueCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDeviceQueueCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkDeviceQueueCreateFlags flags;
    uint32_t queueFamilyIndex;
    uint32_t queueCount;
    const float* pQueuePriorities;
} VkDeviceQueueCreateInfo;
```

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

- **VUID-VkDeviceQueueCreateInfo-queueFamilyIndex-00381**
  queueFamilyIndex must be less than pQueueFamilyPropertyCount returned by vkGetPhysicalDeviceQueueFamilyProperties

- **VUID-VkDeviceQueueCreateInfo-queueCount-00382**
  queueCount must be less than or equal to the queueCount member of the VkQueueFamilyProperties structure, as returned by vkGetPhysicalDeviceQueueFamilyProperties in the pQueueFamilyProperties[queueFamilyIndex]

- **VUID-VkDeviceQueueCreateInfo-pQueuePriorities-00383**
  Each element of pQueuePriorities must be between 0.0 and 1.0 inclusive

- **VUID-VkDeviceQueueCreateInfo-flags-02861**
  If the protectedMemory feature is not enabled, the VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT bit of flags must not be set

- **VUID-VkDeviceQueueCreateInfo-flags-06449**
  If flags includes VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT, queueFamilyIndex must be the index of a queue family that includes the VK_QUEUE_PROTECTED_BIT capability

Valid Usage (Implicit)

- **VUID-VkDeviceQueueCreateInfo-sType-sType**
  sType must be VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO

- **VUID-VkDeviceQueueCreateInfo-pNext-pNext**
  pNext must be NULL

- **VUID-VkDeviceQueueCreateInfo-flags-parameter**
  flags must be a valid combination of VkDeviceQueueCreateFlagBits values

- **VUID-VkDeviceQueueCreateInfo-pQueuePriorities-parameter**
  pQueuePriorities must be a valid pointer to an array of queueCount float values

- **VUID-VkDeviceQueueCreateInfo-queueCount-arraylength**
  queueCount must be greater than 0

Bits which can be set in VkDeviceQueueCreateInfo::flags, specifying usage behavior of a queue, are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkDeviceQueueCreateFlagBits {
// Provided by VK_VERSION_1_1
  VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT = 0x00000001,
} VkDeviceQueueCreateFlagBits;
```

- **VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT** specifies that the device queue is a protected-capable
queue.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkDeviceQueueCreateFlags;
```

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

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

```c
// Provided by VK_VERSION_1_0
void vkGetDeviceQueue(
    VkDevice                      device,
    uint32_t                      queueFamilyIndex,
    uint32_t                      queueIndex,
    VkQueue*                     pQueue);
```

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

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

---

### Valid Usage

- VUID-vkGetDeviceQueue-queueFamilyIndex-00384
  `queueFamilyIndex` must be one of the queue family indices specified when `device` was created, via the `VkDeviceQueueCreateInfo` structure

- VUID-vkGetDeviceQueue-queueIndex-00385
  `queueIndex` must be less than the value of `VkDeviceQueueCreateInfo::queueCount` for the queue family indicated by `queueFamilyIndex` when `device` was created

- VUID-vkGetDeviceQueue-flags-01841
  `VkDeviceQueueCreateInfo::flags` must have been set to zero when `device` was created

---

### Valid Usage (Implicit)

- VUID-vkGetDeviceQueue-device-parameter
  `device` must be a valid `VkDevice` handle

- VUID-vkGetDeviceQueue-pQueue-parameter
To retrieve a handle to a *VkQueue* object with specific *VkDeviceQueueCreateFlags* creation flags, call:

```c
void vkGetDeviceQueue2(
    VkDevice device, 
    const VkDeviceQueueInfo2* pQueueInfo, 
    VkQueue* pQueue);
```

- *device* is the logical device that owns the queue.
- *pQueueInfo* is a pointer to a *VkDeviceQueueInfo2* structure, describing parameters of the device queue to be retrieved.
- *pQueue* is a pointer to a *VkQueue* object that will be filled with the handle for the requested queue.

### Valid Usage (Implicit)

- VUID-vkGetDeviceQueue2-device-parameter
  *device* must be a valid *VkDevice* handle
- VUID-vkGetDeviceQueue2-pQueueInfo-parameter
  *pQueueInfo* must be a valid pointer to a valid *VkDeviceQueueInfo2* structure
- VUID-vkGetDeviceQueue2-pQueue-parameter
  *pQueue* must be a valid pointer to a *VkQueue* handle

The *VkDeviceQueueInfo2* structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkDeviceQueueInfo2 {
    VkStructureType sType;
    const void* pNext;
    VkDeviceQueueCreateFlags flags;
    uint32_t queueFamilyIndex;
    uint32_t queueIndex;
} VkDeviceQueueInfo2;
```

- *sType* is the type of this structure.
- *pNext* is NULL or a pointer to a structure extending this structure. The *pNext* chain of *VkDeviceQueueInfo2* can be used to provide additional device queue parameters to *vkGetDeviceQueue2*.
- *flags* is a *VkDeviceQueueCreateFlags* value indicating the flags used to create the device queue.
- *queueFamilyIndex* is the index of the queue family to which the queue belongs.
• `queueIndex` is the index of the queue to retrieve from within the set of queues that share both the queue family and flags specified.

The queue returned by `vkGetDeviceQueue2` must have the same `flags` value from this structure as that used at device creation time in a `VkDeviceQueueCreateInfo` structure.

*Note*

Normally, if you create both protected-capable and non-protected-capable queues with the same family, they are treated as separate lists of queues and `queueIndex` is relative to the start of the list of queues specified by both `queueFamilyIndex` and `flags`. However, for historical reasons, some implementations may exhibit different behavior. These divergent implementations instead concatenate the lists of queues and treat `queueIndex` as relative to the start of the first list of queues with the given `queueFamilyIndex`. This only matters in cases where an application has created both protected-capable and non-protected-capable queues from the same queue family.

For such divergent implementations, the maximum value of `queueIndex` is equal to the sum of `VkDeviceQueueCreateInfo::queueCount` minus one, for all `VkDeviceQueueCreateInfo` structures that share a common `queueFamilyIndex`.

Such implementations will return `NULL` for either the protected or unprotected queues when calling `vkGetDeviceQueue2` with `queueIndex` in the range zero to `VkDeviceQueueCreateInfo::queueCount` minus one. In cases where these implementations returned `NULL`, the corresponding queues are instead located in the extended range described in the preceding two paragraphs.

This behaviour will not be observed on any driver that has passed Vulkan conformance test suite version 1.3.3.0, or any subsequent version. This information can be found by querying `VkPhysicalDeviceDriverProperties::conformanceVersion`.

### Valid Usage

- **VUID-VkDeviceQueueInfo2-queueFamilyIndex-01842**  
  `queueFamilyIndex` must be one of the queue family indices specified when `device` was created, via the `VkDeviceQueueCreateInfo` structure

- **VUID-VkDeviceQueueInfo2-flags-06225**  
  `flags` must be equal to `VkDeviceQueueCreateInfo::flags` for a `VkDeviceQueueCreateInfo` structure for the queue family indicated by `queueFamilyIndex` when `device` was created

- **VUID-VkDeviceQueueInfo2-queueIndex-01843**  
  `queueIndex` must be less than `VkDeviceQueueCreateInfo::queueCount` for the corresponding queue family and flags indicated by `queueFamilyIndex` and `flags` when `device` was created
Valid Usage (Implicit)

- VUID-VkDeviceQueueInfo2-sType-sType
  sType must be VK_STRUCTURE_TYPE_DEVICE_QUEUE_INFO_2
- VUID-VkDeviceQueueInfo2-pNext-pNext
  pNext must be NULL
- VUID-VkDeviceQueueInfo2-flags-parameter
  flags must be a valid combination of VkDeviceQueueCreateFlagBits values

5.3.3. Queue Family Index

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

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

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

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

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

5.3.4. Queue Priority

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

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

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

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

### 5.3.5. Queue Submission

Work is submitted to a queue via *queue submission* commands such as `vkQueueSubmit2` or `vkQueueSubmit`. Queue submission commands define a set of *queue operations* to be executed by the underlying physical device, including synchronization with semaphores and fences.

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

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

If a fence is present in a queue submission, it describes a *fence signal operation*.

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

**Sparse Memory Binding**

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

### 5.3.6. Queue Destruction

Queues are created along with a logical device during `vkCreateDevice`. All queues associated with a logical device are destroyed when `vkDestroyDevice` is called on that device.
Chapter 6. Command Buffers

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

Command buffers are represented by VkCommandBuffer handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_HANDLE(VkCommandBuffer)
```

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

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

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

6.1. Command Buffer Lifecycle

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

**Initial**

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

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

**Executable**

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

**Pending**

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

**Invalid**

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

![Lifecycle of a command buffer](image)

*Figure 1. Lifecycle of a command buffer*

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

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

Secondary command buffers can be recorded to a primary command buffer via
vkCmdExecuteCommands. This partially ties the lifecycle of the two command buffers together - if the primary is submitted to a queue, both the primary and any secondaries recorded to it move to the pending state. Once execution of the primary completes, so it does for any secondary recorded within it. After all executions of each command buffer complete, they each move to their appropriate completion state (either to the executable state or the invalid state, as specified above).

If a secondary moves to the invalid state or the initial state, then all primary buffers it is recorded in move to the invalid state. A primary moving to any other state does not affect the state of a secondary recorded in it.

**Note**

Resetting or freeing a primary command buffer removes the lifecycle linkage to all secondary command buffers that were recorded into it.

### 6.2. Command Pools

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

Command pools are represented by VkCommandPool handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkCommandPool)
```

To create a command pool, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateCommandPool(
    VkDevice device,
    const VkCommandPoolCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkCommandPool* pCommandPool);
```

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

• VUID-vkCreateCommandPool-queueFamilyIndex-01937
  pCreateInfo->queueFamilyIndex must be the index of a queue family available in the logical device device

Valid Usage (Implicit)

• VUID-vkCreateCommandPool-device-parameter
device must be a valid VkDevice handle

• VUID-vkCreateCommandPool-pCreateInfo-parameter
  pCreateInfo must be a valid pointer to a valid VkCommandPoolCreateInfo structure

• VUID-vkCreateCommandPool-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

• VUID-vkCreateCommandPool-pCommandPool-parameter
  pCommandPool must be a valid pointer to a VkCommandPool handle

Return Codes

Success

• VK_SUCCESS

Failure

• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkCommandPoolCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkCommandPoolCreateInfo {
  VkStructureType sType;
  const void* pNext;
  VkCommandPoolCreateFlags flags;
  uint32_t queueFamilyIndex;
} VkCommandPoolCreateInfo;
```

• sType is the type of this structure.

• pNext is NULL or a pointer to a structure extending this structure.

• flags is a bitmask of VkCommandPoolCreateFlagBits indicating usage behavior for the pool and command buffers allocated from it.
• `queueFamilyIndex` designates a queue family as described in section Queue Family Properties. All command buffers allocated from this command pool must be submitted on queues from the same queue family.

### Valid Usage

- **VUID-VkCommandPoolCreateInfo-flags-02860**
  
  If the `protectedMemory` feature is not enabled, the `VK_COMMAND_POOL_CREATE_PROTECTED_BIT` bit of `flags` must not be set.

### Valid Usage (Implicit)

- **VUID-VkCommandPoolCreateInfo-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO`.

- **VUID-VkCommandPoolCreateInfo-pNext-pNext**
  
  `pNext` must be `NULL`.

- **VUID-VkCommandPoolCreateInfo-flags-parameter**
  
  `flags` must be a valid combination of `VkCommandPoolCreateFlagBits` values.

Bits which can be set in `VkCommandPoolCreateInfo::flags`, specifying usage behavior for a command pool, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkCommandPoolCreateFlagBits {
    VK_COMMAND_POOL_CREATE_TRANSIENT_BIT = 0x00000001,
    VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT = 0x00000002,
    // Provided by VK_VERSION_1_1
    VK_COMMAND_POOL_CREATE_PROTECTED_BIT = 0x00000004,
} VkCommandPoolCreateFlagBits;
```

- **`VK_COMMAND_POOL_CREATE_TRANSIENT_BIT`** specifies that command buffers allocated from the pool will be short-lived, meaning that they will be reset or freed in a relatively short timeframe. This flag may be used by the implementation to control memory allocation behavior within the pool.

- **`VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT`** allows any command buffer allocated from a pool to be individually reset to the initial state; either by calling `vkResetCommandBuffer`, or via the implicit reset when calling `vkBeginCommandBuffer`. If this flag is not set on a pool, then `vkResetCommandBuffer` must not be called for any command buffer allocated from that pool.

- **`VK_COMMAND_POOL_CREATE_PROTECTED_BIT`** specifies that command buffers allocated from the pool are protected command buffers.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkCommandPoolCreateFlags;
```
**VkCommandPoolCreateFlags** is a bitmask type for setting a mask of zero or more **VkCommandPoolCreateFlagBits**.

To trim a command pool, call:

```c
// Provided by VK_VERSION_1_1
void vkTrimCommandPool(
    VkDevice device,
    VkCommandPool commandPool,
    VkCommandPoolTrimFlags flags);
```

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

Trimming a command pool recycles unused memory from the command pool back to the system. Command buffers allocated from the pool are not affected by the command.

**Note**

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

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

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

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

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

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

**Valid Usage (Implicit)**

- **VUID-vkTrimCommandPool-device-parameter**
  
  `device` must be a valid `VkDevice` handle

- **VUID-vkTrimCommandPool-commandPool-parameter**
  
  `commandPool` must be a valid `VkCommandPool` handle

- **VUID-vkTrimCommandPool-flags-zerobitmask**
  
  `flags` must be 0

- **VUID-vkTrimCommandPool-commandPool-parent**
  
  `commandPool` must have been created, allocated, or retrieved from `device`

---

**Host Synchronization**

- Host access to `commandPool` must be externally synchronized

---

```c
// Provided by VK_VERSION_1_1
typedef VkFlags VkCommandPoolTrimFlags;
```

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

To reset a command pool, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkResetCommandPool(
    VkDevice device,
    VkCommandPool commandPool,
    VkCommandPoolResetFlags flags);
```

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

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

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

- VUID-vkResetCommandPool-commandPool-00040
  All VkCommandBuffer objects allocated from commandPool must not be in the pending state.

Valid Usage (Implicit)

- VUID-vkResetCommandPool-device-parameter
  device must be a valid VkDevice handle
- VUID-vkResetCommandPool-commandPool-parameter
  commandPool must be a valid VkCommandPool handle
- VUID-vkResetCommandPool-flags-parameter
  flags must be a valid combination of VkCommandPoolResetFlagBits values
- VUID-vkResetCommandPool-commandPool-parent
  commandPool must have been created, allocated, or retrieved from device

Host Synchronization

- Host access to commandPool must be externally synchronized

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_DEVICE_MEMORY

Bits which can be set in vkResetCommandPool::flags, controlling the reset operation, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkCommandPoolResetFlagBits {
    VK_COMMAND_POOL_RESET_RELEASE_RESOURCES_BIT = 0x00000001,
} VkCommandPoolResetFlagBits;
```

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

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkCommandPoolResetFlags;
```
VkCommandPoolResetFlags is a bitmask type for setting a mask of zero or more VkCommandPoolResetFlagBits.

To destroy a command pool, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyCommandPool(
    VkDevice device,
    VkCommandPool commandPool,
    const VkAllocationCallbacks* pAllocator);
```

- **device** is the logical device that destroys the command pool.
- **commandPool** is the handle of the command pool to destroy.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

When a pool is destroyed, all command buffers allocated from the pool are freed.

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

---

**Valid Usage**

- VUID-vkDestroyCommandPool-commandPool-00041
  All VkCommandBuffer objects allocated from commandPool must not be in the pending state

- VUID-vkDestroyCommandPool-commandPool-00042
  If VkAllocationCallbacks were provided when commandPool was created, a compatible set of callbacks must be provided here

- VUID-vkDestroyCommandPool-commandPool-00043
  If no VkAllocationCallbacks were provided when commandPool was created, pAllocator must be NULL

---

**Valid Usage (Implicit)**

- VUID-vkDestroyCommandPool-device-parameter
  device must be a valid VkDevice handle

- VUID-vkDestroyCommandPool-commandPool-parameter
  If commandPool is not VK_NULL_HANDLE, commandPool must be a valid VkCommandPool handle

- VUID-vkDestroyCommandPool-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkDestroyCommandPool-commandPool-parent
  If commandPool is a valid handle, it must have been created, allocated, or retrieved from
Host Synchronization

- Host access to `commandPool` must be externally synchronized

6.3. Command Buffer Allocation and Management

To allocate command buffers, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkAllocateCommandBuffers(
    VkDevice device,
    const VkCommandBufferAllocateInfo* pAllocateInfo,
    VkCommandBuffer* pCommandBuffers);
```

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

`vkAllocateCommandBuffers` can be used to allocate multiple command buffers. If the allocation of any of those command buffers fails, the implementation must free all successfully allocated command buffer objects from this command, set all entries of the `pCommandBuffers` array to `NULL` and return the error.

**Note**

Filling `pCommandBuffers` with `NULL` values on failure is an exception to the default error behavior that output parameters will have undefined contents.

When command buffers are first allocated, they are in the initial state.

**Valid Usage (Implicit)**

- VUID-vkAllocateCommandBuffers-device-parameter  
  `device` must be a valid `VkDevice` handle
- VUID-vkAllocateCommandBuffers-pAllocateInfo-parameter  
  `pAllocateInfo` must be a valid pointer to a valid `VkCommandBufferAllocateInfo` structure
- VUID-vkAllocateCommandBuffers-pCommandBuffers-parameter  
  `pCommandBuffers` must be a valid pointer to an array of `pAllocateInfo->commandBufferCount`
**VkCommandBuffer handles**

- VUID-vkAllocateCommandBuffers-pAllocateInfo::commandBufferCount-arraylength

  \texttt{pAllocateInfo->commandBufferCount} must be greater than 0

---

**Host Synchronization**

- Host access to \texttt{pAllocateInfo->commandPool} must be externally synchronized

---

**Return Codes**

**Success**

- \texttt{VK_SUCCESS}

**Failure**

- \texttt{VK_ERROR_OUT_OF_HOST_MEMORY}
- \texttt{VK_ERROR_OUT_OFDEVICE_MEMORY}

---

The \texttt{VkCommandBufferAllocateInfo} structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkCommandBufferAllocateInfo {
    VkStructureType sType;
    const void* pNext;
    VkCommandPool commandPool;
    VkCommandBufferLevel level;
    uint32_t commandBufferCount;
} VkCommandBufferAllocateInfo;
```

- \texttt{sType} is the type of this structure.
- \texttt{pNext} is NULL or a pointer to a structure extending this structure.
- \texttt{commandPool} is the command pool from which the command buffers are allocated.
- \texttt{level} is a \texttt{VkCommandBufferLevel} value specifying the command buffer level.
- \texttt{commandBufferCount} is the number of command buffers to allocate from the pool.

---

**Valid Usage (Implicit)**

- VUID-VkCommandBufferAllocateInfo-sType-sType

  \texttt{sType} must be \texttt{VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO}

- VUID-VkCommandBufferAllocateInfo-pNext-pNext

  \texttt{pNext} must be NULL

- VUID-VkCommandBufferAllocateInfo-commandPool-parameter
commandPool must be a valid VkCommandPool handle

- VUID-VkCommandBufferAllocateInfo-level-parameter
  level must be a valid VkCommandBufferLevel value

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

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

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

To reset a command buffer, call:

```c
VkResult vkResetCommandBuffer(
    VkCommandBuffer commandBuffer,      // Provided by VK_VERSION_1_0
    VkCommandBufferResetFlags flags);
```

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

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

**Valid Usage**

- VUID-vkResetCommandBuffer-commandBuffer-00045
  commandBuffer must not be in the pending state

- VUID-vkResetCommandBuffer-commandBuffer-00046
  commandBuffer must have been allocated from a pool that was created with the
  VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT

**Valid Usage (Implicit)**

- VUID-vkResetCommandBuffer-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkResetCommandBuffer-flags-parameter
  flags must be a valid combination of VkCommandBufferResetFlagBits values
Host Synchronization

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

Return Codes

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

Bits which can be set in `vkResetCommandBuffer::flags`, controlling the reset operation, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkCommandBufferResetFlagBits {
    VK_COMMAND_BUFFER_RESET_RELEASE_RESOURCES_BIT = 0x00000001,
} VkCommandBufferResetFlagBits;
```

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

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkCommandBufferResetFlags;
```

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

To free command buffers, call:

```c
// Provided by VK_VERSION_1_0
void vkFreeCommandBuffers(
    VkDevice device,         // Logical device that owns the command pool.
    VkCommandPool commandPool, // Command pool from which the command buffers were allocated.
    uint32_t commandBufferCount, // Number of command buffers to free.
    const VkCommandBuffer* pCommandBuffers); // Pointer to an array of command buffers.
```
• `commandBufferCount` is the length of the `pCommandBuffers` array.

• `pCommandBuffers` is a pointer to an array of handles of command buffers to free.

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

### Valid Usage

- VUID-vkFreeCommandBuffers-pCommandBuffers-00047
  All elements of `pCommandBuffers` must not be in the **pending state**

- VUID-vkFreeCommandBuffers-pCommandBuffers-00048
  `pCommandBuffers` must be a valid pointer to an array of `commandBufferCount` `VkCommandBuffer` handles, each element of which must either be a valid handle or `NULL`.

### Valid Usage (Implicit)

- VUID-vkFreeCommandBuffers-device-parameter
  `device` must be a valid `VkDevice` handle

- VUID-vkFreeCommandBuffers-commandPool-parameter
  `commandPool` must be a valid `VkCommandPool` handle

- VUID-vkFreeCommandBuffers-commandBufferCount-arraylength
  `commandBufferCount` must be greater than 0

- VUID-vkFreeCommandBuffers-commandPool-parent
  `commandPool` must have been created, allocated, or retrieved from `device`

- VUID-vkFreeCommandBuffers-pCommandBuffers-parent
  Each element of `pCommandBuffers` that is a valid handle must have been created, allocated, or retrieved from `commandPool`

### Host Synchronization

- Host access to `commandPool` must be externally synchronized

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

### 6.4. Command Buffer Recording

To begin recording a command buffer, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkBeginCommandBuffer( VkCommandBuffer commandBuffer, const VkCommandBufferBeginInfo* pBeginInfo);
```
• `commandBuffer` is the handle of the command buffer which is to be put in the recording state.
• `pBeginInfo` is a pointer to a `VkCommandBufferBeginInfo` structure defining additional information about how the command buffer begins recording.

### Valid Usage

- VUID-vkBeginCommandBuffer-commandBuffer-00049
  `commandBuffer` must not be in the recording or pending state

- VUID-vkBeginCommandBuffer-commandBuffer-00050
  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

- VUID-vkBeginCommandBuffer-commandBuffer-00051
  If `commandBuffer` is a secondary command buffer, the `pInheritanceInfo` member of `pBeginInfo` must be a valid `VkCommandBufferInheritanceInfo` structure

- VUID-vkBeginCommandBuffer-commandBuffer-00052
  If `commandBuffer` is a secondary command buffer and either the `occlusionQueryEnable` member of the `pInheritanceInfo` member of `pBeginInfo` is `VK_FALSE`, or the `occlusionQueryPrecise` feature is not enabled, then `pBeginInfo->pInheritanceInfo->queryFlags` must not contain `VK_QUERY_CONTROL_PRECISE_BIT`

- VUID-vkBeginCommandBuffer-commandBuffer-02840
  If `commandBuffer` is a primary command buffer, then `pBeginInfo->flags` must not set both the `VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT` and the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flags

### Valid Usage (Implicit)

- VUID-vkBeginCommandBuffer-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkBeginCommandBuffer-pBeginInfo-parameter
  `pBeginInfo` must be a valid pointer to a valid `VkCommandBufferBeginInfo` structure

### Host Synchronization

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

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkCommandBufferBeginInfo structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkCommandBufferBeginInfo {
    VkStructureType sType;
    const void* pNext;
    VkCommandBufferUsageFlags flags;
    const VkCommandBufferInheritanceInfo* pInheritanceInfo;
} VkCommandBufferBeginInfo;
```

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

Valid Usage

• VUID-VkCommandBufferBeginInfo-flags-00055
  If **flags** contains **VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT**, the **framebuffer** member of **pInheritanceInfo** must be either **VK_NULL_HANDLE**, or a valid VkFramebuffer that is compatible with the **renderPass** member of **pInheritanceInfo**

• VUID-VkCommandBufferBeginInfo-flags-06000
  If **flags** contains **VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT** and the **renderPass** member of **pInheritanceInfo** is not **VK_NULL_HANDLE**, **renderPass** must be a valid VkRenderPass

• VUID-VkCommandBufferBeginInfo-flags-06001
  If **flags** contains **VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT** and the **renderPass** member of **pInheritanceInfo** is not **VK_NULL_HANDLE**, the **subpass** member of **pInheritanceInfo** must be a valid subpass index within the **renderPass** member of **pInheritanceInfo**

• VUID-VkCommandBufferBeginInfo-flags-06002
If `flags` contains `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT` and the `renderPass` member of `pInheritanceInfo` is `VK_NULL_HANDLE`, the `pNext` chain of `pInheritanceInfo` must include a `VkCommandBufferInheritanceRenderingInfo` structure.

### Valid Usage (Implicit)

- **VUID-VkCommandBufferBeginInfo-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO`

- **VUID-VkCommandBufferBeginInfo-pNext-pNext**
  
  `pNext` must be `NULL` or a pointer to a valid instance of `VkDeviceGroupCommandBufferBeginInfo`

- **VUID-VkCommandBufferBeginInfo-sType-unique**
  
  The `sType` value of each struct in the `pNext` chain must be unique

- **VUID-VkCommandBufferBeginInfo-flags-parameter**
  
  `flags` must be a valid combination of `VkCommandBufferUsageFlagBits` values

Bits which can be set in `VkCommandBufferBeginInfo::flags`, specifying usage behavior for a command buffer, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkCommandBufferUsageFlagBits {
    VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT = 0x00000001,
    VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT = 0x00000002,
    VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT = 0x00000004,
} VkCommandBufferUsageFlagBits;
```

- **`VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT`** specifies that each recording of the command buffer will only be submitted once, and the command buffer will be reset and recorded again between each submission.

- **`VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT`** specifies that a secondary command buffer is considered to be entirely inside a render pass. If this is a primary command buffer, then this bit is ignored.

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

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkCommandBufferUsageFlags;
```

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

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

```
// Provided by VK_VERSION_1_0
typedef struct VkCommandBufferInheritanceInfo {
    VkStructureType sType;
    const void* pNext;
    VkRenderPass renderPass;
    uint32_t subpass;
    VkFramebuffer framebuffer;
    VkBool32 occlusionQueryEnable;
    VkQueryControlFlags queryFlags;
    VkQueryPipelineStatisticFlags pipelineStatistics;
} VkCommandBufferInheritanceInfo;
```

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

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

- `occlusionQueryEnable` specifies whether the command buffer can be executed while an occlusion query is active in the primary command buffer. If this is `VK_TRUE`, then this command buffer can be executed whether the primary command buffer has an occlusion query active or not. If this is `VK_FALSE`, then the primary command buffer must not have an occlusion query active.
- `queryFlags` specifies the query flags that can be used by an active occlusion query in the primary command buffer when this secondary command buffer is executed. If this value includes the `VK_QUERY_CONTROL_PRECISE_BIT` bit, then the active query can return boolean results or actual sample counts. If this bit is not set, then the active query must not use the `VK_QUERY_CONTROL_PRECISE_BIT` bit.
- `pipelineStatistics` is a bitmask of `VkQueryPipelineStatisticFlagBits` specifying the set of pipeline statistics that can be counted by an active query in the primary command buffer when this secondary command buffer is executed. If this value includes a given bit, then this command buffer can be executed whether the primary command buffer has a pipeline statistics query active that includes this bit or not. If this value excludes a given bit, then the active pipeline statistics query must not be from a query pool that counts that statistic.
If the `VkCommandBuffer` will not be executed within a render pass instance, or if the render pass instance was begun with `vkCmdBeginRendering`, `renderPass`, `subpass`, and `framebuffer` are ignored.

### Valid Usage

- **VUID-VkCommandBufferInheritanceInfo-occlusionQueryEnable-00056**
  If the `inheritedQueries` feature is not enabled, `occlusionQueryEnable` must be `VK_FALSE`

- **VUID-VkCommandBufferInheritanceInfo-queryFlags-00057**
  If the `inheritedQueries` feature is enabled, `queryFlags` must be a valid combination of `VkQueryControlFlagBits` values

- **VUID-VkCommandBufferInheritanceInfo-queryFlags-02788**
  If the `inheritedQueries` feature is not enabled, `queryFlags` must be 0

- **VUID-VkCommandBufferInheritanceInfo-pipelineStatistics-02789**
  If the `pipelineStatisticsQuery` feature is enabled, `pipelineStatistics` must be a valid combination of `VkQueryPipelineStatisticFlagBits` values

- **VUID-VkCommandBufferInheritanceInfo-pipelineStatistics-00058**
  If the `pipelineStatisticsQuery` feature is not enabled, `pipelineStatistics` must be 0

### Valid Usage (Implicit)

- **VUID-VkCommandBufferInheritanceInfo-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_INFO`

- **VUID-VkCommandBufferInheritanceInfo-pNext-pNext**
  `pNext` must be `NULL` or a pointer to a valid instance of `VkCommandBufferInheritanceRenderingInfo`

- **VUID-VkCommandBufferInheritanceInfo-sType-unique**
  The `sType` value of each struct in the `pNext` chain must be unique

- **VUID-VkCommandBufferInheritanceInfo-commonparent**
  Both of `framebuffer`, and `renderPass` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`

---

**Note**

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

If a command buffer is in the `invalid`, or `executable state`, and the command buffer was allocated from a command pool with the `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT` flag set, then `vkBeginCommandBuffer` implicitly resets the command buffer, behaving as if `vkResetCommandBuffer` had been called with `VK_COMMAND_BUFFER_RESET_RELEASE_RESOURCES_BIT` not set. After the implicit reset, `commandBuffer` is moved to the `recording state`.

The `VkCommandBufferInheritanceRenderingInfo` structure is defined as:
typedef struct VkCommandBufferInheritanceRenderingInfo {
    VkStructureType sType;
    const void* pNext;
    VkRenderingFlags flags;
    uint32_t viewMask;
    uint32_t colorAttachmentCount;
    const VkFormat* pColorAttachmentFormats;
    VkFormat depthAttachmentFormat;
    VkFormat stencilAttachmentFormat;
    VkSampleCountFlagBits rasterizationSamples;
} VkCommandBufferInheritanceRenderingInfo;

• sType is the type of this structure
• pNext is NULL or a pointer to a structure extending this structure
• flags is a bitmask of VkRenderingFlagBits used by the render pass instance.
• viewMask is the view mask used for rendering.
• colorAttachmentCount is the number of color attachments specified in the render pass instance.
• pColorAttachmentFormats is a pointer to an array of VkFormat values defining the format of color attachments.
• depthAttachmentFormat is a VkFormat value defining the format of the depth attachment.
• stencilAttachmentFormat is a VkFormat value defining the format of the stencil attachment.
• rasterizationSamples is a VkSampleCountFlagBits specifying the number of samples used in rasterization.

If the pNext chain of VkCommandBufferInheritanceInfo includes a VkCommandBufferInheritanceRenderingInfo structure, then that structure controls parameters of dynamic render pass instances that the VkCommandBuffer can be executed within. If VkCommandBufferInheritanceInfo::renderPass is not VK_NULL_HANDLE, or VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT is not specified in VkCommandBufferBeginInfo ::flags, parameters of this structure are ignored.

If colorAttachmentCount is 0 and the variableMultisampleRate feature is enabled, rasterizationSamples is ignored.

If depthAttachmentFormat, stencilAttachmentFormat, or any element of pColorAttachmentFormats is VK_FORMAT_UNDEFINED, it indicates that the corresponding attachment is unused within the render pass.

Valid Usage

• VUID-VkCommandBufferInheritanceRenderingInfo-colorAttachmentCount-06004
  If colorAttachmentCount is not 0, rasterizationSamples must be a valid VkSampleCountFlagBits value
If the `variableMultisampleRate` feature is not enabled, `rasterizationSamples` must be a valid `VkSampleCountFlagBits` value.

If any element of `pColorAttachmentFormats` is not `VK_FORMAT_UNDEFINED`, it must be a format with potential format features that include `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`.

If `depthAttachmentFormat` is not `VK_FORMAT_UNDEFINED`, it must be a format that includes a depth aspect.

If `depthAttachmentFormat` is not `VK_FORMAT_UNDEFINED`, it must be a format with potential format features that include `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`.

If `stencilAttachmentFormat` is not `VK_FORMAT_UNDEFINED`, it must be a format that includes a stencil aspect.

If `stencilAttachmentFormat` is not `VK_FORMAT_UNDEFINED`, it must be a format with potential format features that include `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`.

If `depthAttachmentFormat` is not `VK_FORMAT_UNDEFINED` and `stencilAttachmentFormat` is not `VK_FORMAT_UNDEFINED`, `depthAttachmentFormat` must equal `stencilAttachmentFormat`.

If the `multiview` feature is not enabled, `viewMask` must be 0.

The index of the most significant bit in `viewMask` must be less than `maxMultiviewViewCount`.

Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_RENDERING_INFO`.
- `flags` must be a valid combination of `VkRenderingFlagBits` values.
- If `colorAttachmentCount` is not 0, `pColorAttachmentFormats` must be a valid pointer to an array of `colorAttachmentCount` valid `VkFormat` values.
- `depthAttachmentFormat` must be a valid `VkFormat` value.
- `stencilAttachmentFormat` must be a valid `VkFormat` value.
- If `rasterizationSamples` is not 0, `rasterizationSamples` must be a valid `VkSampleCountFlagBits` value.
Once recording starts, an application records a sequence of commands (vkCmd*) to set state in the command buffer, draw, dispatch, and other commands.

To complete recording of a command buffer, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkEndCommandBuffer(
    VkCommandBuffer commandBuffer);
```

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

If there was an error during recording, the application will be notified by an unsuccessful return code returned by `vkEndCommandBuffer`. If the application wishes to further use the command buffer, the command buffer **must** be reset.

The command buffer **must** have been in the **recording state**, and is moved to the **executable state**.

---

### Valid Usage

- VUID-vkEndCommandBuffer-commandBuffer-00059
  
  `commandBuffer` **must** be in the **recording state**

- VUID-vkEndCommandBuffer-commandBuffer-00060
  
  If `commandBuffer` is a primary command buffer, there **must** not be an active render pass instance

- VUID-vkEndCommandBuffer-commandBuffer-00061
  
  All queries made **active** during the recording of `commandBuffer` **must** have been made **inactive**

---

### Valid Usage (Implicit)

- VUID-vkEndCommandBuffer-commandBuffer-parameter
  
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle

---

### Host Synchronization

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

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

---

### Return Codes

**Success**

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

### 6.5. Command Buffer Submission

**Note**

Submission can be a high overhead operation, and applications **should** attempt to batch work together into as few calls to `vkQueueSubmit` or `vkQueueSubmit2` as possible.

To submit command buffers to a queue, call:

```c
// Provided by VK_VERSION_1_3
VkResult vkQueueSubmit2(
    VkQueue queue,  // The queue to submit to.
    uint32_t submitCount,  // Number of elements in the pSubmits array.
    const VkSubmitInfo2* pSubmits,  // Pointer to an array of VkSubmitInfo2 structures specifying batches.
    VkFence fence  // Optional fence handle signaling once all submitted command buffers have completed execution.
);
```

- `queue` is the queue that the command buffers will be submitted to.
- `submitCount` is the number of elements in the `pSubmits` array.
- `pSubmits` is a pointer to an array of `VkSubmitInfo2` structures, each specifying a command buffer submission batch.
- `fence` is an **optional** handle to a fence to be signaled once all submitted command buffers have completed execution. If `fence` is not `VK_NULL_HANDLE`, it defines a fence signal operation.

`vkQueueSubmit2` is a **queue submission command**, with each batch defined by an element of `pSubmits`.

Semaphore operations submitted with `vkQueueSubmit2` have additional ordering constraints compared to other submission commands, with dependencies involving previous and subsequent queue operations. Information about these additional constraints can be found in the semaphore section of the **synchronization chapter**.

If any command buffer submitted to this queue is in the **executable state**, it is moved to the **pending state**. Once execution of all submissions of a command buffer complete, it moves from the **pending state**, back to the **executable state**. If a command buffer was recorded with the `VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT` flag, it instead moves back to the **invalid state**.

If `vkQueueSubmit2` fails, it **may** return `VK_ERROR_OUT_OF_HOST_MEMORY` or `VK_ERROR_OUT_OFDEVICE_MEMORY`. If it does, the implementation **must** ensure that the state and contents of any resources or synchronization primitives referenced by the submitted command buffers and any semaphores referenced by `pSubmits` is unaffected by the call or its failure. If `vkQueueSubmit2` fails in such a way...
that the implementation is unable to make that guarantee, the implementation must return \texttt{VK\_ERROR\_DEVICE\_LOST}. See \texttt{Lost\ Device}.

\begin{tcolorbox}
\textbf{Valid Usage}

\begin{itemize}
  \item \textbf{VUID-vkQueueSubmit2-fence-04894}
    \textit{If} \texttt{fence} \textit{is not} \texttt{VK\_NULL\_HANDLE}, \texttt{fence} \textbf{must} be unsignaled
  \item \textbf{VUID-vkQueueSubmit2-fence-04895}
    \textit{If} \texttt{fence} \textit{is not} \texttt{VK\_NULL\_HANDLE}, \texttt{fence} \textbf{must} not be associated with any other queue command that has not yet completed execution on that queue
  \item \textbf{VUID-vkQueueSubmit2-synchronization2-03866}
    The \texttt{synchronization2} feature \textbf{must} be enabled
  \item \textbf{VUID-vkQueueSubmit2-commandBuffer-03867}
    If a command recorded into the \texttt{commandBuffer} member of any element of the \texttt{pCommandBufferInfos} member of any element of \texttt{pSubmits} referenced an \texttt{VkEvent}, that event \textbf{must} not be referenced by a command that has been submitted to another queue and is still in the \textit{pending state}
  \item \textbf{VUID-vkQueueSubmit2-semaphore-03868}
    The \texttt{semaphore} member of any binary semaphore element of the \texttt{pSignalSemaphoreInfos} member of any element of \texttt{pSubmits} \textbf{must} be unsignaled when the semaphore signal operation it defines is executed on the device
  \item \textbf{VUID-vkQueueSubmit2-stageMask-03869}
    The \texttt{stageMask} member of any element of the \texttt{pSignalSemaphoreInfos} member of any element of \texttt{pSubmits} \textbf{must} only include pipeline stages that are supported by the queue family which \texttt{queue} belongs to
  \item \textbf{VUID-vkQueueSubmit2-stageMask-03870}
    The \texttt{stageMask} member of any element of the \texttt{pWaitSemaphoreInfos} member of any element of \texttt{pSubmits} \textbf{must} only include pipeline stages that are supported by the queue family which \texttt{queue} belongs to
  \item \textbf{VUID-vkQueueSubmit2-semaphore-03871}
    When a semaphore wait operation for a binary semaphore is executed, as defined by the \texttt{semaphore} member of any element of the \texttt{pWaitSemaphoreInfos} member of any element of \texttt{pSubmits}, there \textbf{must} be no other queues waiting on the same semaphore
  \item \textbf{VUID-vkQueueSubmit2-semaphore-03872}
    The \texttt{semaphore} member of any element of the \texttt{pWaitSemaphoreInfos} member of any element of \texttt{pSubmits} \textbf{must} be semaphores that are signaled, or have \textit{semaphore signal operations} previously submitted for execution
  \item \textbf{VUID-vkQueueSubmit2-commandBuffer-03874}
    The \texttt{commandBuffer} member of any element of the \texttt{pCommandBufferInfos} member of any element of \texttt{pSubmits} \textbf{must} be in the \textit{pending or executable state}
  \item \textbf{VUID-vkQueueSubmit2-commandBuffer-03875}
    If a command recorded into the \texttt{commandBuffer} member of any element of the \texttt{pCommandBufferInfos} member of any element of \texttt{pSubmits} was not recorded with the
VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT, it must not be in the pending state

- **VUID-vkQueueSubmit2-commandBuffer-03876**
  Any secondary command buffers recorded into the commandBuffer member of any element of the pCommandBufferInfos member of any element of pSubmits must be in the pending or executable state

- **VUID-vkQueueSubmit2-commandBuffer-03877**
  If any secondary command buffers recorded into the commandBuffer member of any element of the pCommandBufferInfos member of any element of pSubmits was not recorded with the VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT, it must not be in the pending state

- **VUID-vkQueueSubmit2-commandBuffer-03878**
  The commandBuffer member of any element of the pCommandBufferInfos member of any element of pSubmits must have been allocated from a VkCommandPool that was created for the same queue family queue belongs to

- **VUID-vkQueueSubmit2-commandBuffer-03879**
  If a command recorded into the commandBuffer member of any element of the pCommandBufferInfos member of any element of pSubmits includes a Queue Family Transfer Acquire Operation, there must exist a previously submitted Queue Family Transfer Release Operation on a queue in the queue family identified by the acquire operation, with parameters matching the acquire operation as defined in the definition of such acquire operations, and which happens before the acquire operation

- **VUID-vkQueueSubmit2-queue-06447**
  If queue was not created with VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT, the flags member of any element of pSubmits must not include VK_SUBMIT_PROTECTED_BIT_KHR

---

**Valid Usage (Implicit)**

- **VUID-vkQueueSubmit2-queue-parameter**
  queue must be a valid VkQueue handle

- **VUID-vkQueueSubmit2-pSubmits-parameter**
  If submitCount is not 0, pSubmits must be a valid pointer to an array of submitCount valid VkSubmitInfo2 structures

- **VUID-vkQueueSubmit2-fence-parameter**
  If fence is not VK_NULL_HANDLE, fence must be a valid VkFence handle

- **VUID-vkQueueSubmit2-commonparent**
  Both of fence, and queue that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same VkDevice

---

**Host Synchronization**

- Host access to queue must be externally synchronized
- Host access to fence must be externally synchronized
# Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
</table>

## Return Codes

### Success
- VK_SUCCESS

### Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST

The VkSubmitInfo2 structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkSubmitInfo2 {
    VkStructureType sType;
    const void* pNext;
   VkSubmitFlags flags;
    uint32_t waitSemaphoreInfoCount;
    const VkSemaphoreSubmitInfo* pWaitSemaphoreInfos;
    uint32_t commandBufferInfoCount;
    const VkCommandBufferSubmitInfo* pCommandBufferInfos;
    uint32_t signalSemaphoreInfoCount;
    const VkSemaphoreSubmitInfo* pSignalSemaphoreInfos;
} VkSubmitInfo2;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is a bitmask of VkSubmitFlagBits.
- **waitSemaphoreInfoCount** is the number of elements in **pWaitSemaphoreInfos**.
- **pWaitSemaphoreInfos** is a pointer to an array of VkSemaphoreSubmitInfo structures defining semaphore wait operations.
- **commandBufferInfoCount** is the number of elements in **pCommandBufferInfos** and the number of command buffers to execute in the batch.
- **pCommandBufferInfos** is a pointer to an array of VkCommandBufferSubmitInfo structures describing command buffers to execute in the batch.
- **signalSemaphoreInfoCount** is the number of elements in **pSignalSemaphoreInfos**.
• **pSignalSemaphoreInfos** is a pointer to an array of **VkSemaphoreSubmitInfo** describing semaphore signal operations.

**Valid Usage**

• VUID-VkSubmitInfo2-flags-03886
  If `flags` includes **VK_SUBMIT_PROTECTED_BIT**, all elements of `pCommandBuffers` **must** be protected command buffers

• VUID-VkSubmitInfo2-flags-03887
  If `flags` does not include **VK_SUBMIT_PROTECTED_BIT**, each element of `pCommandBuffers` **must** not be a protected command buffer

• VUID-VkSubmitInfo2KHR-commandBuffer-06192
  If any `commandBuffer` member of an element of `pCommandBufferInfos` contains any **resumed render pass instances**, they **must** be suspended by a render pass instance earlier in submission order within `pCommandBufferInfos`

• VUID-VkSubmitInfo2KHR-commandBuffer-06010
  If any `commandBuffer` member of an element of `pCommandBufferInfos` contains any **suspended render pass instances**, they **must** be resumed by a render pass instance later in submission order within `pCommandBufferInfos`

• VUID-VkSubmitInfo2KHR-commandBuffer-06011
  If any `commandBuffer` member of an element of `pCommandBufferInfos` contains any **suspended render pass instances**, there **must** be no action or synchronization commands between that render pass instance and the render pass instance that resumes it

• VUID-VkSubmitInfo2KHR-commandBuffer-06012
  If any `commandBuffer` member of an element of `pCommandBufferInfos` contains any **suspended render pass instances**, there **must** be no render pass instances between that render pass instance and the render pass instance that resumes it

**Valid Usage (Implicit)**

• VUID-VkSubmitInfo2-sType-sType
  `sType` **must** be **VK_STRUCTURE_TYPE_SUBMIT_INFO_2**

• VUID-VkSubmitInfo2-pNext-pNext
  `pNext` **must** be **NULL**

• VUID-VkSubmitInfo2-flags-parameter
  `flags` **must** be a valid combination of **VkSubmitFlagBits** values

• VUID-VkSubmitInfo2-pWaitSemaphoreInfos-parameter
  If `waitSemaphoreInfoCount` is not **0**, `pWaitSemaphoreInfos` **must** be a valid pointer to an array of `waitSemaphoreInfoCount` valid **VkSemaphoreSubmitInfo** structures

• VUID-VkSubmitInfo2-pCommandBufferInfos-parameter
  If `commandBufferInfoCount` is not **0**, `pCommandBufferInfos` **must** be a valid pointer to an array of `commandBufferInfoCount` valid **VkCommandBufferSubmitInfo** structures
If `signalSemaphoreInfoCount` is not 0, `pSignalSemaphoreInfos` must be a valid pointer to an array of `signalSemaphoreInfoCount` valid `VkSemaphoreSubmitInfo` structures.

Bits which can be set in `VkSubmitInfo2::flags`, specifying submission behavior, are:

```c
// Provided by VK_VERSION_1_3
typedef enum VkSubmitFlagBits {
    VK_SUBMIT_PROTECTED_BIT = 0x00000001,
    VK_SUBMIT_PROTECTED_BIT_KHR = VK_SUBMIT_PROTECTED_BIT,
} VkSubmitFlagBits;
```

- **VK_SUBMIT_PROTECTED_BIT** specifies that this batch is a protected submission.

```c
// Provided by VK_VERSION_1_3
typedef VkFlags VkSubmitFlags;
```

`VkSubmitFlags` is a bitmask type for setting a mask of zero or more `VkSubmitFlagBits`.

The `VkSemaphoreSubmitInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkSemaphoreSubmitInfo {
    VkStructureType sType;
    const void* pNext;
    VkSemaphore semaphore;
    uint64_t value;
    VkPipelineStageFlags2 stageMask;
    uint32_t deviceIndex;
} VkSemaphoreSubmitInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **semaphore** is a `VkSemaphore` affected by this operation.
- **value** is ignored.
- **stageMask** is a `VkPipelineStageFlags2` mask of pipeline stages which limit the first synchronization scope of a semaphore signal operation, or second synchronization scope of a semaphore wait operation as described in the semaphore wait operation and semaphore signal operation sections of the synchronization chapter.
- **deviceIndex** is the index of the device within a device group that executes the semaphore wait or signal operation.

Whether this structure defines a semaphore wait or signal operation is defined by how it is used.
Valid Usage

• VUID-VkSemaphoreSubmitInfo-stageMask-03929
  If the geometryShader feature is not enabled, stageMask must not contain
  VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT

• VUID-VkSemaphoreSubmitInfo-stageMask-03930
  If the tessellationShader feature is not enabled, stageMask must not contain
  VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT
  or
  VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT

• VUID-VkSemaphoreSubmitInfo-device-03888
  If the device that semaphore was created on is not a device group, deviceIndex must be 0

• VUID-VkSemaphoreSubmitInfo-device-03889
  If the device that semaphore was created on is a device group, deviceIndex must be a valid
  device index

Valid Usage (Implicit)

• VUID-VkSemaphoreSubmitInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_SEMAPHORE_SUBMIT_INFO

• VUID-VkSemaphoreSubmitInfo-pNext-pNext
  pNext must be NULL

• VUID-VkSemaphoreSubmitInfo-semaphore-parameter
  semaphore must be a valid VkSemaphore handle

• VUID-VkSemaphoreSubmitInfo-stageMask-parameter
  stageMask must be a valid combination of VkPipelineStageFlagBits2 values

The VkCommandBufferSubmitInfo structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkCommandBufferSubmitInfo {
    VkStructureType sType;
    const void* pNext;
    VkCommandBuffer commandBuffer;
    uint32_t deviceMask;
} VkCommandBufferSubmitInfo;
```

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• commandBuffer is a VkCommandBuffer to be submitted for execution.
• deviceMask is a bitmask indicating which devices in a device group execute the command buffer.
  A deviceMask of 0 is equivalent to setting all bits corresponding to valid devices in the group to 1.
Valid Usage

• VUID-VkCommandBufferSubmitInfo-commandBuffer-03890
  commandBuffer must not have been allocated with VK_COMMAND_BUFFER_LEVEL_SECONDARY

• VUID-VkCommandBufferSubmitInfo-deviceMask-03891
  If deviceMask is not 0, it must be a valid device mask

Valid Usage (Implicit)

• VUID-VkCommandBufferSubmitInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_COMMAND_BUFFER_SUBMIT_INFO

• VUID-VkCommandBufferSubmitInfo-pNext-pNext
  pNext must be NULL

• VUID-VkCommandBufferSubmitInfo-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

To submit command buffers to a queue, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkQueueSubmit(
  VkQueue queue,
  uint32_t submitCount,
  const VkSubmitInfo* pSubmits,
  VkFence fence);
```

• queue is the queue that the command buffers will be submitted to.

• submitCount is the number of elements in the pSubmits array.

• pSubmits is a pointer to an array of VkSubmitInfo structures, each specifying a command buffer submission batch.

• fence is an optional handle to a fence to be signaled once all submitted command buffers have completed execution. If fence is not VK_NULL_HANDLE, it defines a fence signal operation.

vkQueueSubmit is a queue submission command, with each batch defined by an element of pSubmits. Batches begin execution in the order they appear in pSubmits, but may complete out of order.

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

Details on the interaction of pWaitDstStageMask with synchronization are described in the semaphore wait operation section of the synchronization chapter.

The order that batches appear in pSubmits is used to determine submission order, and thus all the
implicit ordering guarantees that respect it. Other than these implicit ordering guarantees and any explicit synchronization primitives, these batches may overlap or otherwise execute out of order.

If any command buffer submitted to this queue is in the executable state, it is moved to the pending state. Once execution of all submissions of a command buffer complete, it moves from the pending state, back to the executable state. If a command buffer was recorded with the VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT flag, it instead moves to the invalid state.

If vkQueueSubmit fails, it may return VK_ERROR_OUT_OF_HOST_MEMORY or VK_ERROR_OUT_OF_DEVICE_MEMORY. If it does, the implementation must ensure that the state and contents of any resources or synchronization primitives referenced by the submitted command buffers and any semaphores referenced by pSubmits is unaffected by the call or its failure. If vkQueueSubmit fails in such a way that the implementation is unable to make that guarantee, the implementation must return VK_ERROR_DEVICE_LOST. See Lost Device.

Valid Usage

- VUID-vkQueueSubmit-fence-00063
  If fence is not VK_NULL_HANDLE, fence must be unsignaled

- VUID-vkQueueSubmit-fence-00064
  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

- VUID-vkQueueSubmit-pCommandBuffers-00065
  Any calls to vkCmdSetEvent, vkCmdResetEvent or vkCmdWaitEvents that have been recorded into any of the command buffer elements of the pCommandBuffers member of any element of pSubmits, must not reference any VkEvent that is referenced by any of those commands in a command buffer that has been submitted to another queue and is still in the pending state

- VUID-vkQueueSubmit-pWaitDstStageMask-00066
  Any stage flag included in any element of the pWaitDstStageMask member of any element of pSubmits must be a pipeline stage supported by one of the capabilities of queue, as specified in the table of supported pipeline stages

- VUID-vkQueueSubmit-pSignalSemaphores-00067
  Each binary semaphore element of the pSignalSemaphores member of any element of pSubmits must be unsignaled when the semaphore signal operation it defines is executed on the device

- VUID-vkQueueSubmit-pWaitSemaphores-00068
  When a semaphore wait operation referring to a binary semaphore defined by any element of the pWaitSemaphores member of any element of pSubmits executes on queue, there must be no other queues waiting on the same semaphore

- VUID-vkQueueSubmit-pWaitSemaphores-00069
  All elements of the pWaitSemaphores member of all elements of pSubmits must be semaphores that are signaled, or have semaphore signal operations previously submitted for execution

- VUID-vkQueueSubmit-pWaitSemaphores-03238
All elements of the `pWaitSemaphores` member of all elements of `pSubmits` created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_BINARY` must reference a semaphore signal operation that has been submitted for execution and any semaphore signal operations on which it depends (if any) must have also been submitted for execution

- **VUID-vkQueueSubmit-pCommandBuffers-00070**
  Each element of the `pCommandBuffers` member of each element of `pSubmits` must be in the pending or executable state

- **VUID-vkQueueSubmit-pCommandBuffers-00071**
  If any element of the `pCommandBuffers` member of any element of `pSubmits` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT`, it must not be in the pending state

- **VUID-vkQueueSubmit-pCommandBuffers-00072**
  Any secondary command buffers recorded into any element of the `pCommandBuffers` member of any element of `pSubmits` must be in the pending or executable state

- **VUID-vkQueueSubmit-pCommandBuffers-00073**
  If any secondary command buffers recorded into any element of the `pCommandBuffers` member of any element of `pSubmits` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT`, it must not be in the pending state

- **VUID-vkQueueSubmit-pCommandBuffers-00074**
  Each element of the `pCommandBuffers` member of each element of `pSubmits` must have been allocated from a `VkCommandPool` that was created for the same queue family `queue` belongs to

- **VUID-vkQueueSubmit-pSubmits-02207**
  If any element of `pSubmits->pCommandBuffers` includes a Queue Family Transfer Acquire Operation, there must exist a previously submitted Queue Family Transfer Release Operation on a queue in the queue family identified by the acquire operation, with parameters matching the acquire operation as defined in the definition of such acquire operations, and which happens-before the acquire operation

- **VUID-vkQueueSubmit-pSubmits-02808**
  Any resource created with `VK_SHARING_MODE_EXCLUSIVE` that is read by an operation specified by `pSubmits` must not be owned by any queue family other than the one which `queue` belongs to, at the time it is executed

- **VUID-vkQueueSubmit-pSubmits-04626**
  Any resource created with `VK_SHARING_MODE_CONCURRENT` that is accessed by an operation specified by `pSubmits` must have included the queue family of `queue` at resource creation time

- **VUID-vkQueueSubmit-queue-06448**
  If `queue` was not created with `VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT`, there must be no element of `pSubmits` that includes an `VkProtectedSubmitInfo` structure in its `pNext` chain with `protectedSubmit` equal to `VK_TRUE`
Valid Usage (Implicit)

- **VUID-vkQueueSubmit-queue-parameter**
  - `queue` **must** be a valid `VkQueue` handle

- **VUID-vkQueueSubmit-pSubmits-parameter**
  - If `submitCount` is not 0, `pSubmits` **must** be a valid pointer to an array of `submitCount` valid `VkSubmitInfo` structures

- **VUID-vkQueueSubmit-fence-parameter**
  - If `fence` is not `VK_NULL_HANDLE`, `fence` **must** be a valid `VkFence` handle

- **VUID-vkQueueSubmit-commonparent**
  - Both of `fence`, and `queue` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
</table>

Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

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

The `VkSubmitInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSubmitInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t waitSemaphoreCount;
    const VkSemaphore* pWaitSemaphores;
} VkSubmitInfo;
```
const VkPipelineStageFlags* pWaitDstStageMask;
uint32_t commandBufferCount;
const VkCommandBuffer* pCommandBuffers;
uint32_t signalSemaphoreCount;
const VkSemaphore* pSignalSemaphores;

VkSubmitInfo;

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

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

### Valid Usage

- **VUID-VkSubmitInfo-pWaitDstStageMask-04090**
  If the **geometryShader** feature is not enabled, **pWaitDstStageMask** **must** not contain
  **VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT**

- **VUID-VkSubmitInfo-pWaitDstStageMask-04091**
  If the **tessellationShader** feature is not enabled, **pWaitDstStageMask** **must** not contain
  **VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT** or **VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT**

- **VUID-VkSubmitInfo-pWaitDstStageMask-03937**
  If the **synchronization2** feature is not enabled, **pWaitDstStageMask** **must** not be 0

- **VUID-VkSubmitInfo-pCommandBuffers-00075**
  Each element of **pCommandBuffers** **must** not have been allocated with
  **VK_COMMAND_BUFFER_LEVEL_SECONDARY**
Each element of `pWaitDstStageMask` must not include `VK_PIPELINE_STAGE_HOST_BIT`.

If any element of `pWaitSemaphores` or `pSignalSemaphores` was created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE`, then the `pNext` chain must include a `VkTimelineSemaphoreSubmitInfo` structure.

If the `pNext` chain of this structure includes a `VkTimelineSemaphoreSubmitInfo` structure and any element of `pWaitSemaphores` was created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE`, then its `waitSemaphoreValueCount` member must equal `waitSemaphoreCount`.

If the `pNext` chain of this structure includes a `VkTimelineSemaphoreSubmitInfo` structure and any element of `pSignalSemaphores` was created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE`, then its `signalSemaphoreValueCount` member must equal `signalSemaphoreCount`.

For each element of `pSignalSemaphores` created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` the corresponding element of `VkTimelineSemaphoreSubmitInfo::pSignalSemaphoreValues` must have a value greater than the current value of the semaphore when the semaphore signal operation is executed.

For each element of `pWaitSemaphores` created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` the corresponding element of `VkTimelineSemaphoreSubmitInfo::pWaitSemaphoreValues` must have a value which does not differ from the current value of the semaphore or the value of any outstanding semaphore wait or signal operation on that semaphore by more than `maxTimelineSemaphoreValueDifference`.

For each element of `pSignalSemaphores` created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` the corresponding element of `VkTimelineSemaphoreSubmitInfo::pSignalSemaphoreValues` must have a value which does not differ from the current value of the semaphore or the value of any outstanding semaphore wait or signal operation on that semaphore by more than `maxTimelineSemaphoreValueDifference`.

If the `pNext` chain of this structure does not include a `VkProtectedSubmitInfo` structure with `protectedSubmit` set to `VK_TRUE`, then each element of the `pCommandBuffers` array must be an unprotected command buffer.

If the `pNext` chain of this structure includes a `VkProtectedSubmitInfo` structure with `protectedSubmit` set to `VK_TRUE`, then each element of the `pCommandBuffers` array must be a protected command buffer.
If `pCommandBuffers` contains any resumed render pass instances, they must be suspended by a render pass instance earlier in submission order within `pCommandBuffers`.

If `pCommandBuffers` contains any suspended render pass instances, they must be resumed by a render pass instance later in submission order within `pCommandBuffers`.

If `pCommandBuffers` contains any suspended render pass instances, there must be no action or synchronization commands executed in a primary or secondary command buffer between that render pass instance and the render pass instance that resumes it.

If `pCommandBuffers` contains any suspended render pass instances, there must be no render pass instances between that render pass instance and the render pass instance that resumes it.

### Valid Usage (Implicit)

- **VUID-VkSubmitInfo-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_SUBMIT_INFO`

- **VUID-VkSubmitInfo-pNext-pNext**
  
  Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkDeviceGroupSubmitInfo`, `VkProtectedSubmitInfo`, or `VkTimelineSemaphoreSubmitInfo`.

- **VUID-VkSubmitInfo-sType-unique**
  
  The `sType` value of each struct in the `pNext` chain must be unique.

- **VUID-VkSubmitInfo-pWaitSemaphores-parameter**
  
  If `waitSemaphoreCount` is not 0, `pWaitSemaphores` must be a valid pointer to an array of `waitSemaphoreCount` valid `VkSemaphore` handles.

- **VUID-VkSubmitInfo-pWaitDstStageMask-parameter**
  
  If `waitSemaphoreCount` is not 0, `pWaitDstStageMask` must be a valid pointer to an array of `waitSemaphoreCount` valid combinations of `VkPipelineStageFlagBits` values.

- **VUID-VkSubmitInfo-pCommandBuffers-parameter**
  
  If `commandBufferCount` is not 0, `pCommandBuffers` must be a valid pointer to an array of `commandBufferCount` valid `VkCommandBuffer` handles.

- **VUID-VkSubmitInfo-pSignalSemaphores-parameter**
  
  If `signalSemaphoreCount` is not 0, `pSignalSemaphores` must be a valid pointer to an array of `signalSemaphoreCount` valid `VkSemaphore` handles.

- **VUID-VkSubmitInfo-commonparent**
  
  Each of the elements of `pCommandBuffers`, the elements of `pSignalSemaphores`, and the elements of `pWaitSemaphores` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`. 
To specify the values to use when waiting for and signaling semaphores created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE`, add a `VkTimelineSemaphoreSubmitInfo` structure to the `pNext` chain of the `VkSubmitInfo` structure when using `vkQueueSubmit` or the `VkBindSparseInfo` structure when using `vkQueueBindSparse`. The `VkTimelineSemaphoreSubmitInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkTimelineSemaphoreSubmitInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t waitSemaphoreValueCount;
    const uint64_t* pWaitSemaphoreValues;
    uint32_t signalSemaphoreValueCount;
    const uint64_t* pSignalSemaphoreValues;
} VkTimelineSemaphoreSubmitInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `waitSemaphoreValueCount` is the number of semaphore wait values specified in `pWaitSemaphoreValues`.
- `pWaitSemaphoreValues` is a pointer to an array of `waitSemaphoreValueCount` values for the corresponding semaphores in `VkSubmitInfo::pWaitSemaphores` to wait for.
- `signalSemaphoreValueCount` is the number of semaphore signal values specified in `pSignalSemaphoreValues`.
- `pSignalSemaphoreValues` is a pointer to an array of `signalSemaphoreValueCount` values for the corresponding semaphores in `VkSubmitInfo::pSignalSemaphores` to set when signaled.

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

**Valid Usage (Implicit)**

- VUID-VkTimelineSemaphoreSubmitInfo-sType-sType `sType must be VK_STRUCTURE_TYPE_TIMELINE_SEMAPHORE_SUBMIT_INFO`
- VUID-VkTimelineSemaphoreSubmitInfo-pWaitSemaphoreValues-parameter
  If `waitSemaphoreValueCount` is not 0, and `pWaitSemaphoreValues` is not NULL, `pWaitSemaphoreValues must be a valid pointer to an array of `waitSemaphoreValueCount` `uint64_t` values`
- VUID-VkTimelineSemaphoreSubmitInfo-pSignalSemaphoreValues-parameter
  If `signalSemaphoreValueCount` is not 0, and `pSignalSemaphoreValues` is not NULL, `pSignalSemaphoreValues must be a valid pointer to an array of `signalSemaphoreValueCount` `uint64_t` values`
If the `pNext` chain of `VkSubmitInfo` includes a `VkProtectedSubmitInfo` structure, then the structure indicates whether the batch is protected. The `VkProtectedSubmitInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkProtectedSubmitInfo {
    VkStructureType sType;
    const void* pNext;
    VkBool32 protectedSubmit;
} VkProtectedSubmitInfo;
```

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

### Valid Usage (Implicit)

- VUID-VkProtectedSubmitInfo-sType-sType
  - `sType` must be `VK_STRUCTURE_TYPE_PROTECTED_SUBMIT_INFO`

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

The `VkDeviceGroupSubmitInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkDeviceGroupSubmitInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t waitSemaphoreCount;
    const uint32_t* pWaitSemaphoreDeviceIndices;
    uint32_t commandBufferCount;
    const uint32_t* pCommandBufferDeviceMasks;
    uint32_t signalSemaphoreCount;
    const uint32_t* pSignalSemaphoreDeviceIndices;
} VkDeviceGroupSubmitInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `waitSemaphoreCount` is the number of elements in the `pWaitSemaphoreDeviceIndices` array.
- `pWaitSemaphoreDeviceIndices` is a pointer to an array of `waitSemaphoreCount` device indices indicating which physical device executes the semaphore wait operation in the corresponding element of `VkSubmitInfo::pWaitSemaphores`.
- `commandBufferCount` is the number of elements in the `pCommandBufferDeviceMasks` array.
• **pCommandBufferDeviceMasks** is a pointer to an array of `commandBufferCount` device masks indicating which physical devices execute the command buffer in the corresponding element of VkSubmitInfo::pCommandBuffers. A physical device executes the command buffer if the corresponding bit is set in the mask.

• **signalSemaphoreCount** is the number of elements in the pSignalSemaphoreDeviceIndices array.

• **pSignalSemaphoreDeviceIndices** is a pointer to an array of `signalSemaphoreCount` device indices indicating which physical device executes the semaphore signal operation in the corresponding element of VkSubmitInfo::pSignalSemaphores.

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

**Valid Usage**

- VUID-VkDeviceGroupSubmitInfo-waitSemaphoreCount-00082
  waitSemaphoreCount must equal VkSubmitInfo::waitSemaphoreCount

- VUID-VkDeviceGroupSubmitInfo-commandBufferCount-00083
  commandBufferCount must equal VkSubmitInfo::commandBufferCount

- VUID-VkDeviceGroupSubmitInfo-signalSemaphoreCount-00084
  signalSemaphoreCount must equal VkSubmitInfo::signalSemaphoreCount

- VUID-VkDeviceGroupSubmitInfo-pWaitSemaphoreDeviceIndices-00085
  All elements of pWaitSemaphoreDeviceIndices and pSignalSemaphoreDeviceIndices must be valid device indices

- VUID-VkDeviceGroupSubmitInfo-pCommandBufferDeviceMasks-00086
  All elements of pCommandBufferDeviceMasks must be valid device masks

**Valid Usage (Implicit)**

- VUID-VkDeviceGroupSubmitInfo-sType-sType
  sType must be VK_STRUCTURE_TYPEDEVICEGROUP_SUBMIT_INFO

- VUID-VkDeviceGroupSubmitInfo-pWaitSemaphoreDeviceIndices-parameter
  If waitSemaphoreCount is not 0, pWaitSemaphoreDeviceIndices must be a valid pointer to an array of waitSemaphoreCount uint32_t values

- VUID-VkDeviceGroupSubmitInfo-pCommandBufferDeviceMasks-parameter
  If commandBufferCount is not 0, pCommandBufferDeviceMasks must be a valid pointer to an array of commandBufferCount uint32_t values

- VUID-VkDeviceGroupSubmitInfo-pSignalSemaphoreDeviceIndices-parameter
  If signalSemaphoreCount is not 0, pSignalSemaphoreDeviceIndices must be a valid pointer to an array of signalSemaphoreCount uint32_t values
6.6. Queue Forward Progress

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

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

If a command buffer submission waits for any events to be signaled, the application must ensure that command buffer submissions will be able to complete without any subsequent operations by the application. Events signaled by the host must be signaled before the command buffer waits on those events.

**Note**
The ability for commands to wait on the host to set an events was originally added to allow low-latency updates to resources between host and device. However, to ensure quality of service, implementations would necessarily detect extended stalls in execution and timeout after a short period. As this period is not defined in the Vulkan specification, it is impossible to correctly validate any application with any wait period. Since the original users of this functionality were highly limited and platform-specific, this functionality is now considered defunct and should not be used.

6.7. Secondary Command Buffer Execution

A secondary command buffer must not be directly submitted to a queue. Instead, secondary command buffers are recorded to execute as part of a primary command buffer with the command:

```
// Provided by VK_VERSION_1_0
void vkCmdExecuteCommands(
    VkCommandBuffer commandBuffer,  // Must be a handle to a primary command buffer that the secondary command buffers are executed in.
    uint32_t commandBufferCount,    // Length of the pCommandBuffers array.
    const VkCommandBuffer* pCommandBuffers);  // A pointer to an array of commandBufferCount secondary command buffer handles, which are recorded to execute in the primary command buffer in the order they are
```
If any element of \texttt{pCommandBuffers} was not recorded with the \texttt{VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT} flag, and it was recorded into any other primary command buffer which is currently in the \textit{executable or recording state}, that primary command buffer becomes \textit{invalid}.

### Valid Usage

- **VUID-vkCmdExecuteCommands-pCommandBuffers-00088**
  Each element of \texttt{pCommandBuffers} \textbf{must} have been allocated with a level of \texttt{VK_COMMAND_BUFFER_LEVEL_SECONDARY}

- **VUID-vkCmdExecuteCommands-pCommandBuffers-00089**
  Each element of \texttt{pCommandBuffers} \textbf{must} be in the \textit{pending or executable state}

- **VUID-vkCmdExecuteCommands-pCommandBuffers-00091**
  If any element of \texttt{pCommandBuffers} was not recorded with the \texttt{VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT} flag, it \textbf{must} not be in the \textit{pending state}

- **VUID-vkCmdExecuteCommands-pCommandBuffers-00092**
  If any element of \texttt{pCommandBuffers} was not recorded with the \texttt{VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT} flag, it \textbf{must} not have already been recorded to \texttt{commandBuffer}

- **VUID-vkCmdExecuteCommands-pCommandBuffers-00093**
  If any element of \texttt{pCommandBuffers} was not recorded with the \texttt{VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT} flag, it \textbf{must} not appear more than once in \texttt{pCommandBuffers}

- **VUID-vkCmdExecuteCommands-pCommandBuffers-00094**
  Each element of \texttt{pCommandBuffers} \textbf{must} have been allocated from a \texttt{VkCommandPool} that was created for the same queue family as the \texttt{VkCommandPool} from which \texttt{commandBuffer} was allocated

- **VUID-vkCmdExecuteCommands-pCommandBuffers-00096**
  If \texttt{vkCmdExecuteCommands} is being called within a render pass instance, each element of \texttt{pCommandBuffers} \textbf{must} have been recorded with the \texttt{VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT}

- **VUID-vkCmdExecuteCommands-pCommandBuffers-00099**
  If \texttt{vkCmdExecuteCommands} is being called within a render pass instance, and any element of \texttt{pCommandBuffers} was recorded with \texttt{VkCommandBufferInheritanceInfo::framebuffer} not equal to \texttt{VK_NULL_HANDLE}, that \texttt{VkFramebuffer} \textbf{must} match the \texttt{VkFramebuffer} used in the current render pass instance

- **VUID-vkCmdExecuteCommands-contents-06018**
  If \texttt{vkCmdExecuteCommands} is being called within a render pass instance begun with \texttt{vkCmdBeginRenderPass}, its \textit{contents} parameter \textbf{must} have been set to \texttt{VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS}

- **VUID-vkCmdExecuteCommands-pCommandBuffers-06019**
  If \texttt{vkCmdExecuteCommands} is being called within a render pass instance begun with
vkCmdBeginRenderPass, each element of pCommandBuffers must have been recorded with VkCommandBufferInheritanceInfo::subpass set to the index of the subpass which the given command buffer will be executed in

• VUID-vkCmdExecuteCommands-pBeginInfo-06020
  If vkCmdExecuteCommands is being called within a render pass instance begun with vkCmdBeginRenderPass, the render passes specified in the pBeginInfo->pInheritanceInfo->renderPass members of the vkBeginCommandBuffer commands used to begin recording each element of pCommandBuffers must be compatible with the current render pass

• VUID-vkCmdExecuteCommands-pCommandBuffers-00100
  If vkCmdExecuteCommands is not being called within a render pass instance, each element of pCommandBuffers must not have been recorded with the VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT

• VUID-vkCmdExecuteCommands-commandBuffer-00101
  If the inheritedQueries feature is not enabled, commandBuffer must not have any queries active

• VUID-vkCmdExecuteCommands-commandBuffer-00102
  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

• VUID-vkCmdExecuteCommands-commandBuffer-00103
  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

• VUID-vkCmdExecuteCommands-commandBuffer-00104
  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

• VUID-vkCmdExecuteCommands-pCommandBuffers-00105
  Each element of pCommandBuffers must not begin any query types that are active in commandBuffer

• VUID-vkCmdExecuteCommands-commandBuffer-07594
  commandBuffer must not have any queries other than VK_QUERY_TYPE_OCCLUSION and VK_QUERY_TYPE_PIPELINE_STATISTICS active

• VUID-vkCmdExecuteCommands-commandBuffer-01820
  If commandBuffer is a protected command buffer and protectedNoFault is not supported, each element of pCommandBuffers must be a protected command buffer

• VUID-vkCmdExecuteCommands-commandBuffer-01821
  If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, each element of pCommandBuffers must be an unprotected command buffer

• VUID-vkCmdExecuteCommands-commandBuffer-06533
  If vkCmdExecuteCommands is being called within a render pass instance and any recorded command in commandBuffer in the current subpass will write to an image subresource as an attachment, commands recorded in elements of pCommandBuffers must not read from
the memory backing that image subresource in any other way

- VUID-vkCmdExecuteCommands-commandBuffer-06534
  If `vkCmdExecuteCommands` is being called within a render pass instance and any recorded command in `commandBuffer` in the current subpass will read from an image subresource used as an attachment in any way other than as an attachment, commands recorded in elements of `pCommandBuffers` must not write to that image subresource as an attachment

- VUID-vkCmdExecuteCommands-pCommandBuffers-06535
  If `vkCmdExecuteCommands` is being called within a render pass instance and any recorded command in a given element of `pCommandBuffers` will write to an image subresource as an attachment, commands recorded in elements of `pCommandBuffers` at a higher index must not read from the memory backing that image subresource in any other way

- VUID-vkCmdExecuteCommands-pCommandBuffers-06536
  If `vkCmdExecuteCommands` is being called within a render pass instance and any recorded command in a given element of `pCommandBuffers` will read from an image subresource used as an attachment in any way other than as an attachment, commands recorded in elements of `pCommandBuffers` at a higher index must not write to that image subresource as an attachment

- VUID-vkCmdExecuteCommands-pCommandBuffers-06021
  If `pCommandBuffers` contains any suspended render pass instances, there must be no action or synchronization commands between that render pass instance and any render pass instance that resumes it

- VUID-vkCmdExecuteCommands-pCommandBuffers-06022
  If `pCommandBuffers` contains any suspended render pass instances, there must be no render pass instances between that render pass instance and any render pass instance that resumes it

- VUID-vkCmdExecuteCommands-flags-06024
  If `vkCmdExecuteCommands` is being called within a render pass instance begun with `vkCmdBeginRendering`, its `VkRenderingInfo::flags` parameter must have included `VK_RENDERING_CONTENTS_SECONDARY_COMMAND_BUFFERS_BIT`

- VUID-vkCmdExecuteCommands-pBeginInfo-06025
  If `vkCmdExecuteCommands` is being called within a render pass instance begun with `vkCmdBeginRendering`, the render passes specified in the `pBeginInfo->pInheritanceInfo->renderPass` members of the `vkBeginCommandBuffer` commands used to begin recording each element of `pCommandBuffers` must be `VK_NULL_HANDLE`

- VUID-vkCmdExecuteCommands-flags-06026
  If `vkCmdExecuteCommands` is being called within a render pass instance begun with `vkCmdBeginRendering`, the `flags` member of the `VkCommandBufferInheritanceRenderingInfo` structure included in the `pNext` chain of `VkCommandBufferBeginInfo::pInheritanceInfo` used to begin recording each element of `pCommandBuffers` must be equal to the `VkRenderingInfo::flags` parameter to `vkCmdBeginRendering`, excluding `VK_RENDERING_CONTENTS_SECONDARY_COMMAND_BUFFERS_BIT`

- VUID-vkCmdExecuteCommands-colorAttachmentCount-06027
  If `vkCmdExecuteCommands` is being called within a render pass instance begun with `vkCmdBeginRendering`, the `colorAttachmentCount` member of the
VkCommandBufferInheritanceRenderingInfo structure included in the pNext chain of VkCommandBufferBeginInfo::pInheritanceInfo used to begin recording each element of pCommandBuffers must be equal to the VkRenderingInfo::colorAttachmentCount parameter to vkCmdBeginRendering

- VUID-vkCmdExecuteCommands-imageView-06028
  If vkCmdExecuteCommands is being called within a render pass instance begun with vkCmdBeginRendering, if the imageView member of an element of the VkRenderingInfo::pColorAttachments parameter to vkCmdBeginRendering is not VK_NULL_HANDLE, the corresponding element of the pColorAttachmentFormats member of the VkCommandBufferInheritanceRenderingInfo structure included in the pNext chain of VkCommandBufferBeginInfo::pInheritanceInfo used to begin recording each element of pCommandBuffers must be equal to the format used to create that image view

- VUID-vkCmdExecuteCommands-imageView-07606
  If vkCmdExecuteCommands is being called within a render pass instance begun with vkCmdBeginRendering, if the imageView member of an element of the VkRenderingInfo::pColorAttachments parameter to vkCmdBeginRendering is VK_NULL_HANDLE, the corresponding element of the pColorAttachmentFormats member of the VkCommandBufferInheritanceRenderingInfo structure included in the pNext chain of VkCommandBufferBeginInfo::pInheritanceInfo used to begin recording each element of pCommandBuffers must be VK_FORMAT_UNDEFINED

- VUID-vkCmdExecuteCommands-pDepthAttachment-06029
  If vkCmdExecuteCommands is being called within a render pass instance begun with vkCmdBeginRendering, if the VkRenderingInfo::pDepthAttachment->imageView parameter to vkCmdBeginRendering is not VK_NULL_HANDLE, the value of the depthAttachmentFormat member of the VkCommandBufferInheritanceRenderingInfo structure included in the pNext chain of VkCommandBufferBeginInfo::pInheritanceInfo used to begin recording each element of pCommandBuffers must be equal to the format used to create that image view

- VUID-vkCmdExecuteCommands-pStencilAttachment-06030
  If vkCmdExecuteCommands is being called within a render pass instance begun with vkCmdBeginRendering, if the VkRenderingInfo::pStencilAttachment->imageView parameter to vkCmdBeginRendering is not VK_NULL_HANDLE, the value of the stencilAttachmentFormat member of the VkCommandBufferInheritanceRenderingInfo structure included in the pNext chain of VkCommandBufferBeginInfo::pInheritanceInfo used to begin recording each element of pCommandBuffers must be equal to the format used to create that image view

- VUID-vkCmdExecuteCommands-pDepthAttachment-06774
  If vkCmdExecuteCommands is being called within a render pass instance begun with vkCmdBeginRendering and the VkRenderingInfo::pDepthAttachment->imageView parameter to vkCmdBeginRendering was VK_NULL_HANDLE, the value of the depthAttachmentFormat member of the VkCommandBufferInheritanceRenderingInfo structure included in the pNext chain of VkCommandBufferBeginInfo::pInheritanceInfo used to begin recording each element of pCommandBuffers must be VK_FORMAT_UNDEFINED

- VUID-vkCmdExecuteCommands-pStencilAttachment-06775
  If vkCmdExecuteCommands is being called within a render pass instance begun with
vkCmdBeginRendering and the VkRenderingInfo::pStencilAttachment->imageView parameter to vkCmdBeginRendering was VK_NULL_HANDLE, the value of the stencilAttachmentFormat member of the VkCommandBufferInheritanceRenderingInfo structure included in the pNext chain of VkCommandBufferBeginInfo::pInheritanceInfo used to begin recording each element of pCommandBuffers must be VK_FORMAT_UNDEFINED

- VUID-vkCmdExecuteCommands-viewMask-06031
  If vkCmdExecuteCommands is being called within a render pass instance begun with vkCmdBeginRendering, the viewMask member of the VkCommandBufferInheritanceRenderingInfo structure included in the pNext chain of VkCommandBufferBeginInfo::pInheritanceInfo used to begin recording each element of pCommandBuffers must be equal to the VkRenderingInfo::viewMask parameter to vkCmdBeginRendering

Valid Usage (Implicit)

- VUID-vkCmdExecuteCommands-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdExecuteCommands-pCommandBuffers-parameter
  pCommandBuffers must be a valid pointer to an array of commandBufferCount valid VkCommandBuffer handles

- VUID-vkCmdExecuteCommands-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdExecuteCommands-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support transfer, graphics, or compute operations

- VUID-vkCmdExecuteCommands-bufferlevel
  commandBuffer must be a primary VkCommandBuffer

- VUID-vkCmdExecuteCommands-commandBufferCount-arraylength
  commandBufferCount must be greater than 0

- VUID-vkCmdExecuteCommands-commonparent
  Both of commandBuffer, and the elements of pCommandBuffers must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized
6.8. Command Buffer Device Mask

Each command buffer has a piece of state storing the current device mask of the command buffer. This mask controls which physical devices within the logical device all subsequent commands will execute on, including state-setting commands, action commands, and synchronization commands.

Scissor and viewport state (excluding the count of each) can be set to different values on each physical device (only when set as dynamic state), and each physical device will render using its local copy of the state. Other state is shared between physical devices, such that all physical devices use the most recently set values for the state. However, when recording an action command that uses a piece of state, the most recent command that set that state must have included all physical devices that execute the action command in its current device mask.

The command buffer's device mask is orthogonal to the pCommandBufferDeviceMasks member of VkDeviceGroupSubmitInfo. Commands only execute on a physical device if the device index is set in both device masks.

If the pNext chain of VkCommandBufferBeginInfo includes a VkDeviceGroupCommandBufferBeginInfo structure, then that structure includes an initial device mask for the command buffer.

The VkDeviceGroupCommandBufferBeginInfo structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkDeviceGroupCommandBufferBeginInfo {
    VkStructureType sType;
    const void*   pNext;
    uint32_t      deviceMask;
} VkDeviceGroupCommandBufferBeginInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **deviceMask** is the initial value of the command buffer's device mask.

The initial device mask also acts as an upper bound on the set of devices that can ever be in the device mask in the command buffer.

If this structure is not present, the initial value of a command buffer's device mask is set to include
all physical devices in the logical device when the command buffer begins recording.

Valid Usage

- VUID-VkDeviceGroupCommandBufferBeginInfo-deviceMask-00106
deviceMask must be a valid device mask value
- VUID-VkDeviceGroupCommandBufferBeginInfo-deviceMask-00107
deviceMask must not be zero

Valid Usage (Implicit)

- VUID-VkDeviceGroupCommandBufferBeginInfo-sType-sType
sType must be VK_STRUCTURE_TYPE_DEVICE_GROUP_COMMAND_BUFFER_BEGIN_INFO

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

```c
// Provided by VK_VERSION_1_1
def void vkCmdSetDeviceMask(
    VkCommandBuffer commandBuffer,
    uint32_t deviceMask);
```

- commandBuffer is command buffer whose current device mask is modified.
- deviceMask is the new value of the current device mask.

deviceMask is used to filter out subsequent commands from executing on all physical devices whose bit indices are not set in the mask, except commands beginning a render pass instance, commands transitioning to the next subpass in the render pass instance, and commands ending a render pass instance, which always execute on the set of physical devices whose bit indices are included in the deviceMask member of the VkDeviceGroupRenderPassBeginInfo structure passed to the command beginning the corresponding render pass instance.

Valid Usage

- VUID-vkCmdSetDeviceMask-deviceMask-00108
deviceMask must be a valid device mask value
- VUID-vkCmdSetDeviceMask-deviceMask-00109
deviceMask must not be zero
- VUID-vkCmdSetDeviceMask-deviceMask-00110
deviceMask must not include any set bits that were not in the VkDeviceGroupCommandBufferBeginInfo::deviceMask value when the command buffer began recording
- VUID-vkCmdSetDeviceMask-deviceMask-00111
If vkCmdSetDeviceMask is called inside a render pass instance, deviceMask must not include
any set bits that were not in the `VkDeviceGroupRenderPassBeginInfo::deviceMask` value when the render pass instance began recording

### Valid Usage (Implicit)

- VUID-vkCmdSetDeviceMask-commandBuffer-parameter
  
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetDeviceMask-commandBuffer-recording
  
  `commandBuffer` **must** be in the recording state

- VUID-vkCmdSetDeviceMask-commandBuffer-cmdpool
  
  The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, compute, or transfer operations

### Host Synchronization

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

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics Compute Transfer</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 7. Synchronization and Cache Control

Synchronization of access to resources is primarily the responsibility of the application in Vulkan. The order of execution of commands with respect to the host and other commands on the device has few implicit guarantees, and needs to be explicitly specified. Memory caches and other optimizations are also explicitly managed, requiring that the flow of data through the system is largely under application control.

Whilst some implicit guarantees exist between commands, five explicit synchronization mechanisms are exposed by Vulkan:

Fences
Fences can be used to communicate to the host that execution of some task on the device has completed.

Semaphores
Semaphores can be used to control resource access across multiple queues.

Events
Events provide a fine-grained synchronization primitive which can be signaled either within a command buffer or by the host, and can be waited upon within a command buffer or queried on the host.

Pipeline Barriers
Pipeline barriers also provide synchronization control within a command buffer, but at a single point, rather than with separate signal and wait operations.

Render Passes
Render passes provide a useful synchronization framework for most rendering tasks, built upon the concepts in this chapter. Many cases that would otherwise need an application to use other synchronization primitives can be expressed more efficiently as part of a render pass.

7.1. Execution and Memory Dependencies

An operation is an arbitrary amount of work to be executed on the host, a device, or an external entity such as a presentation engine. Synchronization commands introduce explicit execution dependencies, and memory dependencies between two sets of operations defined by the command’s two synchronization scopes.

The synchronization scopes define which other operations a synchronization command is able to create execution dependencies with. Any type of operation that is not in a synchronization command’s synchronization scopes will not be included in the resulting dependency. For example, for many synchronization commands, the synchronization scopes can be limited to just operations executing in specific pipeline stages, which allows other pipeline stages to be excluded from a dependency. Other scoping options are possible, depending on the particular command.
An execution dependency is a guarantee that for two sets of operations, the first set must happen-before the second set. If an operation happens-before another operation, then the first operation must complete before the second operation is initiated. More precisely:

- Let $\text{Ops}_1$ and $\text{Ops}_2$ be separate sets of operations.
- Let $\text{Sync}$ be a synchronization command.
- Let $\text{Scope}_{1st}$ and $\text{Scope}_{2nd}$ be the synchronization scopes of $\text{Sync}$.
- Let $\text{ScopedOps}_1$ be the intersection of sets $\text{Ops}_1$ and $\text{Scope}_{1st}$.
- Let $\text{ScopedOps}_2$ be the intersection of sets $\text{Ops}_2$ and $\text{Scope}_{2nd}$.
- Submitting $\text{Ops}_1$, $\text{Sync}$ and $\text{Ops}_2$ for execution, in that order, will result in execution dependency $\text{ExeDep}$ between $\text{ScopedOps}_1$ and $\text{ScopedOps}_2$.
- Execution dependency $\text{ExeDep}$ guarantees that $\text{ScopedOps}_1$ happen-before $\text{ScopedOps}_2$.

An execution dependency chain is a sequence of execution dependencies that form a happens-before relation between the first dependency’s $\text{ScopedOps}_1$ and the final dependency’s $\text{ScopedOps}_2$. For each consecutive pair of execution dependencies, a chain exists if the intersection of $\text{Scope}_{2nd}$ in the first dependency and $\text{Scope}_{1st}$ in the second dependency is not an empty set. The formation of a single execution dependency from an execution dependency chain can be described by substituting the following in the description of execution dependencies:

- Let $\text{Sync}$ be a set of synchronization commands that generate an execution dependency chain.
- Let $\text{Scope}_{1st}$ be the first synchronization scope of the first command in $\text{Sync}$.
- Let $\text{Scope}_{2nd}$ be the second synchronization scope of the last command in $\text{Sync}$.

Execution dependencies alone are not sufficient to guarantee that values resulting from writes in one set of operations can be read from another set of operations.

Three additional types of operations are used to control memory access. Availability operations cause the values generated by specified memory write accesses to become available to a memory domain for future access. Any available value remains available until a subsequent write to the same memory location occurs (whether it is made available or not) or the memory is freed. Memory domain operations cause writes that are available to a source memory domain to become available to a destination memory domain (an example of this is making writes available to the host domain available to the device domain). Visibility operations cause values available to a memory domain to become visible to specified memory accesses.

Availability, visibility, memory domains, and memory domain operations are formally defined in the Availability and Visibility section of the Memory Model chapter. Which API operations perform each of these operations is defined in Availability, Visibility, and Domain Operations.

A memory dependency is an execution dependency which includes availability and visibility operations such that:

- The first set of operations happens-before the availability operation.
- The availability operation happens-before the visibility operation.
- The visibility operation happens-before the second set of operations.
Once written values are made visible to a particular type of memory access, they can be read or written by that type of memory access. Most synchronization commands in Vulkan define a memory dependency.

The specific memory accesses that are made available and visible are defined by the access scopes of a memory dependency. Any type of access that is in a memory dependency’s first access scope and occurs in ScopedOps₁ is made available. Any type of access that is in a memory dependency’s second access scope and occurs in ScopedOps₂ has any available writes made visible to it. Any type of operation that is not in a synchronization command’s access scopes will not be included in the resulting dependency.

A memory dependency enforces availability and visibility of memory accesses and execution order between two sets of operations. Adding to the description of execution dependency chains:

- Let MemOps₁ be the set of memory accesses performed by ScopedOps₁.
- Let MemOps₂ be the set of memory accesses performed by ScopedOps₂.
- Let AccessScope₁st be the first access scope of the first command in the Sync chain.
- Let AccessScope₂nd be the second access scope of the last command in the Sync chain.
- Let ScopedMemOps₁ be the intersection of sets MemOps₁ and AccessScope₁st.
- Let ScopedMemOps₂ be the intersection of sets MemOps₂ and AccessScope₂nd.
- Submitting Ops₁, Sync, and Ops₂ for execution, in that order, will result in a memory dependency MemDep between ScopedOps₁ and ScopedOps₂.
- Memory dependency MemDep guarantees that:
  - Memory writes in ScopedMemOps₁ are made available.
  - Available memory writes, including those from ScopedMemOps₁, are made visible to ScopedMemOps₂.

Note

Execution and memory dependencies are used to solve data hazards, i.e. to ensure that read and write operations occur in a well-defined order. Write-after-read hazards can be solved with just an execution dependency, but read-after-write and write-after-write hazards need appropriate memory dependencies to be included between them. If an application does not include dependencies to solve these hazards, the results and execution orders of memory accesses are undefined.

7.1.1. Image Layout Transitions

Image subresources can be transitioned from one layout to another as part of a memory dependency (e.g. by using an image memory barrier). When a layout transition is specified in a memory dependency, it happens-after the availability operations in the memory dependency, and happens-before the visibility operations. Image layout transitions may perform read and write accesses on all memory bound to the image subresource range, so applications must ensure that all memory writes have been made available before a layout transition is executed. Available memory is automatically made visible to a layout transition, and writes performed by a layout transition are automatically made available.
Layout transitions always apply to a particular image subresource range, and specify both an old layout and new layout. The old layout must either be `VK_IMAGE_LAYOUT_UNDEFINED`, or match the current layout of the image subresource range. If the old layout matches the current layout of the image subresource range, the transition preserves the contents of that range. If the old layout is `VK_IMAGE_LAYOUT_UNDEFINED`, the contents of that range may be discarded.

As image layout transitions may perform read and write accesses on the memory bound to the image, if the image subresource affected by the layout transition is bound to peer memory for any device in the current device mask then the memory heap the bound memory comes from must support the `VK_PEER_MEMORY_FEATURE_GENERIC_SRC_BIT` and `VK_PEER_MEMORY_FEATURE_GENERIC_DST_BIT` capabilities as returned by `vkGetDeviceGroupPeerMemoryFeatures`.

**Note**

Applications must ensure that layout transitions happen-after all operations accessing the image with the old layout, and happen-before any operations that will access the image with the new layout. Layout transitions are potentially read/write operations, so not defining appropriate memory dependencies to guarantee this will result in a data race.

Image layout transitions interact with memory aliasing.

Layout transitions that are performed via image memory barriers execute in their entirety in submission order, relative to other image layout transitions submitted to the same queue, including those performed by render passes. In effect there is an implicit execution dependency from each such layout transition to all layout transitions previously submitted to the same queue.

### 7.1.2. Pipeline Stages

The work performed by an action or synchronization command consists of multiple operations, which are performed as a sequence of logically independent steps known as pipeline stages. The exact pipeline stages executed depend on the particular command that is used, and current command buffer state when the command was recorded. Drawing commands, dispatching commands, copy commands, clear commands, and synchronization commands all execute in different sets of pipeline stages. Synchronization commands do not execute in a defined pipeline stage.

**Note**

Operations performed by synchronization commands (e.g. availability and visibility operations) are not executed by a defined pipeline stage. However other commands can still synchronize with them by using the synchronization scopes to create a dependency chain.

Execution of operations across pipeline stages must adhere to implicit ordering guarantees, particularly including pipeline stage order. Otherwise, execution across pipeline stages may overlap or execute out of order with regards to other stages, unless otherwise enforced by an execution dependency.

Several of the synchronization commands include pipeline stage parameters, restricting the
synchronization scopes for that command to just those stages. This allows fine grained control over the exact execution dependencies and accesses performed by action commands. Implementations should use these pipeline stages to avoid unnecessary stalls or cache flushing.

Bits which can be set in a VkPipelineStageFlags2 mask, specifying stages of execution, are:

```c
// Provided by VK_VERSION_1_3
// Flag bits for VkPipelineStageFlagBits2
typedef VkFlags64 VkPipelineStageFlagBits2;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_NONE = 0ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_NONE_KHR = 0ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_TOP_OF_PIPE_BIT = 0x00000001ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_TOP_OF_PIPE_BIT_KHR = 0x00000001ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT = 0x00000002ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR = 0x00000002ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT = 0x00000004ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR = 0x00000004ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT = 0x00000008ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT_KHR = 0x00000008ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT = 0x00000010ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR = 0x00000010ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT = 0x00000020ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR = 0x00000020ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT = 0x00000040ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR = 0x00000040ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT = 0x00000080ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR = 0x00000080ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT = 0x00000100ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR = 0x00000100ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT = 0x00000200ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR = 0x00000200ULL;
```
= 0x00000200ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT
= 0x00000400ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR
= 0x00000400ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_COMPUTE_SHADER_BIT
= 0x00000800ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_COMPUTE_SHADER_BIT_KHR
= 0x00000800ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT
= 0x00001000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR
= 0x00001000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_TRANSFER_BIT
= 0x00001000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_TRANSFER_BIT_KHR
= 0x00001000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_BOTTOM_OF_PIPE_BIT
= 0x00002000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_BOTTOM_OF_PIPE_BIT_KHR
= 0x00002000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_HOST_BIT
= 0x00004000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_HOST_BIT_KHR
= 0x00004000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT
= 0x00008000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR
= 0x00008000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT
= 0x00010000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR
= 0x00010000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_COPY_BIT
= 0x100000000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_COPY_BIT_KHR
= 0x100000000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_RESOLVE_BIT
= 0x200000000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR
= 0x200000000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_BLIT_BIT
= 0x400000000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_BLIT_BIT_KHR
= 0x400000000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_CLEAR_BIT
= 0x800000000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR
= 0x800000000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT
= 0x2000000000ULL;
static const VkPipelineStageFlagBits2 VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR
= 0x2000000000ULL;
- **VK_PIPELINE_STAGE_2_NONE** specifies no stages of execution.
- **VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT** specifies the stage of the pipeline where indirect command parameters are consumed.
- **VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT** specifies the stage of the pipeline where index buffers are consumed.
- **VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT** specifies the stage of the pipeline where vertex buffers are consumed.
- **VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT** is equivalent to the logical OR of:
  - **VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT**
  - **VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT**
- **VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT** specifies the vertex shader stage.
- **VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT** specifies the tessellation control shader stage.
- **VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT** specifies the tessellation evaluation shader stage.
- **VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT** specifies the geometry shader stage.
- **VK_PIPELINE_STAGE_2_PRE_RASTERIZATION_SHADERS_BIT** is equivalent to specifying all supported pre-rasterization shader stages:
  - **VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT**
  - **VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT**
  - **VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT**
  - **VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT**
- **VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT** specifies the fragment shader stage.
- **VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT** specifies the stage of the pipeline where early fragment tests (depth and stencil tests before fragment shading) are performed. This stage also includes subpass load operations for framebuffer attachments with a depth/stencil format.
- **VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT** specifies the stage of the pipeline where late fragment tests (depth and stencil tests after fragment shading) are performed. This stage also includes subpass store operations for framebuffer attachments with a depth/stencil format.
- **VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT** specifies the stage of the pipeline after blending where the final color values are output from the pipeline. This stage also includes subpass load and store operations, multisample resolve operations for framebuffer attachments with a color or depth/stencil format, and **vkCmdClearAttachments**.
- **VK_PIPELINE_STAGE_2_COMPUTE_SHADER_BIT** specifies the compute shader stage.
- **VK_PIPELINE_STAGE_2_HOST_BIT** specifies a pseudo-stage indicating execution on the host of reads/writes of device memory. This stage is not invoked by any commands recorded in a command buffer.
- **VK_PIPELINE_STAGE_2_COPY_BIT** specifies the execution of all copy commands, including `vkCmdCopyQueryPoolResults`.
- **VK_PIPELINE_STAGE_2_BLIT_BIT** specifies the execution of `vkCmdBlitImage`.
- **VK_PIPELINE_STAGE_2_RESOLVE_BIT** specifies the execution of `vkCmdResolveImage`.
- **VK_PIPELINE_STAGE_2_CLEAR_BIT** specifies the execution of clear commands, with the exception of `vkCmdClearAttachments`.
- **VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT** is equivalent to specifying all of:
  - **VK_PIPELINE_STAGE_2_COPY_BIT**
  - **VK_PIPELINE_STAGE_2_BLIT_BIT**
  - **VK_PIPELINE_STAGE_2_RESOLVE_BIT**
  - **VK_PIPELINE_STAGE_2_CLEAR_BIT**
- **VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT** specifies the execution of all graphics pipeline stages, and is equivalent to the logical OR of:
  - **VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT**
  - **VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT**
  - **VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT**
  - **VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT**
  - **VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT**
  - **VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT**
  - **VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT**
  - **VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT**
  - **VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT**
  - **VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT**
- **VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT** specifies all operations performed by all commands supported on the queue it is used with.
- **VK_PIPELINE_STAGE_2_TOP_OF_PIPE_BIT** is equivalent to **VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT** with `VkAccessFlags2` set to 0 when specified in the second synchronization scope, but equivalent to **VK_PIPELINE_STAGE_2_NONE** in the first scope.
- **VK_PIPELINE_STAGE_2_BOTTOM_OF_PIPE_BIT** is equivalent to **VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT** with `VkAccessFlags2` set to 0 when specified in the first synchronization scope, but equivalent to **VK_PIPELINE_STAGE_2_NONE** in the second scope.

**Note**
The **TOP** and **BOTTOM** pipeline stages are deprecated, and applications should prefer
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT and VK_PIPELINE_STAGE_2_NONE.

*Note*
The VkPipelineStageFlags2 bitmask goes beyond the 31 individual bit flags allowable within a C99 enum, which is how VkPipelineStageFlagBits is defined. The first 31 values are common to both, and are interchangeable.

VkPipelineStageFlags2 is a bitmask type for setting a mask of zero or more VkPipelineStageFlagBits2 flags:

```c
// Provided by VK_VERSION_1_3
typedef VkFlags64 VkPipelineStageFlags2;
```

Bits which can be set in a VkPipelineStageFlags mask, specifying stages of execution, are:

```c
// Provided by VK_VERSION_1_0
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,
    // Provided by VK_VERSION_1_3
    VK_PIPELINE_STAGE_NONE = 0,
} VkPipelineStageFlagBits;
```

These values all have the same meaning as the equivalently named values for VkPipelineStageFlags2.

- **VK_PIPELINE_STAGE_NONE** specifies no stages of execution.
- **VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT** specifies the stage of the pipeline where VkDrawIndirect* / VkDispatchIndirect* / VkTraceRaysIndirect* data structures are consumed.
- **VK_PIPELINE_STAGE_VERTEX_INPUT_BIT** specifies the stage of the pipeline where vertex and index buffers are consumed.
- **VK_PIPELINE_STAGE_VERTEX_SHADER_BIT** specifies the vertex shader stage.
- **VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT** specifies the tessellation control shader stage.
- **VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT** specifies the tessellation evaluation shader stage.
- **VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT** specifies the geometry shader stage.
- **VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT** specifies the fragment shader stage.
- **VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT** specifies the stage of the pipeline where early fragment tests (depth and stencil tests before fragment shading) are performed. This stage also includes subpass load operations for framebuffer attachments with a depth/stencil format.
- **VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT** specifies the stage of the pipeline where late fragment tests (depth and stencil tests after fragment shading) are performed. This stage also includes subpass store operations for framebuffer attachments with a depth/stencil format.
- **VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT** specifies the stage of the pipeline after blending where the final color values are output from the pipeline. This stage also includes subpass load and store operations, multisample resolve operations for framebuffer attachments with a color or depth/stencil format, and **vkCmdClearAttachments**.
- **VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT** specifies the execution of a compute shader.
- **VK_PIPELINE_STAGE_TRANSFER_BIT** specifies the following commands:
  - All copy commands, including **vkCmdCopyQueryPoolResults**
  - **vkCmdBlitImage2** and **vkCmdBlitImage**
  - **vkCmdResolveImage2** and **vkCmdResolveImage**
  - All clear commands, with the exception of **vkCmdClearAttachments**
- **VK_PIPELINE_STAGE_HOST_BIT** specifies a pseudo-stage indicating execution on the host of reads/writes of device memory. This stage is not invoked by any commands recorded in a command buffer.
- **VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT** specifies the execution of all graphics pipeline stages, and is equivalent to the logical OR of:
  - **VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT**
  - **VK_PIPELINE_STAGE_VERTEX_INPUT_BIT**
  - **VK_PIPELINE_STAGE_VERTEX_SHADER_BIT**
  - **VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT**
  - **VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT**
  - **VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT**
  - **VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT**
  - **VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT**
  - **VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT**
  - **VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT**
VK_PIPELINE_STAGE_ALL_COMMANDS_BIT specifies all operations performed by all commands supported on the queue it is used with.

VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT is equivalent to VK_PIPELINE_STAGE_ALL_COMMANDS_BIT with VkAccessFlags set to 0 when specified in the second synchronization scope, but specifies no stage of execution when specified in the first scope.

VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT is equivalent to VK_PIPELINE_STAGE_ALL_COMMANDS_BIT with VkAccessFlags set to 0 when specified in the first synchronization scope, but specifies no stage of execution when specified in the second scope.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineStageFlags;
```

VkPipelineStageFlags is a bitmask type for setting a mask of zero or more VkPipelineStageFlagBits.

If a synchronization command includes a source stage mask, its first synchronization scope only includes execution of the pipeline stages specified in that mask, and its first access scope only includes memory accesses performed by pipeline stages specified in that mask.

If a synchronization command includes a destination stage mask, its second synchronization scope only includes execution of the pipeline stages specified in that mask, and its second access scope only includes memory access performed by pipeline stages specified in that mask.

---

**Note**

Including a particular pipeline stage in the first synchronization scope of a command implicitly includes logically earlier pipeline stages in the synchronization scope. Similarly, the second synchronization scope includes logically later pipeline stages.

However, note that access scopes are not affected in this way - only the precise stages specified are considered part of each access scope.

Certain pipeline stages are only available on queues that support a particular set of operations. The following table lists, for each pipeline stage flag, which queue capability flag must be supported by the queue. When multiple flags are enumerated in the second column of the table, it means that the pipeline stage is supported on the queue if it supports any of the listed capability flags. For further details on queue capabilities see Physical Device Enumeration and Queues.

**Table 3. Supported pipeline stage flags**

<table>
<thead>
<tr>
<th>Pipeline stage flag</th>
<th>Required queue capability flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_PIPELINE_STAGE_NONE</td>
<td>None required</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT</td>
</tr>
</tbody>
</table>
Pipeline stages that execute as a result of a command logically complete execution in a specific order, such that completion of a logically later pipeline stage must not happen-before completion of a logically earlier stage. This means that including any stage in the source stage mask for a particular synchronization command also implies that any logically earlier stages are included in Scope\textsubscript{1st} for that command.

Similarly, initiation of a logically earlier pipeline stage must not happen-after initiation of a logically later pipeline stage. Including any given stage in the destination stage mask for a particular synchronization command also implies that any logically later stages are included in Scope\textsubscript{2nd} for that command.

Note: Implementations may not support synchronization at every pipeline stage for every synchronization operation. If a pipeline stage that an implementation does not support synchronization for appears in a source stage mask, it may substitute any logically later stage in its place for the first synchronization scope. If a pipeline

<table>
<thead>
<tr>
<th>Pipeline stage flag</th>
<th>Required queue capability flag</th>
<th>Required queue capability flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_PIPELINE_STAGE_2_PRE_RASTERIZATION_SHADERS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>None required</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSellation_CONTROL_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSellation_EVALUATION_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_COMPUTE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_HOST_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_COMPUTE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>(VK_PIPELINE_STAGE_TRANSFER_BIT)</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_COPY_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_BLIT_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_RESOLVE_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_CLEAR_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_COPY_BIT_KHR</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_COMMANDS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
</tbody>
</table>
stage that an implementation does not support synchronization for appears in a destination stage mask, it may substitute any logically earlier stage in its place for the second synchronization scope.

For example, if an implementation is unable to signal an event immediately after vertex shader execution is complete, it may instead signal the event after color attachment output has completed.

If an implementation makes such a substitution, it must not affect the semantics of execution or memory dependencies or image and buffer memory barriers.

Graphics pipelines are executable on queues supporting VK_QUEUE_GRAPHICS_BIT. Stages executed by graphics pipelines can only be specified in commands recorded for queues supporting VK_QUEUE_GRAPHICS_BIT.

The graphics pipeline executes the following stages, with the logical ordering of the stages matching the order specified here:

- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT
- VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT
- VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
- VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT

For the compute pipeline, the following stages occur in this order:

- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT

For the transfer pipeline, the following stages occur in this order:

- VK_PIPELINE_STAGE_TRANSFER_BIT

For host operations, only one pipeline stage occurs, so no order is guaranteed:

- VK_PIPELINE_STAGE_HOST_BIT

7.1.3. Access Types

Memory in Vulkan can be accessed from within shader invocations and via some fixed-function
stages of the pipeline. The access type is a function of the descriptor type used, or how a fixed-function stage accesses memory.

Some synchronization commands take sets of access types as parameters to define the access scopes of a memory dependency. If a synchronization command includes a source access mask, its first access scope only includes accesses via the access types specified in that mask. Similarly, if a synchronization command includes a destination access mask, its second access scope only includes accesses via the access types specified in that mask.

Bits which can be set in the srcAccessMask and dstAccessMask members of VkMemoryBarrier2KHR, VkImageMemoryBarrier2KHR, and VkBufferMemoryBarrier2KHR, specifying access behavior, are:

```c
// Provided by VK_VERSION_1_3
// Flag bits for VkAccessFlagBits2
typedef VkFlags64 VkAccessFlagBits2;
static const VkAccessFlagBits2 VK_ACCESS_2_NONE = 0ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_NONE_KHR = 0ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT = 0x00000001ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT_KHR = 0x00000001ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_INDEX_READ_BIT = 0x00000002ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_INDEX_READ_BIT_KHR = 0x00000002ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT = 0x00000004ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT_KHR = 0x00000004ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_UNIFORM_READ_BIT = 0x00000008ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_UNIFORM_READ_BIT_KHR = 0x00000008ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT = 0x00000010ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT_KHR = 0x00000010ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_SHADER_READ_BIT = 0x00000020ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_SHADER_READ_BIT_KHR = 0x00000020ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_SHADER_WRITE_BIT = 0x00000040ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_SHADER_WRITE_BIT_KHR = 0x00000040ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT = 0x00000080ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR = 0x00000080ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT = 0x00000100ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT_KHR = 0x00000100ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT = 0x00000200ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR = 0x00000200ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT = 0x00000400ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT_KHR = 0x00000400ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_TRANSFER_READ_BIT = 0x00000800ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_TRANSFER_READ_BIT_KHR = 0x00000800ULL;
```

168
static const VkAccessFlagBits2 VK_ACCESS_2_TRANSFER_WRITE_BIT = 0x00001000ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_TRANSFER_WRITE_BIT_KHR = 0x00001000ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_HOST_READ_BIT = 0x00002000ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_HOST_READ_BIT_KHR = 0x00002000ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_HOST_WRITE_BIT = 0x00004000ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_HOST_WRITE_BIT_KHR = 0x00004000ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_MEMORY_READ_BIT = 0x00008000ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_MEMORY_READ_BIT_KHR = 0x00008000ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_MEMORY_WRITE_BIT = 0x00010000ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_MEMORY_WRITE_BIT_KHR = 0x00010000ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_SHADER_SAMPLED_READ_BIT = 0x100000000ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_SHADER_SAMPLED_READ_BIT_KHR = 0x100000000ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_SHADER_STORAGE_READ_BIT = 0x200000000ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR = 0x200000000ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT = 0x400000000ULL;
static const VkAccessFlagBits2 VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT_KHR = 0x400000000ULL;

• VK_ACCESS_2_NONE specifies no accesses.
• VK_ACCESS_2_MEMORY_READ_BIT specifies all read accesses. It is always valid in any access mask, and is treated as equivalent to setting all READ access flags that are valid where it is used.
• VK_ACCESS_2_MEMORY_WRITE_BIT specifies all write accesses. It is always valid in any access mask, and is treated as equivalent to setting all WRITE access flags that are valid where it is used.
• VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT specifies read access to command data read from indirect buffers as part of an indirect drawing or dispatch command. Such access occurs in the VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT pipeline stage.
• VK_ACCESS_2_INDEX_READ_BIT specifies read access to an index buffer as part of an indexed drawing command, bound by vkCmdBindIndexBuffer. Such access occurs in the VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT pipeline stage.
• VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT specifies read access to a vertex buffer as part of a drawing command, bound by vkCmdBindVertexBuffers. Such access occurs in the VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT pipeline stage.
• VK_ACCESS_2_UNIFORM_READ_BIT specifies read access to a uniform buffer in any shader pipeline stage.
• VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT specifies read access to an input attachment within a render pass during fragment shading. Such access occurs in the VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT pipeline stage.
• VK_ACCESS_2_SHADER_SAMPLED_READ_BIT specifies read access to a uniform texel buffer or sampled image in any shader pipeline stage.
• VK_ACCESS_2_SHADER_STORAGE_READ_BIT specifies read access to a storage buffer, physical storage buffer, storage texel buffer, or storage image in any shader pipeline stage.
• VK_ACCESS_2_SHADER_READ_BIT is equivalent to the logical OR of:
- **VK_ACCESS_2_UNIFORM_READ_BIT**
- **VK_ACCESS_2_SHADER_SAMPLED_READ_BIT**
- **VK_ACCESS_2_SHADER_STORAGE_READ_BIT**

**VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT** specifies write access to a storage buffer, physical storage buffer, storage texel buffer, or storage image in any shader pipeline stage.

**VK_ACCESS_2_SHADER_WRITE_BIT** is equivalent to **VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT**.

**VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT** specifies read access to a color attachment, such as via blending, logic operations, or via certain subpass load operations. Such access occurs in the **VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT** pipeline stage.

**VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT** specifies write access to a color, resolve, or depth/stencil resolve attachment during a render pass or via certain subpass load and store operations. Such access occurs in the **VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT** pipeline stage.

**VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT** specifies read access to a depth/stencil attachment, via depth or stencil operations or via certain subpass load operations. Such access occurs in the **VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT** or **VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT** pipeline stages.

**VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT** specifies write access to a depth/stencil attachment, via depth or stencil operations or via certain subpass load and store operations. Such access occurs in the **VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT** or **VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT** pipeline stages.

**VK_ACCESS_2_TRANSFER_READ_BIT** specifies read access to an image or buffer in a copy operation. Such access occurs in the **VK_PIPELINE_STAGE_2_COPY_BIT**, **VK_PIPELINE_STAGE_2_BLIT_BIT**, or **VK_PIPELINE_STAGE_2_RESOLVE_BIT** pipeline stages.

**VK_ACCESS_2_TRANSFER_WRITE_BIT** specifies write access to an image or buffer in a clear or copy operation. Such access occurs in the **VK_PIPELINE_STAGE_2_COPY_BIT**, **VK_PIPELINE_STAGE_2_BLIT_BIT**, **VK_PIPELINE_STAGE_2_CLEAR_BIT**, or **VK_PIPELINE_STAGE_2_RESOLVE_BIT** pipeline stages.

**VK_ACCESS_2_HOST_READ_BIT** specifies read access by a host operation. Accesses of this type are not performed through a resource, but directly on memory. Such access occurs in the **VK_PIPELINE_STAGE_2_HOST_BIT** pipeline stage.

**VK_ACCESS_2_HOST_WRITE_BIT** specifies write access by a host operation. Accesses of this type are not performed through a resource, but directly on memory. Such access occurs in the **VK_PIPELINE_STAGE_2_HOST_BIT** pipeline stage.

---

**Note**

In situations where an application wishes to select all access types for a given set of pipeline stages, **VK_ACCESS_2_MEMORY_READ_BIT** or **VK_ACCESS_2_MEMORY_WRITE_BIT** can be used. This is particularly useful when specifying stages that only have a single access type.

---

**Note**
The `VkAccessFlags2` bitmask goes beyond the 31 individual bit flags allowable within a C99 enum, which is how `VkAccessFlagBits` is defined. The first 31 values are common to both, and are interchangeable.

`VkAccessFlags2` is a bitmask type for setting a mask of zero or more `VkAccessFlagBits`:

```
// Provided by VK_VERSION_1_3
typedef VkFlags64 VkAccessFlags2;
```

Bits which can be set in the `srcAccessMask` and `dstAccessMask` members of `VkSubpassDependency`, `VkMemoryBarrier`, `VkBufferMemoryBarrier`, and `VkImageMemoryBarrier`, specifying access behavior, are:

```
// Provided by VK_VERSION_1_0
typedef enum VkAccessFlagBits {
    VK_ACCESS_INDIRECT_COMMAND_READ_BIT = 0x00000001,
    VK_ACCESS_INDEX_READ_BIT = 0x00000002,
    VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT = 0x00000004,
    VK_ACCESS_UNIFORM_READ_BIT = 0x00000008,
    VK_ACCESS_INPUT_ATTACHMENT_READ_BIT = 0x00000010,
    VK_ACCESS_SHADER_READ_BIT = 0x00000020,
    VK_ACCESS_SHADER_WRITE_BIT = 0x00000040,
    VK_ACCESS_COLOR_ATTACHMENT_READ_BIT = 0x00000080,
    VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT = 0x00000100,
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT = 0x00000200,
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT = 0x00000400,
    VK_ACCESS_TRANSFER_READ_BIT = 0x00000800,
    VK_ACCESS_TRANSFER_WRITE_BIT = 0x00001000,
    VK_ACCESS_HOST_READ_BIT = 0x00002000,
    VK_ACCESS_HOST_WRITE_BIT = 0x00004000,
    VK_ACCESS_MEMORY_READ_BIT = 0x00008000,
    VK_ACCESS_MEMORY_WRITE_BIT = 0x00010000,
    // Provided by VK_VERSION_1_3
    VK_ACCESS_NONE = 0,
} VkAccessFlagBits;
```

These values all have the same meaning as the equivalently named values for `VkAccessFlags2`.

- `VK_ACCESS_NONE` specifies no accesses.
- `VK_ACCESS_MEMORY_READ_BIT` specifies all read accesses. It is always valid in any access mask, and is treated as equivalent to setting all `READ` access flags that are valid where it is used.
- `VK_ACCESS_MEMORY_WRITE_BIT` specifies all write accesses. It is always valid in any access mask, and is treated as equivalent to setting all `WRITE` access flags that are valid where it is used.
- `VK_ACCESS_INDIRECT_COMMAND_READ_BIT` specifies read access to indirect command data read as part of an indirect drawing or dispatching command. Such access occurs in the `VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT` pipeline stage.
• **VK_ACCESS_INDEX_READ_BIT** specifies read access to an index buffer as part of an indexed drawing command, bound by `vkCmdBindIndexBuffer`. Such access occurs in the **VK_PIPELINE_STAGE_VERTEX_INPUT_BIT** pipeline stage.

• **VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT** specifies read access to a vertex buffer as part of a drawing command, bound by `vkCmdBindVertexBuffers`. Such access occurs in the **VK_PIPELINE_STAGE_VERTEX_INPUT_BIT** pipeline stage.

• **VK_ACCESS_UNIFORM_READ_BIT** specifies read access to a **uniform buffer** in any shader pipeline stage.

• **VK_ACCESS_INPUT_ATTACHMENT_READ_BIT** specifies read access to an **input attachment** within a render pass during fragment shading. Such access occurs in the **VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT** pipeline stage.

• **VK_ACCESS_SHADER_READ_BIT** specifies read access to a **uniform buffer**, **uniform texel buffer**, **sampled image**, **storage buffer**, **physical storage buffer**, **storage texel buffer**, or **storage image** in any shader pipeline stage.

• **VK_ACCESS_SHADER_WRITE_BIT** specifies write access to a **storage buffer**, **physical storage buffer**, **storage texel buffer**, or **storage image** in any shader pipeline stage.

• **VK_ACCESS_COLOR_ATTACHMENT_READ_BIT** specifies read access to a **color attachment**, such as via blending, logic operations, or via certain subpass load operations.

• **VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT** specifies write access to a **color**, **resolve**, or **depth/stencil resolve attachment** during a render pass or via certain subpass load and store operations. Such access occurs in the **VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT** pipeline stage.

• **VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT** specifies read access to a **depth/stencil attachment**, via depth or stencil operations or via certain subpass load operations. Such access occurs in the **VK_PIPELINE_STAGE_1_EARLY_FRAGMENT_TESTS_BIT** or **VK_PIPELINE_STAGE_1_LATE_FRAGMENT_TESTS_BIT** pipeline stages.

• **VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT** specifies write access to a **depth/stencil attachment**, via depth or stencil operations or via certain subpass load and store operations. Such access occurs in the **VK_PIPELINE_STAGE_1_EARLY_FRAGMENT_TESTS_BIT** or **VK_PIPELINE_STAGE_1_LATE_FRAGMENT_TESTS_BIT** pipeline stages.

• **VK_ACCESS_TRANSFER_READ_BIT** specifies read access to an image or buffer in a **copy** operation. Such access occurs in the **VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT** pipeline stage.

• **VK_ACCESS_TRANSFER_WRITE_BIT** specifies write access to an image or buffer in a **clear** or **copy** operation. Such access occurs in the **VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT** pipeline stage.

• **VK_ACCESS_HOST_READ_BIT** specifies read access by a host operation. Accesses of this type are not performed through a resource, but directly on memory. Such access occurs in the **VK_PIPELINE_STAGE_HOST_BIT** pipeline stage.

• **VK_ACCESS_HOST_WRITE_BIT** specifies write access by a host operation. Accesses of this type are not performed through a resource, but directly on memory. Such access occurs in the **VK_PIPELINE_STAGE_HOST_BIT** pipeline stage.

Certain access types are only performed by a subset of pipeline stages. Any synchronization command that takes both stage masks and access masks uses both to define the **access scopes** - only the specified access types performed by the specified stages are included in the access scope. An
application **must** not specify an access flag in a synchronization command if it does not include a pipeline stage in the corresponding stage mask that is able to perform accesses of that type. The following table lists, for each access flag, which pipeline stages **can** perform that type of access.

**Table 4. Supported access types**

<table>
<thead>
<tr>
<th>Access flag</th>
<th>Supported pipeline stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_ACCESS_INDIRECT_COMMAND_READ_BIT</td>
<td>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_INDEX_READ_BIT</td>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT</td>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_UNIFORM_READ_BIT</td>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT, VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT, VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT, VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, or VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_READ_BIT</td>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT, VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT, VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT, VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, or VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT, VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT, VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT, VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, or VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_INPUT_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT, or VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT, or VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_READ_BIT</td>
<td>VK_PIPELINE_STAGE_TRANSFER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_TRANSFER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_HOST_READ_BIT</td>
<td>VK_PIPELINE_STAGE_HOST_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_HOST_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_HOST_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_READ_BIT</td>
<td>Any</td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_WRITE_BIT</td>
<td>Any</td>
</tr>
</tbody>
</table>
typedef VkFlags VkAccessFlags;

VkAccessFlags is a bitmask type for setting a mask of zero or more VkAccessFlagBits.

If a memory object does not have the VK_MEMORY_PROPERTY_HOST_COHERENT_BIT property, then vkFlushMappedMemoryRanges must be called in order to guarantee that writes to the memory object from the host are made available to the host domain, where they can be further made available to the device domain via a domain operation. Similarly, vkInvalidateMappedMemoryRanges must be called to guarantee that writes which are available to the host domain are made visible to host operations.

If the memory object does have the VK_MEMORY_PROPERTY_HOST_COHERENT_BIT property flag, writes to the memory object from the host are automatically made available to the host domain. Similarly, writes made available to the host domain are automatically made visible to the host.

Note
Queue submission commands automatically perform a domain operation from host to device for all writes performed before the command executes, so in most cases an explicit memory barrier is not needed for this case. In the few circumstances where a submit does not occur between the host write and the device read access, writes can be made available by using an explicit memory barrier.

7.1.4. Framebuffer Region Dependencies

Pipeline stages that operate on, or with respect to, the framebuffer are collectively the framebuffer-space pipeline stages. These stages are:

- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT

For these pipeline stages, an execution or memory dependency from the first set of operations to the second set can either be a single framebuffer-global dependency, or split into multiple framebuffer-local dependencies. A dependency with non-framebuffer-space pipeline stages is neither framebuffer-global nor framebuffer-local.

A framebuffer region is a set of sample (x, y, layer, sample) coordinates that is a subset of the entire framebuffer.

Both synchronization scopes of a framebuffer-local dependency include only the operations performed within corresponding framebuffer regions (as defined below). No ordering guarantees are made between different framebuffer regions for a framebuffer-local dependency.

Both synchronization scopes of a framebuffer-global dependency include operations on all
framebuffer-regions.

If the first synchronization scope includes operations on pixels/fragments with $N$ samples and the second synchronization scope includes operations on pixels/fragments with $M$ samples, where $N$ does not equal $M$, then a framebuffer region containing all samples at a given $(x, y, \text{layer})$ coordinate in the first synchronization scope corresponds to a region containing all samples at the same coordinate in the second synchronization scope. In other words, it is a pixel granularity dependency. If $N$ equals $M$, then a framebuffer region containing a single $(x, y, \text{layer, sample})$ coordinate in the first synchronization scope corresponds to a region containing the same sample at the same coordinate in the second synchronization scope. In other words, it is a sample granularity dependency.

**Note**

Since fragment shader invocations are not specified to run in any particular groupings, the size of a framebuffer region is implementation-dependent, not known to the application, and must be assumed to be no larger than specified above.

**Note**

Practically, the pixel vs. sample granularity dependency means that if an input attachment has a different number of samples than the pipeline’s `rasterizationSamples`, then a fragment can access any sample in the input attachment’s pixel even if it only uses framebuffer-local dependencies. If the input attachment has the same number of samples, then the fragment can only access the covered samples in its input `SampleMask` (i.e. the fragment operations happen-after a framebuffer-local dependency for each sample the fragment covers). To access samples that are not covered, a framebuffer-global dependency is required.

If a synchronization command includes a `dependencyFlags` parameter, and specifies the `VK_DEPENDENCY_BY_REGION_BIT` flag, then it defines framebuffer-local dependencies for the framebuffer-space pipeline stages in that synchronization command, for all framebuffer regions. If no `dependencyFlags` parameter is included, or the `VK_DEPENDENCY_BY_REGION_BIT` flag is not specified, then a framebuffer-global dependency is specified for those stages. The `VK_DEPENDENCY_BY_REGION_BIT` flag does not affect the dependencies between non-framebuffer-space pipeline stages, nor does it affect the dependencies between framebuffer-space and non-framebuffer-space pipeline stages.

**Note**

Framebuffer-local dependencies are more efficient for most architectures; particularly tile-based architectures - which can keep framebuffer-regions entirely in on-chip registers and thus avoid external bandwidth across such a dependency. Including a framebuffer-global dependency in your rendering will usually force all implementations to flush data to memory, or to a higher level cache, breaking any potential locality optimizations.
7.1.5. View-Local Dependencies

In a render pass instance that has multiview enabled, dependencies can be either view-local or view-global.

A view-local dependency only includes operations from a single source view from the source subpass in the first synchronization scope, and only includes operations from a single destination view from the destination subpass in the second synchronization scope. A view-global dependency includes all views in the view mask of the source and destination subpasses in the corresponding synchronization scopes.

If a synchronization command includes a dependencyFlags parameter and specifies the VK_DEPENDENCY_VIEW_LOCAL_BIT flag, then it defines view-local dependencies for that synchronization command, for all views. If no dependencyFlags parameter is included or the VK_DEPENDENCY_VIEW_LOCAL_BIT flag is not specified, then a view-global dependency is specified.

7.1.6. Device-Local Dependencies

Dependencies can be either device-local or non-device-local. A device-local dependency acts as multiple separate dependencies, one for each physical device that executes the synchronization command, where each dependency only includes operations from that physical device in both synchronization scopes. A non-device-local dependency is a single dependency where both synchronization scopes include operations from all physical devices that participate in the synchronization command. For subpass dependencies, all physical devices in the VkDeviceGroupRenderPassBeginInfo::deviceMask participate in the dependency, and for pipeline barriers all physical devices that are set in the command buffer's current device mask participate in the dependency.

If a synchronization command includes a dependencyFlags parameter and specifies the VK_DEPENDENCY_DEVICE_GROUP_BIT flag, then it defines a non-device-local dependency for that synchronization command. If no dependencyFlags parameter is included or the VK_DEPENDENCY_DEVICE_GROUP_BIT flag is not specified, then it defines device-local dependencies for that synchronization command, for all participating physical devices.

Semaphore and event dependencies are device-local and only execute on the one physical device that performs the dependency.

7.2. Implicit Synchronization Guarantees

A small number of implicit ordering guarantees are provided by Vulkan, ensuring that the order in which commands are submitted is meaningful, and avoiding unnecessary complexity in common operations.

Submission order is a fundamental ordering in Vulkan, giving meaning to the order in which action and synchronization commands are recorded and submitted to a single queue. Explicit and implicit ordering guarantees between commands in Vulkan all work on the premise that this ordering is meaningful. This order does not itself define any execution or memory dependencies; synchronization commands and other orderings within the API use this ordering to define their scopes.
Submission order for any given set of commands is based on the order in which they were recorded to command buffers and then submitted. This order is determined as follows:

1. The initial order is determined by the order in which `vkQueueSubmit` and `vkQueueSubmit2` commands are executed on the host, for a single queue, from first to last.

2. The order in which `VkSubmitInfo` structures are specified in the `pSubmits` parameter of `vkQueueSubmit`, or in which `VkSubmitInfo2` structures are specified in the `pSubmits` parameter of `vkQueueSubmit2`, from lowest index to highest.

3. The order in which command buffers are specified in the `pCommandBuffers` member of `VkSubmitInfo` or `VkSubmitInfo2` from lowest index to highest.

4. The order in which commands were recorded to a command buffer on the host, from first to last:
   - For commands recorded outside a render pass, this includes all other commands recorded outside a render pass, including `vkCmdBeginRenderPass` and `vkCmdEndRenderPass` commands; it does not directly include commands inside a render pass.
   - For commands recorded inside a render pass, this includes all other commands recorded inside the same subpass, including the `vkCmdBeginRenderPass` and `vkCmdEndRenderPass` commands that delimit the same render pass instance; it does not include commands recorded to other subpasses. State commands do not execute any operations on the device, instead they set the state of the command buffer when they execute on the host, in the order that they are recorded. Action commands consume the current state of the command buffer when they are recorded, and will execute state changes on the device as required to match the recorded state.

The order of primitives passing through the graphics pipeline and image layout transitions as part of an image memory barrier provide additional guarantees based on submission order.

Execution of pipeline stages within a given command also has a loose ordering, dependent only on a single command.

Signal operation order is a fundamental ordering in Vulkan, giving meaning to the order in which semaphore and fence signal operations occur when submitted to a single queue. The signal operation order for queue operations is determined as follows:

1. The initial order is determined by the order in which `vkQueueSubmit` and `vkQueueSubmit2` commands are executed on the host, for a single queue, from first to last.

2. The order in which `VkSubmitInfo` structures are specified in the `pSubmits` parameter of `vkQueueSubmit`, or in which `VkSubmitInfo2` structures are specified in the `pSubmits` parameter of `vkQueueSubmit2`, from lowest index to highest.

3. The fence signal operation defined by the `fence` parameter of a `vkQueueSubmit`, `vkQueueSubmit2`, or `vkQueueBindSparse` command is ordered after all semaphore signal operations defined by that command.

Semaphore signal operations defined by a single `VkSubmitInfo`, `VkSubmitInfo2`, or `VkBindSparseInfo` structure are unordered with respect to other semaphore signal operations defined within the same structure.
The `vkSignalSemaphore` command does not execute on a queue but instead performs the signal operation from the host. The semaphore signal operation defined by executing a `vkSignalSemaphore` command happens-after the `vkSignalSemaphore` command is invoked and happens-before the command returns.

**Note**

When signaling timeline semaphores, it is the responsibility of the application to ensure that they are ordered such that the semaphore value is strictly increasing. Because the first synchronization scope for a semaphore signal operation contains all semaphore signal operations which occur earlier in submission order, all semaphore signal operations contained in any given batch are guaranteed to happen-after all semaphore signal operations contained in any previous batches. However, no ordering guarantee is provided between the semaphore signal operations defined within a single batch. This, combined with the requirement that timeline semaphore values strictly increase, means that it is invalid to signal the same timeline semaphore twice within a single batch.

If an application wishes to ensure that some semaphore signal operation happens-after some other semaphore signal operation, it can submit a separate batch containing only semaphore signal operations, which will happen-after the semaphore signal operations in any earlier batches.

When signaling a semaphore from the host, the only ordering guarantee is that the signal operation happens-after when `vkSignalSemaphore` is called and happens-before it returns. Therefore, it is invalid to call `vkSignalSemaphore` while there are any outstanding signal operations on that semaphore from any queue submissions unless those queue submissions have some dependency which ensures that they happen-after the host signal operation. One example of this would be if the pending signal operation is, itself, waiting on the same semaphore at a lower value and the call to `vkSignalSemaphore` signals that lower value. Furthermore, if there are two or more processes or threads signaling the same timeline semaphore from the host, the application must ensure that the `vkSignalSemaphore` with the lower semaphore value returns before `vkSignalSemaphore` is called with the higher value.

### 7.3. Fences

Fences are a synchronization primitive that **can** be used to insert a dependency from a queue to the host. Fences have two states - signaled and unsignaled. A fence **can** be signaled as part of the execution of a `queue submission` command. Fences **can** be unsignaled on the host with `vkResetFences`. Fences **can** be waited on by the host with the `vkWaitForFences` command, and the current state **can** be queried with `vkGetFenceStatus`.

The internal data of a fence **may** include a reference to any resources and pending work associated with signal or unsignal operations performed on that fence object, collectively referred to as the fence’s *payload*. Mechanisms to import and export that internal data to and from fences are provided **below**. These mechanisms indirectly enable applications to share fence state between two or more fences and other synchronization primitives across process and API boundaries.
Fences are represented by **VkFence** handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkFence)
```

To create a fence, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateFence(
    VkDevice device,
    const VkFenceCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkFence* pFence);
```

- **device** is the logical device that creates the fence.
- **pCreateInfo** is a pointer to a **VkFenceCreateInfo** structure containing information about how the fence is to be created.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
- **pFence** is a pointer to a handle in which the resulting fence object is returned.

---

### Valid Usage (Implicit)

- VUID-vkCreateFence-device-parameter
  - **device** must be a valid **VkDevice** handle
- VUID-vkCreateFence-pCreateInfo-parameter
  - **pCreateInfo** must be a valid pointer to a valid **VkFenceCreateInfo** structure
- VUID-vkCreateFence-pAllocator-parameter
  - If **pAllocator** is not **NULL**, **pAllocator** must be a valid pointer to a valid **VkAllocationCallbacks** structure
- VUID-vkCreateFence-pFence-parameter
  - **pFence** must be a valid pointer to a **VkFence** handle

---

### Return Codes

**Success**
- **VK_SUCCESS**

**Failure**
- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**

The **VkFenceCreateInfo** structure is defined as:
typedef struct VkFenceCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkFenceCreateFlags flags;
} VkFenceCreateInfo;

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **flags** is a bitmask of `VkFenceCreateFlagBits` specifying the initial state and behavior of the fence.

### Valid Usage (Implicit)

- VUID-VkFenceCreateInfo-sType-sType
  sType **must** be `VK_STRUCTURE_TYPE_FENCE_CREATE_INFO`
- VUID-VkFenceCreateInfo-pNext-pNext
  pNext **must** be `NULL` or a pointer to a valid instance of `VkExportFenceCreateInfo`
- VUID-VkFenceCreateInfo-sType-unique
  The sType value of each struct in the pNext chain **must** be unique
- VUID-VkFenceCreateInfo-flags-parameter
  flags **must** be a valid combination of `VkFenceCreateFlagBits` values

// Provided by VK_VERSION_1_0
typedef enum VkFenceCreateFlagBits {
    VK_FENCE_CREATE_SIGNALED_BIT = 0x00000001,
} VkFenceCreateFlagBits;

- **VK_FENCE_CREATE_SIGNALED_BIT** specifies that the fence object is created in the signaled state. Otherwise, it is created in the unsignaled state.

// Provided by VK_VERSION_1_0
typedef VkFlags VkFenceCreateFlags;

**VkFenceCreateFlags** is a bitmask type for setting a mask of zero or more `VkFenceCreateFlagBits`.

To create a fence whose payload **can** be exported to external handles, add a `VkExportFenceCreateInfo` structure to the `pNext` chain of the `VkFenceCreateInfo` structure. The `VkExportFenceCreateInfo` structure is defined as:

// Provided by VK_VERSION_1_1
typedef struct VkExportFenceCreateInfo {

VkExportFenceCreateInfo

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **handleTypes** is a bitmask of VkExternalFenceHandleTypeFlagBits specifying one or more fence handle types the application can export from the resulting fence. The application can request multiple handle types for the same fence.

### Valid Usage

- VUID-VkExportFenceCreateInfo-handleTypes-01446
  The bits in **handleTypes** must be supported and compatible, as reported by VkExternalFenceProperties

### Valid Usage (Implicit)

- VUID-VkExportFenceCreateInfo-sType-sType
  **sType** must be VK_STRUCTURE_TYPE_EXPORT_FENCE_CREATE_INFO

- VUID-VkExportFenceCreateInfo-handleTypes-parameter
  **handleTypes** must be a valid combination of VkExternalFenceHandleTypeFlagBits values

To destroy a fence, call:

```c
// Provided by VK_VERSION_1_0
define vkDestroyFence

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

- **device** is the logical device that destroys the fence.
- **fence** is the handle of the fence to destroy.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage

- VUID-vkDestroyFence-fence-01120
  All queue submission commands that refer to **fence** must have completed execution

- VUID-vkDestroyFence-fence-01121
  If **VkAllocationCallbacks** were provided when **fence** was created, a compatible set of
callbacks must be provided here

• VUID-vkDestroyFence-fence-01122
  If no VkAllocationCallbacks were provided when fence was created, pAllocator must be NULL.

Valid Usage (Implicit)

• VUID-vkDestroyFence-device-parameter
device must be a valid VkDevice handle

• VUID-vkDestroyFence-fence-parameter
  If fence is not VK_NULL_HANDLE, fence must be a valid VkFence handle

• VUID-vkDestroyFence-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

• VUID-vkDestroyFence-fence-parent
  If fence is a valid handle, it must have been created, allocated, or retrieved from device

Host Synchronization

• Host access to fence must be externally synchronized

To query the status of a fence from the host, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkGetFenceStatus(                      
    VkDevice device,                            
    VkFence fence);                             
```

• device is the logical device that owns the fence.
• fence is the handle of the fence to query.

Upon success, vkGetFenceStatus returns the status of the fence object, with the following return codes:

Table 5. Fence Object Status Codes

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_SUCCESS</td>
<td>The fence specified by fence is signaled.</td>
</tr>
<tr>
<td>VK_NOT_READY</td>
<td>The fence specified by fence is unsignaled.</td>
</tr>
</tbody>
</table>
### Status

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_ERROR_DEVICE_LOST</td>
<td>The device has been lost. See Lost Device.</td>
</tr>
</tbody>
</table>

If a queue submission command is pending execution, then the value returned by this command may immediately be out of date.

If the device has been lost (see Lost Device), vkGetFenceStatus may return any of the above status codes. If the device has been lost and vkGetFenceStatus is called repeatedly, it will eventually return either VK_SUCCESS or VK_ERROR_DEVICE_LOST.

#### Valid Usage (Implicit)

- VUID-vkGetFenceStatus-device-parameter
  - device must be a valid VkDevice handle
- VUID-vkGetFenceStatus-fence-parameter
  - fence must be a valid VkFence handle
- VUID-vkGetFenceStatus-fence-parent
  - fence must have been created, allocated, or retrieved from device

#### Return Codes

**Success**
- VK_SUCCESS
- VK_NOT_READY

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST

To set the state of fences to unsignaled from the host, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkResetFences(
    VkDevice device,          // device
    uint32_t fenceCount,      // fenceCount
    const VkFence* pFences);  // pFences
```

- device is the logical device that owns the fences.
- fenceCount is the number of fences to reset.
- pFences is a pointer to an array of fence handles to reset.
If any member of $pFences$ currently has its payload imported with temporary permanence, that fence’s prior permanent payload is first restored. The remaining operations described therefore operate on the restored payload.

When $vkResetFences$ is executed on the host, it defines a fence unsignal operation for each fence, which resets the fence to the unsignaled state.

If any member of $pFences$ is already in the unsignaled state when $vkResetFences$ is executed, then $vkResetFences$ has no effect on that fence.

**Valid Usage**

- VUID-vkResetFences-pFences-01123
  Each element of $pFences$ must not be currently associated with any queue command that has not yet completed execution on that queue

**Valid Usage (Implicit)**

- VUID-vkResetFences-device-parameter
device must be a valid VkDevice handle
- VUID-vkResetFences-pFences-parameter
  $pFences$ must be a valid pointer to an array of $fenceCount$ valid VkFence handles
- VUID-vkResetFences-fenceCount-arraylength
  $fenceCount$ must be greater than $0$
- VUID-vkResetFences-pFences-parent
  Each element of $pFences$ must have been created, allocated, or retrieved from device

**Host Synchronization**

- Host access to each member of $pFences$ must be externally synchronized

**Return Codes**

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_DEVICE_MEMORY

When a fence is submitted to a queue as part of a queue submission command, it defines a memory dependency on the batches that were submitted as part of that command, and defines a fence signal operation which sets the fence to the signaled state.
The first synchronization scope includes every batch submitted in the same queue submission command. Fence signal operations that are defined by `vkQueueSubmit` or `vkQueueSubmit2` additionally include in the first synchronization scope all commands that occur earlier in submission order. Fence signal operations that are defined by `vkQueueSubmit`, `vkQueueSubmit2` or `vkQueueBindSparse` additionally include in the first synchronization scope any semaphore and fence signal operations that occur earlier in signal operation order.

The second synchronization scope only includes the fence signal operation.

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

The second access scope is empty.

To wait for one or more fences to enter the signaled state on the host, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkWaitForFences(
    VkDevice device,
    uint32_t fenceCount,
    const VkFence* pFences,
    VkBool32 waitAll,
    uint64_t timeout);
```

- `device` is the logical device that owns the fences.
- `fenceCount` is the number of fences to wait on.
- `pFences` is a pointer to an array of `fenceCount` fence handles.
- `waitAll` is the condition that must be satisfied to successfully unblock the wait. If `waitAll` is `VK_TRUE`, then the condition is that all fences in `pFences` are signaled. Otherwise, the condition is that at least one fence in `pFences` is signaled.
- `timeout` is the timeout period in units of nanoseconds. `timeout` is adjusted to the closest value allowed by the implementation-dependent timeout accuracy, which may be substantially longer than one nanosecond, and may be longer than the requested period.

If the condition is satisfied when `vkWaitForFences` is called, then `vkWaitForFences` returns immediately. If the condition is not satisfied at the time `vkWaitForFences` is called, then `vkWaitForFences` will block and wait until the condition is satisfied or the `timeout` has expired, whichever is sooner.

If `timeout` is zero, then `vkWaitForFences` does not wait, but simply returns the current state of the fences. `VK_TIMEOUT` will be returned in this case if the condition is not satisfied, even though no actual wait was performed.

If the condition is satisfied before the `timeout` has expired, `vkWaitForFences` returns `VK_SUCCESS`. Otherwise, `vkWaitForFences` returns `VK_TIMEOUT` after the `timeout` has expired.

If device loss occurs (see Lost Device) before the timeout has expired, `vkWaitForFences` must return in finite time with either `VK_SUCCESS` or `VK_ERROR_DEVICE_LOST`. 
Note

While we guarantee that `vkWaitForFences` must return in finite time, no guarantees are made that it returns immediately upon device loss. However, the client can reasonably expect that the delay will be on the order of seconds and that calling `vkWaitForFences` will not result in a permanently (or seemingly permanently) dead process.

Valid Usage (Implicit)

- **VUID-vkWaitForFences-device-parameter**
  
  `device` must be a valid `VkDevice` handle

- **VUID-vkWaitForFences-pFences-parameter**
  
  `pFences` must be a valid pointer to an array of `fenceCount` valid `VkFence` handles

- **VUID-vkWaitForFences-fenceCount-arraylength**
  
  `fenceCount` must be greater than 0

- **VUID-vkWaitForFences-pFences-parent**
  
  Each element of `pFences` must have been created, allocated, or retrieved from `device`

Return Codes

**Success**

- `VK_SUCCESS`
- `VK_TIMEOUT`

**Failure**

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

An execution dependency is defined by waiting for a fence to become signaled, either via `vkWaitForFences` or by polling on `vkGetFenceStatus`.

The first synchronization scope includes only the fence signal operation.

The second synchronization scope includes the host operations of `vkWaitForFences` or `vkGetFenceStatus` indicating that the fence has become signaled.

Note

Signaling a fence and waiting on the host does not guarantee that the results of memory accesses will be visible to the host, as the access scope of a memory dependency defined by a fence only includes device access. A memory barrier or other memory dependency must be used to guarantee this. See the description of host access types for more information.
7.3.1. Importing Fence Payloads

Applications can import a fence payload into an existing fence using an external fence handle. The effects of the import operation will be either temporary or permanent, as specified by the application. If the import is temporary, the fence will be restored to its permanent state the next time that fence is passed to `vkResetFences`.

**Note**

Restoring a fence to its prior permanent payload is a distinct operation from resetting a fence payload. See `vkResetFences` for more detail.

Performing a subsequent temporary import on a fence before resetting it has no effect on this requirement; the next unsignal of the fence must still restore its last permanent state. A permanent payload import behaves as if the target fence was destroyed, and a new fence was created with the same handle but the imported payload. Because importing a fence payload temporarily or permanently detaches the existing payload from a fence, similar usage restrictions to those applied to `vkDestroyFence` are applied to any command that imports a fence payload. Which of these import types is used is referred to as the import operation's permanence. Each handle type supports either one or both types of permanence.

The implementation must perform the import operation by either referencing or copying the payload referred to by the specified external fence handle, depending on the handle's type. The import method used is referred to as the handle type's transference. When using handle types with reference transference, importing a payload to a fence adds the fence to the set of all fences sharing that payload. This set includes the fence from which the payload was exported. Fence signaling, waiting, and resetting operations performed on any fence in the set must behave as if the set were a single fence. Importing a payload using handle types with copy transference creates a duplicate copy of the payload at the time of import, but makes no further reference to it. Fence signaling, waiting, and resetting operations performed on the target of copy imports must not affect any other fence or payload.

Export operations have the same transference as the specified handle type's import operations. Additionally, exporting a fence payload to a handle with copy transference has the same side effects on the source fence's payload as executing a fence reset operation. If the fence was using a temporarily imported payload, the fence's prior permanent payload will be restored.

**External synchronization** allows implementations to modify an object's internal state, i.e. payload, without internal synchronization. However, for fences sharing a payload across processes, satisfying the external synchronization requirements of `VkFence` parameters as if all fences in the set were the same object is sometimes infeasible. Satisfying valid usage constraints on the state of a fence would similarly require impractical coordination or levels of trust between processes. Therefore, these constraints only apply to a specific fence handle, not to its payload. For distinct fence objects which share a payload:

- If multiple commands which queue a signal operation, or which unsignal a fence, are called concurrently, behavior will be as if the commands were called in an arbitrary sequential order.
- If a queue submission command is called with a fence that is sharing a payload, and the payload is already associated with another queue command that has not yet completed execution, either
one or both of the commands will cause the fence to become signaled when they complete execution.

- If a fence payload is reset while it is associated with a queue command that has not yet completed execution, the payload will become unsignaled, but may become signaled again when the command completes execution.

- In the preceding cases, any of the devices associated with the fences sharing the payload may be lost, or any of the queue submission or fence reset commands may return VK_ERROR_INITIALIZATION_FAILED.

Other than these non-deterministic results, behavior is well defined. In particular:

- The implementation must not crash or enter an internally inconsistent state where future valid Vulkan commands might cause undefined results,

- Timeouts on future wait commands on fences sharing the payload must be effective.

Note

These rules allow processes to synchronize access to shared memory without trusting each other. However, such processes must still be cautious not to use the shared fence for more than synchronizing access to the shared memory. For example, a process should not use a fence with shared payload to tell when commands it submitted to a queue have completed and objects used by those commands may be destroyed, since the other process can accidentally or maliciously cause the fence to signal before the commands actually complete.

When a fence is using an imported payload, its VkExportFenceCreateInfo::handleTypes value is specified when creating the fence from which the payload was exported, rather than specified when creating the fence. Additionally, VkExternalFenceProperties::exportFromImportedHandleTypes restricts which handle types can be exported from such a fence based on the specific handle type used to import the current payload.

When importing a fence payload, it is the responsibility of the application to ensure the external handles meet all valid usage requirements. However, implementations must perform sufficient validation of external handles to ensure that the operation results in a valid fence which will not cause program termination, device loss, queue stalls, host thread stalls, or corruption of other resources when used as allowed according to its import parameters. If the external handle provided does not meet these requirements, the implementation must fail the fence payload import operation with the error code VK_ERROR_INVALID_EXTERNAL_HANDLE.

7.4. Semaphores

Semaphores are a synchronization primitive that can be used to insert a dependency between queue operations or between a queue operation and the host. Binary semaphores have two states—signaled and unsignaled. Timeline semaphores have a strictly increasing 64-bit unsigned integer payload and are signaled with respect to a particular reference value. A semaphore can be signaled after execution of a queue operation is completed, and a queue operation can wait for a semaphore to become signaled before it begins execution. A timeline semaphore can additionally be signaled from the host with the vkSignalSemaphore command and waited on from the host with the
The internal data of a semaphore may include a reference to any resources and pending work associated with signal or unsignal operations performed on that semaphore object, collectively referred to as the semaphore's payload. Mechanisms to import and export that internal data to and from semaphores are provided below. These mechanisms indirectly enable applications to share semaphore state between two or more semaphores and other synchronization primitives across process and API boundaries.

Semaphores are represented by VkSemaphore handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSemaphore)
```

To create a semaphore, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateSemaphore(VkDevice device,
	const VkSemaphoreCreateInfo* pCreateInfo,
	const VkAllocationCallbacks* pAllocator,
	VkSemaphore* pSemaphore);
```

- `device` is the logical device that creates the semaphore.
- `pCreateInfo` is a pointer to a VkSemaphoreCreateInfo structure containing information about how the semaphore is to be created.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pSemaphore` is a pointer to a handle in which the resulting semaphore object is returned.

### Valid Usage (Implicit)
- VUID-vkCreateSemaphore-device-parameter
  - `device` must be a valid VkDevice handle
- VUID-vkCreateSemaphore-pCreateInfo-parameter
  - `pCreateInfo` must be a valid pointer to a valid VkSemaphoreCreateInfo structure
- VUID-vkCreateSemaphore-pAllocator-parameter
  - If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid VkAllocationCallbacks structure
- VUID-vkCreateSemaphore-pSemaphore-parameter
  - `pSemaphore` must be a valid pointer to a VkSemaphore handle
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkSemaphoreCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSemaphoreCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkSemaphoreCreateFlags flags;
} VkSemaphoreCreateInfo;
```

• `sType` is the type of this structure.
• `pNext` is `NULL` or a pointer to a structure extending this structure.
• `flags` is reserved for future use.

Valid Usage (Implicit)

• VUID-VkSemaphoreCreateInfo-sType-sType `sType` must be VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO

• VUID-VkSemaphoreCreateInfo-pNext-pNext Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of VkExportSemaphoreCreateInfo or VkSemaphoreTypeCreateInfo

• VUID-VkSemaphoreCreateInfo-sType-unique The `sType` value of each struct in the `pNext` chain must be unique

• VUID-VkSemaphoreCreateInfo-flags-zerobitmask `flags` must be 0

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkSemaphoreCreateFlags;
```

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

The VkSemaphoreTypeCreateInfo structure is defined as:
typedef struct VkSemaphoreTypeCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkSemaphoreType semaphoreType;
    uint64_t initialValue;
} VkSemaphoreTypeCreateInfo;

• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **semaphoreType** is a VkSemaphoreType value specifying the type of the semaphore.
• **initialValue** is the initial payload value if semaphoreType is VK_SEMAPHORE_TYPE_TIMELINE.

To create a semaphore of a specific type, add a VkSemaphoreTypeCreateInfo structure to the VkSemaphoreCreateInfo::pNext chain.

If no VkSemaphoreTypeCreateInfo structure is included in the pNext chain of VkSemaphoreCreateInfo, then the created semaphore will have a default VkSemaphoreType of VK_SEMAPHORE_TYPE_BINARY.

---

### Valid Usage

- **VUID-VkSemaphoreTypeCreateInfo-timelineSemaphore-03252**
  If the timelineSemaphore feature is not enabled, semaphoreType must not equal VK_SEMAPHORE_TYPE_TIMELINE

- **VUID-VkSemaphoreTypeCreateInfo-semaphoreType-03279**
  If semaphoreType is VK_SEMAPHORE_TYPE_BINARY, initialValue must be zero

### Valid Usage (Implicit)

- **VUID-VkSemaphoreTypeCreateInfo-sType-sType**
  sType must be VK_STRUCTURE_TYPE_SEMAPHORE_TYPE_CREATE_INFO

- **VUID-VkSemaphoreTypeCreateInfo-semaphoreType-parameter semaphoreType must be a valid VkSemaphoreType value**

Possible values of VkSemaphoreTypeCreateInfo::semaphoreType, specifying the type of a semaphore, are:

typedef enum VkSemaphoreType {
    VK_SEMAPHORE_TYPE_BINARY = 0,
    VK_SEMAPHORE_TYPE_TIMELINE = 1,
} VkSemaphoreType;
• **VK_SEMAPHORE_TYPE_BINARY** specifies a *binary semaphore* type that has a boolean payload indicating whether the semaphore is currently signaled or unsignaled. When created, the semaphore is in the unsignaled state.

• **VK_SEMAPHORE_TYPE_TIMELINE** specifies a *timeline semaphore* type that has a strictly increasing 64-bit unsigned integer payload indicating whether the semaphore is signaled with respect to a particular reference value. When created, the semaphore payload has the value given by the `initialValue` field of `VkSemaphoreTypeCreateInfo`.

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

```c
// Provided by VK_VERSION_1_1
typedef struct VkExportSemaphoreCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkExternalSemaphoreHandleTypeFlags handleTypes;
} VkExportSemaphoreCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `handleTypes` is a bitmask of `VkExternalSemaphoreHandleTypeFlagBits` specifying one or more semaphore handle types the application can export from the resulting semaphore. The application can request multiple handle types for the same semaphore.

**Valid Usage**

- VUID-VkExportSemaphoreCreateInfo-handleTypes-01124
  The bits in `handleTypes` must be supported and compatible, as reported by `VkExternalSemaphoreProperties`

**Valid Usage (Implicit)**

- VUID-VkExportSemaphoreCreateInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO`

- VUID-VkExportSemaphoreCreateInfo-handleTypes-parameter
  `handleTypes` must be a valid combination of `VkExternalSemaphoreHandleTypeFlagBits` values

To destroy a semaphore, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroySemaphore(
    VkDevice device,
```
VkSemaphore

const VkAllocationCallbacks*

pAllocator);

• device is the logical device that destroys the semaphore.
• semaphore is the handle of the semaphore to destroy.
• pAllocator controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

• VUID-vkDestroySemaphore-semaphore-01137
  All submitted batches that refer to semaphore must have completed execution
• VUID-vkDestroySemaphore-semaphore-01138
  If VkAllocationCallbacks were provided when semaphore was created, a compatible set of callbacks must be provided here
• VUID-vkDestroySemaphore-semaphore-01139
  If no VkAllocationCallbacks were provided when semaphore was created, pAllocator must be NULL

Valid Usage (Implicit)

• VUID-vkDestroySemaphore-device-parameter
device must be a valid VkDevice handle
• VUID-vkDestroySemaphore-semaphore-parameter
  If semaphore is not VK_NULL_HANDLE, semaphore must be a valid VkSemaphore handle
• VUID-vkDestroySemaphore-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure
• VUID-vkDestroySemaphore-semaphore-parent
  If semaphore is a valid handle, it must have been created, allocated, or retrieved from device

Host Synchronization

• Host access to semaphore must be externally synchronized

7.4.1. Semaphore Signaling

When a batch is submitted to a queue via a queue submission, and it includes semaphores to be signaled, it defines a memory dependency on the batch, and defines semaphore signal operations which set the semaphores to the signaled state.

In case of semaphores created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_TIMELINE the
Semaphore is considered signaled with respect to the counter value set to be signaled as specified in `VkTimelineSemaphoreSubmitInfo` or `VkSemaphoreSignalInfo`.

The first synchronization scope includes every command submitted in the same batch. In the case of `vkQueueSubmit2`, the first synchronization scope is limited to the pipeline stage specified by `VkSemaphoreSubmitInfo::stageMask`. Semaphore signal operations that are defined by `vkQueueSubmit` or `vkQueueSubmit2` additionally include all commands that occur earlier in submission order. Semaphore signal operations that are defined by `vkQueueSubmit`, `vkQueueSubmit2` or `vkQueueBindSparse` additionally include in the first synchronization scope any semaphore and fence signal operations that occur earlier in signal operation order.

The second synchronization scope includes only the semaphore signal operation.

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

The second access scope is empty.

**7.4.2. Semaphore Waiting**

When a batch is submitted to a queue via a queue submission, and it includes semaphores to be waited on, it defines a memory dependency between prior semaphore signal operations and the batch, and defines semaphore wait operations.

Such semaphore wait operations set the semaphores created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_BINARY` to the unsignaled state. In case of semaphores created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` a prior semaphore signal operation defines a memory dependency with a semaphore wait operation if the value the semaphore is signaled with is greater than or equal to the value the semaphore is waited with, thus the semaphore will continue to be considered signaled with respect to the counter value waited on as specified in `VkTimelineSemaphoreSubmitInfo`.

The first synchronization scope includes all semaphore signal operations that operate on semaphores waited on in the same batch, and that happen-before the wait completes.

The second synchronization scope includes every command submitted in the same batch. In the case of `vkQueueSubmit`, the second synchronization scope is limited to operations on the pipeline stages determined by the destination stage mask specified by the corresponding element of `pWaitDstStageMask`. In the case of `vkQueueSubmit2`, the second synchronization scope is limited to the pipeline stage specified by `VkSemaphoreSubmitInfo::stageMask`. Also, in the case of either `vkQueueSubmit2` or `vkQueueSubmit`, the second synchronization scope additionally includes all commands that occur later in submission order.

The first access scope is empty.

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

The semaphore wait operation happens-after the first set of operations in the execution dependency, and happens-before the second set of operations in the execution dependency.
Unlike timeline semaphores, fences or events, the act of waiting for a binary semaphore also unsignals that semaphore. Applications must ensure that between two such wait operations, the semaphore is signaled again, with execution dependencies used to ensure these occur in order. Binary semaphore waits and signals should thus occur in discrete 1:1 pairs.

### 7.4.3. Semaphore State Requirements For Wait Operations

Before waiting on a semaphore, the application must ensure the semaphore is in a valid state for a wait operation. Specifically, when a semaphore wait operation is submitted to a queue:

- A binary semaphore must be signaled, or have an associated semaphore signal operation that is pending execution.
- Any semaphore signal operations on which the pending binary semaphore signal operation depends must also be completed or pending execution.
- There must be no other queue waiting on the same binary semaphore when the operation executes.

### 7.4.4. Host Operations on Semaphores

In addition to semaphore signal operations and semaphore wait operations submitted to device queues, timeline semaphores support the following host operations:

- Query the current counter value of the semaphore using the `vkGetSemaphoreCounterValue` command.
- Wait for a set of semaphores to reach particular counter values using the `vkWaitSemaphores` command.
- Signal the semaphore with a particular counter value from the host using the `vkSignalSemaphore` command.

To query the current counter value of a semaphore created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` from the host, call:

```c
// Provided by VK_VERSION_1_2
VkResult vkGetSemaphoreCounterValue(VkDevice device, VkSemaphore semaphore, uint64_t* pValue);
```

- `device` is the logical device that owns the semaphore.
- `semaphore` is the handle of the semaphore to query.
- `pValue` is a pointer to a 64-bit integer value in which the current counter value of the semaphore is returned.

### Note

...
If a *queue submission* command is pending execution, then the value returned by this command *may* immediately be out of date.

### Valid Usage

- VUID-vkGetSemaphoreCounterValue-semaphore-03255
  
  `semaphore` *must* have been created with a *VkSemaphoreType* of `VK_SEMAPHORE_TYPE_TIMELINE`

### Valid Usage (Implicit)

- VUID-vkGetSemaphoreCounterValue-device-parameter
  
  `device` *must* be a valid *VkDevice* handle

- VUID-vkGetSemaphoreCounterValue-semaphore-parameter
  
  `semaphore` *must* be a valid *VkSemaphore* handle

- VUID-vkGetSemaphoreCounterValue-pValue-parameter
  
  `pValue` *must* be a valid pointer to a `uint64_t` value

- VUID-vkGetSemaphoreCounterValue-semaphore-parent
  
  `semaphore` *must* have been created, allocated, or retrieved from `device`

### Return Codes

#### Success

- `VK_SUCCESS`

#### Failure

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

To wait for a set of semaphores created with a *VkSemaphoreType* of `VK_SEMAPHORE_TYPE_TIMELINE` to reach particular counter values on the host, call:

```c
// Provided by VK_VERSION_1_2
VkResult vkWaitSemaphores(
    VkDevice device,
    const VkSemaphoreWaitInfo* pWaitInfo,
    uint64_t timeout);
```

- `device` is the logical device that owns the semaphores.
- `pWaitInfo` is a pointer to a *VkSemaphoreWaitInfo* structure containing information about the wait condition.
timeout is the timeout period in units of nanoseconds. timeout is adjusted to the closest value allowed by the implementation-dependent timeout accuracy, which may be substantially longer than one nanosecond, and may be longer than the requested period.

If the condition is satisfied when vkWaitSemaphores is called, then vkWaitSemaphores returns immediately. If the condition is not satisfied at the time vkWaitSemaphores is called, then vkWaitSemaphores will block and wait until the condition is satisfied or the timeout has expired, whichever is sooner.

If timeout is zero, then vkWaitSemaphores does not wait, but simply returns information about the current state of the semaphores. VK_TIMEOUT will be returned in this case if the condition is not satisfied, even though no actual wait was performed.

If the condition is satisfied before the timeout has expired, vkWaitSemaphores returns VK_SUCCESS. Otherwise, vkWaitSemaphores returns VK_TIMEOUT after the timeout has expired.

If device loss occurs (see Lost Device) before the timeout has expired, vkWaitSemaphores must return in finite time with either VK_SUCCESS or VK_ERROR_DEVICE_LOST.

Valid Usage (Implicit)

- VUID-vkWaitSemaphores-device-parameter
  device must be a valid VkDevice handle
- VUID-vkWaitSemaphores-pWaitInfo-parameter
  pWaitInfo must be a valid pointer to a valid VkSemaphoreWaitInfo structure

Return Codes

Success
- VK_SUCCESS
- VK_TIMEOUT

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST

The VkSemaphoreWaitInfo structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkSemaphoreWaitInfo {
    VkStructureType sType;
    const void* pNext;
    VkSemaphoreWaitFlags flags;
    uint32_t semaphoreCount;
} VkSemaphoreWaitInfo;
```
const VkSemaphore* pSemaphores;
const uint64_t* pValues;
} VkSemaphoreWaitInfo;

• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **flags** is a bitmask of **VkSemaphoreWaitFlagBits** specifying additional parameters for the semaphore wait operation.
• **semaphoreCount** is the number of semaphores to wait on.
• **pSemaphores** is a pointer to an array of **semaphoreCount** semaphore handles to wait on.
• **pValues** is a pointer to an array of **semaphoreCount** timeline semaphore values.

### Valid Usage

• VUID-VkSemaphoreWaitInfo-pSemaphores-03256
  All of the elements of **pSemaphores** **must** reference a semaphore that was created with a **VkSemaphoreType** of **VK_SEMAPHORE_TYPE_TIMELINE**

### Valid Usage (Implicit)

• VUID-VkSemaphoreWaitInfo-sType-sType
  **sType** **must** be **VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO**

• VUID-VkSemaphoreWaitInfo-pNext-pNext
  **pNext** **must** be **NULL**

• VUID-VkSemaphoreWaitInfo-flags-parameter
  **flags** **must** be a valid combination of **VkSemaphoreWaitFlagBits** values

• VUID-VkSemaphoreWaitInfo-pSemaphores-parameter
  **pSemaphores** **must** be a valid pointer to an array of **semaphoreCount** valid **VkSemaphore** handles

• VUID-VkSemaphoreWaitInfo-pValues-parameter
  **pValues** **must** be a valid pointer to an array of **semaphoreCount** **uint64_t** values

• VUID-VkSemaphoreWaitInfo-semaphoreCount-arraylength
  **semaphoreCount** **must** be greater than **0**

Bits which **can** be set in **VkSemaphoreWaitInfo::flags**, specifying additional parameters of a semaphore wait operation, are:

```c
// Provided by VK_VERSION_1_2
typedef enum VkSemaphoreWaitFlagBits {
    VK_SEMAPHORE_WAIT_ANY_BIT = 0x00000001,
} VkSemaphoreWaitFlagBits;
```
• **VK_SEMAPHORE_WAIT_ANY_BIT** specifies that the semaphore wait condition is that at least one of the semaphores in `VkSemaphoreWaitInfo::pSemaphores` has reached the value specified by the corresponding element of `VkSemaphoreWaitInfo::pValues`. If **VK_SEMAPHORE_WAIT_ANY_BIT** is not set, the semaphore wait condition is that all of the semaphores in `VkSemaphoreWaitInfo::pSemaphores` have reached the value specified by the corresponding element of `VkSemaphoreWaitInfo::pValues`.

```c
// Provided by VK_VERSION_1_2
typedef VkFlags VkSemaphoreWaitFlags;
```

`VkSemaphoreWaitFlags` is a bitmask type for setting a mask of zero or more `VkSemaphoreWaitFlagBits`.

To signal a semaphore created with a `VkSemaphoreType` of **VK_SEMAPHORE_TYPE_TIMELINE** with a particular counter value, on the host, call:

```c
// Provided by VK_VERSION_1_2
VkResult vkSignalSemaphore(  
  VkDevice device,  
  const VkSemaphoreSignalInfo* pSignalInfo);
```

• **device** is the logical device that owns the semaphore.

• **pSignalInfo** is a pointer to a `VkSemaphoreSignalInfo` structure containing information about the signal operation.

When `vkSignalSemaphore` is executed on the host, it defines and immediately executes a *semaphore signal operation* which sets the timeline semaphore to the given value.

The first synchronization scope is defined by the host execution model, but includes execution of `vkSignalSemaphore` on the host and anything that happened-before it.

The second synchronization scope is empty.

### Valid Usage (Implicit)

- **VUID-vkSignalSemaphore-device-parameter**  
  **device** must be a valid `VkDevice` handle

- **VUID-vkSignalSemaphore-pSignalInfo-parameter**  
  **pSignalInfo** must be a valid pointer to a valid `VkSemaphoreSignalInfo` structure

### Return Codes

**Success**

- **VK_SUCCESS**
The `VkSemaphoreSignalInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkSemaphoreSignalInfo {
    VkStructureType sType;
    const void* pNext;
    VkSemaphore semaphore;
    uint64_t value;
} VkSemaphoreSignalInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `semaphore` is the handle of the semaphore to signal.
- `value` is the value to signal.

**Valid Usage**

- VUID-VkSemaphoreSignalInfo-semaphore-03257
  `semaphore` must have been created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE`

- VUID-VkSemaphoreSignalInfo-value-03258
  `value` must have a value greater than the current value of the semaphore

- VUID-VkSemaphoreSignalInfo-value-03259
  `value` must be less than the value of any pending semaphore signal operations

- VUID-VkSemaphoreSignalInfo-value-03260
  `value` must have a value which does not differ from the current value of the semaphore or the value of any outstanding semaphore wait or signal operation on `semaphore` by more than `maxTimelineSemaphoreValueDifference`

**Valid Usage (Implicit)**

- VUID-VkSemaphoreSignalInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_SEMAPHORE_SIGNAL_INFO`

- VUID-VkSemaphoreSignalInfo-pNext-pNext
  `pNext` must be `NULL`

- VUID-VkSemaphoreSignalInfo-semaphore-parameter
  `semaphore` must be a valid `VkSemaphore` handle
7.4.5. Importing Semaphore Payloads

Applications can import a semaphore payload into an existing semaphore using an external semaphore handle. The effects of the import operation will be either temporary or permanent, as specified by the application. If the import is temporary, the implementation must restore the semaphore to its prior permanent state after submitting the next semaphore wait operation. Performing a subsequent temporary import on a semaphore before performing a semaphore wait has no effect on this requirement; the next wait submitted on the semaphore must still restore its last permanent state. A permanent payload import behaves as if the target semaphore was destroyed, and a new semaphore was created with the same handle but the imported payload. Because importing a semaphore payload temporarily or permanently detaches the existing payload from a semaphore, similar usage restrictions to those applied to vkDestroySemaphore are applied to any command that imports a semaphore payload. Which of these import types is used is referred to as the import operation's permanence. Each handle type supports either one or both types of permanence.

The implementation must perform the import operation by either referencing or copying the payload referred to by the specified external semaphore handle, depending on the handle's type. The import method used is referred to as the handle type's transference. When using handle types with reference transference, importing a payload to a semaphore adds the semaphore to the set of all semaphores sharing that payload. This set includes the semaphore from which the payload was exported. Semaphore signaling and waiting operations performed on any semaphore in the set must behave as if the set were a single semaphore. Importing a payload using handle types with copy transference creates a duplicate copy of the payload at the time of import, but makes no further reference to it. Semaphore signaling and waiting operations performed on the target of copy imports must not affect any other semaphore or payload.

Export operations have the same transference as the specified handle type's import operations. Additionally, exporting a semaphore payload to a handle with copy transference has the same side effects on the source semaphore's payload as executing a semaphore wait operation. If the semaphore was using a temporarily imported payload, the semaphore's prior permanent payload will be restored.

External synchronization allows implementations to modify an object's internal state, i.e. payload, without internal synchronization. However, for semaphores sharing a payload across processes, satisfying the external synchronization requirements of VkSemaphore parameters as if all semaphores in the set were the same object is sometimes infeasible. Satisfying the wait operation state requirements would similarly require impractical coordination or levels of trust between processes. Therefore, these constraints only apply to a specific semaphore handle, not to its payload. For distinct semaphore objects which share a payload, if the semaphores are passed to separate queue submission commands concurrently, behavior will be as if the commands were called in an arbitrary sequential order. If the wait operation state requirements are violated for the shared payload by a queue submission command, or if a signal operation is queued for a shared payload that is already signaled or has a pending signal operation, effects must be limited to one or more of the following:

- Returning VK_ERROR_INITIALIZATION_FAILED from the command which resulted in the violation.
- Losing the logical device on which the violation occurred immediately or at a future time,
resulting in a VK_ERROR_DEVICE_LOST error from subsequent commands, including the one causing the violation.

- Continuing execution of the violating command or operation as if the semaphore wait completed successfully after an implementation-dependent timeout. In this case, the state of the payload becomes undefined, and future operations on semaphores sharing the payload will be subject to these same rules. The semaphore must be destroyed or have its payload replaced by an import operation to again have a well-defined state.

Note

These rules allow processes to synchronize access to shared memory without trusting each other. However, such processes must still be cautious not to use the shared semaphore for more than synchronizing access to the shared memory. For example, a process should not use a shared semaphore as part of an execution dependency chain that, when complete, leads to objects being destroyed, if it does not trust other processes sharing the semaphore payload.

When a semaphore is using an imported payload, its VkExportSemaphoreCreateInfo::handleTypes value is specified when creating the semaphore from which the payload was exported, rather than specified when creating the semaphore. Additionally, VkExternalSemaphoreProperties::exportFromImportedHandleTypes restricts which handle types can be exported from such a semaphore based on the specific handle type used to import the current payload.

When importing a semaphore payload, it is the responsibility of the application to ensure the external handles meet all valid usage requirements. However, implementations must perform sufficient validation of external handles to ensure that the operation results in a valid semaphore which will not cause program termination, device loss, queue stalls, or corruption of other resources when used as allowed according to its import parameters, and excepting those side effects allowed for violations of the valid semaphore state for wait operations rules. If the external handle provided does not meet these requirements, the implementation must fail the semaphore payload import operation with the error code VK_ERROR_INVALID_EXTERNAL_HANDLE.

In addition, when importing a semaphore payload that is not compatible with the payload type corresponding to the VkSemaphoreType the semaphore was created with, the implementation may fail the semaphore payload import operation with the error code VK_ERROR_INVALID_EXTERNAL_HANDLE.

Note

As the introduction of the external semaphore handle type VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT predates that of timeline semaphores, support for importing semaphore payloads from external handles of that type into semaphores created (implicitly or explicitly) with a VkSemaphoreType of VK_SEMAPHORE_TYPE_BINARY is preserved for backwards compatibility. However, applications should prefer importing such handle types into semaphores created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_TIMELINE.
7.5. Events

Events are a synchronization primitive that can be used to insert a fine-grained dependency between commands submitted to the same queue, or between the host and a queue. Events must not be used to insert a dependency between commands submitted to different queues. Events have two states - signaled and unsignaled. An application can signal or unsignal an event either on the host or on the device. A device can be made to wait for an event to become signaled before executing further operations. No command exists to wait for an event to become signaled on the host, but the current state of an event can be queried.

Events are represented by VkEvent handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkEvent)
```

To create an event, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateEvent(
    VkDevice device,
    const VkEventCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkEvent* pEvent);
```

- `device` is the logical device that creates the event.
- `pCreateInfo` is a pointer to a VkEventCreateInfo structure containing information about how the event is to be created.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pEvent` is a pointer to a handle in which the resulting event object is returned.

When created, the event object is in the unsignaled state.

**Valid Usage (Implicit)**

- VUID-vkCreateEvent-device-parameter `device` must be a valid VkDevice handle
- VUID-vkCreateEvent-pCreateInfo-parameter `pCreateInfo` must be a valid pointer to a valid VkEventCreateInfo structure
- VUID-vkCreateEvent-pAllocator-parameter If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid VkAllocationCallbacks structure
- VUID-vkCreateEvent-pEvent-parameter `pEvent` must be a valid pointer to a VkEvent handle
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkEventCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkEventCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkEventCreateFlags flags;
} VkEventCreateInfo;
```

• `sType` is the type of this structure.
• `pNext` is NULL or a pointer to a structure extending this structure.
• `flags` is a bitmask of `VkEventCreateFlagBits` defining additional creation parameters.

Valid Usage (Implicit)

• VUID-VkEventCreateInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_EVENT_CREATE_INFO`

• VUID-VkEventCreateInfo-pNext-pNext
  `pNext` must be NULL

• VUID-VkEventCreateInfo-flags-parameter
  `flags` must be a valid combination of `VkEventCreateFlagBits` values

```c
// Provided by VK_VERSION_1_0
typedef enum VkEventCreateFlagBits {
    // Provided by VK_VERSION_1_3
    VK_EVENT_CREATE_DEVICE_ONLY_BIT = 0x00000001,
} VkEventCreateFlagBits;
```

• `VK_EVENT_CREATE_DEVICE_ONLY_BIT` specifies that host event commands will not be used with this event.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkEventCreateFlags;
```
**VkEventCreateFlags** is a bitmask type for setting a mask of **VkEventCreateFlagBits**.

To destroy an event, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyEvent(
    VkDevice device,
    VkEvent event,
    const VkAllocationCallbacks* pAllocator);
```

- **device** is the logical device that destroys the event.
- **event** is the handle of the event to destroy.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage

- VUID-vkDestroyEvent-event-01145
  All submitted commands that refer to **event** must have completed execution

- VUID-vkDestroyEvent-event-01146
  If **VkAllocationCallbacks** were provided when **event** was created, a compatible set of callbacks must be provided here

- VUID-vkDestroyEvent-event-01147
  If no **VkAllocationCallbacks** were provided when **event** was created, **pAllocator** must be NULL

### Valid Usage (Implicit)

- VUID-vkDestroyEvent-device-parameter
  **device** must be a valid **VkDevice** handle

- VUID-vkDestroyEvent-event-parameter
  If **event** is not **VK_NULL_HANDLE**, **event** must be a valid **VkEvent** handle

- VUID-vkDestroyEvent-pAllocator-parameter
  If **pAllocator** is not NULL, **pAllocator** must be a valid pointer to a valid **VkAllocationCallbacks** structure

- VUID-vkDestroyEvent-event-parent
  If **event** is a valid handle, it must have been created, allocated, or retrieved from **device**

### Host Synchronization

- Host access to **event** must be externally synchronized

To query the state of an event from the host, call:
VkResult vkGetEventStatus(
    VkDevice device,
    VkEvent event);

- `device` is the logical device that owns the event.
- `event` is the handle of the event to query.

Upon success, `vkGetEventStatus` returns the state of the event object with the following return codes:

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_EVENT_SET</td>
<td>The event specified by <code>event</code> is signaled.</td>
</tr>
<tr>
<td>VK_EVENT_RESET</td>
<td>The event specified by <code>event</code> is unsignaled.</td>
</tr>
</tbody>
</table>

If a `vkCmdSetEvent` or `vkCmdResetEvent` command is in a command buffer that is in the pending state, then the value returned by this command may immediately be out of date.

The state of an event can be updated by the host. The state of the event is immediately changed, and subsequent calls to `vkGetEventStatus` will return the new state. If an event is already in the requested state, then updating it to the same state has no effect.

**Valid Usage**

- VUID-vkGetEventStatus-event-03940
  - `event` must not have been created with `VK_EVENT_CREATE_DEVICE_ONLY_BIT`

**Valid Usage (Implicit)**

- VUID-vkGetEventStatus-device-parameter
  - `device` must be a valid `VkDevice` handle
- VUID-vkGetEventStatus-event-parameter
  - `event` must be a valid `VkEvent` handle
- VUID-vkGetEventStatus-event-parent
  - `event` must have been created, allocated, or retrieved from `device`

**Return Codes**

**Success**

- `VK_EVENT_SET`
To set the state of an event to signaled from the host, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkSetEvent(
    VkDevice device,
    VkEvent event);
```

- `device` is the logical device that owns the event.
- `event` is the event to set.

When `vkSetEvent` is executed on the host, it defines an event signal operation which sets the event to the signaled state.

If `event` is already in the signaled state when `vkSetEvent` is executed, then `vkSetEvent` has no effect, and no event signal operation occurs.

**Note**

If a command buffer is waiting for an event to be signaled from the host, the application must signal the event before submitting the command buffer, as described in the queue forward progress section.

### Valid Usage

- VUID-vkSetEvent-event-03941
  - `event` must not have been created with `VK_EVENT_CREATE_DEVICE_ONLY_BIT`

### Valid Usage (Implicit)

- VUID-vkSetEvent-device-parameter
  - `device` must be a valid `VkDevice` handle
- VUID-vkSetEvent-event-parameter
  - `event` must be a valid `VkEvent` handle
- VUID-vkSetEvent-event-parent
  - `event` must have been created, allocated, or retrieved from `device`
Host Synchronization

- Host access to event must be externally synchronized

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

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

```c
// Provided by VK_VERSION_1_0
VkResult vkResetEvent(
    VkDevice device,
    VkEvent event);
```

- device is the logical device that owns the event.
- event is the event to reset.

When vkResetEvent is executed on the host, it defines an event unsignal operation which resets the event to the unsignaled state.

If event is already in the unsignaled state when vkResetEvent is executed, then vkResetEvent has no effect, and no event unsignal operation occurs.

Valid Usage

- VUID-vkResetEvent-event-03821
  There must be an execution dependency between vkResetEvent and the execution of any vkCmdWaitEvents that includes event in its pEvents parameter

- VUID-vkResetEvent-event-03822
  There must be an execution dependency between vkResetEvent and the execution of any vkCmdWaitEvents2 that includes event in its pEvents parameter

- VUID-vkResetEvent-event-03823
  event must not have been created with VK_EVENT_CREATE_DEVICE_ONLY_BIT

Valid Usage (Implicit)

- VUID-vkResetEvent-device-parameter
device must be a valid VkDevice handle

- VUID-vkResetEvent-event-parameter
  event must be a valid VkEvent handle
- VUID-vkResetEvent-event-parent
  event must have been created, allocated, or retrieved from device

Host Synchronization

- Host access to event must be externally synchronized

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The state of an event can also be updated on the device by commands inserted in command buffers.

To signal an event from a device, call:

```
// Provided by VK_VERSION_1_3
void vkCmdSetEvent2(
    VkCommandBuffer commandBuffer,
    VkEvent event,
    const VkDependencyInfo* pDependencyInfo);
```

- commandBuffer is the command buffer into which the command is recorded.
- event is the event that will be signaled.
- pDependencyInfo is a pointer to a VkDependencyInfo structure defining the first scopes of this operation.

When vkCmdSetEvent2 is submitted to a queue, it defines the first half of memory dependencies defined by pDependencyInfo, as well as an event signal operation which sets the event to the signaled state. A memory dependency is defined between the event signal operation and commands that occur earlier in submission order.

The first synchronization scope and access scope are defined by the union of all the memory dependencies defined by pDependencyInfo, and are applied to all operations that occur earlier in submission order. Queue family ownership transfers and image layout transitions defined by pDependencyInfo are also included in the first scopes.
The second synchronization scope includes only the event signal operation, and any queue family ownership transfers and image layout transitions defined by pDependencyInfo.

The second access scope includes only queue family ownership transfers and image layout transitions.

Future vkCmdWaitEvents2 commands rely on all values of each element in pDependencyInfo matching exactly with those used to signal the corresponding event. vkCmdWaitEvents must not be used to wait on the result of a signal operation defined by vkCmdSetEvent2.

Note
The extra information provided by vkCmdSetEvent2 compared to vkCmdSetEvent allows implementations to more efficiently schedule the operations required to satisfy the requested dependencies. With vkCmdSetEvent, the full dependency information is not known until vkCmdWaitEvents is recorded, forcing implementations to insert the required operations at that point and not before.

If event is already in the signaled state when vkCmdSetEvent2 is executed on the device, then vkCmdSetEvent2 has no effect, no event signal operation occurs, and no dependency is generated.

Valid Usage

- VUID-vkCmdSetEvent2-synchronization2-03824
  The synchronization2 feature must be enabled

- VUID-vkCmdSetEvent2-dependencyFlags-03825
  The dependencyFlags member of pDependencyInfo must be 0

- VUID-vkCmdSetEvent2-commandBuffer-03826
  The current device mask of commandBuffer must include exactly one physical device

- VUID-vkCmdSetEvent2-srcStageMask-03827
  The srcStageMask member of any element of the pMemoryBarriers, pBufferMemoryBarriers, or pImageMemoryBarriers members of pDependencyInfo must only include pipeline stages valid for the queue family that was used to create the command pool that commandBuffer was allocated from

- VUID-vkCmdSetEvent2-dstStageMask-03828
  The dstStageMask member of any element of the pMemoryBarriers, pBufferMemoryBarriers, or pImageMemoryBarriers members of pDependencyInfo must only include pipeline stages valid for the queue family that was used to create the command pool that commandBuffer was allocated from

Valid Usage (Implicit)

- VUID-vkCmdSetEvent2-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdSetEvent2-event-parameter
**Host Synchronization**

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

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td>Synchronization</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Decode</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Encode</td>
<td></td>
</tr>
</tbody>
</table>

The `VkDependencyInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkDependencyInfo {
    VkStructureType sType;
    const void* pNext;
    VkDependencyFlags dependencyFlags;
    VkMemoryBarrier2* pMemoryBarriers;
    uint32_t memoryBarrierCount;
    uint32_t bufferMemoryBarrierCount;
    const VkBufferMemoryBarrier2* pBufferMemoryBarriers;
    uint32_t imageMemoryBarrierCount;
    const VkImageMemoryBarrier2* pImageMemoryBarriers;
} VkDependencyInfo;
```
• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• dependencyFlags is a bitmask of VkDependencyFlagBits specifying how execution and memory dependencies are formed.
• memoryBarrierCount is the length of the pMemoryBarriers array.
• pMemoryBarriers is a pointer to an array of VkMemoryBarrier2 structures defining memory dependencies between any memory accesses.
• bufferMemoryBarrierCount is the length of the pBufferMemoryBarriers array.
• pBufferMemoryBarriers is a pointer to an array of VkBufferMemoryBarrier2 structures defining memory dependencies between buffer ranges.
• imageMemoryBarrierCount is the length of the pImageMemoryBarriers array.
• pImageMemoryBarriers is a pointer to an array of VkImageMemoryBarrier2 structures defining memory dependencies between image subresources.

This structure defines a set of memory dependencies, as well as queue family transfer operations and image layout transitions.

Each member of pMemoryBarriers, pBufferMemoryBarriers, and pImageMemoryBarriers defines a separate memory dependency.

Valid Usage (Implicit)

• VUID-VkDependencyInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_DEPENDENCY_INFO
• VUID-VkDependencyInfo-pNext-pNext
  pNext must be NULL
• VUID-VkDependencyInfo-dependencyFlags-parameter
  dependencyFlags must be a valid combination of VkDependencyFlagBits values
• VUID-VkDependencyInfo-pMemoryBarriers-parameter
  If memoryBarrierCount is not 0, pMemoryBarriers must be a valid pointer to an array of memoryBarrierCount valid VkMemoryBarrier2 structures
• VUID-VkDependencyInfo-pBufferMemoryBarriers-parameter
  If bufferMemoryBarrierCount is not 0, pBufferMemoryBarriers must be a valid pointer to an array of bufferMemoryBarrierCount valid VkBufferMemoryBarrier2 structures
• VUID-VkDependencyInfo-pImageMemoryBarriers-parameter
  If imageMemoryBarrierCount is not 0, pImageMemoryBarriers must be a valid pointer to an array of imageMemoryBarrierCount valid VkImageMemoryBarrier2 structures

To set the state of an event to signaled from a device, call:
```c
void vkCmdSetEvent(
    VkCommandBuffer commandBuffer,
    VkEvent event,
    VkPipelineStageFlags stageMask);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `event` is the event that will be signaled.
- `stageMask` specifies the source stage mask used to determine the first synchronization scope.

`vkCmdSetEvent` behaves identically to `vkCmdSetEvent2`, except that it does not define an access scope, and must only be used with `vkCmdWaitEvents`, not `vkCmdWaitEvents2`.

**Valid Usage**

- VUID-vkCmdSetEvent-stageMask-04090
  If the `geometryShader` feature is not enabled, `stageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`

- VUID-vkCmdSetEvent-stageMask-04091
  If the `tessellationShader` feature is not enabled, `stageMask` must not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`

- VUID-vkCmdSetEvent-stageMask-03937
  If the `synchronization2` feature is not enabled, `stageMask` must not be 0

- VUID-vkCmdSetEvent-stageMask-06457
  Any pipeline stage included in `stageMask` must be supported by the capabilities of the queue family specified by the `queueFamilyIndex` member of the `VkCommandPoolCreateInfo` structure that was used to create the `VkCommandPool` that `commandBuffer` was allocated from, as specified in the table of supported pipeline stages

- VUID-vkCmdSetEvent-stageMask-01149
  `stageMask` must not include `VK_PIPELINE_STAGE_HOST_BIT`

- VUID-vkCmdSetEvent-commandBuffer-01152
  `commandBuffer`'s current device mask must include exactly one physical device

**Valid Usage (Implicit)**

- VUID-vkCmdSetEvent-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetEvent-event-parameter
  `event` must be a valid `VkEvent` handle

- VUID-vkCmdSetEvent-stageMask-parameter
  `stageMask` must be a valid combination of `VkPipelineStageFlagBits` values
• VUID-vkCmdSetEvent-commandBuffer-recording
  \text{commandBuffer must be in the recording state}

• VUID-vkCmdSetEvent-commandBuffer-cmdpool
  The \text{VkCommandPool} that \text{commandBuffer} was allocated from must support graphics, compute, decode, or encode operations

• VUID-vkCmdSetEvent-renderpass
  This command must only be called outside of a render pass instance

• VUID-vkCmdSetEvent-commonparent
  Both of \text{commandBuffer}, and \text{event} must have been created, allocated, or retrieved from the same \text{VkDevice}

\begin{center}
\begin{tabular}{|c|c|c|}
  \hline
  \textbf{Command Buffer Levels} & \textbf{Render Pass Scope} & \textbf{Supported Queue Types} \\
  \hline
  Primary & Outside & Graphics \\
  Secondary & & Compute \\
  & & Decode \\
  & & Encode \\
  & & Synchronization \\
  \hline
\end{tabular}
\end{center}

Host Synchronization

• Host access to \text{commandBuffer} must be externally synchronized

• Host access to the \text{VkCommandPool} that \text{commandBuffer} was allocated from must be externally synchronized

Command Properties

To unsignal the event from a device, call:

\begin{verbatim}
// Provided by VK_VERSION_1_3
void vkCmdResetEvent2(
    VkCommandBuffer commandBuffer, 
    VkEvent event, 
    VkPipelineStageFlags2 stageMask);
\end{verbatim}

• \text{commandBuffer} is the command buffer into which the command is recorded.

• \text{event} is the event that will be unsignaled.

• \text{stageMask} is a \text{VkPipelineStageFlags2} mask of pipeline stages used to determine the first synchronization scope.

When \text{vkCmdResetEvent2} is submitted to a queue, it defines an execution dependency on commands that were submitted before it, and defines an event unsignal operation which resets the
event to the unsignaled state.

The first synchronization scope includes all commands that occur earlier in submission order. The synchronization scope is limited to operations by stageMask or stages that are logically earlier than stageMask.

The second synchronization scope includes only the event unsignal operation.

If event is already in the unsignaled state when vkCmdResetEvent2 is executed on the device, then this command has no effect, no event unsignal operation occurs, and no execution dependency is generated.

**Valid Usage**

- VUID-vkCmdResetEvent2-stageMask-03929
  If the geometryShader feature is not enabled, stageMask **must** not contain VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT

- VUID-vkCmdResetEvent2-stageMask-03930
  If the tessellationShader feature is not enabled, stageMask **must** not contain VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT or VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT

- VUID-vkCmdResetEvent2-synchronization2-03829
  The synchronization2 feature **must** be enabled

- VUID-vkCmdResetEvent2-stageMask-03830
  stageMask **must** not include VK_PIPELINE_STAGE_2_HOST_BIT

- VUID-vkCmdResetEvent2-event-03831
  There **must** be an execution dependency between vkCmdResetEvent2 and the execution of any vkCmdWaitEvents that includes event in its pEvents parameter

- VUID-vkCmdResetEvent2-event-03832
  There **must** be an execution dependency between vkCmdResetEvent2 and the execution of any vkCmdWaitEvents2 that includes event in its pEvents parameter

- VUID-vkCmdResetEvent2-commandBuffer-03833
  commandBuffer’s current device mask **must** include exactly one physical device

**Valid Usage (Implicit)**

- VUID-vkCmdResetEvent2-commandBuffer-parameter
  commandBuffer **must** be a valid VkCommandBuffer handle

- VUID-vkCmdResetEvent2-event-parameter
  event **must** be a valid VkEvent handle

- VUID-vkCmdResetEvent2-stageMask-parameter
  stageMask **must** be a valid combination of VkPipelineStageFlagBits2 values

- VUID-vkCmdResetEvent2-commandBuffer-recording

215
**commandBuffer** must be in the recording state

- **VUID-vkCmdResetEvent2-commandBuffer-cmdpool**
  The **VkCommandPool** that **commandBuffer** was allocated from **must** support graphics, compute, decode, or encode operations

- **VUID-vkCmdResetEvent2-renderpass**
  This command **must** only be called outside of a render pass instance

- **VUID-vkCmdResetEvent2-commonparent**
  Both of **commandBuffer**, and **event** **must** have been created, allocated, or retrieved from the same **VkDevice**

### Host Synchronization

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Secondary</td>
<td>Outside</td>
<td>Graphics Compute Decode Encode</td>
<td>Synchronization</td>
</tr>
</tbody>
</table>

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

```c
// Provided by VK_VERSION_1_0
void vkCmdResetEvent(
    VkCommandBuffer commandBuffer,
    VkEvent event,
    VkPipelineStageFlags stageMask);
```

- **commandBuffer** is the command buffer into which the command is recorded.
- **event** is the event that will be unsignaled.
- **stageMask** is a bitmask of **VkPipelineStageFlagBits** specifying the source stage mask used to determine when the **event** is unsignaled.

**vkCmdResetEvent** behaves identically to **vkCmdResetEvent2**.
Valid Usage

• VUID-vkCmdResetEvent-stageMask-04090
  If the geometryShader feature is not enabled, stageMask must not contain
  VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT

• VUID-vkCmdResetEvent-stageMask-04091
  If the tessellationShader feature is not enabled, stageMask must not contain
  VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT or
  VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT

• VUID-vkCmdResetEvent-stageMask-03937
  If the synchronization2 feature is not enabled, stageMask must not be 0

• VUID-vkCmdResetEvent-stageMask-06458
  Any pipeline stage included in stageMask must be supported by the capabilities of the
  queue family specified by the queueFamilyIndex member of the V kiến CommandPoolCreateInfo structure that was used to create the V kiến CommandPool that
  commandBuffer was allocated from, as specified in the table of supported pipeline stages

• VUID-vkCmdResetEvent-stageMask-01153
  stageMask must not include VK_PIPELINE_STAGE_HOST_BIT

• VUID-vkCmdResetEvent-event-03834
  There must be an execution dependency between vkCmdResetEvent and the execution of
  any vkCmdWaitEvents that includes event in its pEvents parameter

• VUID-vkCmdResetEvent-event-03835
  There must be an execution dependency between vkCmdResetEvent and the execution of
  any vkCmdWaitEvents2 that includes event in its pEvents parameter

• VUID-vkCmdResetEvent-commandBuffer-01157
  commandBuffer’s current device mask must include exactly one physical device

Valid Usage (Implicit)

• VUID-vkCmdResetEvent-commandBuffer-parameter
  commandBuffer must be a valid V kiến CommandBuffer handle

• VUID-vkCmdResetEvent-event-parameter
  event must be a valid V kiến Event handle

• VUID-vkCmdResetEvent-stageMask-parameter
  stageMask must be a valid combination of V kiến PipelineStageFlagBits values

• VUID-vkCmdResetEvent-commandBuffer-recording
  commandBuffer must be in the recording state

• VUID-vkCmdResetEvent-commandBuffer-cmdpool
  The V kiến CommandPool that commandBuffer was allocated from must support graphics, compute,
  decode, or encode operations

• VUID-vkCmdResetEvent-renderpass
This command **must** only be called outside of a render pass instance

- VUID-vkCmdResetEvent-commonparent
  Both of `commandBuffer`, and `event` **must** have been created, allocated, or retrieved from the same `VkDevice`

---

**Host Synchronization**

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

---

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td>Synchronization</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Decode</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Encode</td>
<td></td>
</tr>
</tbody>
</table>

---

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

```c
// Provided by VK_VERSION_1_3
void vkCmdWaitEvents2(
    VkCommandBuffer commandBuffer,
    uint32_t eventCount,
    const VkEvent* pEvents,
    const VkDependencyInfo* pDependencyInfos);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `eventCount` is the length of the `pEvents` array.
- `pEvents` is a pointer to an array of `eventCount` events to wait on.
- `pDependencyInfos` is a pointer to an array of `eventCount` `VkDependencyInfo` structures, defining the second synchronization scope.

When `vkCmdWaitEvents2` is submitted to a queue, it inserts memory dependencies according to the elements of `pDependencyInfos` and each corresponding element of `pEvents`. `vkCmdWaitEvents2` **must** not be used to wait on event signal operations occurring on other queues, or signal operations executed by `vkCmdSetEvent`.

The first synchronization scope and access scope of each memory dependency defined by any element `i` of `pDependencyInfos` are applied to operations that occurred earlier in submission order...
than the last event signal operation on element \( i \) of \( p\text{Events} \).

Signal operations for an event at index \( i \) are only included if:

- The event was signaled by a \texttt{vkCmdSetEvent2} command that occurred earlier in submission order with a \texttt{dependencyInfo} parameter exactly equal to the element of \( p\text{DependencyInfos} \) at index \( i \); or
- The event was created without \( \text{VK\_EVENT\_CREATE\_DEVICE\_ONLY\_BIT} \), and the first synchronization scope defined by the element of \( p\text{DependencyInfos} \) at index \( i \) only includes host operations \( (\text{VK\_PIPELINE\_STAGE\_2\_HOST\_BIT}) \).

The second synchronization scope and access scope of each memory dependency defined by any element \( i \) of \( p\text{DependencyInfos} \) are applied to operations that occurred later in submission order than \texttt{vkCmdWaitEvents2}.

\textit{Note}

\texttt{vkCmdWaitEvents2} is used with \texttt{vkCmdSetEvent2} to define a memory dependency between two sets of action commands, roughly in the same way as pipeline barriers, but split into two commands such that work between the two \textbf{may} execute unhindered.

\textit{Note}

Applications should be careful to avoid race conditions when using events. There is no direct ordering guarantee between \texttt{vkCmdSetEvent2} and \texttt{vkCmdResetEvent2}, \texttt{vkCmdResetEvent}, or \texttt{vkCmdSetEvent}. Another execution dependency (e.g. a pipeline barrier or semaphore with \texttt{VK\_PIPELINE\_STAGE\_2\_ALL\_COMMANDS\_BIT}) is needed to prevent such a race condition.

\textbf{Valid Usage}

- VUID-vkCmdWaitEvents2-synchronization2-03836
  The \textit{synchronization2} feature \textbf{must} be enabled
- VUID-vkCmdWaitEvents2-pEvents-03837
  Members of \( p\text{Events} \) \textbf{must} not have been signaled by \texttt{vkCmdSetEvent}
- VUID-vkCmdWaitEvents2-pEvents-03838
  For any element \( i \) of \( p\text{Events} \), if that event is signaled by \texttt{vkCmdSetEvent2}, that command's \texttt{dependencyInfo} parameter \textbf{must} be exactly equal to the \( i \)th element of \( p\text{DependencyInfos} \)
- VUID-vkCmdWaitEvents2-pEvents-03839
  For any element \( i \) of \( p\text{Events} \), if that event is signaled by \texttt{vkSetEvent}, barriers in the \( i \)th element of \( p\text{DependencyInfos} \) \textbf{must} include only host operations in their first synchronization scope
- VUID-vkCmdWaitEvents2-pEvents-03840
  For any element \( i \) of \( p\text{Events} \), if barriers in the \( i \)th element of \( p\text{DependencyInfos} \) include only host operations, the \( i \)th element of \( p\text{Events} \) \textbf{must} be signaled before \texttt{vkCmdWaitEvents2} is executed
For any element \(i\) of \(pEvents\), if barriers in the \(i\)th element of \(pDependencyInfos\) do not include host operations, the \(i\)th element of \(pEvents\) must be signaled by a corresponding \(vkCmdSetEvent2\) that occurred earlier in submission order.

The \(srcStageMask\) member of any element of the \(pMemoryBarriers\), \(pBufferMemoryBarriers\), or \(pImageMemoryBarriers\) members of \(pDependencyInfos\) must either include only pipeline stages valid for the queue family that was used to create the command pool that \(commandBuffer\) was allocated from, or include only \(VK\_PIPELINE\_STAGE\_2\_HOST\_BIT\).

The \(dstStageMask\) member of any element of the \(pMemoryBarriers\), \(pBufferMemoryBarriers\), or \(pImageMemoryBarriers\) members of \(pDependencyInfos\) must only include pipeline stages valid for the queue family that was used to create the command pool that \(commandBuffer\) was allocated from.

If \(vkCmdWaitEvents2\) is being called inside a render pass instance, the \(srcStageMask\) member of any element of the \(pMemoryBarriers\), \(pBufferMemoryBarriers\), or \(pImageMemoryBarriers\) members of \(pDependencyInfos\) must not include \(VK\_PIPELINE\_STAGE\_2\_HOST\_BIT\).

\(commandBuffer\)'s current device mask must include exactly one physical device.

Valid Usage (Implicit)

- \(commandBuffer\) must be a valid \(VkCommandBuffer\) handle.
- \(pEvents\) must be a valid pointer to an array of \(eventCount\) valid \(VkEvent\) handles.
- \(pDependencyInfos\) must be a valid pointer to an array of \(eventCount\) valid \(VkDependencyInfo\) structures.
- \(commandBuffer\) must be in the recording state.
- \(commandBuffer\) must support graphics, compute, decode, or encode operations.
- \(eventCount\) must be greater than 0.
- Both of \(commandBuffer\), and the elements of \(pEvents\) must have been created, allocated, or retrieved from the same \(VkDevice\).
Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>Synchronization</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Decode</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Encode</td>
<td></td>
</tr>
</tbody>
</table>

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

```c
// Provided by VK_VERSION_1_0
void vkCmdWaitEvents(
    VkCommandBuffer commandBuffer,  
    uint32_t eventCount,            
    const VkEvent* pEvents,        
    VkPipelineStageFlags srcStageMask,  
    VkPipelineStageFlags dstStageMask,  
    uint32_t memoryBarrierCount,    
    const VkMemoryBarrier* pMemoryBarriers,  
    uint32_t bufferMemoryBarrierCount,  
    const VkBufferMemoryBarrier* pBufferMemoryBarriers,  
    uint32_t imageMemoryBarrierCount,  
    const VkImageMemoryBarrier* pImageMemoryBarriers);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `eventCount` is the length of the `pEvents` array.
- `pEvents` is a pointer to an array of event object handles to wait on.
- `srcStageMask` is a bitmask of `VkPipelineStageFlagBits` specifying the source stage mask.
- `dstStageMask` is a bitmask of `VkPipelineStageFlagBits` specifying the destination stage mask.
- `memoryBarrierCount` is the length of the `pMemoryBarriers` array.
- `pMemoryBarriers` is a pointer to an array of `VkMemoryBarrier` structures.
- `bufferMemoryBarrierCount` is the length of the `pBufferMemoryBarriers` array.
- `pBufferMemoryBarriers` is a pointer to an array of `VkBufferMemoryBarrier` structures.
- `imageMemoryBarrierCount` is the length of the `pImageMemoryBarriers` array.
• **pImageMemoryBarriers** is a pointer to an array of VkImageMemoryBarrier structures.

**vkCmdWaitEvents** is largely similar to **vkCmdWaitEvents2**, but **can** only wait on signal operations defined by **vkCmdSetEvent**. As **vkCmdSetEvent** does not define any access scopes, **vkCmdWaitEvents** defines the first access scope for each event signal operation in addition to its own access scopes.

**Note**
Since **vkCmdSetEvent** does not have any dependency information beyond a stage mask, implementations do not have the same opportunity to perform availability and visibility operations or image layout transitions in advance as they do with **vkCmdSetEvent2** and **vkCmdWaitEvents2**.

When **vkCmdWaitEvents** is submitted to a queue, it defines a memory dependency between prior event signal operations on the same queue or the host, and subsequent commands. **vkCmdWaitEvents** must not be used to wait on event signal operations occurring on other queues.

The first synchronization scope only includes event signal operations that operate on members of **pEvents**, and the operations that happened-before the event signal operations. Event signal operations performed by **vkCmdSetEvent** that occur earlier in submission order are included in the first synchronization scope, if the logically latest pipeline stage in their **stageMask** parameter is logically earlier than or equal to the logically latest pipeline stage in **srcStageMask**. Event signal operations performed by **vkSetEvent** are only included in the first synchronization scope if **VK_PIPELINE_STAGE_HOST_BIT** is included in **srcStageMask**.

The second synchronization scope includes all commands that occur later in submission order. The second synchronization scope is limited to operations on the pipeline stages determined by the destination stage mask specified by **dstStageMask**.

The first access scope is limited to accesses in the pipeline stages determined by the source stage mask specified by **srcStageMask**. Within that, the first access scope only includes the first access scopes defined by elements of the **pMemoryBarriers**, **pBufferMemoryBarriers** and **pImageMemoryBarriers** arrays, which each define a set of memory barriers. If no memory barriers are specified, then the first access scope includes no accesses.

The second access scope is limited to accesses in the pipeline stages determined by the destination stage mask specified by **dstStageMask**. Within that, the second access scope only includes the second access scopes defined by elements of the **pMemoryBarriers**, **pBufferMemoryBarriers** and **pImageMemoryBarriers** arrays, which each define a set of memory barriers. If no memory barriers are specified, then the second access scope includes no accesses.

---

**Valid Usage**

- **VUID-vkCmdWaitEvents-srcStageMask-04090**
  If the geometryShader feature is not enabled, **srcStageMask** must not contain **VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT**

- **VUID-vkCmdWaitEvents-srcStageMask-04091**
  If the tessellationShader feature is not enabled, **srcStageMask** must not contain **VK_PIPELINE_STAGE_TESSellation_CONTROL_SHADER_BIT** or
• VUID-vkCmdWaitEvents-srcStageMask-03937
  If the synchronization feature is not enabled, srcStageMask must not be 0

• VUID-vkCmdWaitEvents-dstStageMask-04090
  If the geometryShader feature is not enabled, dstStageMask must not contain VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT

• VUID-vkCmdWaitEvents-dstStageMask-04091
  If the tessellationShader feature is not enabled, dstStageMask must not contain VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT or VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT

• VUID-vkCmdWaitEvents-dstStageMask-03937
  If the synchronization feature is not enabled, dstStageMask must not be 0

• VUID-vkCmdWaitEvents-srAccessMask-02815
  The srcAccessMask member of each element of pMemoryBarriers must only include access flags that are supported by one or more of the pipeline stages in srcStageMask, as specified in the table of supported access types

• VUID-vkCmdWaitEvents-dstAccessMask-02816
  The dstAccessMask member of each element of pMemoryBarriers must only include access flags that are supported by one or more of the pipeline stages in dstStageMask, as specified in the table of supported access types

• VUID-vkCmdWaitEvents-pBufferMemoryBarriers-02817
  For any element of pBufferMemoryBarriers, if its srcQueueFamilyIndex and dstQueueFamilyIndex members are equal, or if its srcQueueFamilyIndex is the queue family index that was used to create the command pool that commandBuffer was allocated from, then its srcAccessMask member must only contain access flags that are supported by one or more of the pipeline stages in srcStageMask, as specified in the table of supported access types

• VUID-vkCmdWaitEvents-pBufferMemoryBarriers-02818
  For any element of pBufferMemoryBarriers, if its srcQueueFamilyIndex and dstQueueFamilyIndex members are equal, or if its dstQueueFamilyIndex is the queue family index that was used to create the command pool that commandBuffer was allocated from, then its dstAccessMask member must only contain access flags that are supported by one or more of the pipeline stages in dstStageMask, as specified in the table of supported access types

• VUID-vkCmdWaitEvents-pImageMemoryBarriers-02819
  For any element of pImageMemoryBarriers, if its srcQueueFamilyIndex and dstQueueFamilyIndex members are equal, or if its srcQueueFamilyIndex is the queue family index that was used to create the command pool that commandBuffer was allocated from, then its srcAccessMask member must only contain access flags that are supported by one or more of the pipeline stages in srcStageMask, as specified in the table of supported access types

• VUID-vkCmdWaitEvents-pImageMemoryBarriers-02820
  For any element of pImageMemoryBarriers, if its srcQueueFamilyIndex and
dstQueueFamilyIndex members are equal, or if its dstQueueFamilyIndex is the queue family index that was used to create the command pool that commandBuffer was allocated from, then its dstAccessMask member must only contain access flags that are supported by one or more of the pipeline stages in dstStageMask, as specified in the table of supported access types.

- VUID-vkCmdWaitEvents-srcStageMask-06459
  Any pipeline stage included in srcStageMask must be supported by the capabilities of the queue family specified by the queueFamilyIndex member of the VkCommandPoolCreateInfo structure that was used to create the VkCommandPool that commandBuffer was allocated from, as specified in the table of supported pipeline stages.

- VUID-vkCmdWaitEvents-dstStageMask-06460
  Any pipeline stage included in dstStageMask must be supported by the capabilities of the queue family specified by the queueFamilyIndex member of the VkCommandPoolCreateInfo structure that was used to create the VkCommandPool that commandBuffer was allocated from, as specified in the table of supported pipeline stages.

- VUID-vkCmdWaitEvents-srcStageMask-01158
  srcStageMask must be the bitwise OR of the stageMask parameter used in previous calls to vkCmdSetEvent with any of the elements of pEvents and VK_PIPELINE_STAGE_HOST_BIT if any of the elements of pEvents was set using vkSetEvent.

- VUID-vkCmdWaitEvents-srcQueueFamilyIndex-02803
  The srcQueueFamilyIndex and dstQueueFamilyIndex members of any element of pBufferMemoryBarriers or pImageMemoryBarriers must be equal.

- VUID-vkCmdWaitEvents-commandBuffer-01167
  commandBuffer’s current device mask must include exactly one physical device.

- VUID-vkCmdWaitEvents-pEvents-03847
  Elements of pEvents must not have been signaled by vkCmdSetEvent2.

Valid Usage (Implicit)

- VUID-vkCmdWaitEvents-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle.

- VUID-vkCmdWaitEvents-pEvents-parameter
  pEvents must be a valid pointer to an array of eventCount valid VkEvent handles.

- VUID-vkCmdWaitEvents-srcStageMask-parameter
  srcStageMask must be a valid combination of VkPipelineStageFlagBits values.

- VUID-vkCmdWaitEvents-dstStageMask-parameter
  dstStageMask must be a valid combination of VkPipelineStageFlagBits values.

- VUID-vkCmdWaitEvents-pMemoryBarriers-parameter
  If memoryBarrierCount is not 0, pMemoryBarriers must be a valid pointer to an array of...
memoryBarrierCount valid VkMemoryBarrier structures

• VUID-vkCmdWaitEvents-pBufferMemoryBarriers-parameter
  If bufferMemoryBarrierCount is not 0, pBufferMemoryBarriers must be a valid pointer to an array of bufferMemoryBarrierCount valid VkBufferMemoryBarrier structures

• VUID-vkCmdWaitEvents-pImageMemoryBarriers-parameter
  If imageMemoryBarrierCount is not 0, pImageMemoryBarriers must be a valid pointer to an array of imageMemoryBarrierCount valid VkImageMemoryBarrier structures

• VUID-vkCmdWaitEvents-commandBuffer-recording
  commandBuffer must be in the recording state

• VUID-vkCmdWaitEvents-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics, compute, decode, or encode operations

• VUID-vkCmdWaitEvents-eventCount-arraylength
  eventCount must be greater than 0

• VUID-vkCmdWaitEvents-commonparent
  Both of commandBuffer, and the elements of pEvents must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

• Host access to commandBuffer must be externally synchronized

• Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>Synchronization</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Decode</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Encode</td>
<td></td>
</tr>
</tbody>
</table>

7.6. Pipeline Barriers

To record a pipeline barrier, call:

// Provided by VK_VERSION_1_3
def void vkCmdPipelineBarrier2(
    VkCommandBuffer commandBuffer,
const VkDependencyInfo* pDependencyInfo);

- `commandBuffer` is the command buffer into which the command is recorded.
- `pDependencyInfo` is a pointer to a `VkDependencyInfo` structure defining the scopes of this operation.

When `vkCmdPipelineBarrier2` is submitted to a queue, it defines memory dependencies between commands that were submitted before it, and those submitted after it.

The first synchronization scope and access scope of each memory dependency defined by `pDependencyInfo` are applied to operations that occurred earlier in submission order.

The second synchronization scope and access scope of each memory dependency defined by `pDependencyInfo` are applied to operations that occurred later in submission order.

If `vkCmdPipelineBarrier2` is recorded within a render pass instance, the synchronization scopes are limited to operations within the same subpass.

---

**Valid Usage**

- **VUID-vkCmdPipelineBarrier2-pDependencies-02285**
  If `vkCmdPipelineBarrier2` is called within a render pass instance, the render pass must have been created with at least one `VkSubpassDependency` instance in `VkRenderPassCreateInfo::pDependencies` that expresses a dependency from the current subpass to itself, with synchronization scopes and access scopes that are all supersets of the scopes defined in this command.

- **VUID-vkCmdPipelineBarrier2-bufferMemoryBarrierCount-01178**
  If `vkCmdPipelineBarrier2` is called within a render pass instance, it must not include any buffer memory barriers.

- **VUID-vkCmdPipelineBarrier2-image-04073**
  If `vkCmdPipelineBarrier2` is called within a render pass instance, the image member of any image memory barrier included in this command must be an attachment used in the current subpass both as an input attachment, and as either a color or depth/stencil attachment.

- **VUID-vkCmdPipelineBarrier2-oldLayout-01181**
  If `vkCmdPipelineBarrier2` is called within a render pass instance, the oldLayout and newLayout members of any image memory barrier included in this command must be equal.

- **VUID-vkCmdPipelineBarrier2-srcQueueFamilyIndex-01182**
  If `vkCmdPipelineBarrier2` is called within a render pass instance, the srcQueueFamilyIndex and dstQueueFamilyIndex members of any image memory barrier included in this command must be equal.

- **VUID-vkCmdPipelineBarrier2-dependencyFlags-01186**
  If `vkCmdPipelineBarrier2` is called outside of a render pass instance, `VK_DEPENDENCY_VIEW_LOCAL_BIT` must not be included in the dependency flags.
If `vkCmdPipelineBarrier2` is called within a render pass instance, the render pass must not have been started with `vkCmdBeginRendering`.

The synchronization2 feature must be enabled.

The srcStageMask member of any element of the `pMemoryBarriers`, `pBufferMemoryBarriers`, or `pImageMemoryBarriers` members of `pDependencyInfo` must only include pipeline stages valid for the queue family that was used to create the command pool that `commandBuffer` was allocated from.

The dstStageMask member of any element of the `pMemoryBarriers`, `pBufferMemoryBarriers`, or `pImageMemoryBarriers` members of `pDependencyInfo` must only include pipeline stages valid for the queue family that was used to create the command pool that `commandBuffer` was allocated from.

**Valid Usage (Implicit)**

- `commandBuffer` must be a valid `VkCommandBuffer` handle.
- `pDependencyInfo` must be a valid pointer to a valid `VkDependencyInfo` structure.
- `commandBuffer` must be in the recording state.
- The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, compute, decode, or encode operations.

**Host Synchronization**

- Host access to `commandBuffer` must be externally synchronized.
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized.
To record a pipeline barrier, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdPipelineBarrier(
    VkCommandBuffer commandBuffer,
    VkPipelineStageFlags srcStageMask,
    VkPipelineStageFlags dstStageMask,
    VkDependencyFlags dependencyFlags,
    uint32_t memoryBarrierCount,
    const VkMemoryBarrier* pMemoryBarriers,
    uint32_t bufferMemoryBarrierCount,
    const VkBufferMemoryBarrier* pBufferMemoryBarriers,
    uint32_t imageMemoryBarrierCount,
    const VkImageMemoryBarrier* pImageMemoryBarriers);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `srcStageMask` is a bitmask of `VkPipelineStageFlagBits` specifying the source stages.
- `dstStageMask` is a bitmask of `VkPipelineStageFlagBits` specifying the destination stages.
- `dependencyFlags` is a bitmask of `VkDependencyFlagBits` specifying how execution and memory dependencies are formed.
- `memoryBarrierCount` is the length of the `pMemoryBarriers` array.
- `pMemoryBarriers` is a pointer to an array of `VkMemoryBarrier` structures.
- `bufferMemoryBarrierCount` is the length of the `pBufferMemoryBarriers` array.
- `pBufferMemoryBarriers` is a pointer to an array of `VkBufferMemoryBarrier` structures.
- `imageMemoryBarrierCount` is the length of the `pImageMemoryBarriers` array.
- `pImageMemoryBarriers` is a pointer to an array of `VkImageMemoryBarrier` structures.

`vkCmdPipelineBarrier` operates almost identically to `vkCmdPipelineBarrier2`, except that the scopes and barriers are defined as direct parameters rather than being defined by an `VkDependencyInfo`.

When `vkCmdPipelineBarrier` is submitted to a queue, it defines a memory dependency between commands that were submitted before it, and those submitted after it.
If `vkCmdPipelineBarrier` was recorded outside a render pass instance, the first synchronization scope includes all commands that occur earlier in submission order. If `vkCmdPipelineBarrier` was recorded inside a render pass instance, the first synchronization scope includes only commands that occur earlier in submission order within the same subpass. In either case, the first synchronization scope is limited to operations on the pipeline stages determined by the source stage mask specified by `srcStageMask`.

If `vkCmdPipelineBarrier` was recorded outside a render pass instance, the second synchronization scope includes all commands that occur later in submission order. If `vkCmdPipelineBarrier` was recorded inside a render pass instance, the second synchronization scope includes only commands that occur later in submission order within the same subpass. In either case, the second synchronization scope is limited to operations on the pipeline stages determined by the destination stage mask specified by `dstStageMask`.

The first access scope is limited to accesses in the pipeline stages determined by the source stage mask specified by `srcStageMask`. Within that, the first access scope only includes the first access scopes defined by elements of the `pMemoryBarriers`, `pBufferMemoryBarriers` and `pImageMemoryBarriers` arrays, which each define a set of memory barriers. If no memory barriers are specified, then the first access scope includes no accesses.

The second access scope is limited to accesses in the pipeline stages determined by the destination stage mask specified by `dstStageMask`. Within that, the second access scope only includes the second access scopes defined by elements of the `pMemoryBarriers`, `pBufferMemoryBarriers` and `pImageMemoryBarriers` arrays, which each define a set of memory barriers. If no memory barriers are specified, then the second access scope includes no accesses.

If `dependencyFlags` includes `VK_DEPENDENCY_BY_REGION_BIT`, then any dependency between framebuffer-space pipeline stages is framebuffer-local - otherwise it is framebuffer-global.

### Valid Usage

- **VUID-vkCmdPipelineBarrier-srcStageMask-04090**
  
  If the *geometryShader* feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`

- **VUID-vkCmdPipelineBarrier-srcStageMask-04091**
  
  If the *tessellationShader* feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`

- **VUID-vkCmdPipelineBarrier-srcStageMask-03937**
  
  If the *synchronization2* feature is not enabled, `srcStageMask` must not be `0`

- **VUID-vkCmdPipelineBarrier-dstStageMask-04090**
  
  If the *geometryShader* feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`

- **VUID-vkCmdPipelineBarrier-dstStageMask-04091**
  
  If the *tessellationShader* feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT

- VUID-vkCmdPipelineBarrier-dstStageMask-03937
  If the synchronization2 feature is not enabled, dstStageMask must not be 0.

- VUID-vkCmdPipelineBarrier-srcAccessMask-02815
  The srcAccessMask member of each element of pMemoryBarriers must only include access flags that are supported by one or more of the pipeline stages in srcStageMask, as specified in the table of supported access types.

- VUID-vkCmdPipelineBarrier-dstAccessMask-02816
  The dstAccessMask member of each element of pMemoryBarriers must only include access flags that are supported by one or more of the pipeline stages in dstStageMask, as specified in the table of supported access types.

- VUID-vkCmdPipelineBarrier-pBufferMemoryBarriers-02817
  For any element of pBufferMemoryBarriers, if its srcQueueFamilyIndex and dstQueueFamilyIndex members are equal, or if its srcQueueFamilyIndex is the queue family index that was used to create the command pool that commandBuffer was allocated from, then its srcAccessMask member must only contain access flags that are supported by one or more of the pipeline stages in srcStageMask, as specified in the table of supported access types.

- VUID-vkCmdPipelineBarrier-pBufferMemoryBarriers-02818
  For any element of pBufferMemoryBarriers, if its dstQueueFamilyIndex is the queue family index that was used to create the command pool that commandBuffer was allocated from, then its dstAccessMask member must only contain access flags that are supported by one or more of the pipeline stages in dstStageMask, as specified in the table of supported access types.

- VUID-vkCmdPipelineBarrier-pImageMemoryBarriers-02819
  For any element of pImageMemoryBarriers, if its srcQueueFamilyIndex and dstQueueFamilyIndex members are equal, or if its srcQueueFamilyIndex is the queue family index that was used to create the command pool that commandBuffer was allocated from, then its srcAccessMask member must only contain access flags that are supported by one or more of the pipeline stages in srcStageMask, as specified in the table of supported access types.

- VUID-vkCmdPipelineBarrier-pImageMemoryBarriers-02820
  For any element of pImageMemoryBarriers, if its dstQueueFamilyIndex is the queue family index that was used to create the command pool that commandBuffer was allocated from, then its dstAccessMask member must only contain access flags that are supported by one or more of the pipeline stages in dstStageMask, as specified in the table of supported access types.

- VUID-vkCmdPipelineBarrier-pDependencies-02285
  If vkCmdPipelineBarrier is called within a render pass instance, the render pass must have been created with at least one VkSubpassDependency instance in VkRenderPassCreateInfo::pDependencies that expresses a dependency from the current subpass to itself, with synchronization scopes and access scopes that are all supersets of the scopes defined in...
this command

- VUID-vkCmdPipelineBarrier-bufferMemoryBarrierCount-01178
  If `vkCmdPipelineBarrier` is called within a render pass instance, it **must** not include any buffer memory barriers

- VUID-vkCmdPipelineBarrier-image-04073
  If `vkCmdPipelineBarrier` is called within a render pass instance, the `image` member of any image memory barrier included in this command **must** be an attachment used in the current subpass both as an input attachment, and as either a color or depth/stencil attachment

- VUID-vkCmdPipelineBarrier-oldLayout-01181
  If `vkCmdPipelineBarrier` is called within a render pass instance, the `oldLayout` and `newLayout` members of any image memory barrier included in this command **must** be equal

- VUID-vkCmdPipelineBarrier-srcQueueFamilyIndex-01182
  If `vkCmdPipelineBarrier` is called within a render pass instance, the `srcQueueFamilyIndex` and `dstQueueFamilyIndex` members of any image memory barrier included in this command **must** be equal

- VUID-vkCmdPipelineBarrier-dependencyFlags-01186
  If `vkCmdPipelineBarrier` is called outside of a render pass instance, `VK_DEPENDENCY_VIEW_LOCAL_BIT` must not be included in the dependency flags

- VUID-vkCmdPipelineBarrier-None-06191
  If `vkCmdPipelineBarrier` is called within a render pass instance, the render pass **must** not have been started with `vkCmdBeginRendering`

- VUID-vkCmdPipelineBarrier-srcStageMask-06461
  Any pipeline stage included in `srcStageMask` **must** be supported by the capabilities of the queue family specified by the `queueFamilyIndex` member of the `VkCommandPoolCreateInfo` structure that was used to create the `VkCommandPool` that `commandBuffer` was allocated from, as specified in the table of supported pipeline stages

- VUID-vkCmdPipelineBarrier-dstStageMask-06462
  Any pipeline stage included in `dstStageMask` **must** be supported by the capabilities of the queue family specified by the `queueFamilyIndex` member of the `VkCommandPoolCreateInfo` structure that was used to create the `VkCommandPool` that `commandBuffer` was allocated from, as specified in the table of supported pipeline stages

### Valid Usage (Implicit)

- VUID-vkCmdPipelineBarrier-commandBuffer-parameter
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle

- VUID-vkCmdPipelineBarrier-srcStageMask-parameter
  `srcStageMask` **must** be a valid combination of `VkPipelineStageFlagBits` values

- VUID-vkCmdPipelineBarrier-dstStageMask-parameter
  `dstStageMask` **must** be a valid combination of `VkPipelineStageFlagBits` values
• VUID-vkCmdPipelineBarrier-dependencyFlags-parameter
dependencyFlags must be a valid combination of VkDependencyFlagBits values

• VUID-vkCmdPipelineBarrier-pMemoryBarriers-parameter
If memoryBarrierCount is not 0, pMemoryBarriers must be a valid pointer to an array of memoryBarrierCount valid VkMemoryBarrier structures

• VUID-vkCmdPipelineBarrier-pBufferMemoryBarriers-parameter
If bufferMemoryBarrierCount is not 0, pBufferMemoryBarriers must be a valid pointer to an array of bufferMemoryBarrierCount valid VkBufferMemoryBarrier structures

• VUID-vkCmdPipelineBarrier-pImageMemoryBarriers-parameter
If imageMemoryBarrierCount is not 0, pImageMemoryBarriers must be a valid pointer to an array of imageMemoryBarrierCount valid VkImageMemoryBarrier structures

• VUID-vkCmdPipelineBarrier-commandBuffer-recording
commandBuffer must be in the recording state

• VUID-vkCmdPipelineBarrier-commandBuffer-cmdpool
The VkCommandPool that commandBuffer was allocated from must support transfer, graphics, compute, decode, or encode operations

Host Synchronization

• Host access to commandBuffer must be externally synchronized

• Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Transfer Graphics Compute Decode Encode</td>
<td>Synchronization</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits which can be set in VkCmdPipelineBarrier::dependencyFlags, specifying how execution and memory dependencies are formed, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkDependencyFlagBits {
    VK_DEPENDENCY_BY_REGION_BIT = 0x00000001,
    // Provided by VK_VERSION_1_1
    VK_DEPENDENCY_DEVICE_GROUP_BIT = 0x00000004,
    // Provided by VK_VERSION_1_1
```
VK_DEPENDENCY_VIEW_LOCAL_BIT = 0x00000002,
} VkDependencyFlagBits;

• VK_DEPENDENCY_BY_REGION_BIT specifies that dependencies will be framebuffer-local.
• VK_DEPENDENCY_VIEW_LOCAL_BIT specifies that a subpass has more than one view.
• VK_DEPENDENCY_DEVICE_GROUP_BIT specifies that dependencies are non-device-local.

// Provided by VK_VERSION_1_0
typedef VkFlags VkDependencyFlags;

VkDependencyFlags is a bitmask type for setting a mask of zero or more VkDependencyFlagBits.

### 7.6.1. Subpass Self-dependency

vkCmdPipelineBarrier or vkCmdPipelineBarrier2 must not be called within a render pass instance started with vkCmdBeginRendering.

If vkCmdPipelineBarrier or vkCmdPipelineBarrier2 is called inside a render pass instance, the following restrictions apply. For a given subpass to allow a pipeline barrier, the render pass must declare a self-dependency from that subpass to itself. That is, there must exist a subpass dependency with srcSubpass and dstSubpass both equal to that subpass index. More than one self-dependency can be declared for each subpass.

Self-dependencies must only include pipeline stage bits that are graphics stages. If any of the stages in srcStageMask are framebuffer-space stages, dstStageMask must only contain framebuffer-space stages. This means that pseudo-stages like VK_PIPELINE_STAGE_ALL_COMMANDS_BIT which include the execution of both framebuffer-space stages and non-framebuffer-space stages must not be used.

If the source and destination stage masks both include framebuffer-space stages, then dependencyFlags must include VK_DEPENDENCY_BY_REGION_BIT. If the subpass has more than one view, then dependencyFlags must include VK_DEPENDENCY_VIEW_LOCAL_BIT.

Each of the synchronization scopes and access scopes of a vkCmdPipelineBarrier2 or vkCmdPipelineBarrier command inside a render pass instance must be a subset of the scopes of one of the self-dependencies for the current subpass.

If the self-dependency has VK_DEPENDENCY_BY_REGION_BIT or VK_DEPENDENCY_VIEW_LOCAL_BIT set, then so must the pipeline barrier. Pipeline barriers within a render pass instance must not include buffer memory barriers. Image memory barriers must only specify image subresources that are used as attachments within the subpass, and must not define an image layout transition or queue family ownership transfer.

### 7.7. Memory Barriers

Memory barriers are used to explicitly control access to buffer and image subresource ranges. Memory barriers are used to transfer ownership between queue families, change image layouts, and define availability and visibility operations. They explicitly define the access types and buffer
and image subresource ranges that are included in the access scopes of a memory dependency that is created by a synchronization command that includes them.

7.7.1. Global Memory Barriers

Global memory barriers apply to memory accesses involving all memory objects that exist at the time of its execution.

The VkMemoryBarrier2 structure is defined as:

```
// Provided by VK_VERSION_1_3
typedef struct VkMemoryBarrier2 {
    VkStructureType sType;
    const void* pNext;
    VkPipelineStageFlags2 srcStageMask;
    VkAccessFlags2 srcAccessMask;
    VkPipelineStageFlags2 dstStageMask;
    VkAccessFlags2 dstAccessMask;
} VkMemoryBarrier2;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **srcStageMask** is a VkPipelineStageFlags2 mask of pipeline stages to be included in the first synchronization scope.
- **srcAccessMask** is a VkAccessFlags2 mask of access flags to be included in the first access scope.
- **dstStageMask** is a VkPipelineStageFlags2 mask of pipeline stages to be included in the second synchronization scope.
- **dstAccessMask** is a VkAccessFlags2 mask of access flags to be included in the second access scope.

This structure defines a memory dependency affecting all device memory.

The first synchronization scope and access scope described by this structure include only operations and memory accesses specified by srcStageMask and srcAccessMask.

The second synchronization scope and access scope described by this structure include only operations and memory accesses specified by dstStageMask and dstAccessMask.

**Valid Usage**

- **VUID-VkMemoryBarrier2-srcStageMask-03929**
  If the geometryShader feature is not enabled, srcStageMask must not contain VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT

- **VUID-VkMemoryBarrier2-srcStageMask-03930**
  If the tessellationShader feature is not enabled, srcStageMask must not contain VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT or VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT
• VUID-VkMemoryBarrier2-srcAccessMask-03900
  If srcAccessMask includes VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT, VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkMemoryBarrier2-srcAccessMask-03901
  If srcAccessMask includes VK_ACCESS_2_INDEX_READ_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT, VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkMemoryBarrier2-srcAccessMask-03902
  If srcAccessMask includes VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT, VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkMemoryBarrier2-srcAccessMask-03903
  If srcAccessMask includes VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT, VK_PIPELINE_STAGE_2_SUBPASS_SHADING_BIT_HUAWEI,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkMemoryBarrier2-srcAccessMask-03904
  If srcAccessMask includes VK_ACCESS_2_UNIFORM_READ_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT, or one of
  the VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkMemoryBarrier2-srcAccessMask-03905
  If srcAccessMask includes VK_ACCESS_2_SHADER_SAMPLED_READ_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT, or one of
  the VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkMemoryBarrier2-srcAccessMask-03906
  If srcAccessMask includes VK_ACCESS_2_SHADER_STORAGE_READ_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT, or one of
  the VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkMemoryBarrier2-srcAccessMask-03907
  If srcAccessMask includes VK_ACCESS_2_SHADER_WRITE_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT, or one of
  the VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkMemoryBarrier2-srcAccessMask-03908
  If srcAccessMask includes VK_ACCESS_2_SHADER_READ_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT,
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or one of the
  VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkMemoryBarrier2-srcAccessMask-03909
  If srcAccessMask includes VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT, or one of
  the VK_PIPELINE_STAGE_*_SHADER_BIT stages
If `srcAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT`, `srcStageMask` must include
```
VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT,
VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT
```

If `srcAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT`, `srcStageMask` must include
```
VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT,
VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT
```

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT`, `srcStageMask` must include
```
VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT,
VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT
```

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`, `srcStageMask` must include
```
VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT,
VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT
```

If `srcAccessMask` includes `VK_ACCESS_2_TRANSFER_READ_BIT`, `srcStageMask` must include
```
VK_PIPELINE_STAGE_2_COPY_BIT, VK_PIPELINE_STAGE_2_BLIT_BIT,
VK_PIPELINE_STAGE_2_RESOLVE_BIT, VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT,
VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT
```

If `srcAccessMask` includes `VK_ACCESS_2_TRANSFER_WRITE_BIT`, `srcStageMask` must include
```
VK_PIPELINE_STAGE_2_COPY_BIT, VK_PIPELINE_STAGE_2_BLIT_BIT,
VK_PIPELINE_STAGE_2_RESOLVE_BIT, VK_PIPELINE_STAGE_2_CLEAR_BIT,
VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT
```

If `srcAccessMask` includes `VK_ACCESS_2_HOST_READ_BIT`, `srcStageMask` must include
```
VK_PIPELINE_STAGE_2_HOST_BIT
```

If `srcAccessMask` includes `VK_ACCESS_2_HOST_WRITE_BIT`, `srcStageMask` must include
```
VK_PIPELINE_STAGE_2_HOST_BIT
```

If the `geometryShader` feature is not enabled, `dstStageMask` must not contain
```
VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT
```

If the `tessellationShader` feature is not enabled, `dstStageMask` must not contain
```
VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT or VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT
```
If `dstAccessMask` includes `VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

If `dstAccessMask` includes `VK_ACCESS_2_INDEX_READ_BIT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT`, `VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

If `dstAccessMask` includes `VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT`, `VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

If `dstAccessMask` includes `VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT`, `VK_PIPELINE_STAGE_2_SUBPASS_SHADING_BIT_HUAWEI`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

If `dstAccessMask` includes `VK_ACCESS_2_UNIFORM_READ_BIT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `dstAccessMask` includes `VK_ACCESS_2_SHADER_SAMPLED_READ_BIT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `dstAccessMask` includes `VK_ACCESS_2_SHADER_STORAGE_READ_BIT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `dstAccessMask` includes `VK_ACCESS_2_SHADER_WRITE_BIT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `dstAccessMask` includes `VK_ACCESS_2_SHADER_READ_BIT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `dstAccessMask` includes `VK_ACCESS_2_SHADER_WRITE_BIT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.
• VUID-VkMemoryBarrier2-dstAccessMask-03910
If dstAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT, dstStageMask must include
  VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkMemoryBarrier2-dstAccessMask-03911
If dstAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT, dstStageMask must include
  VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkMemoryBarrier2-dstAccessMask-03912
If dstAccessMask includes VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT, dstStageMask must include
  VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT,
  VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkMemoryBarrier2-dstAccessMask-03913
If dstAccessMask includes VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT, dstStageMask must include
  VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT,
  VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkMemoryBarrier2-dstAccessMask-03914
If dstAccessMask includes VK_ACCESS_2_TRANSFER_READ_BIT, dstStageMask must include
  VK_PIPELINE_STAGE_2_COPY_BIT,
  VK_PIPELINE_STAGE_2_BLIT_BIT,
  VK_PIPELINE_STAGE_2_RESOLVE_BIT,
  VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT,
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkMemoryBarrier2-dstAccessMask-03915
If dstAccessMask includes VK_ACCESS_2_TRANSFER_WRITE_BIT, dstStageMask must include
  VK_PIPELINE_STAGE_2_COPY_BIT,
  VK_PIPELINE_STAGE_2_BLIT_BIT,
  VK_PIPELINE_STAGE_2_RESOLVE_BIT,
  VK_PIPELINE_STAGE_2_CLEAR_BIT,
  VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT,
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkMemoryBarrier2-dstAccessMask-03916
If dstAccessMask includes VK_ACCESS_2_HOST_READ_BIT, dstStageMask must include

• VUID-VkMemoryBarrier2-dstAccessMask-03917
If dstAccessMask includes VK_ACCESS_2_HOST_WRITE_BIT, dstStageMask must include

Valid Usage (Implicit)

• VUID-VkMemoryBarrier2-sType-sType
  sType must be VK_STRUCTURE_TYPE_MEMORY_BARRIER_2

• VUID-VkMemoryBarrier2-srcStageMask-parameter
**srcStageMask** must be a valid combination of VkPipelineStageFlagBits2 values

- VUID-VkMemoryBarrier2-srcAccessMask-parameter
  **srcAccessMask** must be a valid combination of VkAccessFlagBits2 values

- VUID-VkMemoryBarrier2-dstStageMask-parameter
  **dstStageMask** must be a valid combination of VkPipelineStageFlagBits2 values

- VUID-VkMemoryBarrier2-dstAccessMask-parameter
  **dstAccessMask** must be a valid combination of VkAccessFlagBits2 values

The **VkMemoryBarrier** structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkMemoryBarrier {
    VkStructureType sType;
    const void* pNext;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
} VkMemoryBarrier;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **srcAccessMask** is a bitmask of VkAccessFlagBits specifying a source access mask.
- **dstAccessMask** is a bitmask of VkAccessFlagBits specifying a destination access mask.

The first access scope is limited to access types in the source access mask specified by **srcAccessMask**.

The second access scope is limited to access types in the destination access mask specified by **dstAccessMask**.

### Valid Usage (Implicit)

- VUID-VkMemoryBarrier-sType-sType
  **sType** must be VK_STRUCTURE_TYPE_MEMORY_BARRIER

- VUID-VkMemoryBarrier-pNext-pNext
  **pNext** must be NULL

- VUID-VkMemoryBarrier-srcAccessMask-parameter
  **srcAccessMask** must be a valid combination of VkAccessFlagBits values

- VUID-VkMemoryBarrier-dstAccessMask-parameter
  **dstAccessMask** must be a valid combination of VkAccessFlagBits values

### 7.7.2. Buffer Memory Barriers

Buffer memory barriers only apply to memory accesses involving a specific buffer range. That is, a memory dependency formed from a buffer memory barrier is scoped to access via the specified range.
buffer range. Buffer memory barriers can also be used to define a queue family ownership transfer for the specified buffer range.

The `VkBufferMemoryBarrier2` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkBufferMemoryBarrier2 {
    VkStructureType sType;
    const void* pNext;
    VkPipelineStageFlags2 srcStageMask;
    VkAccessFlags2 srcAccessMask;
    VkPipelineStageFlags2 dstStageMask;
    VkAccessFlags2 dstAccessMask;
    uint32_t srcQueueFamilyIndex;
    uint32_t dstQueueFamilyIndex;
    VkBuffer buffer;
    VkDeviceSize offset;
    VkDeviceSize size;
} VkBufferMemoryBarrier2;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcStageMask` is a `VkPipelineStageFlags2` mask of pipeline stages to be included in the first synchronization scope.
- `srcAccessMask` is a `VkAccessFlags2` mask of access flags to be included in the first access scope.
- `dstStageMask` is a `VkPipelineStageFlags2` mask of pipeline stages to be included in the second synchronization scope.
- `dstAccessMask` is a `VkAccessFlags2` mask of access flags to be included in the second access scope.
- `srcQueueFamilyIndex` is the source queue family for a queue family ownership transfer.
- `dstQueueFamilyIndex` is the destination queue family for a queue family ownership transfer.
- `buffer` is a handle to the buffer whose backing memory is affected by the barrier.
- `offset` is an offset in bytes into the backing memory for `buffer`; this is relative to the base offset as bound to the buffer (see `vkBindBufferMemory`).
- `size` is a size in bytes of the affected area of backing memory for `buffer`, or `VK_WHOLE_SIZE` to use the range from `offset` to the end of the buffer.

This structure defines a memory dependency limited to a range of a buffer, and can define a queue family transfer operation for that range.

The first synchronization scope and access scope described by this structure include only operations and memory accesses specified by `srcStageMask` and `srcAccessMask`.

The second synchronization scope and access scope described by this structure include only operations and memory accesses specified by `dstStageMask` and `dstAccessMask`.

240
Both access scopes are limited to only memory accesses to buffer in the range defined by offset and size.

If buffer was created with VK_SHARING_MODE_EXCLUSIVE, and srcQueueFamilyIndex is not equal to dstQueueFamilyIndex, this memory barrier defines a queue family transfer operation. When executed on a queue in the family identified by srcQueueFamilyIndex, this barrier defines a queue family release operation for the specified buffer range, and the second synchronization and access scopes do not synchronize operations on that queue. When executed on a queue in the family identified by dstQueueFamilyIndex, this barrier defines a queue family acquire operation for the specified buffer range, and the first synchronization and access scopes do not synchronize operations on that queue.

A queue family transfer operation is also defined if the values are not equal, and either is one of the special queue family values reserved for external memory ownership transfers, as described in Queue Family Ownership Transfer. A queue family release operation is defined when dstQueueFamilyIndex is one of those values, and a queue family acquire operation is defined when srcQueueFamilyIndex is one of those values.

### Valid Usage

- **VUID-VkBufferMemoryBarrier2-srcStageMask-03929**
  If the geometryShader feature is not enabled, srcStageMask must not contain VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT

- **VUID-VkBufferMemoryBarrier2-srcStageMask-03930**
  If the tessellationShader feature is not enabled, srcStageMask must not contain VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT or VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT

- **VUID-VkBufferMemoryBarrier2-srcAccessMask-03900**
  If srcAccessMask includes VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT, srcStageMask must include VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT, VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

- **VUID-VkBufferMemoryBarrier2-srcAccessMask-03901**
  If srcAccessMask includes VK_ACCESS_2_INDEX_READ_BIT, srcStageMask must include VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT, VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

- **VUID-VkBufferMemoryBarrier2-srcAccessMask-03902**
  If srcAccessMask includes VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT, srcStageMask must include VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT, VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

- **VUID-VkBufferMemoryBarrier2-srcAccessMask-03903**
  If srcAccessMask includes VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT, srcStageMask must include VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT, VK_PIPELINE_STAGE_2_SUBPASS_SHADING_BIT_HUAWEI, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT
If `srcAccessMask` includes `VK_ACCESS_2_UNIFORM_READ_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_SAMPLED_READ_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_STORAGE_READ_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_READ_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_WRITE_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT` or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

If `srcAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT` or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.
If `srcAccessMask` includes `VK_ACCESS_2_TRANSFER_READ_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COPY_BIT`, `VK_PIPELINE_STAGE_2_BLIT_BIT`, `VK_PIPELINE_STAGE_2_RESOLVE_BIT`, `VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTUREBUILD_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

- **VUID-VkBufferMemoryBarrier2-srcAccessMask-03915**
  If `srcAccessMask` includes `VK_ACCESS_2_TRANSFER_WRITE_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COPY_BIT`, `VK_PIPELINE_STAGE_2_BLIT_BIT`, `VK_PIPELINE_STAGE_2_RESOLVE_BIT`, `VK_PIPELINE_STAGE_2_CLEAR_BIT`, `VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTUREBUILD_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

- **VUID-VkBufferMemoryBarrier2-srcAccessMask-03916**
  If `srcAccessMask` includes `VK_ACCESS_2_HOST_READ_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_HOST_BIT`.

- **VUID-VkBufferMemoryBarrier2-srcAccessMask-03917**
  If `srcAccessMask` includes `VK_ACCESS_2_HOST_WRITE_BIT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_HOST_BIT`.

If the `geometryShader` feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT`.

- **VUID-VkBufferMemoryBarrier2-dstStageMask-03929**
  If the `tessellationShader` feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_2_TESSELLATIONCONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT`.

- **VUID-VkBufferMemoryBarrier2-dstStageMask-03930**
  If `dstAccessMask` includes `VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTUREBUILD_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

- **VUID-VkBufferMemoryBarrier2-dstAccessMask-03900**
  If `dstAccessMask` includes `VK_ACCESS_2_INDEX_READ_BIT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT`, `VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

- **VUID-VkBufferMemoryBarrier2-dstAccessMask-03901**
  If `dstAccessMask` includes `VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTEINPUT_BIT`, `VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

- **VUID-VkBufferMemoryBarrier2-dstAccessMask-03902**
  If `dstAccessMask` includes `VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT`, `VK_PIPELINE_STAGE_2_SUBPASSSHADING_BIT_HUAWEI`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.

- **VUID-VkBufferMemoryBarrier2-dstAccessMask-03903**
  If `dstAccessMask` includes `VK_ACCESS_2_INDEX_READ_BIT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_DRAW_INDEX_BIT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT`.
If \( \text{dstAccessMask} \) includes \( \text{VK_ACCESS_2_UNIFORM_READ_BIT} \), \( \text{dstStageMask} \) must include \( \text{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT}, \text{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT} \), or one of the \( \text{VK_PIPELINE_STAGE_2_\*_SHADER_BIT} \) stages

If \( \text{dstAccessMask} \) includes \( \text{VK_ACCESS_2_SHADER_SAMPLED_READ_BIT} \), \( \text{dstStageMask} \) must include \( \text{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT}, \text{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT} \), or one of the \( \text{VK_PIPELINE_STAGE_2_\*_SHADER_BIT} \) stages

If \( \text{dstAccessMask} \) includes \( \text{VK_ACCESS_2_SHADER_STORAGE_READ_BIT} \), \( \text{dstStageMask} \) must include \( \text{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT}, \text{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT} \), or one of the \( \text{VK_PIPELINE_STAGE_2_\*_SHADER_BIT} \) stages

If \( \text{dstAccessMask} \) includes \( \text{VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT} \), \( \text{dstStageMask} \) must include \( \text{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT}, \text{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT} \), or one of the \( \text{VK_PIPELINE_STAGE_2_\*_SHADER_BIT} \) stages

If \( \text{dstAccessMask} \) includes \( \text{VK_ACCESS_2_SHADER_READ_BIT} \), \( \text{dstStageMask} \) must include \( \text{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT}, \text{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT}, \text{VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR} \), or one of the \( \text{VK_PIPELINE_STAGE_2_\*_SHADER_BIT} \) stages

If \( \text{dstAccessMask} \) includes \( \text{VK_ACCESS_2_SHADER_WRITE_BIT} \), \( \text{dstStageMask} \) must include \( \text{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT}, \text{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT} \), or one of the \( \text{VK_PIPELINE_STAGE_2_\*_SHADER_BIT} \) stages

If \( \text{dstAccessMask} \) includes \( \text{VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT} \), \( \text{dstStageMask} \) must include \( \text{VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT}, \text{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT}, \text{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT} \)

If \( \text{dstAccessMask} \) includes \( \text{VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT} \), \( \text{dstStageMask} \) must include \( \text{VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT}, \text{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT}, \text{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT} \)

If \( \text{dstAccessMask} \) includes \( \text{VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT} \), \( \text{dstStageMask} \) must include \( \text{VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT}, \text{VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT}, \text{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT}, \text{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT} \)

If \( \text{dstAccessMask} \) includes \( \text{VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT} \), \( \text{dstStageMask} \) must include \( \text{VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT}, \text{VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT}, \text{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT}, \text{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT} \)
If dstAccessMask includes VK_ACCESS_2_TRANSFER_READ_BIT, dstStageMask must include
VK_PIPELINE_STAGE_2_COPY_BIT,
VK_PIPELINE_STAGE_2_BLIT_BIT,
VK_PIPELINE_STAGE_2_RESOLVE_BIT,
VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT,
VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
or
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkBufferMemoryBarrier2-dstAccessMask-03915
  If dstAccessMask includes VK_ACCESS_2_TRANSFER_WRITE_BIT, dstStageMask must include
  VK_PIPELINE_STAGE_2_COPY_BIT,
  VK_PIPELINE_STAGE_2_BLIT_BIT,
  VK_PIPELINE_STAGE_2_RESOLVE_BIT,
  VK_PIPELINE_STAGE_2_CLEAR_BIT,
  VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT,
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkBufferMemoryBarrier2-dstAccessMask-03916
  If dstAccessMask includes VK_ACCESS_2_HOST_READ_BIT, dstStageMask must include
  VK_PIPELINE_STAGE_2_HOST_BIT

• VUID-VkBufferMemoryBarrier2-dstAccessMask-03917
  If dstAccessMask includes VK_ACCESS_2_HOST_WRITE_BIT, dstStageMask must include
  VK_PIPELINE_STAGE_2_HOST_BIT

• VUID-VkBufferMemoryBarrier2-offset-01187
  offset must be less than the size of buffer

• VUID-VkBufferMemoryBarrier2-size-01188
  If size is not equal to VK_WHOLE_SIZE, size must be greater than 0

• VUID-VkBufferMemoryBarrier2-size-01189
  If size is not equal to VK_WHOLE_SIZE, size must be less than or equal to than the size of buffer minus offset

• VUID-VkBufferMemoryBarrier2-buffer-01931
  If buffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

• VUID-VkBufferMemoryBarrier2-srcQueueFamilyIndex-04087
  If srcQueueFamilyIndex is not equal to dstQueueFamilyIndex, at least one must not be a special queue family reserved for external memory ownership transfers, as described in Queue Family Ownership Transfer

• VUID-VkBufferMemoryBarrier2-buffer-04088
  If buffer was created with a sharing mode of VK_SHARING_MODE_CONCURRENT, srcQueueFamilyIndex and dstQueueFamilyIndex are not equal, and one of srcQueueFamilyIndex and dstQueueFamilyIndex is one of the special queue family values reserved for external memory transfers, the other must be VK_QUEUE_FAMILY_IGNORED

• VUID-VkBufferMemoryBarrier2-buffer-04089
  If buffer was created with a sharing mode of VK_SHARING_MODE_EXCLUSIVE, and srcQueueFamilyIndex and dstQueueFamilyIndex are not equal, srcQueueFamilyIndex and dstQueueFamilyIndex must both be valid queue families, or one of the special queue family values reserved for external memory transfers, as described in Queue Family Ownership Transfer
If either `srcStageMask` or `dstStageMask` includes `VK_PIPELINE_STAGE_2_HOST_BIT`, `srcQueueFamilyIndex` and `dstQueueFamilyIndex` must be equal.

**Valid Usage (Implicit)**

- **VUID-VkBufferMemoryBarrier2-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER_2`

- **VUID-VkBufferMemoryBarrier2-pNext-pNext**
  
  `pNext` must be `NULL`

- **VUID-VkBufferMemoryBarrier2-srcStageMask-parameter**
  
  `srcStageMask` must be a valid combination of `VkPipelineStageFlagBits2` values

- **VUID-VkBufferMemoryBarrier2-srcAccessMask-parameter**
  
  `srcAccessMask` must be a valid combination of `VkAccessFlagBits2` values

- **VUID-VkBufferMemoryBarrier2-dstStageMask-parameter**
  
  `dstStageMask` must be a valid combination of `VkPipelineStageFlagBits2` values

- **VUID-VkBufferMemoryBarrier2-dstAccessMask-parameter**
  
  `dstAccessMask` must be a valid combination of `VkAccessFlagBits2` values

- **VUID-VkBufferMemoryBarrier2-buffer-parameter**
  
  `buffer` must be a valid `VkBuffer` handle

The `VkBufferMemoryBarrier` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkBufferMemoryBarrier {
    VkStructureType    sType;
    const void*        pNext;
    VkAccessFlags      srcAccessMask;
    VkAccessFlags      dstAccessMask;
    uint32_t           srcQueueFamilyIndex;
    uint32_t           dstQueueFamilyIndex;
    VkBuffer           buffer;
    VkDeviceSize       offset;
    VkDeviceSize       size;
} VkBufferMemoryBarrier;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcAccessMask` is a bitmask of `VkAccessFlagBits` specifying a source access mask.
- `dstAccessMask` is a bitmask of `VkAccessFlagBits` specifying a destination access mask.
- `srcQueueFamilyIndex` is the source queue family for a queue family ownership transfer.
- `dstQueueFamilyIndex` is the destination queue family for a queue family ownership transfer.
• **buffer** is a handle to the buffer whose backing memory is affected by the barrier.
• **offset** is an offset in bytes into the backing memory for **buffer**; this is relative to the base offset as bound to the buffer (see vkBindBufferMemory).
• **size** is a size in bytes of the affected area of backing memory for **buffer**, or **VK_WHOLE_SIZE** to use the range from **offset** to the end of the buffer.

The first access scope is limited to access to memory through the specified buffer range, via access types in the source access mask specified by **srcAccessMask**. If **srcAccessMask** includes **VK_ACCESS_HOST_WRITE_BIT**, a memory domain operation is performed where available memory in the host domain is also made available to the device domain.

The second access scope is limited to access to memory through the specified buffer range, via access types in the destination access mask specified by **dstAccessMask**. If **dstAccessMask** includes **VK_ACCESS_HOST_WRITE_BIT** or **VK_ACCESS_HOST_READ_BIT**, a memory domain operation is performed where available memory in the device domain is also made available to the host domain.

**Note**

When **VK_MEMORY_PROPERTY_HOST_COHERENT_BIT** is used, available memory in host domain is automatically made visible to host domain, and any host write is automatically made available to host domain.

If **srcQueueFamilyIndex** is not equal to **dstQueueFamilyIndex**, and **srcQueueFamilyIndex** is equal to the current queue family, then the memory barrier defines a queue family release operation for the specified buffer range, and the second access scope includes no access, as if **dstAccessMask** was **0**.

If **dstQueueFamilyIndex** is not equal to **srcQueueFamilyIndex**, and **dstQueueFamilyIndex** is equal to the current queue family, then the memory barrier defines a queue family acquire operation for the specified buffer range, and the first access scope includes no access, as if **srcAccessMask** was **0**.

### Valid Usage

• VUID-VkBufferMemoryBarrier-offset-01187
  
  **offset must** be less than the size of **buffer**

• VUID-VkBufferMemoryBarrier-size-01188
  
  If **size** is not equal to **VK_WHOLE_SIZE**, **size must** be greater than **0**

• VUID-VkBufferMemoryBarrier-size-01189
  
  If **size** is not equal to **VK_WHOLE_SIZE**, **size must** be less than or equal to than the size of **buffer** minus **offset**

• VUID-VkBufferMemoryBarrier-buffer-01931
  
  If **buffer** is non-sparse then it **must** be bound completely and contiguously to a single **VkDeviceMemory** object

• VUID-VkBufferMemoryBarrier-srcQueueFamilyIndex-04087
  
  If **srcQueueFamilyIndex** is not equal to **dstQueueFamilyIndex**, at least one **must** not be a special queue family reserved for external memory ownership transfers, as described in Queue Family Ownership Transfer
If buffer was created with a sharing mode ofVK_SHARING_MODE_CONCURRENT, srcQueueFamilyIndex and dstQueueFamilyIndex are not equal, and one of srcQueueFamilyIndex and dstQueueFamilyIndex is one of the special queue family values reserved for external memory transfers, the other must be VK_QUEUE_FAMILY_IGNORED.

If buffer was created with a sharing mode ofVK_SHARING_MODE_EXCLUSIVE, and srcQueueFamilyIndex and dstQueueFamilyIndex are not equal, srcQueueFamilyIndex and dstQueueFamilyIndex must both be valid queue families, or one of the special queue family values reserved for external memory transfers, as described in Queue Family Ownership Transfer.

If the synchronization2 feature is not enabled, and buffer was created with a sharing mode ofVK_SHARING_MODE_CONCURRENT, at least one of srcQueueFamilyIndex and dstQueueFamilyIndex must be VK_QUEUE_FAMILY_IGNORED.

Valid Usage (Implicit)

- VUID-VkBufferMemoryBarrier-sType-sType
  
sType must beVK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER

- VUID-VkBufferMemoryBarrier-pNext-pNext
  
pNext must be NULL

- VUID-VkBufferMemoryBarrier-buffer-parameter
  
buffer must be a valid VkBuffer handle

VK_WHOLE_SIZE is a special value indicating that the entire remaining length of a buffer following a given offset should be used. It can be specified for VkBufferMemoryBarrier::size and other structures.

#define VK_WHOLE_SIZE (~0ULL)

7.7.3. Image Memory Barriers

Image memory barriers only apply to memory accesses involving a specific image subresource range. That is, a memory dependency formed from an image memory barrier is scoped to access via the specified image subresource range. Image memory barriers can also be used to define image layout transitions or a queue family ownership transfer for the specified image subresource range.

The VkImageMemoryBarrier2 structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkImageMemoryBarrier2 {
```
 VkStructureType sType;
const void* pNext;
VkPipelineStageFlags2 srcStageMask;
VkAccessFlags2 srcAccessMask;
VkPipelineStageFlags2 dstStageMask;
VkAccessFlags2 dstAccessMask;
VkImageLayout oldLayout;
VkImageLayout newLayout;
uint32_t srcQueueFamilyIndex;
uint32_t dstQueueFamilyIndex;
VkImage image;
VkImageSubresourceRange subresourceRange;
} VkImageMemoryBarrier2;

• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **srcStageMask** is a VkPipelineStageFlags2 mask of pipeline stages to be included in the first synchronization scope.
• **srcAccessMask** is a VkAccessFlags2 mask of access flags to be included in the first access scope.
• **dstStageMask** is a VkPipelineStageFlags2 mask of pipeline stages to be included in the second synchronization scope.
• **dstAccessMask** is a VkAccessFlags2 mask of access flags to be included in the second access scope.
• **oldLayout** is the old layout in an image layout transition.
• **newLayout** is the new layout in an image layout transition.
• **srcQueueFamilyIndex** is the source queue family for a queue family ownership transfer.
• **dstQueueFamilyIndex** is the destination queue family for a queue family ownership transfer.
• **image** is a handle to the image affected by this barrier.
• **subresourceRange** describes the image subresource range within image that is affected by this barrier.

This structure defines a memory dependency limited to an image subresource range, and can define a queue family transfer operation and image layout transition for that subresource range.

The first synchronization scope and access scope described by this structure include only operations and memory accesses specified by srcStageMask and srcAccessMask.

The second synchronization scope and access scope described by this structure include only operations and memory accesses specified by dstStageMask and dstAccessMask.

Both access scopes are limited to only memory accesses to image in the subresource range defined by subresourceRange.

If image was created with VK_SHARING_MODE_EXCLUSIVE, and srcQueueFamilyIndex is not equal to dstQueueFamilyIndex, this memory barrier defines a queue family transfer operation. When executed on a queue in the family identified by srcQueueFamilyIndex, this barrier defines a queue
family release operation for the specified image subresource range, and the second synchronization and access scopes do not synchronize operations on that queue. When executed on a queue in the family identified by dstQueueFamilyIndex, this barrier defines a queue family acquire operation for the specified image subresource range, and the first synchronization and access scopes do not synchronize operations on that queue.

A queue family transfer operation is also defined if the values are not equal, and either is one of the special queue family values reserved for external memory ownership transfers, as described in Queue Family Ownership Transfer. A queue family release operation is defined when dstQueueFamilyIndex is one of those values, and a queue family acquire operation is defined when srcQueueFamilyIndex is one of those values.

If oldLayout is not equal to newLayout, then the memory barrier defines an image layout transition for the specified image subresource range. If this memory barrier defines a queue family transfer operation, the layout transition is only executed once between the queues.

Note
When the old and new layout are equal, the layout values are ignored - data is preserved no matter what values are specified, or what layout the image is currently in.

If image has a multi-planar format and the image is disjoint, then including VK_IMAGE_ASPECT_COLOR_BIT in the aspectMask member of subresourceRange is equivalent to including VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, and (for three-plane formats only) VK_IMAGE_ASPECT_PLANE_2_BIT.

Valid Usage

• VUID-VkImageMemoryBarrier2-srcStageMask-03929
  If the geometryShader feature is not enabled, srcStageMask must not contain VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT

• VUID-VkImageMemoryBarrier2-srcStageMask-03930
  If the tessellationShader feature is not enabled, srcStageMask must not contain VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT or VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT

• VUID-VkImageMemoryBarrier2-srcAccessMask-03900
  If srcAccessMask includes VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT, VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkImageMemoryBarrier2-srcAccessMask-03901
  If srcAccessMask includes VK_ACCESS_2_INDEX_READ_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT, VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkImageMemoryBarrier2-srcAccessMask-03902
  If srcAccessMask includes VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT, srcStageMask must
include VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT,
   VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT,
   VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or
   VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

- VUID-VkImageMemoryBarrier2-srcAccessMask-03903
  If srcAccessMask includes VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT,
  VK_PIPELINE_STAGE_2_SUBPASS_SHADING_BIT_HUAWEI, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT,
  or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

- VUID-VkImageMemoryBarrier2-srcAccessMask-03904
  If srcAccessMask includes VK_ACCESS_2_UNIFORM_READ_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT, or one of
  the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- VUID-VkImageMemoryBarrier2-srcAccessMask-03905
  If srcAccessMask includes VK_ACCESS_2_SHADER_SAMPLED_READ_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT, or one of
  the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- VUID-VkImageMemoryBarrier2-srcAccessMask-03906
  If srcAccessMask includes VK_ACCESS_2_SHADER_STORAGE_READ_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT, or one of
  the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- VUID-VkImageMemoryBarrier2-srcAccessMask-03907
  If srcAccessMask includes VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT, or one of
  the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- VUID-VkImageMemoryBarrier2-srcAccessMask-03908
  If srcAccessMask includes VK_ACCESS_2_SHADER_READ_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT,
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or one of the
  VK_PIPELINE_STAGE_*_SHADER_BIT stages

- VUID-VkImageMemoryBarrier2-srcAccessMask-03909
  If srcAccessMask includes VK_ACCESS_2_SHADER_WRITE_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT, or one of
  the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- VUID-VkImageMemoryBarrier2-srcAccessMask-03910
  If srcAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

- VUID-VkImageMemoryBarrier2-srcAccessMask-03911
  If srcAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

- VUID-VkImageMemoryBarrier2-srcAccessMask-03912
  If srcAccessMask includes VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT, srcStageMask must include
  VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT,
VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

- VUID-VkImageMemoryBarrier2-srcAccessMask-03913
  If srcAccessMask includes VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT, srcStageMask must include VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT, VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

- VUID-VkImageMemoryBarrier2-srcAccessMask-03914
  If srcAccessMask includes VK_ACCESS_2_TRANSFER_READ_BIT, srcStageMask must include VK_PIPELINE_STAGE_2_COPY_BIT, VK_PIPELINE_STAGE_2_BLIT_BIT, VK_PIPELINE_STAGE_2_RESOLVE_BIT, VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT, VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

- VUID-VkImageMemoryBarrier2-srcAccessMask-03915
  If srcAccessMask includes VK_ACCESS_2_TRANSFER_WRITE_BIT, srcStageMask must include VK_PIPELINE_STAGE_2_COPY_BIT, VK_PIPELINE_STAGE_2_BLIT_BIT, VK_PIPELINE_STAGE_2_RESOLVE_BIT, VK_PIPELINE_STAGE_2_CLEAR_BIT, VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT, VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

- VUID-VkImageMemoryBarrier2-srcAccessMask-03916
  If srcAccessMask includes VK_ACCESS_2_HOST_READ_BIT, srcStageMask must include VK_PIPELINE_STAGE_2_HOST_BIT

- VUID-VkImageMemoryBarrier2-srcAccessMask-03917
  If srcAccessMask includes VK_ACCESS_2_HOST_WRITE_BIT, srcStageMask must include VK_PIPELINE_STAGE_2_HOST_BIT

- VUID-VkImageMemoryBarrier2-dstStageMask-03929
  If the geometryShader feature is not enabled, dstStageMask must not contain VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT

- VUID-VkImageMemoryBarrier2-dstStageMask-03930
  If the tessellationShader feature is not enabled, dstStageMask must not contain VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT or VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT

- VUID-VkImageMemoryBarrier2-dstAccessMask-03900
  If dstAccessMask includes VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT, dstStageMask must include VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT, VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

- VUID-VkImageMemoryBarrier2-dstAccessMask-03901
  If dstAccessMask includes VK_ACCESS_2_INDEX_READ_BIT, dstStageMask must include VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT, VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

- VUID-VkImageMemoryBarrier2-dstAccessMask-03902
If dstAccessMask includes VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT, dstStageMask must include
VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkImageMemoryBarrier2-dstAccessMask-03903
If dstAccessMask includes VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT, dstStageMask must include
VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT, VK_PIPELINE_STAGE_2_SUBPASS_SHADING_BIT_HUAWEI, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkImageMemoryBarrier2-dstAccessMask-03904
If dstAccessMask includes VK_ACCESS_2_UNIFORM_READ_BIT, dstStageMask must include
VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT, or one of the
VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkImageMemoryBarrier2-dstAccessMask-03905
If dstAccessMask includes VK_ACCESS_2_SHADER_SAMPLED_READ_BIT, dstStageMask must include
VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT, or one of the
VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkImageMemoryBarrier2-dstAccessMask-03906
If dstAccessMask includes VK_ACCESS_2_SHADER_STORAGE_READ_BIT, dstStageMask must include
VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT, or one of the
VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkImageMemoryBarrier2-dstAccessMask-03907
If dstAccessMask includes VK_ACCESS_2_SHADER_WRITE_BIT, dstStageMask must include
VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT, or one of the
VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkImageMemoryBarrier2-dstAccessMask-03908
If dstAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT, dstStageMask must include
VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkImageMemoryBarrier2-dstAccessMask-03909
If dstAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT, dstStageMask must include
VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkImageMemoryBarrier2-dstAccessMask-03910
If dstAccessMask includes VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT, dstStageMask must include
VK_PIPELINE_STAGE_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkImageMemoryBarrier2-dstAccessMask-03911
If dstAccessMask includes VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT, dstStageMask must include
VK_PIPELINE_STAGE_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT, or
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT

• VUID-VkImageMemoryBarrier2-dstAccessMask-03912
If dstAccessMask includes VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT, dstStageMask
• VUID-VkImageMemoryBarrier2-dstAccessMask-03913
  If \( \text{dstAccessMask} \) includes \( VK\_ACCESS\_2\_DEPTH\_STENCIL\_ATTACHMENT\_WRITE\_BIT \), \( \text{dstStageMask} \) must include
  \( VK\_PIPELINE\_STAGE\_2\_EARLY\_FRAGMENT\_TESTS\_BIT, VK\_PIPELINE\_STAGE\_2\_LATE\_FRAGMENT\_TESTS\_BIT, VK\_PIPELINE\_STAGE\_2\_ALL\_GRAPHICS\_BIT, \) or
  \( VK\_PIPELINE\_STAGE\_2\_ALL\_COMMANDS\_BIT \)

• VUID-VkImageMemoryBarrier2-dstAccessMask-03914
  If \( \text{dstAccessMask} \) includes \( VK\_ACCESS\_2\_TRANSFER\_READ\_BIT \), \( \text{dstStageMask} \) must include
  \( VK\_PIPELINE\_STAGE\_2\_COPY\_BIT, VK\_PIPELINE\_STAGE\_2\_BLIT\_BIT, VK\_PIPELINE\_STAGE\_2\_RESOLVE\_BIT, VK\_PIPELINE\_STAGE\_2\_CLEAR\_BIT, VK\_PIPELINE\_STAGE\_2\_ALL\_TRANSFER\_BIT, VK\_PIPELINE\_STAGE\_2\_ACCELERATION\_STRUCTURE\_BUILD\_BIT\_KHR, \) or
  \( VK\_PIPELINE\_STAGE\_2\_ALL\_COMMANDS\_BIT \)

• VUID-VkImageMemoryBarrier2-dstAccessMask-03915
  If \( \text{dstAccessMask} \) includes \( VK\_ACCESS\_2\_TRANSFER\_WRITE\_BIT \), \( \text{dstStageMask} \) must include
  \( VK\_PIPELINE\_STAGE\_2\_COPY\_BIT, VK\_PIPELINE\_STAGE\_2\_BLIT\_BIT, VK\_PIPELINE\_STAGE\_2\_RESOLVE\_BIT, VK\_PIPELINE\_STAGE\_2\_CLEAR\_BIT, VK\_PIPELINE\_STAGE\_2\_ALL\_TRANSFER\_BIT, VK\_PIPELINE\_STAGE\_2\_ACCELERATION\_STRUCTURE\_BUILD\_BIT\_KHR, \) or
  \( VK\_PIPELINE\_STAGE\_2\_ALL\_COMMANDS\_BIT \)

• VUID-VkImageMemoryBarrier2-dstAccessMask-03916
  If \( \text{dstAccessMask} \) includes \( VK\_ACCESS\_2\_HOST\_READ\_BIT \), \( \text{dstStageMask} \) must include
  \( VK\_PIPELINE\_STAGE\_2\_HOST\_BIT \)

• VUID-VkImageMemoryBarrier2-dstAccessMask-03917
  If \( \text{dstAccessMask} \) includes \( VK\_ACCESS\_2\_HOST\_WRITE\_BIT \), \( \text{dstStageMask} \) must include
  \( VK\_PIPELINE\_STAGE\_2\_HOST\_BIT \)

• VUID-VkImageMemoryBarrier2-subresourceRange-01486
  \( \text{subresourceRange.baseMipLevel} \) must be less than the \( \text{mipLevels} \) specified in \( \text{VkImageCreateInfo} \) when image was created

• VUID-VkImageMemoryBarrier2-subresourceRange-01724
  If \( \text{subresourceRange.levelCount} \) is not \( VK\_REMAINING\_MIP\_LEVELS \), \( \text{subresourceRange.baseMipLevel} + \text{subresourceRange.levelCount} \) must be less than or equal to the \( \text{mipLevels} \) specified in \( \text{VkImageCreateInfo} \) when image was created

• VUID-VkImageMemoryBarrier2-subresourceRange-01488
  \( \text{subresourceRange.baseArrayLayer} \) must be less than the \( \text{arrayLayers} \) specified in \( \text{VkImageCreateInfo} \) when image was created

• VUID-VkImageMemoryBarrier2-subresourceRange-01725
  If \( \text{subresourceRange.layerCount} \) is not \( VK\_REMAINING\_ARRAY\_LAYERS \), \( \text{subresourceRange.baseArrayLayer} + \text{subresourceRange.layerCount} \) must be less than or equal to the \( \text{arrayLayers} \) specified in \( \text{VkImageCreateInfo} \) when image was created

• VUID-VkImageMemoryBarrier2-image-01932
  If image is non-sparse then it must be bound completely and contiguously to a single
VkDeviceMemory object

- VUID-VkImageMemoryBarrier2-oldLayout-01208
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` then image must have been created with `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`

- VUID-VkImageMemoryBarrier2-oldLayout-01209
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `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`

- VUID-VkImageMemoryBarrier2-oldLayout-01210
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `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`

- VUID-VkImageMemoryBarrier2-oldLayout-01211
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `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`

- VUID-VkImageMemoryBarrier2-oldLayout-01212
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` then image must have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT`

- VUID-VkImageMemoryBarrier2-oldLayout-01213
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` then image must have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT`

- VUID-VkImageMemoryBarrier2-newLayout-01197
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, oldLayout must be `VK_IMAGE_LAYOUT_UNDEFINED` or the current layout of the image subresources affected by the barrier

- VUID-VkImageMemory Barrier2-oldLayout-01198
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, newLayout must not be `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`

- VUID-VkImageMemoryBarrier2-oldLayout-01658
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL` then image must have been
• VUID-VkImageMemoryBarrier2-oldLayout-01659
  If srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define an image layout transition, and oldLayout or newLayout is VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL then image must have been created with VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT

• VUID-VkImageMemoryBarrier2-oldLayout-01660
  If srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define an image layout transition, and oldLayout or newLayout is VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL then image must have been created with at least one of VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, VK_IMAGE_USAGE_SAMPLED_BIT, or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT

• VUID-VkImageMemoryBarrier2-sync2-06911
  If the synchronization2 feature is not enabled, layout must not be VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL_KHR or VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR

• VUID-VkImageMemoryBarrier2-synchronization2-06911
  If the synchronization2 feature is not enabled, layout must not be VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL_KHR or VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR

• VUID-VkImageMemoryBarrier2-srcQueueFamilyIndex-03938
  If srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define an image layout transition, and oldLayout or newLayout is VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, image must have been created with VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT or VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT

• VUID-VkImageMemoryBarrier2-srcQueueFamilyIndex-03939
  If srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define an image layout transition, and oldLayout or newLayout is VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, image must have been created with at least one of VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, VK_IMAGE_USAGE_SAMPLED_BIT, or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT

• VUID-VkImageMemoryBarrier2-image-01671
  If image has a single-plane color format or is not disjoint, then the aspectMask member of
subresourceRange must be VK_IMAGE_ASPECT_COLOR_BIT

• VUID-VkImageMemoryBarrier2-image-01672
  If image has a multi-planar format and the image is disjoint, then the aspectMask member of subresourceRange must include either at least one of VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, and VK_IMAGE_ASPECT_PLANE_2_BIT; or must include VK_IMAGE_ASPECT_COLOR_BIT

• VUID-VkImageMemoryBarrier2-image-01673
  If image has a multi-planar format with only two planes, then the aspectMask member of subresourceRange must not include VK_IMAGE_ASPECT_PLANE_2_BIT

• VUID-VkImageMemoryBarrier2-image-03319
  If image has a depth/stencil format with both depth and stencil and the separateDepthStencilLayouts feature is enabled, then the aspectMask member of subresourceRange must include either or both VK_IMAGE_ASPECT_DEPTH_BIT and VK_IMAGE_ASPECT_STENCIL_BIT

• VUID-VkImageMemoryBarrier2-image-03320
  If image has a depth/stencil format with both depth and stencil and the separateDepthStencilLayouts feature is not enabled, then the aspectMask member of subresourceRange must include both VK_IMAGE_ASPECT_DEPTH_BIT and VK_IMAGE_ASPECT_STENCIL_BIT

• VUID-VkImageMemoryBarrier2-srcQueueFamilyIndex-04070
  If srcQueueFamilyIndex is not equal to dstQueueFamilyIndex, at least one must not be a special queue family reserved for external memory ownership transfers, as described in Queue Family Ownership Transfer

• VUID-VkImageMemoryBarrier2-image-04071
  If image was created with a sharing mode of VK_SHARING_MODE_CONCURRENT, srcQueueFamilyIndex and dstQueueFamilyIndex are not equal, and one of srcQueueFamilyIndex and dstQueueFamilyIndex is one of the special queue family values reserved for external memory transfers, the other must be VK_QUEUE_FAMILY_IGNORED

• VUID-VkImageMemoryBarrier2-srcStageMask-03854
  If either srcStageMask or dstStageMask includes VK_PIPELINE_STAGE_2_HOST_BIT, srcQueueFamilyIndex and dstQueueFamilyIndex must be equal

• VUID-VkImageMemoryBarrier2-srcStageMask-03855
  If srcStageMask includes VK_PIPELINE_STAGE_2_HOST_BIT, and srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define an image layout transition, oldLayout must be one of VK_IMAGE_LAYOUT_PREINITIALIZED, VK_IMAGE_LAYOUT_UNDEFINED, or VK_IMAGE_LAYOUT_GENERAL
**Valid Usage (Implicit)**

- **VUID-VkImageMemoryBarrier2-sType-sType**
  
  *sType* **must** be `VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER_2`

- **VUID-VkImageMemoryBarrier2-pNext-pNext**
  
  *pNext* **must** be `NULL`

- **VUID-VkImageMemoryBarrier2-srcStageMask-parameter**
  
  *srcStageMask* **must** be a valid combination of `VkPipelineStageFlagBits2` values

- **VUID-VkImageMemoryBarrier2-srcAccessMask-parameter**
  
  *srcAccessMask* **must** be a valid combination of `VkAccessFlagBits2` values

- **VUID-VkImageMemoryBarrier2-dstStageMask-parameter**
  
  *dstStageMask* **must** be a valid combination of `VkPipelineStageFlagBits2` values

- **VUID-VkImageMemoryBarrier2-dstAccessMask-parameter**
  
  *dstAccessMask* **must** be a valid combination of `VkAccessFlagBits2` values

- **VUID-VkImageMemoryBarrier2-oldLayout-parameter**
  
  *oldLayout* **must** be a valid `VkImageLayout` value

- **VUID-VkImageMemoryBarrier2-newLayout-parameter**
  
  *newLayout* **must** be a valid `VkImageLayout` value

- **VUID-VkImageMemoryBarrier2-image-parameter**
  
  *image* **must** be a valid `VkImage` handle

- **VUID-VkImageMemoryBarrier2-subresourceRange-parameter**
  
  *subresourceRange* **must** be a valid `VkImageSubresourceRange` structure

The `VkImageMemoryBarrier` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkImageMemoryBarrier {
    VkStructureType sType;
    const void* pNext;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    VkImageLayout oldLayout;
    VkImageLayout newLayout;
    uint32_t srcQueueFamilyIndex;
    uint32_t dstQueueFamilyIndex;
    VkImage image;
    VkImageSubresourceRange subresourceRange;
} VkImageMemoryBarrier;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **srcAccessMask** is a bitmask of `VkAccessFlagBits` specifying a source access mask.
• `dstAccessMask` is a bitmask of `VkAccessFlagBits` specifying a destination access mask.

• `oldLayout` is the old layout in an image layout transition.

• `newLayout` is the new layout in an image layout transition.

• `srcQueueFamilyIndex` is the source queue family for a queue family ownership transfer.

• `dstQueueFamilyIndex` is the destination queue family for a queue family ownership transfer.

• `image` is a handle to the image affected by this barrier.

• `subresourceRange` describes the image subresource range within `image` that is affected by this barrier.

The first access scope is limited to access to memory through the specified image subresource range, via access types in the source access mask specified by `srcAccessMask`. If `srcAccessMask` includes `VK_ACCESS_HOST_WRITE_BIT`, memory writes performed by that access type are also made visible, as that access type is not performed through a resource.

The second access scope is limited to access to memory through the specified image subresource range, via access types in the destination access mask specified by `dstAccessMask`. If `dstAccessMask` includes `VK_ACCESS_HOST_WRITE_BIT` or `VK_ACCESS_HOST_READ_BIT`, available memory writes are also made visible to accesses of those types, as those access types are not performed through a resource.

If `srcQueueFamilyIndex` is not equal to `dstQueueFamilyIndex`, and `srcQueueFamilyIndex` is equal to the current queue family, then the memory barrier defines a queue family release operation for the specified image subresource range, and the second access scope includes no access, as if `dstAccessMask` was 0.

If `dstQueueFamilyIndex` is not equal to `srcQueueFamilyIndex`, and `dstQueueFamilyIndex` is equal to the current queue family, then the memory barrier defines a queue family acquire operation for the specified image subresource range, and the first access scope includes no access, as if `srcAccessMask` was 0.

If the synchronization2 feature is not enabled or `oldLayout` is not equal to `newLayout`, `oldLayout` and `newLayout` define an image layout transition for the specified image subresource range.

Note

If the synchronization2 feature is enabled, when the old and new layout are equal, the layout values are ignored - data is preserved no matter what values are specified, or what layout the image is currently in.

If `image` has a multi-planar format and the image is disjoint, then including `VK_IMAGE_ASPECT_COLOR_BIT` in the `aspectMask` member of `subresourceRange` is equivalent to including `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, and (for three-plane formats only) `VK_IMAGE_ASPECT_PLANE_2_BIT`.

Valid Usage

• VUID-VkImageMemoryBarrier-subresourceRange-01486

  `subresourceRange.baseMipLevel` must be less than the `mipLevels` specified in
• VUID-VkImageMemoryBarrier-subresourceRange-01724
  If subresourceRange.levelCount is not VK_REMAINING_MIP_LEVELS, subresourceRange.baseMipLevel + subresourceRange.levelCount must be less than or equal to the mipLevels specified in VkImageCreateInfo when image was created

• VUID-VkImageMemoryBarrier-subresourceRange-01488
  subresourceRange.baseArrayLayer must be less than the arrayLayers specified in VkImageCreateInfo when image was created

• VUID-VkImageMemoryBarrier-subresourceRange-01725
  If subresourceRange.layerCount is not VK_REMAINING_ARRAY_LAYERS, subresourceRange.baseArrayLayer + subresourceRange.layerCount must be less than or equal to the arrayLayers specified in VkImageCreateInfo when image was created

• VUID-VkImageMemoryBarrier-image-01932
  If image is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

• VUID-VkImageMemoryBarrier-oldLayout-01208
  If srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define an image layout transition, and oldLayout or newLayout is VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL then image must have been created with VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT

• VUID-VkImageMemoryBarrier-oldLayout-01209
  If srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define an image layout transition, and 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

• VUID-VkImageMemoryBarrier-oldLayout-01210
  If srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define an image layout transition, and 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

• VUID-VkImageMemoryBarrier-oldLayout-01211
  If srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define an image layout transition, and 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

• VUID-VkImageMemoryBarrier-oldLayout-01212
  If srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define an image layout transition, and oldLayout or newLayout is VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL then image must have been created with VK_IMAGE_USAGE_TRANSFER_SRC_BIT
VK_IMAGE_USAGE_TRANSFER_DST_BIT

- VUID-VkImageMemoryBarrier-oldLayout-01197
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, `oldLayout` must be `VK_IMAGE_LAYOUT_UNDEFINED` or the current layout of the image subresources affected by the barrier.

- VUID-VkImageMemoryBarrier-newLayout-01198
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, `newLayout` must not be `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`.

- VUID-VkImageMemoryBarrier-oldLayout-01658
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL` then image must have been created with `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`.

- VUID-VkImageMemoryBarrier-oldLayout-01659
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL` then image must have been created with `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`.

- VUID-VkImageMemoryBarrier-srcQueueFamilyIndex-04065
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL` then image must have been created with at least one of `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_SAMPLED_BIT`, or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`.

- VUID-VkImageMemoryBarrier-srcQueueFamilyIndex-04066
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL` then image must have been created with at least one of `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_SAMPLED_BIT`, or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`.

- VUID-VkImageMemoryBarrier-srcQueueFamilyIndex-04067
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL` then image must have been created with at least one of `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_SAMPLED_BIT`, or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`.

- VUID-VkImageMemoryBarrier-srcQueueFamilyIndex-04068
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL` then image must have been created with `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT` set.

- VUID-VkImageMemoryBarrier-synchronization2-06911
  If the `synchronization2` feature is not enabled, layout must not be...
VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL_KHR or VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR

• VUID-VkImageMemoryBarrier-srcQueueFamilyIndex-03938
  If srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define an image layout transition, and oldLayout or newLayout is VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL, image must have been created with VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT or VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT

• VUID-VkImageMemoryBarrier-srcQueueFamilyIndex-03939
  If srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define an image layout transition, and oldLayout or newLayout is VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL, image must have been created with at least one of VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, VK_IMAGE_USAGE_SAMPLED_BIT, or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT

• VUID-VkImageMemoryBarrier-image-01671
  If image has a single-plane color format or is not disjoint, then the aspectMask member of subresourceRange must be VK_IMAGE_ASPECT_COLOR_BIT

• VUID-VkImageMemoryBarrier-image-01672
  If image has a multi-planar format and the image is disjoint, then the aspectMask member of subresourceRange must include either at least one of VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, and VK_IMAGE_ASPECT_PLANE_2_BIT; or must include VK_IMAGE_ASPECT_COLOR_BIT

• VUID-VkImageMemoryBarrier-image-01673
  If image has a multi-planar format with only two planes, then the aspectMask member of subresourceRange must not include VK_IMAGE_ASPECT_PLANE_2_BIT

• VUID-VkImageMemoryBarrier-image-03319
  If image has a depth/stencil format with both depth and stencil and the separateDepthStencilLayouts feature is enabled, then the aspectMask member of subresourceRange must include either or both VK_IMAGE_ASPECT_DEPTH_BIT and VK_IMAGE_ASPECT_STENCIL_BIT

• VUID-VkImageMemoryBarrier-image-03320
  If image has a depth/stencil format with both depth and stencil and the separateDepthStencilLayouts feature is not enabled, then the aspectMask member of subresourceRange must include both VK_IMAGE_ASPECT_DEPTH_BIT and VK_IMAGE_ASPECT_STENCIL_BIT

• VUID-VkImageMemoryBarrier-srcQueueFamilyIndex-04070
  If srcQueueFamilyIndex is not equal to dstQueueFamilyIndex, at least one must not be a special queue family reserved for external memory ownership transfers, as described in Queue Family Ownership Transfer

• VUID-VkImageMemoryBarrier-image-04071
  If image was created with a sharing mode of VK_SHARING_MODE_CONCURRENT, srcQueueFamilyIndex and dstQueueFamilyIndex are not equal, and one of srcQueueFamilyIndex and dstQueueFamilyIndex is one of the special queue family values reserved for external memory transfers, the other must be VK_QUEUE_FAMILY_IGNORED

• VUID-VkImageMemoryBarrier-image-04072
  If image was created with a sharing mode of VK_SHARING_MODE_EXCLUSIVE, and
**srcQueueFamilyIndex** and **dstQueueFamilyIndex** are not equal, **srcQueueFamilyIndex** and **dstQueueFamilyIndex** must both be valid queue families, or one of the special queue family values reserved for external memory transfers, as described in **Queue Family Ownership Transfer**

- **VUID-VkImageMemoryBarrier-synchronization2-03857**
  If the synchronization2 feature is not enabled, and **image** was created with a sharing mode of **VK_SHARING_MODE_CONCURRENT**, at least one of **srcQueueFamilyIndex** and **dstQueueFamilyIndex** must be **VK_QUEUE_FAMILY_IGNORED**

---

**Valid Usage (Implicit)**

- **VUID-VkImageMemoryBarrier-sType-sType**
  **sType** must be **VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER**

- **VUID-VkImageMemoryBarrier-pNext-pNext**
  **pNext** must be **NULL**

- **VUID-VkImageMemoryBarrier-oldLayout-parameter**
  **oldLayout** must be a valid **VkImageLayout** value

- **VUID-VkImageMemoryBarrier-newLayout-parameter**
  **newLayout** must be a valid **VkImageLayout** value

- **VUID-VkImageMemoryBarrier-image-parameter**
  **image** must be a valid **VkImage** handle

- **VUID-VkImageMemoryBarrier-subresourceRange-parameter**
  **subresourceRange** must be a valid **VkImageSubresourceRange** structure

---

**7.7.4. Queue Family Ownership Transfer**

Resources created with a **VkSharingMode** of **VK_SHARING_MODE_EXCLUSIVE** must have their ownership explicitly transferred from one queue family to another in order to access their content in a well-defined manner on a queue in a different queue family.

The special queue family index **VK_QUEUE_FAMILY_IGNORED** indicates that a queue family parameter or member is ignored.

```c
#define VK_QUEUE_FAMILY_IGNORED           (~0U)
```

Resources shared with external APIs or instances using external memory must also explicitly manage ownership transfers between local and external queues (or equivalent constructs in external APIs) regardless of the **VkSharingMode** specified when creating them.

The special queue family index **VK_QUEUE_FAMILY_EXTERNAL** represents any queue external to the resource's current Vulkan instance, as long as the queue uses the same underlying device group or physical device, and the same driver version as the resource's **VkDevice**, as indicated by **VkPhysicalDeviceIDProperties::deviceUUID** and **VkPhysicalDeviceIDProperties::driverUUID**.
If memory dependencies are correctly expressed between uses of such a resource between two queues in different families, but no ownership transfer is defined, the contents of that resource are undefined for any read accesses performed by the second queue family.

**Note**

If an application does not need the contents of a resource to remain valid when transferring from one queue family to another, then the ownership transfer **should** be skipped.

A queue family ownership transfer consists of two distinct parts:

1. Release exclusive ownership from the source queue family
2. Acquire exclusive ownership for the destination queue family

An application **must** ensure that these operations occur in the correct order by defining an execution dependency between them, e.g. using a semaphore.

A **release operation** is used to release exclusive ownership of a range of a buffer or image subresource range. A release operation is defined by executing a buffer memory barrier (for a buffer range) or an image memory barrier (for an image subresource range) using a pipeline barrier command, on a queue from the source queue family. The **srcQueueFamilyIndex** parameter of the barrier **must** be set to the source queue family index, and the **dstQueueFamilyIndex** parameter to the destination queue family index. **dstAccessMask** is ignored for such a barrier, such that no visibility operation is executed - the value of this mask does not affect the validity of the barrier. The release operation happens-after the availability operation, and happens-before operations specified in the second synchronization scope of the calling command.

An **acquire operation** is used to acquire exclusive ownership of a range of a buffer or image subresource range. An acquire operation is defined by executing a buffer memory barrier (for a buffer range) or an image memory barrier (for an image subresource range) using a pipeline barrier command, on a queue from the destination queue family. The buffer range or image subresource range specified in an acquire operation **must** match exactly that of a previous release operation. The **srcQueueFamilyIndex** parameter of the barrier **must** be set to the source queue family index, and the **dstQueueFamilyIndex** parameter to the destination queue family index. **srcAccessMask** is ignored for such a barrier, such that no availability operation is executed - the value of this mask does not affect the validity of the barrier. The acquire operation happens-after operations in the first synchronization scope of the calling command, and happens-before the visibility operation.

**Note**

Whilst it is not invalid to provide destination or source access masks for memory barriers used for release or acquire operations, respectively, they have no practical effect. Access after a release operation has undefined results, and so visibility for those accesses has no practical effect. Similarly, write access before an acquire operation will produce undefined results for future access, so availability of those writes has no practical use. In an earlier version of the specification, these were

264
required to match on both sides - but this was subsequently relaxed. These masks \textbf{should} be set to 0.

If the transfer is via an image memory barrier, and an image layout transition is desired, then the values of \texttt{oldLayout} and \texttt{newLayout} in the \textit{release operation}'s memory barrier \textbf{must} be equal to values of \texttt{oldLayout} and \texttt{newLayout} in the \textit{acquire operation}'s memory barrier. Although the image layout transition is submitted twice, it will only be executed once. A layout transition specified in this way happens-after the \textit{release operation} and happens-before the \textit{acquire operation}.

If the values of \texttt{srcQueueFamilyIndex} and \texttt{dstQueueFamilyIndex} are equal, no ownership transfer is performed, and the barrier operates as if they were both set to \texttt{VK_QUEUE_FAMILY_IGNORED}.

Queue family ownership transfers \textbf{may} perform read and write accesses on all memory bound to the image subresource or buffer range, so applications \textbf{must} ensure that all memory writes have been made \texttt{available} before a queue family ownership transfer is executed. Available memory is automatically made visible to queue family release and acquire operations, and writes performed by those operations are automatically made available.

Once a queue family has acquired ownership of a buffer range or image subresource range of a \texttt{VK_SHARING_MODE_EXCLUSIVE} resource, its contents are undefined to other queue families unless ownership is transferred. The contents of any portion of another resource which aliases memory that is bound to the transferred buffer or image subresource range are undefined after a release or acquire operation.

\begin{quote}
\textit{Note}

Because events \textbf{cannot} be used directly for inter-queue synchronization, and because \texttt{vkCmdSetEvent} does not have the queue family index or memory barrier parameters needed by a \textit{release operation}, the release and acquire operations of a queue family ownership transfer \textbf{can} only be performed using \texttt{vkCmdPipelineBarrier}.
\end{quote}

\section*{7.8. Wait Idle Operations}

To wait on the host for the completion of outstanding queue operations for a given queue, call:

\begin{verbatim}
// Provided by VK_VERSION_1_0
VkResult vkQueueWaitIdle(
    VkQueue queue);
\end{verbatim}

\begin{itemize}
\item \texttt{queue} is the queue on which to wait.
\end{itemize}

\texttt{vkQueueWaitIdle} is equivalent to having submitted a valid fence to every previously executed queue submission command that accepts a fence, then waiting for all of those fences to signal using \texttt{vkWaitForFences} with an infinite timeout and \texttt{waitAll} set to \texttt{VK_TRUE}.  


Valid Usage (Implicit)

- VUID-vkQueueWaitIdle-queue-parameter
  queue must be a valid VkQueue handle

Host Synchronization

- Host access to queue must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
</table>

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST

To wait on the host for the completion of outstanding queue operations for all queues on a given logical device, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkDeviceWaitIdle(
    VkDevice device);
```

- device is the logical device to idle.

vkDeviceWaitIdle is equivalent to calling vkQueueWaitIdle for all queues owned by device.

Valid Usage (Implicit)

- VUID-vkDeviceWaitIdle-device-parameter
  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

### 7.9. Host Write Ordering Guarantees

When batches of command buffers are submitted to a queue via a queue submission command, it defines a memory dependency with prior host operations, and execution of command buffers submitted to the queue.

The first **synchronization scope** is defined by the host execution model, but includes execution of `vkQueueSubmit` on the host and anything that happened-before it.

The second **synchronization scope** includes all commands submitted in the same queue submission, and all commands that occur later in submission order.

The first **access scope** includes all host writes to mappable device memory that are available to the host memory domain.

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

### 7.10. Synchronization and Multiple Physical Devices

If a logical device includes more than one physical device, then fences, semaphores, and events all still have a single instance of the signaled state.

A fence becomes signaled when all physical devices complete the necessary queue operations.

Semaphore wait and signal operations all include a device index that is the sole physical device that performs the operation. These indices are provided in the `VkDeviceGroupSubmitInfo` and `VkDeviceGroupBindSparseInfo` structures. Semaphores are not exclusively owned by any physical device. For example, a semaphore can be signaled by one physical device and then waited on by a different physical device.

An event **can** only be waited on by the same physical device that signaled it (or the host).
Chapter 8. Render Pass

Draw commands must be recorded within a render pass instance. Each render pass instance defines a set of image resources, referred to as attachments, used during rendering.

To begin a render pass instance, call:

```c
// Provided by VK_VERSION_1_3
void vkCmdBeginRendering(
    VkCommandBuffer commandBuffer, // commandBuffer is the command buffer in which to record the command.
    const VkRenderingInfo* pRenderingInfo); // pRenderingInfo is a pointer to a VkRenderingInfo structure specifying details of the render pass instance to begin.
```

After beginning a render pass instance, the command buffer is ready to record draw commands.

If pRenderingInfo->flags includes VK_RENDERING_RESUMING_BIT then this render pass is resumed from a render pass instance that has been suspended earlier in submission order.

Valid Usage

- VUID-vkCmdBeginRendering-dynamicRendering-06446 The dynamicRendering feature must be enabled.
- VUID-vkCmdBeginRendering-commandBuffer-06068 If commandBuffer is a secondary command buffer, pRenderingInfo->flags must not include VK_RENDERING_CONTENTS_SECONDARY_COMMAND_BUFFERS_BIT.

Valid Usage (Implicit)

- VUID-vkCmdBeginRendering-commandBuffer-parameter commandBuffer must be a valid VkCommandBuffer handle.
- VUID-vkCmdBeginRendering-pRenderingInfo-parameter pRenderingInfo must be a valid pointer to a valid VkRenderingInfo structure.
- VUID-vkCmdBeginRendering-commandBuffer-recording commandBuffer must be in the recording state.
- VUID-vkCmdBeginRendering-commandBuffer-cmdpool The VkCommandPool that commandBuffer was allocated from must support graphics operations.
- VUID-vkCmdBeginRendering-renderpass This command must only be called outside of a render pass instance.
Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td>Action State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The VkRenderingInfo structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkRenderingInfo {
    VkStructureType sType;
    const void* pNext;
    VkRenderingFlags flags;
    VkRect2D renderArea;
    uint32_t layerCount;
    uint32_t viewMask;
    uint32_t colorAttachmentCount;
    const VkRenderingAttachmentInfo* pColorAttachments;
    const VkRenderingAttachmentInfo* pDepthAttachment;
    const VkRenderingAttachmentInfo* pStencilAttachment;
} VkRenderingInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is a bitmask of VkRenderingFlagBits.
- **renderArea** is the render area that is affected by the render pass instance.
- **layerCount** is the number of layers rendered to in each attachment when **viewMask** is 0.
- **viewMask** is the view mask indicating the indices of attachment layers that will be rendered when it is not 0.
- **colorAttachmentCount** is the number of elements in **pColorAttachments**.
- **pColorAttachments** is a pointer to an array of colorAttachmentCount VkRenderingAttachmentInfo structures describing any color attachments used.
- **pDepthAttachment** is a pointer to a VkRenderingAttachmentInfo structure describing a depth attachment.
• pStencilAttachment is a pointer to a VkRenderingAttachmentInfo structure describing a stencil attachment.

If viewMask is not 0, multiview is enabled.

If there is an instance of VkDeviceGroupRenderPassBeginInfo included in the pNext chain and its deviceCount member is not 0, then renderArea is ignored, and the render area is defined per-device by that structure.

Each element of the pColorAttachments array corresponds to an output location in the shader, i.e. if the shader declares an output variable decorated with a Location value of X, then it uses the attachment provided in pColorAttachments[X]. If the imageView member of any element of pColorAttachments is VK_NULL_HANDLE, writes to the corresponding location by a fragment are discarded.

Valid Usage

• VUID-VkRenderingInfo-viewMask-06069
  If viewMask is 0, layerCount must not be 0

• VUID-VkRenderingInfo-multisampledRenderToSingleSampled-06857
  If none of the VK_AMD_mixed_attachment_samples extension, the VK_NV_framebuffer_mixed_samples extension, or the multisampledRenderToSingleSampled feature are enabled, imageView members of pDepthAttachment, pStencilAttachment, and elements of pColorAttachments that are not VK_NULL_HANDLE must have been created with the same sampleCount

• VUID-VkRenderingInfo-pNext-06077
  If the pNext chain does not contain VkDeviceGroupRenderPassBeginInfo or its deviceRenderAreaCount member is equal to 0, renderArea.offset.x must be greater than or equal to 0

• VUID-VkRenderingInfo-pNext-06078
  If the pNext chain does not contain VkDeviceGroupRenderPassBeginInfo or its deviceRenderAreaCount member is equal to 0, renderArea.offset.y must be greater than or equal to 0

• VUID-VkRenderingInfo-pNext-06079
  If the pNext chain does not contain VkDeviceGroupRenderPassBeginInfo or its deviceRenderAreaCount member is equal to 0, the width of the imageView member of any element of pColorAttachments, pDepthAttachment, or pStencilAttachment that is not VK_NULL_HANDLE must be greater than or equal to renderArea.offset.x + renderArea.extent.width

• VUID-VkRenderingInfo-pNext-06080
  If the pNext chain does not contain VkDeviceGroupRenderPassBeginInfo or its deviceRenderAreaCount member is equal to 0, the height of the imageView member of any element of pColorAttachments, pDepthAttachment, or pStencilAttachment that is not VK_NULL_HANDLE must be greater than or equal to renderArea.offset.y + renderArea.extent.height

• VUID-VkRenderingInfo-pNext-06083
If the `pNext` chain contains `VkDeviceGroupRenderPassBeginInfo`, the width of the `imageView` member of any element of `pColorAttachments`, `pDepthAttachment`, or `pStencilAttachment` that is not `VK_NULL_HANDLE` must be greater than or equal to the sum of the `offset.x` and `extent.width` members of each element of `pDeviceRenderAreas`.

- **VUID-VkRenderingInfo-pNext-06084**
  If the `pNext` chain contains `VkDeviceGroupRenderPassBeginInfo`, the height of the `imageView` member of any element of `pColorAttachments`, `pDepthAttachment`, or `pStencilAttachment` that is not `VK_NULL_HANDLE` must be greater than or equal to the sum of the `offset.y` and `extent.height` members of each element of `pDeviceRenderAreas`.

- **VUID-VkRenderingInfo-pNext-06085**
  If neither `pDepthAttachment` or `pStencilAttachment` are `NULL` and the `imageView` member of either structure is not `VK_NULL_HANDLE`, the `imageView` member of each structure must be the same.

- **VUID-VkRenderingInfo-pNext-06086**
  If neither `pDepthAttachment` or `pStencilAttachment` are `NULL`, and the `resolveMode` member of each is not `VK_RESOLVE_MODE_NONE`, the `resolveImageView` member of each structure must be the same.

- **VUID-VkRenderingInfo-colorAttachmentCount-06087**
  If `colorAttachmentCount` is not `0` and the `imageView` member of an element of `pColorAttachments` is not `VK_NULL_HANDLE`, that `imageView` must have been created with `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`.

- **VUID-VkRenderingInfo-pDepthAttachment-06547**
  If `pDepthAttachment` is not `NULL` and `pDepthAttachment->imageView` is not `VK_NULL_HANDLE`, `pDepthAttachment->imageView` must have been created with a format that includes a depth aspect.

- **VUID-VkRenderingInfo-pDepthAttachment-06088**
  If `pDepthAttachment` is not `NULL` and `pDepthAttachment->imageView` is not `VK_NULL_HANDLE`, `pDepthAttachment->imageView` must have been created with `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`.

- **VUID-VkRenderingInfo-pStencilAttachment-06548**
  If `pStencilAttachment` is not `NULL` and `pStencilAttachment->imageView` is not `VK_NULL_HANDLE`, `pStencilAttachment->imageView` must have been created with a format that includes a stencil aspect.

- **VUID-VkRenderingInfo-pStencilAttachment-06089**
  If `pStencilAttachment` is not `NULL` and `pStencilAttachment->imageView` is not `VK_NULL_HANDLE`, `pStencilAttachment->imageView` must have been created with a stencil usage including `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`.

- **VUID-VkRenderingInfo-colorAttachmentCount-06090**
  If `colorAttachmentCount` is not `0` and the `imageView` member of an element of `pColorAttachments` is not `VK_NULL_HANDLE`, the layout member of that element of `pColorAttachments` must not be `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`.

- **VUID-VkRenderingInfo-colorAttachmentCount-06091**
  If `colorAttachmentCount` is not `0` and the `imageView` member of an element of
pColorAttachments is not VK_NULL_HANDLE, if the resolveMode member of that element of pColorAttachments is not VK_RESOLVE_MODE_NONE, its resolveImageLayout member must not be VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL.

- VUID-VkRenderingInfo-pDepthAttachment-06092
  If pDepthAttachment is not NULL and pDepthAttachment->imageView is not VK_NULL_HANDLE, pDepthAttachment->layout must not be VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL.

- VUID-VkRenderingInfo-pDepthAttachment-06093
  If pDepthAttachment is not NULL, pDepthAttachment->imageView is not VK_NULL_HANDLE, and pDepthAttachment->resolveMode is not VK_RESOLVE_MODE_NONE, pDepthAttachment->resolveImageLayout must not be VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL.

- VUID-VkRenderingInfo-pStencilAttachment-06094
  If pStencilAttachment is not NULL and pStencilAttachment->imageView is not VK_NULL_HANDLE, pStencilAttachment->layout must not be VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL.

- VUID-VkRenderingInfo-pStencilAttachment-06095
  If pStencilAttachment is not NULL, pStencilAttachment->imageView is not VK_NULL_HANDLE, and pStencilAttachment->resolveMode is not VK_RESOLVE_MODE_NONE, pStencilAttachment->resolveImageLayout must not be VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL.

- VUID-VkRenderingInfo-colorAttachmentCount-06096
  If colorAttachmentCount is not 0 and the imageView member of an element of pColorAttachments is not VK_NULL_HANDLE, the layout member of that element of pColorAttachments must not be VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL.

- VUID-VkRenderingInfo-colorAttachmentCount-06097
  If colorAttachmentCount is not 0 and the imageView member of an element of pColorAttachments is not VK_NULL_HANDLE, if the resolveMode member of that element of pColorAttachments is not VK_RESOLVE_MODE_NONE, its resolveImageLayout member must not be VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL.

- VUID-VkRenderingInfo-pDepthAttachment-06098
  If pDepthAttachment is not NULL, pDepthAttachment->imageView is not VK_NULL_HANDLE, and pDepthAttachment->resolveMode is not VK_RESOLVE_MODE_NONE, pDepthAttachment->resolveImageLayout must not be VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL.

- VUID-VkRenderingInfo-pStencilAttachment-06099
  If pStencilAttachment is not NULL, pStencilAttachment->imageView is not VK_NULL_HANDLE, and pStencilAttachment->resolveMode is not VK_RESOLVE_MODE_NONE, pStencilAttachment->resolveImageLayout must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL.

- VUID-VkRenderingInfo-colorAttachmentCount-06100
  If colorAttachmentCount is not 0 and the imageView member of an element of pColorAttachments is not VK_NULL_HANDLE, the layout member of that element of
pColorAttachments must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL

• VUID-VkRenderingInfo-colorAttachmentCount-06101
  If colorAttachmentCount is not 0 and the imageView member of an element of pColorAttachments is not VK_NULL_HANDLE, if the resolveMode member of that element of pColorAttachments is not VK_RESOLVE_MODE_NONE, its resolveImageLayout member must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL

• VUID-VkRenderingInfo-pDepthAttachment-07732
  If pDepthAttachment is not NULL and pDepthAttachment->imageView is not VK_NULL_HANDLE, pDepthAttachment->layout must not be VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL

• VUID-VkRenderingInfo-pDepthAttachment-07733
  If pDepthAttachment is not NULL, pDepthAttachment->imageView is not VK_NULL_HANDLE, and pDepthAttachment->resolveMode is not VK_RESOLVE_MODE_NONE, pDepthAttachment->resolveImageLayout must not be VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL

• VUID-VkRenderingInfo-pStencilAttachment-07734
  If pStencilAttachment is not NULL and pStencilAttachment->imageView is not VK_NULL_HANDLE, pStencilAttachment->layout must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL

• VUID-VkRenderingInfo-pStencilAttachment-07735
  If pStencilAttachment is not NULL, pStencilAttachment->imageView is not VK_NULL_HANDLE, and pStencilAttachment->resolveMode is not VK_RESOLVE_MODE_NONE, pStencilAttachment->resolveImageLayout must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL

• VUID-VkRenderingInfo-pDepthAttachment-06102
  If pDepthAttachment or pStencilAttachment are both not NULL, pDepthAttachment->imageView and pStencilAttachment->imageView are both not VK_NULL_HANDLE, pDepthAttachment->resolveMode must be one of the bits set in VkPhysicalDeviceDepthStencilResolveProperties::supportedDepthResolveModes

• VUID-VkRenderingInfo-pStencilAttachment-06103
  If pStencilAttachment or pStencilAttachment->imageView are both not NULL, pStencilAttachment->resolveMode must be one of the bits set in VkPhysicalDeviceDepthStencilResolveProperties::supportedStencilResolveModes

• VUID-VkRenderingInfo-pDepthAttachment-06104
  If pDepthAttachment or pStencilAttachment are both not NULL, pDepthAttachment->imageView and pStencilAttachment->imageView are both not VK_NULL_HANDLE, and VkPhysicalDeviceDepthStencilResolveProperties::independentResolveNone is VK_FALSE, the resolveMode of both structures must be the same value

• VUID-VkRenderingInfo-pDepthAttachment-06105
  If pDepthAttachment or pStencilAttachment are both not NULL, pDepthAttachment->imageView and pStencilAttachment->imageView are both not VK_NULL_HANDLE,
VkPhysicalDeviceDepthStencilResolveProperties::independentResolve is VK_FALSE, and the resolveMode of neither structure is VK_RESOLVE_MODE_NONE, the resolveMode of both structures must be the same value

- VUID-VkRenderingInfo-colorAttachmentCount-06106
  colorAttachmentCount must be less than or equal to VkPhysicalDeviceLimits::maxColorAttachments

- VUID-VkRenderingInfo-multiview-06127
  If the multiview feature is not enabled, viewMask must be 0

- VUID-VkRenderingInfo-viewMask-06128
  The index of the most significant bit in viewMask must be less than maxMultiviewViewCount

Valid Usage (Implicit)

- VUID-VkRenderingInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_RENDERING_INFO

- VUID-VkRenderingInfo-pNext-pNext
  pNext must be NULL or a pointer to a valid instance of VkDeviceGroupRenderPassBeginInfo

- VUID-VkRenderingInfo-sType-unique
  The sType value of each struct in the pNext chain must be unique

- VUID-VkRenderingInfo-flags-parameter
  flags must be a valid combination of VkRenderingFlagBits values

- VUID-VkRenderingInfo-pColorAttachments-parameter
  If colorAttachmentCount is not 0, pColorAttachments must be a valid pointer to an array of colorAttachmentCount valid VkRenderingAttachmentInfo structures

- VUID-VkRenderingInfo-pDepthAttachment-parameter
  If pDepthAttachment is not NULL, pDepthAttachment must be a valid pointer to a valid VkRenderingAttachmentInfo structure

- VUID-VkRenderingInfo-pStencilAttachment-parameter
  If pStencilAttachment is not NULL, pStencilAttachment must be a valid pointer to a valid VkRenderingAttachmentInfo structure

Bits which can be set in VkRenderingInfo::flags describing additional properties of the render pass are:

```c
// Provided by VK_VERSION_1_3
typedef enum VkRenderingFlagBits {
    VK_RENDERING_CONTENTS_SECONDARY_COMMAND_BUFFERS_BIT = 0x00000001,
    VK_RENDERING_SUSPENDING_BIT = 0x00000002,
    VK_RENDERING_RESUMING_BIT = 0x00000004,
    VK_RENDERING_CONTENTS_SECONDARY_COMMAND_BUFFERS_BIT_KHR = VK_RENDERING_CONTENTS_SECONDARY_COMMAND_BUFFERS_BIT,
    VK_RENDERING_SUSPENDING_BIT_KHR = VK_RENDERING_SUSPENDING_BIT,
    VK_RENDERING_RESUMING_BIT_KHR = VK_RENDERING_RESUMING_BIT,
```
• **VK_RENDERING_CONTENTS_SECONDARY_COMMAND_BUFFERS_BIT** specifies that draw calls for the render pass instance will be recorded in secondary command buffers.

• **VK_RENDERING_RESUMING_BIT** specifies that the render pass instance is resuming an earlier suspended render pass instance.

• **VK_RENDERING_SUSPENDING_BIT** specifies that the render pass instance will be suspended.

The contents of `pRenderingInfo` must match between suspended render pass instances and the render pass instances that resume them, other than the presence or absence of the **VK_RENDERING_RESUMING_BIT, VK_RENDERING_SUSPENDING_BIT, and VK_RENDERING_CONTENTS_SECONDARY_COMMAND_BUFFERS_BIT** flags. No action or synchronization commands, or other render pass instances, are allowed between suspending and resuming render pass instances.

```c
// Provided by VK_VERSION_1_3
typedef VkFlags VkRenderingFlags;
```

**VkRenderingFlags** is a bitmask type for setting a mask of zero or more **VkRenderingFlagBits**.

The **VkRenderingAttachmentInfo** structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkRenderingAttachmentInfo {
    VkStructureType sType;
    const void* pNext;
    VkImageView imageView;
    VkImageLayout imageLayout;
    VkResolveModeFlagBits resolveMode;
    VkImageView resolveImageView;
    VkImageLayout resolveImageLayout;
    VkAttachmentLoadOp loadOp;
    VkAttachmentStoreOp storeOp;
    VkClearValue clearValue;
} VkRenderingAttachmentInfo;
```

• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.

• **imageView** is the image view that will be used for rendering.

• **imageLayout** is the layout that **imageView** will be in during rendering.

• **resolveMode** is a **VkResolveModeFlagBits** value defining how multisampled data written to **imageView** will be resolved.

• **resolveImageView** is an image view used to write resolved multisample data at the end of rendering.
• `resolveImageLayout` is the layout that `resolveImageView` will be in during rendering.

• `loadOp` is a `VkAttachmentLoadOp` value specifying how the contents of `imageView` are treated at the start of the render pass instance.

• `storeOp` is a `VkAttachmentStoreOp` value specifying how the contents of `imageView` are treated at the end of the render pass instance.

• `clearValue` is a `VkClearValue` structure defining values used to clear `imageView` when `loadOp` is `VK_ATTACHMENT_LOAD_OP_CLEAR`.

Values in `imageView` are loaded and stored according to the values of `loadOp` and `storeOp`, within the render area for each device specified in `VkRenderingInfo`. If `imageView` is `VK_NULL_HANDLE`, other members of this structure are ignored; writes to this attachment will be discarded, and no load, store, or resolve operations will be performed.

If `resolveMode` is `VK_RESOLVE_MODE_NONE`, then `resolveImageView` is ignored. If `resolveMode` is not `VK_RESOLVE_MODE_NONE`, and `resolveImageView` is not `VK_NULL_HANDLE`, values in `resolveImageView` within the render area become undefined once rendering begins. Only values in the aspect corresponding to the use of this attachment become undefined (the depth aspect if this attachment is used as `VkRenderingInfo::pDepthAttachment`, and the stencil aspect if it is used as `pStencilAttachment`).

At the end of rendering, the values written to each pixel location in `imageView` will be resolved according to `resolveMode` and stored into the the same location in `resolveImageView`.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>The resolve mode and store operation are independent; it is valid to write both resolved and unresolved values, and equally valid to discard the unresolved values while writing the resolved ones.</td>
</tr>
</tbody>
</table>

Store and resolve operations are only performed at the end of a render pass instance that does not specify the `VK_RENDERING_SUSPENDING_BIT_KHR` flag.

Load operations are only performed at the beginning of a render pass instance that does not specify the `VK_RENDERING_RESUMING_BIT_KHR` flag.

Image contents at the end of a suspended render pass instance remain defined for access by a resuming render pass instance.

### Valid Usage

- **VUID-VkRenderingAttachmentInfo-imageView-06129**
  If `imageView` is not `VK_NULL_HANDLE` and has a non-integer color format, `resolveMode` must be `VK_RESOLVE_MODE_NONE` or `VK_RESOLVE_MODE_AVERAGE_BIT`

- **VUID-VkRenderingAttachmentInfo-imageView-06130**
  If `imageView` is not `VK_NULL_HANDLE` and has an integer color format, `resolveMode` must be `VK_RESOLVE_MODE_NONE` or `VK_RESOLVE_MODE_SAMPLE_ZERO_BIT`

- **VUID-VkRenderingAttachmentInfo-imageView-06132**
If `imageView` is not `VK_NULL_HANDLE` and `resolveMode` is not `VK_RESOLVE_MODE_NONE`, `imageView` must not have a sample count of `VK_SAMPLE_COUNT_1_BIT`

- VUID-VkRenderingAttachmentInfo-imageView-06860
  If `imageView` is not `VK_NULL_HANDLE` and `resolveMode` is not `VK_RESOLVE_MODE_NONE`, `resolveImageView` must not be `VK_NULL_HANDLE`

- VUID-VkRenderingAttachmentInfo-imageView-06864
  If `imageView` is not `VK_NULL_HANDLE`, `resolveImageView` is not `VK_NULL_HANDLE`, and `resolveMode` is not `VK_RESOLVE_MODE_NONE`, `resolveImageView` must have a sample count of `VK_SAMPLE_COUNT_1_BIT`

- VUID-VkRenderingAttachmentInfo-imageView-06865
  If `imageView` is not `VK_NULL_HANDLE`, `resolveImageView` is not `VK_NULL_HANDLE`, and `resolveMode` is not `VK_RESOLVE_MODE_NONE`, `imageView` and `resolveImageView` must have the same `VkFormat`

- VUID-VkRenderingAttachmentInfo-imageView-06135
  If `imageView` is not `VK_NULL_HANDLE`, `imageLayout` must not be `VK_IMAGE_LAYOUT_UNDEFINED`, `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL`, `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`, or `VK_IMAGE_LAYOUT_PREINITIALIZED`

- VUID-VkRenderingAttachmentInfo-imageView-06136
  If `imageView` is not `VK_NULL_HANDLE` and `resolveMode` is not `VK_RESOLVE_MODE_NONE`, `resolveImageLayout` must not be `VK_IMAGE_LAYOUT_UNDEFINED`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL`, `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`, or `VK_IMAGE_LAYOUT_PREINITIALIZED`

- VUID-VkRenderingAttachmentInfo-imageView-06137
  If `imageView` is not `VK_NULL_HANDLE` and `resolveMode` is not `VK_RESOLVE_MODE_NONE`, `resolveImageLayout` must not be `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL` or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`

- VUID-VkRenderingAttachmentInfo-imageView-06142
  If `imageView` is not `VK_NULL_HANDLE` and `resolveMode` is not `VK_RESOLVE_MODE_NONE`, `resolveImageLayout` must not be `VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR`

---

**Valid Usage (Implicit)**

- VUID-VkRenderingAttachmentInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_RENDERING_ATTACHMENT_INFO`

- VUID-VkRenderingAttachmentInfo-pNext-pNext
  `pNext` must be `NULL`

- VUID-VkRenderingAttachmentInfo-imageView-parameter
  If `imageView` is not `VK_NULL_HANDLE`, `imageView` must be a valid `VkImageView` handle

- VUID-VkRenderingAttachmentInfo-imageLayout-parameter
  `imageLayout` must be a valid `VkImageLayout` value

- VUID-VkRenderingAttachmentInfo-resolveMode-parameter
If `resolveMode` is not 0, `resolveMode` must be a valid `VkResolveModeFlagBits` value.

- **VUID-VkRenderingAttachmentInfo-resolveImageView-parameter**
  If `resolveImageView` is not `VK_NULL_HANDLE`, `resolveImageView` must be a valid `VkImageView` handle.

- **VUID-VkRenderingAttachmentInfo-resolveImageLayout-parameter**
  `resolveImageLayout` must be a valid `VkImageLayout` value.

- **VUID-VkRenderingAttachmentInfo-loadOp-parameter**
  `loadOp` must be a valid `VkAttachmentLoadOp` value.

- **VUID-VkRenderingAttachmentInfo-storeOp-parameter**
  `storeOp` must be a valid `VkAttachmentStoreOp` value.

- **VUID-VkRenderingAttachmentInfo-clearValue-parameter**
  `clearValue` must be a valid `VkClearValue` union.

- **VUID-VkRenderingAttachmentInfo-commonparent**
  Both of `imageView`, and `resolveImageView` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`.

To end a render pass instance, call:

```c
// Provided by VK_VERSION_1_3
void vkCmdEndRendering(
    VkCommandBuffer commandBuffer);
```

- `commandBuffer` is the command buffer in which to record the command.

If the value of `pRenderingInfo->flags` used to begin this render pass instance included `VK_RENDERING_SUSPENDING_BIT`, then this render pass is suspended and will be resumed later in submission order.

### Valid Usage

- **VUID-vkCmdEndRendering-None-06161**
  The current render pass instance must have been begun with `vkCmdBeginRendering`.

- **VUID-vkCmdEndRendering-commandBuffer-06162**
  The current render pass instance must have been begun in `commandBuffer`.

- **VUID-vkCmdEndRendering-None-06999**
  If `vkCmdBeginQuery*` was called within the render pass, the corresponding `vkCmdEndQuery*` must have been called subsequently within the same subpass.

### Valid Usage (Implicit)

- **VUID-vkCmdEndRendering-commandBuffer-parameter**
  `commandBuffer` must be a valid `VkCommandBuffer` handle.
VUID-vkCmdEndRendering-commandBuffer-recording

**commandBuffer** must be in the recording state

VUID-vkCmdEndRendering-commandBuffer-cmdpool

The *VkCommandPool* that **commandBuffer** was allocated from must support graphics operations

VUID-vkCmdEndRendering-renderpass

This command must only be called inside of a render pass instance

### Host Synchronization

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Action State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note**

For more complex rendering graphs, it is possible to pre-define a static *render pass* object, which as well as allowing draw commands, allows the definition of framebuffer-local dependencies between multiple subpasses. These objects have a lot of setup cost compared to **vkCmdBeginRendering**, but use of subpass dependencies can confer important performance benefits on some devices.

A render pass object represents a collection of attachments, subpasses, and dependencies between the subpasses, and describes how the attachments are used over the course of the subpasses.

Render passes are represented by **VkRenderPass** handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkRenderPass)
```

An *attachment description* describes the properties of an attachment including its format, sample count, and how its contents are treated at the beginning and end of each render pass instance.

A *subpass* represents a phase of rendering that reads and writes a subset of the attachments in a render pass. Rendering commands are recorded into a particular subpass of a render pass instance.
A subpass description describes the subset of attachments that is involved in the execution of a subpass. Each subpass can read from some attachments as input attachments, write to some as color attachments or depth/stencil attachments, and perform multisample resolve operations to resolve attachments. A subpass description can also include a set of preserve attachments, which are attachments that are not read or written by the subpass but whose contents must be preserved throughout the subpass.

A subpass uses an attachment if the attachment is a color, depth/stencil, resolve, depth/stencil resolve, or input attachment for that subpass (as determined by the pColorAttachments, pDepthStencilAttachment, pResolveAttachments, VkSubpassDescriptionDepthStencilResolve::pDepthStencilResolveAttachment, and pInputAttachments members of VkSubpassDescription, respectively). A subpass does not use an attachment if that attachment is preserved by the subpass. The first use of an attachment is in the lowest numbered subpass that uses that attachment. Similarly, the last use of an attachment is in the highest numbered subpass that uses that attachment.

The subpasses in a render pass all render to the same dimensions, and fragments for pixel (x,y,layer) in one subpass can only read attachment contents written by previous subpasses at that same (x,y,layer) location.

---

**Note**

By describing a complete set of subpasses in advance, render passes provide the implementation an opportunity to optimize the storage and transfer of attachment data between subpasses.

In practice, this means that subpasses with a simple framebuffer-space dependency may be merged into a single tiled rendering pass, keeping the attachment data on-chip for the duration of a render pass instance. However, it is also quite common for a render pass to only contain a single subpass.

---

Subpass dependencies describe execution and memory dependencies between subpasses.

A subpass dependency chain is a sequence of subpass dependencies in a render pass, where the source subpass of each subpass dependency (after the first) equals the destination subpass of the previous dependency.

Execution of subpasses may overlap or execute out of order with regards to other subpasses, unless otherwise enforced by an execution dependency. Each subpass only respects submission order for commands recorded in the same subpass, and the vkCmdBeginRenderPass and vkCmdEndRenderPass commands that delimit the render pass - commands within other subpasses are not included. This affects most other implicit ordering guarantees.

A render pass describes the structure of subpasses and attachments independent of any specific image views for the attachments. The specific image views that will be used for the attachments, and their dimensions, are specified in VkFramebuffer objects. Framebuffers are created with respect to a specific render pass that the framebuffer is compatible with (see Render Pass Compatibility). Collectively, a render pass and a framebuffer define the complete render target state for one or more subpasses as well as the algorithmic dependencies between the subpasses.
The various pipeline stages of the drawing commands for a given subpass may execute concurrently and/or out of order, both within and across drawing commands, whilst still respecting pipeline order. However for a given (x,y,layer,sample) sample location, certain per-sample operations are performed in rasterization order.

VK_ATTACHMENT_UNUSED is a constant indicating that a render pass attachment is not used.

```
#define VK_ATTACHMENT_UNUSED (~0U)
```

## 8.1. Render Pass Creation

To create a render pass, call:

```
// Provided by VK_VERSION_1_0
VkResult vkCreateRenderPass(
    VkDevice device,
    const VkRenderPassCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkRenderPass* pRenderPass);
```

- `device` is the logical device that creates the render pass.
- `pCreateInfo` is a pointer to a VkRenderPassCreateInfo structure describing the parameters of the render pass.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pRenderPass` is a pointer to a VkRenderPass handle in which the resulting render pass object is returned.

### Valid Usage (Implicit)

- VUID-vkCreateRenderPass-device-parameter
  device must be a valid VkDevice handle

- VUID-vkCreateRenderPass-pCreateInfo-parameter
  pCreateInfo must be a valid pointer to a valid VkRenderPassCreateInfo structure

- VUID-vkCreateRenderPass-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkCreateRenderPass-pRenderPass-parameter
  pRenderPass must be a valid pointer to a VkRenderPass handle
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkRenderPassCreateInfo structure is defined as:

```
// Provided by VK_VERSION_1_0
typedef struct VkRenderPassCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkRenderPassCreateFlags flags;
    uint32_t attachmentCount;
    const VkAttachmentDescription* pAttachments;
    uint32_t subpassCount;
    const VkSubpassDescription* pSubpasses;
    uint32_t dependencyCount;
    const VkSubpassDependency* pDependencies;
} VkRenderPassCreateInfo;
```

• `sType` is the type of this structure.
• `pNext` is NULL or a pointer to a structure extending this structure.
• `flags` is reserved for future use.
• `attachmentCount` is the number of attachments used by this render pass.
• `pAttachments` is a pointer to an array of `attachmentCount` `VkAttachmentDescription` structures describing the attachments used by the render pass.
• `subpassCount` is the number of subpasses to create.
• `pSubpasses` is a pointer to an array of `subpassCount` `VkSubpassDescription` structures describing each subpass.
• `dependencyCount` is the number of memory dependencies between pairs of subpasses.
• `pDependencies` is a pointer to an array of `dependencyCount` `VkSubpassDependency` structures describing dependencies between pairs of subpasses.

Note
Care should be taken to avoid a data race here; if any subpasses access attachments with overlapping memory locations, and one of those accesses is a write, a subpass dependency needs to be included between them.
Valid Usage

- **VUID-VkRenderPassCreateInfo-attachment-00834**
  If the attachment member of any element of pInputAttachments, pColorAttachments, pResolveAttachments or pDepthStencilAttachment, or any element of pPreserveAttachments in any element of pSubpasses is not VK_ATTACHMENT_UNUSED, then it must be less than attachmentCount.

- **VUID-VkRenderPassCreateInfo-pAttachments-00836**
  For any member of pAttachments with a loadOp equal to VK_ATTACHMENT_LOAD_OP_CLEAR, the first use of that attachment must not specify a layout equal to VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL.

- **VUID-VkRenderPassCreateInfo-pAttachments-02511**
  For any member of pAttachments with a stencilLoadOp equal to VK_ATTACHMENT_LOAD_OP_CLEAR, the first use of that attachment must not specify a layout equal to VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL.

- **VUID-VkRenderPassCreateInfo-pAttachments-01566**
  For any member of pAttachments with a stencilLoadOp equal to VK_ATTACHMENT_LOAD_OP_CLEAR, the first use of that attachment must not specify a layout equal to VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL.

- **VUID-VkRenderPassCreateInfo-pNext-01926**
  If the pNext chain includes a VkRenderPassInputAttachmentAspectCreateInfo structure, the subpass member of each element of its pAspectReferences member must be less than subpassCount.

- **VUID-VkRenderPassCreateInfo-pNext-01927**
  If the pNext chain includes a VkRenderPassInputAttachmentAspectCreateInfo structure, the inputAttachmentIndex member of each element of its pAspectReferences member must be less than the value of inputAttachmentCount in the element of pSubpasses identified by its subpass member.

- **VUID-VkRenderPassCreateInfo-pNext-01963**
  If the pNext chain includes a VkRenderPassInputAttachmentAspectCreateInfo structure, for any element of the pInputAttachments member of any element of pSubpasses where the attachment member is not VK_ATTACHMENT_UNUSED, the aspectMask member of the corresponding element of VkRenderPassInputAttachmentAspectCreateInfo::pAspectReferences must only include aspects that are present in images of the format specified by the element of pAttachments at attachment.

- **VUID-VkRenderPassCreateInfo-pNext-01928**
  If the pNext chain includes a VkRenderPassMultiviewCreateInfo structure, and its subpassCount member is not zero, that member must be equal to the value of subpassCount.
If the `pNext` chain includes a `VkRenderPassMultiviewCreateInfo` structure, if its `dependencyCount` member is not zero, it **must** be equal to `dependencyCount`.

If the `pNext` chain includes a `VkRenderPassMultiviewCreateInfo` structure, for each non-zero element of `pViewOffsets`, the `srcSubpass` and `dstSubpass` members of `pDependencies` at the same index **must** not be equal.

If the `pNext` chain includes a `VkRenderPassMultiviewCreateInfo` structure, for any element of `pDependencies` with a `dependencyFlags` member that does not include `VK_DEPENDENCY_VIEW_LOCAL_BIT`, the corresponding element of the `pViewOffsets` member of that `VkRenderPassMultiviewCreateInfo` instance **must** be 0.

If the `pNext` chain includes a `VkRenderPassMultiviewCreateInfo` structure, for any element of `pViewMasks` member **must** either all be 0, or all not be 0.

If the `pNext` chain includes a `VkRenderPassMultiviewCreateInfo` structure, and each element of its `pViewMasks` member is 0, its `correlationMaskCount` member **must** be 0.

For any element of `pDependencies`, if the `srcSubpass` is not `VK_SUBPASS_EXTERNAL`, all stage flags included in the `srcStageMask` member of that dependency **must** be a pipeline stage supported by the pipeline identified by the `pipelineBindPoint` member of the source subpass.

For any element of `pDependencies`, if the `dstSubpass` is not `VK_SUBPASS_EXTERNAL`, all stage flags included in the `dstStageMask` member of that dependency **must** be a pipeline stage supported by the pipeline identified by the `pipelineBindPoint` member of the destination subpass.

For any element of `pDependencies`, if its `srcSubpass` is not `VK_SUBPASS_EXTERNAL`, it **must** be less than `subpassCount`.

For any element of `pDependencies`, if its `dstSubpass` is not `VK_SUBPASS_EXTERNAL`, it **must** be less than `subpassCount`.

**Valid Usage (Implicit)**

- **VUID-VkRenderPassCreateInfo-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO`
Each `pNext` member of any structure (including this one) in the `pNext` chain **must** be either `NULL` or a pointer to a valid instance of `VkRenderPassInputAttachmentAspectCreateInfo` or `VkRenderPassMultiviewCreateInfo`.

The `sType` value of each struct in the `pNext` chain **must** be unique.

`flags` must be `0`.

If `attachmentCount` is not `0`, `pAttachments` **must** be a valid pointer to an array of `VkAttachmentDescription` structures.

`pSubpasses` **must** be a valid pointer to an array of `subpassCount` valid `VkSubpassDescription` structures.

If `dependencyCount` is not `0`, `pDependencies` **must** be a valid pointer to an array of `dependencyCount` valid `VkSubpassDependency` structures.

`subpassCount` **must** be greater than `0`.

Bits which **can** be set in `VkRenderPassCreateInfo::flags`, describing additional properties of the render pass, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkRenderPassCreateFlagBits {
} VkRenderPassCreateFlagBits;
```

**Note**

All bits for this type are defined by extensions, and none of those extensions are enabled in this build of the specification.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkRenderPassCreateFlags;
```

`VkRenderPassCreateFlags` is a bitmask type for setting a mask of zero or more `VkRenderPassCreateFlagBits`.

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

The `VkRenderPassMultiviewCreateInfo` structure is defined as:
typedef struct VkRenderPassMultiviewCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t subpassCount;
    const uint32_t* pViewMasks;
    uint32_t dependencyCount;
    const int32_t* pViewOffsets;
    uint32_t correlationMaskCount;
    const uint32_t* pCorrelationMasks;
} VkRenderPassMultiviewCreateInfo;

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• subpassCount is zero or the number of subpasses in the render pass.
• pViewMasks is a pointer to an array of subpassCount view masks, where each mask is a bitfield of view indices describing which views rendering is broadcast to in each subpass, when multiview is enabled. If subpassCount is zero, each view mask is treated as zero.
• dependencyCount is zero or the number of dependencies in the render pass.
• pViewOffsets is a pointer to an array of dependencyCount view offsets, one for each dependency. If dependencyCount is zero, each dependency’s view offset is treated as zero. Each view offset controls which views in the source subpass the views in the destination subpass depend on.
• correlationMaskCount is zero or the number of correlation masks.
• pCorrelationMasks is a pointer to an array of correlationMaskCount view masks indicating sets of views that may be more efficient to render concurrently.

When a subpass uses a non-zero view mask, multiview functionality is considered to be enabled. Multiview is all-or-nothing for a render pass - that is, either all subpasses must have a non-zero view mask (though some subpasses may have only one view) or all must be zero. Multiview causes all drawing and clear commands in the subpass to behave as if they were broadcast to each view, where a view is represented by one layer of the framebuffer attachments. All draws and clears are broadcast to each view index whose bit is set in the view mask. The view index is provided in the ViewIndex shader input variable, and color, depth/stencil, and input attachments all read/write the layer of the framebuffer corresponding to the view index.

If the view mask is zero for all subpasses, multiview is considered to be disabled and all drawing commands execute normally, without this additional broadcasting.

Some implementations may not support multiview in conjunction with geometry shaders or tessellation shaders.

When multiview is enabled, the VK_DEPENDENCY_VIEW_LOCAL_BIT bit in a dependency can be used to express a view-local dependency, meaning that each view in the destination subpass depends on a single view in the source subpass. Unlike pipeline barriers, a subpass dependency can potentially have a different view mask in the source subpass and the destination subpass. If the dependency is
view-local, then each view (dstView) in the destination subpass depends on the view dstView + pViewOffsets[dependency] in the source subpass. If there is not such a view in the source subpass, then this dependency does not affect that view in the destination subpass. If the dependency is not view-local, then all views in the destination subpass depend on all views in the source subpass, and the view offset is ignored. A non-zero view offset is not allowed in a self-dependency.

The elements of pCorrelationMasks are a set of masks of views indicating that views in the same mask may exhibit spatial coherency between the views, making it more efficient to render them concurrently. Correlation masks must not have a functional effect on the results of the multiview rendering.

When multiview is enabled, at the beginning of each subpass all non-render pass state is undefined. In particular, each time vkCmdBeginRenderPass or vkCmdNextSubpass is called the graphics pipeline must be bound, any relevant descriptor sets or vertex/index buffers must be bound, and any relevant dynamic state or push constants must be set before they are used.

### Valid Usage

- **VUID-VkRenderPassMultiviewCreateInfo-pCorrelationMasks-00841**
  Each view index must not be set in more than one element of pCorrelationMasks

- **VUID-VkRenderPassMultiviewCreateInfo-multiview-06555**
  If the multiview feature is not enabled, each element of pViewMasks must be 0

- **VUID-VkRenderPassMultiviewCreateInfo-pViewMasks-06697**
  The index of the most significant bit in each element of pViewMasks must be less than maxMultiviewViewCount

### Valid Usage (Implicit)

- **VUID-VkRenderPassMultiviewCreateInfo-sType-sType**
  sType must be VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO

- **VUID-VkRenderPassMultiviewCreateInfo-pViewMasks-parameter**
  If subpassCount is not 0, pViewMasks must be a valid pointer to an array of subpassCount uint32_t values

- **VUID-VkRenderPassMultiviewCreateInfo-pViewOffsets-parameter**
  If dependencyCount is not 0, pViewOffsets must be a valid pointer to an array of dependencyCount int32_t values

- **VUID-VkRenderPassMultiviewCreateInfo-pCorrelationMasks-parameter**
  If correlationMaskCount is not 0, pCorrelationMasks must be a valid pointer to an array of correlationMaskCount uint32_t values

The VkAttachmentDescription structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkAttachmentDescription {
```
VkAttachmentDescriptionFlags flags;
VkFormat format;
VkSampleCountFlagBits samples;
VkAttachmentLoadOp loadOp;
VkAttachmentStoreOp storeOp;
VkAttachmentLoadOp stencilLoadOp;
VkAttachmentStoreOp stencilStoreOp;
VkImageLayout initialLayout;
VkImageLayout finalLayout;
}

VkAttachmentDescription;

- **flags** is a bitmask of VkAttachmentDescriptionFlagBits specifying additional properties of the attachment.
- **format** is a VkFormat value specifying the format of the image view that will be used for the attachment.
- **samples** is a VkSampleCountFlagBits value specifying the number of samples of the image.
- **loadOp** is a VkAttachmentLoadOp value specifying how the contents of color and depth components of the attachment are treated at the beginning of the subpass where it is first used.
- **storeOp** is a VkAttachmentStoreOp value specifying how the contents of color and depth components of the attachment are treated at the end of the subpass where it is last used.
- **stencilLoadOp** is a VkAttachmentLoadOp value specifying how the contents of stencil components of the attachment are treated at the beginning of the subpass where it is first used.
- **stencilStoreOp** is a VkAttachmentStoreOp value specifying how the contents of stencil components of the attachment are treated at the end of the last subpass where it is used.
- **initialLayout** is the layout the attachment image subresource will be in when a render pass instance begins.
- **finalLayout** is the layout the attachment image subresource will be transitioned to when a render pass instance ends.

If the attachment uses a color format, then loadOp and storeOp are used, and stencilLoadOp and stencilStoreOp are ignored. If the format has depth and/or stencil components, loadOp and storeOp apply only to the depth data, while stencilLoadOp and stencilStoreOp define how the stencil data is handled. loadOp and stencilLoadOp define the **load operations** that execute as part of the first subpass that uses the attachment. storeOp and stencilStoreOp define the **store operations** that execute as part of the last subpass that uses the attachment.

The load operation for each sample in an attachment happens-before any recorded command which accesses the sample in the first subpass where the attachment is used. Load operations for attachments with a depth/stencil format execute in the VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT pipeline stage. Load operations for attachments with a color format execute in the VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT pipeline stage.

The store operation for each sample in an attachment happens-after any recorded command which accesses the sample in the last subpass where the attachment is used. Store operations for attachments with a depth/stencil format execute in the VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT pipeline stage.
pipeline stage. Store operations for attachments with a color format execute in the `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` pipeline stage.

If an attachment is not used by any subpass, `loadOp`, `storeOp`, `stencilStoreOp`, and `stencilLoadOp` will be ignored for that attachment, and no load or store ops will be performed. However, any transition specified by `initialLayout` and `finalLayout` will still be executed.

The load and store operations apply on the first and last use of each view in the render pass, respectively. If a view index of an attachment is not included in the view mask in any subpass that uses it, then the load and store operations are ignored, and the attachment's memory contents will not be modified by execution of a render pass instance.

During a render pass instance, input/color attachments with color formats that have a component size of 8, 16, or 32 bits must be represented in the attachment's format throughout the instance. Attachments with other floating- or fixed-point color formats, or with depth components may be represented in a format with a precision higher than the attachment format, but must be represented with the same range. When such a component is loaded via the `loadOp`, it will be converted into an implementation-dependent format used by the render pass. Such components must be converted from the render pass format, to the format of the attachment, before they are resolved or stored at the end of a render pass instance via `storeOp`. Conversions occur as described in [Numeric Representation and Computation](#) and [Fixed-Point Data Conversions](#).

If `flags` includes `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT`, then the attachment is treated as if it shares physical memory with another attachment in the same render pass. This information limits the ability of the implementation to reorder certain operations (like layout transitions and the `loadOp`) such that it is not improperly reordered against other uses of the same physical memory via a different attachment. This is described in more detail below.

If a render pass uses multiple attachments that alias the same device memory, those attachments must each include the `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT` bit in their attachment description flags. Attachments aliasing the same memory occurs in multiple ways:

- Multiple attachments being assigned the same image view as part of framebuffer creation.
- Attachments using distinct image views that correspond to the same image subresource of an image.
- Attachments using views of distinct image subresources which are bound to overlapping memory ranges.

**Note**

Render passes must include subpass dependencies (either directly or via a subpass dependency chain) between any two subpasses that operate on the same attachment or aliasing attachments and those subpass dependencies must include execution and memory dependencies separating uses of the aliases, if at least one of those subpasses writes to one of the aliases. These dependencies must not include the `VK_DEPENDENCY_BY_REGION_BIT` if the aliases are views of distinct image subresources which overlap in memory.

Multiple attachments that alias the same memory must not be used in a single subpass. A given
attachment index **must** not be used multiple times in a single subpass, with one exception: two subpass attachments **can** use the same attachment index if at least one use is as an input attachment and neither use is as a resolve or preserve attachment. In other words, the same view **can** be used simultaneously as an input and color or depth/stencil attachment, but **must** not be used as multiple color or depth/stencil attachments nor as resolve or preserve attachments. The precise set of valid scenarios is described in more detail **below**.

If a set of attachments alias each other, then all except the first to be used in the render pass **must** use an `initialLayout` of `VK_IMAGE_LAYOUT_UNDEFINED`, since the earlier uses of the other aliases make their contents undefined. Once an alias has been used and a different alias has been used after it, the first alias **must** not be used in any later subpasses. However, an application **can** assign the same image view to multiple aliasing attachment indices, which allows that image view to be used multiple times even if other aliases are used in between.

---

**Note**

Once an attachment needs the `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT` bit, there **should** be no additional cost of introducing additional aliases, and using these additional aliases **may** allow more efficient clearing of the attachments on multiple uses via `VK_ATTACHMENT_LOAD_OP_CLEAR`.

---

### Valid Usage

- VUID-VkAttachmentDescription-format-06698
  
  `format` **must** not be `VK_FORMAT_UNDEFINED`

- VUID-VkAttachmentDescription-format-06699
  
  If `format` includes a color or depth aspect and `loadOp` is `VK_ATTACHMENT_LOAD_OP_LOAD`, then `initialLayout` **must** not be `VK_IMAGE_LAYOUT_UNDEFINED`

- VUID-VkAttachmentDescription-finalLayout-00843
  
  `finalLayout` **must** not be `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`

- VUID-VkAttachmentDescription-format-03280
  
  If `format` is a color format, `initialLayout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`

- VUID-VkAttachmentDescription-format-03281
  
  If `format` is a depth/stencil format, `initialLayout` **must** not be `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`

- VUID-VkAttachmentDescription-format-03282
  
  If `format` is a color format, `finalLayout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`

- VUID-VkAttachmentDescription-format-03283
  
  If `format` is a depth/stencil format, `finalLayout` **must** not be `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`

- VUID-VkAttachmentDescription-format-06487
If \textit{format} is a color format, \textit{initialLayout} must not be \code{VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL} or \code{VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL}.

- \textbf{VUID-VkAttachmentDescription-format-06488}
  If \textit{format} is a color format, \textit{finalLayout} must not be \code{VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL} or \code{VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL}.

- \textbf{VUID-VkAttachmentDescription-separateDepthStencilLayouts-03284}
  If the \textit{separateDepthStencilLayouts} feature is not enabled, \textit{initialLayout} must not be \code{VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL}, \code{VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL}, \code{VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL}, or \code{VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL}.

- \textbf{VUID-VkAttachmentDescription-separateDepthStencilLayouts-03285}
  If the \textit{separateDepthStencilLayouts} feature is not enabled, \textit{finalLayout} must not be \code{VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL}, \code{VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL}, \code{VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL}, or \code{VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL}.

- \textbf{VUID-VkAttachmentDescription-format-03286}
  If \textit{format} is a color format, \textit{initialLayout} must not be \code{VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL}, \code{VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL}, \code{VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL}, or \code{VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL}.

- \textbf{VUID-VkAttachmentDescription-format-03287}
  If \textit{format} is a color format, \textit{finalLayout} must not be \code{VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL}, \code{VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL}, \code{VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL}, or \code{VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL}.

- \textbf{VUID-VkAttachmentDescription-format-06906}
  If \textit{format} is a depth/stencil format which includes both depth and stencil aspects, \textit{initialLayout} must not be \code{VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL} or \code{VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL}.

- \textbf{VUID-VkAttachmentDescription-format-06907}
  If \textit{format} is a depth/stencil format which includes both depth and stencil aspects, \textit{finalLayout} must not be \code{VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL} or \code{VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL}.

- \textbf{VUID-VkAttachmentDescription-format-03290}
  If \textit{format} is a depth/stencil format which includes only the depth aspect, \textit{initialLayout} must not be \code{VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL} or \code{VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL}.

- \textbf{VUID-VkAttachmentDescription-format-03291}
  If \textit{format} is a depth/stencil format which includes only the depth aspect, \textit{finalLayout} must not be \code{VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL} or \code{VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL}.

- \textbf{VUID-VkAttachmentDescription-format-06700}
If `format` includes a stencil aspect and `stencilLoadOp` is `VK_ATTACHMENT_LOAD_OP_LOAD`, then `initialLayout must not be VK_IMAGE_LAYOUT_UNDEFINED`.

- **VUID-VkAttachmentDescription-format-03292**
  If `format` is a depth/stencil format which includes only the stencil aspect, `initialLayout must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`.

- **VUID-VkAttachmentDescription-format-03293**
  If `format` is a depth/stencil format which includes only the stencil aspect, `finalLayout must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`.

- **VUID-VkAttachmentDescription-format-06242**
  If `format` is a depth/stencil format which includes both depth and stencil aspects, `initialLayout must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`.

- **VUID-VkAttachmentDescription-format-06243**
  If `format` is a depth/stencil format which includes both depth and stencil aspects, `finalLayout must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`.

**Valid Usage (Implicit)**

- **VUID-VkAttachmentDescription-flags-parameter**
  `flags must be a valid combination of VkAttachmentDescriptionFlagBits values`.

- **VUID-VkAttachmentDescription-format-parameter**
  `format must be a valid VkFormat value`.

- **VUID-VkAttachmentDescription-samples-parameter**
  `samples must be a valid VkSampleCountFlagBits value`.

- **VUID-VkAttachmentDescription-loadOp-parameter**
  `loadOp must be a valid VkAttachmentLoadOp value`.

- **VUID-VkAttachmentDescription-storeOp-parameter**
  `storeOp must be a valid VkAttachmentStoreOp value`.

- **VUID-VkAttachmentDescription-stencilLoadOp-parameter**
  `stencilLoadOp must be a valid VkAttachmentLoadOp value`.

- **VUID-VkAttachmentDescription-stencilStoreOp-parameter**
  `stencilStoreOp must be a valid VkAttachmentStoreOp value`.

- **VUID-VkAttachmentDescription-initialLayout-parameter**
  `initialLayout must be a valid VkImageLayout value`.

- **VUID-VkAttachmentDescription-finalLayout-parameter**
  `finalLayout must be a valid VkImageLayout value`.

Bits which `can be set in VkAttachmentDescription::flags`, describing additional properties of the
attachment, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkAttachmentDescriptionFlagBits {
    VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT = 0x00000001,
} VkAttachmentDescriptionFlagBits;
```

- **VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT** specifies that the attachment aliases the same device memory as other attachments.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkAttachmentDescriptionFlags;
```

`VkAttachmentDescriptionFlags` is a bitmask type for setting a mask of zero or more `VkAttachmentDescriptionFlagBits`.

Possible values of `VkAttachmentDescription::loadOp` and `stencilLoadOp`, specifying how the contents of the attachment are treated, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkAttachmentLoadOp {
    VK_ATTACHMENT_LOAD_OP_LOAD = 0,
    VK_ATTACHMENT_LOAD_OP_CLEAR = 1,
    VK_ATTACHMENT_LOAD_OP_DONT_CARE = 2,
} VkAttachmentLoadOp;
```

- **VK_ATTACHMENT_LOAD_OP_LOAD** specifies that the previous contents of the image within the render area will be preserved. For attachments with a depth/stencil format, this uses the access type `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT`. For attachments with a color format, this uses the access type `VK_ACCESS_COLOR_ATTACHMENT_READ_BIT`.

- **VK_ATTACHMENT_LOAD_OP_CLEAR** specifies that the contents within the render area will be cleared to a uniform value, which is specified when a render pass instance is begun. For attachments with a depth/stencil format, this uses the access type `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`. For attachments with a color format, this uses the access type `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`.

- **VK_ATTACHMENT_LOAD_OP_DONT_CARE** specifies that the previous contents within the area need not be preserved; the contents of the attachment will be undefined inside the render area. For attachments with a depth/stencil format, this uses the access type `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`. For attachments with a color format, this uses the access type `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`.

Possible values of `VkAttachmentDescription::storeOp` and `stencilStoreOp`, specifying how the contents of the attachment are treated, are:

```c
// Provided by VK_VERSION_1_0
```
typedef enum VkAttachmentStoreOp {
    VK_ATTACHMENT_STORE_OP_STORE = 0,
    VK_ATTACHMENT_STORE_OP_DONT_CARE = 1,
    // Provided by VK_VERSION_1.3
    VK_ATTACHMENT_STORE_OP_NONE = 0x301000,
} VkAttachmentStoreOp;

• **VK_ATTACHMENT_STORE_OP_STORE** specifies the contents generated during the render pass and within the render area are written to memory. For attachments with a depth/stencil format, this uses the access type `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`. For attachments with a color format, this uses the access type `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`.

• **VK_ATTACHMENT_STORE_OP_DONT_CARE** specifies the contents within the render area are not needed after rendering, and **may** be discarded; the contents of the attachment will be undefined inside the render area. For attachments with a depth/stencil format, this uses the access type `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`. For attachments with a color format, this uses the access type `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`.

• **VK_ATTACHMENT_STORE_OP_NONE** specifies the contents within the render area are not accessed by the store operation. However, if the attachment was written to during the render pass, the contents of the attachment will be undefined inside the render area.

Note

**VK_ATTACHMENT_STORE_OP_DONT_CARE** can cause contents generated during previous render passes to be discarded before reaching memory, even if no write to the attachment occurs during the current render pass.

The **VkRenderPassInputAttachmentAspectCreateInfo** structure is defined as:

```c
// Provided by VK_VERSION_1.1
typedef struct VkRenderPassInputAttachmentAspectCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t aspectReferenceCount;
    const VkInputAttachmentAspectReference* pAspectReferences;
} VkRenderPassInputAttachmentAspectCreateInfo;
```

• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.

• **aspectReferenceCount** is the number of elements in the **pAspectReferences** array.

• **pAspectReferences** is a pointer to an array of **aspectReferenceCount** `VkInputAttachmentAspectReference` structures containing a mask describing which aspect(s) **can** be accessed for a given input attachment within a given subpass.

To specify which aspects of an input attachment **can** be read, add a **VkRenderPassInputAttachmentAspectCreateInfo** structure to the **pNext** chain of the **VkRenderPassCreateInfo** structure:
An application can access any aspect of an input attachment that does not have a specified aspect mask in the pAspectReferences array. Otherwise, an application must not access aspect(s) of an input attachment other than those in its specified aspect mask.

Valid Usage (Implicit)

- VUID-VkRenderPassInputAttachmentAspectCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_RENDER_PASS_INPUT_ATTACHMENT_ASPECT_CREATE_INFO

- VUID-VkRenderPassInputAttachmentAspectCreateInfo-pAspectReferences-parameter
  pAspectReferences must be a valid pointer to an array of aspectReferenceCount valid
  VkInputAttachmentAspectReference structures

- VUID-VkRenderPassInputAttachmentAspectCreateInfo-aspectReferenceCount-arraylength
  aspectReferenceCount must be greater than 0

The VkInputAttachmentAspectReference structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkInputAttachmentAspectReference {
    uint32_t subpass;
    uint32_t inputAttachmentIndex;
    VkImageAspectFlags aspectMask;
} VkInputAttachmentAspectReference;
```

- subpass is an index into the pSubpasses array of the parent VkRenderPassCreateInfo structure.
- inputAttachmentIndex is an index into the pInputAttachments of the specified subpass.
- aspectMask is a mask of which aspect(s) can be accessed within the specified subpass.

This structure specifies an aspect mask for a specific input attachment of a specific subpass in the render pass.

subpass and inputAttachmentIndex index into the render pass as:

```c
pCreateInfo->pSubpasses[subpass].pInputAttachments[inputAttachmentIndex]
```

Valid Usage

- VUID-VkInputAttachmentAspectReference-aspectReference-aspectMask-01964
  aspectMask must not include VK_IMAGE_ASPECT_METADATA_BIT

Valid Usage (Implicit)

- VUID-VkInputAttachmentAspectReference-aspectReference-aspectMask-parameter
  aspectMask must be a valid combination of VkImageAspectFlagBits values
The *VkSubpassDescription* structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSubpassDescription {
    VkSubpassDescriptionFlags flags;
    VkPipelineBindPoint pipelineBindPoint;
    uint32_t inputAttachmentCount;
    const VkAttachmentReference* pInputAttachments;
    uint32_t colorAttachmentCount;
    const VkAttachmentReference* pColorAttachments;
    const VkAttachmentReference* pResolveAttachments;
    const VkAttachmentReference* pDepthStencilAttachment;
    uint32_t preserveAttachmentCount;
    const uint32_t* pPreserveAttachments;
} VkSubpassDescription;
```

- **flags** is a bitmask of *VkSubpassDescriptionFlagBits* specifying usage of the subpass.
- **pipelineBindPoint** is a *VkPipelineBindPoint* value specifying the pipeline type supported for this subpass.
- **inputAttachmentCount** is the number of input attachments.
- **pInputAttachments** is a pointer to an array of *VkAttachmentReference* structures defining the input attachments for this subpass and their layouts.
- **colorAttachmentCount** is the number of color attachments.
- **pColorAttachments** is a pointer to an array of *colorAttachmentCount* *VkAttachmentReference* structures defining the color attachments for this subpass and their layouts.
- **pResolveAttachments** is *NULL* or a pointer to an array of *colorAttachmentCount* *VkAttachmentReference* structures defining the resolve attachments for this subpass and their layouts.
- **pDepthStencilAttachment** is a pointer to a *VkAttachmentReference* structure specifying the depth/stencil attachment for this subpass and its layout.
- **preserveAttachmentCount** is the number of preserved attachments.
- **pPreserveAttachments** is a pointer to an array of *preserveAttachmentCount* render pass attachment indices identifying attachments that are not used by this subpass, but whose contents **must** be preserved throughout the subpass.

Each element of the *pInputAttachments* array corresponds to an input attachment index in a fragment shader, i.e. if a shader declares an image variable decorated with a *InputAttachmentIndex* value of *X*, then it uses the attachment provided in *pInputAttachments[X]*. Input attachments **must** also be bound to the pipeline in a descriptor set. If the _attachment_ member of any element of *pInputAttachments* is *VK_ATTACHMENT_UNUSED*, the application **must** not read from the corresponding input attachment index. Fragment shaders can use subpass input variables to access the contents of
an input attachment at the fragment's (x, y, layer) framebuffer coordinates.

Each element of the pColorAttachments array corresponds to an output location in the shader, i.e. if the shader declares an output variable decorated with a Location value of X, then it uses the attachment provided in pColorAttachments[X]. If the attachment member of any element of pColorAttachments is VK_ATTACHMENT_UNUSED, then writes to the corresponding location by a fragment shader are discarded.

If pResolveAttachments is not NULL, each of its elements corresponds to a color attachment (the element in pColorAttachments at the same index), and a multisample resolve operation is defined for each attachment. At the end of each subpass, multisample resolve operations read the subpass’s color attachments, and resolve the samples for each pixel within the render area to the same pixel location in the corresponding resolve attachments, unless the resolve attachment index is VK_ATTACHMENT_UNUSED.

Similarly, if VkSubpassDescriptionDepthStencilResolve::pDepthStencilResolveAttachment is not NULL and does not have the value VK_ATTACHMENT_UNUSED, it corresponds to the depth/stencil attachment in pDepthStencilAttachment, and multisample resolve operations for depth and stencil are defined by VkSubpassDescriptionDepthStencilResolve::depthResolveMode and VkSubpassDescriptionDepthStencilResolve::stencilResolveMode, respectively. At the end of each subpass, multisample resolve operations read the subpass’s depth/stencil attachment, and resolve the samples for each pixel to the same pixel location in the corresponding resolve attachment. If VkSubpassDescriptionDepthStencilResolve::depthResolveMode is VK_RESOLVE_MODE_NONE, then the depth component of the resolve attachment is not written to and its contents are preserved. Similarly, if VkSubpassDescriptionDepthStencilResolve::stencilResolveMode is VK_RESOLVE_MODE_NONE, then the stencil component of the resolve attachment is not written to and its contents are preserved. VkSubpassDescriptionDepthStencilResolve::depthResolveMode is ignored if the VkFormat of the pDepthStencilResolveAttachment does not have a depth component. Similarly, VkSubpassDescriptionDepthStencilResolve::stencilResolveMode is ignored if the VkFormat of the pDepthStencilResolveAttachment does not have a stencil component.

If pDepthStencilAttachment is NULL, or if its attachment index is VK_ATTACHMENT_UNUSED, it indicates that no depth/stencil attachment will be used in the subpass.

The contents of an attachment within the render area become undefined at the start of a subpass S if all of the following conditions are true:

- The attachment is used as a color, depth/stencil, or resolve attachment in any subpass in the render pass.
- There is a subpass S₁ that uses or preserves the attachment, and a subpass dependency from S₁ to S.
- The attachment is not used or preserved in subpass S.

Once the contents of an attachment become undefined in subpass S, they remain undefined for subpasses in subpass dependency chains starting with subpass S until they are written again. However, they remain valid for subpasses in other subpass dependency chains starting with subpass S₁ if those subpasses use or preserve the attachment.
Valid Usage

• VUID-VkSubpassDescription-attachment-06912
  If the attachment member of an element of pInputAttachments is not VK_ATTACHMENT_UNUSED, its layout member must not be VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL

• VUID-VkSubpassDescription-attachment-06913
  If the attachment member of an element of pColorAttachments is not VK_ATTACHMENT_UNUSED, its layout member must not be VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL

• VUID-VkSubpassDescription-attachment-06914
  If the attachment member of an element of pResolveAttachments is not VK_ATTACHMENT_UNUSED, its layout member must not be VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL

• VUID-VkSubpassDescription-attachment-06915
  If the attachment member of pDepthStencilAttachment is not VK_ATTACHMENT_UNUSED, its layout member must not be VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL

• VUID-VkSubpassDescription-attachment-06916
  If the attachment member of an element of pColorAttachments is not VK_ATTACHMENT_UNUSED, its layout member must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL

• VUID-VkSubpassDescription-attachment-06917
  If the attachment member of an element of pResolveAttachments is not VK_ATTACHMENT_UNUSED, its layout member must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL

• VUID-VkSubpassDescription-attachment-06918
  If the attachment member of an element of pInputAttachments is not VK_ATTACHMENT_UNUSED, its layout member must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_ATTACHMENT_OPTIMAL

• VUID-VkSubpassDescription-attachment-06919
  If the attachment member of an element of pColorAttachments is not VK_ATTACHMENT_UNUSED, its layout member must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL

• VUID-VkSubpassDescription-attachment-06920
  If the attachment member of an element of pResolveAttachments is not VK_ATTACHMENT_UNUSED, its layout member must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL
• VUID-VkSubpassDescription-pipelineBindPoint-00844
  pipelineBindPoint must be VK_PIPELINE_BIND_POINT_GRAPHICS

• VUID-VkSubpassDescription-colorAttachmentCount-00845
  colorAttachmentCount must be less than or equal to VkPhysicalDeviceLimits::maxColorAttachments

• VUID-VkSubpassDescription-loadOp-00846
  If the first use of an attachment in this render pass is as an input attachment, and the attachment is not also used as a color or depth/stencil attachment in the same subpass, then loadOp must not be VK_ATTACHMENT_LOAD_OP_CLEAR

• VUID-VkSubpassDescription-pResolveAttachments-00847
  If pResolveAttachments is not NULL, for each resolve attachment that is not VK_ATTACHMENT_UNUSED, the corresponding color attachment must not be VK_ATTACHMENT_UNUSED

• VUID-VkSubpassDescription-pResolveAttachments-00848
  If pResolveAttachments is not NULL, for each resolve attachment that is not VK_ATTACHMENT_UNUSED, the corresponding color attachment must not have a sample count of VK_SAMPLE_COUNT_1_BIT

• VUID-VkSubpassDescription-pResolveAttachments-00849
  If pResolveAttachments is not NULL, each resolve attachment that is not VK_ATTACHMENT_UNUSED must have a sample count of VK_SAMPLE_COUNT_1_BIT

• VUID-VkSubpassDescription-pResolveAttachments-00850
  If pResolveAttachments is not NULL, each resolve attachment that is not VK_ATTACHMENT_UNUSED must have the same VkFormat as its corresponding color attachment

• VUID-VkSubpassDescription-pColorAttachments-06868
  If neither the VK_AMD_mixed_attachment_samples extension nor the VK_NV_framebuffer_mixed_samples extension is enabled, all attachments in pColorAttachments that are not VK_ATTACHMENT_UNUSED must have the same sample count

• VUID-VkSubpassDescription-pInputAttachments-02647
  All attachments in pInputAttachments that are not VK_ATTACHMENT_UNUSED must have image formats whose potential format features contain at least VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT or VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT

• VUID-VkSubpassDescription-pColorAttachments-02648
  All attachments in pColorAttachments that are not VK_ATTACHMENT_UNUSED must have image formats whose potential format features contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT

• VUID-VkSubpassDescription-pResolveAttachments-02649
  All attachments in pResolveAttachments that are not VK_ATTACHMENT_UNUSED must have image formats whose potential format features contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT

• VUID-VkSubpassDescription-pDepthStencilAttachment-02650
  If pDepthStencilAttachment is not NULL and the attachment is not VK_ATTACHMENT_UNUSED then it must have an image format whose potential format features contain
• VUID-VkSubpassDescription-pDepthStencilAttachment-01418
  If neither the `VK_AMD_mixed_attachment_samples` nor the `VK_NV_framebuffer_mixed_samples` extensions are enabled, and if `pDepthStencilAttachment` is not `VK_ATTACHMENT_UNUSED` and any attachments in `pColorAttachments` are not `VK_ATTACHMENT_UNUSED`, they must have the same sample count.

• VUID-VkSubpassDescription-attachment-00853
  Each element of `pPreserveAttachments` must not be `VK_ATTACHMENT_UNUSED`.

• VUID-VkSubpassDescription-pPreserveAttachments-00854
  Each element of `pPreserveAttachments` must not also be an element of any other member of the subpass description.

• VUID-VkSubpassDescription-layout-02519
  If any attachment is used by more than one `VkAttachmentReference` member, then each use must use the same layout.

• VUID-VkSubpassDescription-pDepthStencilAttachment-04438
  `pDepthStencilAttachment` and `pColorAttachments` must not contain references to the same attachment.

---

**Valid Usage (Implicit)**

• VUID-VkSubpassDescription-flags-zero bitmask
  `flags` must be 0.

• VUID-VkSubpassDescription-pipelineBindPoint-parameter
  `pipelineBindPoint` must be a valid `VkPipelineBindPoint` value.

• VUID-VkSubpassDescription-pInputAttachments-parameter
  If `inputAttachmentCount` is not 0, `pInputAttachments` must be a valid pointer to an array of `inputAttachmentCount` valid `VkAttachmentReference` structures.

• VUID-VkSubpassDescription-pColorAttachments-parameter
  If `colorAttachmentCount` is not 0, `pColorAttachments` must be a valid pointer to an array of `colorAttachmentCount` valid `VkAttachmentReference` structures.

• VUID-VkSubpassDescription-pResolveAttachments-parameter
  If `colorAttachmentCount` is not 0, and `pResolveAttachments` is not NULL, `pResolveAttachments` must be a valid pointer to an array of `colorAttachmentCount` valid `VkAttachmentReference` structures.

• VUID-VkSubpassDescription-pDepthStencilAttachment-parameter
  If `pDepthStencilAttachment` is not NULL, `pDepthStencilAttachment` must be a valid pointer to a valid `VkAttachmentReference` structure.

• VUID-VkSubpassDescription-pPreserveAttachments-parameter
  If `preserveAttachmentCount` is not 0, `pPreserveAttachments` must be a valid pointer to an array of `preserveAttachmentCount` `uint32_t` values.

---

Bits which can be set in `VkSubpassDescription::flags`, specifying usage of the subpass, are:
typedef enum VkSubpassDescriptionFlagBits {
} VkSubpassDescriptionFlagBits;

Note
All bits for this type are defined by extensions, and none of those extensions are enabled in this build of the specification.

typedef VkFlags VkSubpassDescriptionFlags;

VkSubpassDescriptionFlags is a bitmask type for setting a mask of zero or more VkSubpassDescriptionFlagBits.

The VkAttachmentReference structure is defined as:

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

- attachment is either an integer value identifying an attachment at the corresponding index in VkRenderPassCreateInfo::pAttachments, or VK_ATTACHMENT_UNUSED to signify that this attachment is not used.
- layout is a VkImageLayout value specifying the layout the attachment uses during the subpass.

Valid Usage

- VUID-VkAttachmentReference-layout-03077
  If attachment is not VK_ATTACHMENT_UNUSED, layout must not be VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_PREINITIALIZED, or VK_IMAGE_LAYOUT_PRESENT_SRC_KHR

- VUID-VkAttachmentReference-separateDepthStencilLayouts-03313
  If the separateDepthStencilLayouts feature is not enabled, and attachment is not VK_ATTACHMENT_UNUSED, layout must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL,

Valid Usage (Implicit)

- VUID-VkAttachmentReference-layout-parameter
  layout must be a valid VkImageLayout value
\textbf{VK\_SUBPASS\_EXTERNAL} is a special subpass index value expanding synchronization scope outside a subpass. It is described in more detail by \texttt{VkSubpassDependency}.

\begin{verbatim}
#define VK_SUBPASS_EXTERNAL (-0U)
\end{verbatim}

The \texttt{VkSubpassDependency} structure is defined as:

\begin{verbatim}
// Provided by VK\_VERSION\_1\_0
typedef struct VkSubpassDependency {
    uint32_t srcSubpass;
    uint32_t dstSubpass;
    VkPipelineStageFlags srcStageMask;
    VkPipelineStageFlags dstStageMask;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    VkDependencyFlags dependencyFlags;
} VkSubpassDependency;
\end{verbatim}

- \texttt{srcSubpass} is the subpass index of the first subpass in the dependency, or \texttt{VK\_SUBPASS\_EXTERNAL}.
- \texttt{dstSubpass} is the subpass index of the second subpass in the dependency, or \texttt{VK\_SUBPASS\_EXTERNAL}.
- \texttt{srcStageMask} is a bitmask of \texttt{VkPipelineStageFlagBits} specifying the source stage mask.
- \texttt{dstStageMask} is a bitmask of \texttt{VkPipelineStageFlagBits} specifying the destination stage mask.
- \texttt{srcAccessMask} is a bitmask of \texttt{VkAccessFlagBits} specifying a source access mask.
- \texttt{dstAccessMask} is a bitmask of \texttt{VkAccessFlagBits} specifying a destination access mask.
- \texttt{dependencyFlags} is a bitmask of \texttt{VkDependencyFlagBits}.

If \texttt{srcSubpass} is equal to \texttt{dstSubpass} then the \texttt{VkSubpassDependency} describes a subpass self-dependency, and only constrains the pipeline barriers allowed within a subpass instance. Otherwise, when a render pass instance which includes a subpass dependency is submitted to a queue, it defines a memory dependency between the subpasses identified by \texttt{srcSubpass} and \texttt{dstSubpass}.

If \texttt{srcSubpass} is equal to \texttt{VK\_SUBPASS\_EXTERNAL}, the first synchronization scope includes commands that occur earlier in submission order than the \texttt{vkCmdBeginRenderPass} used to begin the render pass instance. Otherwise, the first set of commands includes all commands submitted as part of the subpass instance identified by \texttt{srcSubpass} and any load, store or multisample resolve operations on attachments used in \texttt{srcSubpass}. In either case, the first synchronization scope is limited to operations on the pipeline stages determined by the source stage mask specified by \texttt{srcStageMask}.

If \texttt{dstSubpass} is equal to \texttt{VK\_SUBPASS\_EXTERNAL}, the second synchronization scope includes commands that occur later in submission order than the \texttt{vkCmdEndRenderPass} used to end the render pass instance. Otherwise, the second set of commands includes all commands submitted as part of the subpass instance identified by \texttt{dstSubpass} and any load, store or multisample resolve operations on attachments used in \texttt{dstSubpass}. In either case, the second synchronization scope is limited to
operations on the pipeline stages determined by the destination stage mask specified by dstStageMask.

The first access scope is limited to accesses in the pipeline stages determined by the source stage mask specified by srcStageMask. It is also limited to access types in the source access mask specified by srcAccessMask.

The second access scope is limited to accesses in the pipeline stages determined by the destination stage mask specified by dstStageMask. It is also limited to access types in the destination access mask specified by dstAccessMask.

The availability and visibility operations defined by a subpass dependency affect the execution of image layout transitions within the render pass.

**Note**

For non-attachment resources, the memory dependency expressed by subpass dependency is nearly identical to that of a VkMemoryBarrier (with matching srcAccessMask and dstAccessMask parameters) submitted as a part of a vkCmdPipelineBarrier (with matching srcStageMask and dstStageMask parameters). The only difference being that its scopes are limited to the identified subpasses rather than potentially affecting everything before and after.

For attachments however, subpass dependencies work more like a VkImageMemoryBarrier defined similarly to the VkMemoryBarrier above, the queue family indices set to VK_QUEUE_FAMILY_IGNORED, and layouts as follows:

- The equivalent to oldLayout is the attachment's layout according to the subpass description for srcSubpass.
- The equivalent to newLayout is the attachment's layout according to the subpass description for dstSubpass.

---

**Valid Usage**

- VUID-VkSubpassDependency-srcStageMask-04090  
  If the geometryShader feature is not enabled, srcStageMask must not contain VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT

- VUID-VkSubpassDependency-srcStageMask-04091  
  If the tessellationShader feature is not enabled, srcStageMask must not contain VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT or VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT

- VUID-VkSubpassDependency-srcStageMask-03937  
  If the synchronization2 feature is not enabled, srcStageMask must not be 0

- VUID-VkSubpassDependency-dstStageMask-04090  
  If the geometryShader feature is not enabled, dstStageMask must not contain VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT

- VUID-VkSubpassDependency-dstStageMask-04091
If the \texttt{tessellationShader} feature is not enabled, \texttt{dstStageMask} \textbf{must} not contain \texttt{VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT} or \texttt{VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT}

- VUID-VkSubpassDependency-dstStageMask-03937
  If the \texttt{synchronization2} feature is not enabled, \texttt{dstStageMask} \textbf{must} not be 0

- VUID-VkSubpassDependency-srcSubpass-00864
  \texttt{srcSubpass} \textbf{must} be less than or equal to \texttt{dstSubpass}, unless one of them is \texttt{VK_SUBPASS_EXTERNAL}, to avoid cyclic dependencies and ensure a valid execution order

- VUID-VkSubpassDependency-srcSubpass-00865
  \texttt{srcSubpass} and \texttt{dstSubpass} \textbf{must} not both be equal to \texttt{VK_SUBPASS_EXTERNAL}

- VUID-VkSubpassDependency-srcSubpass-06809
  If \texttt{srcSubpass} is equal to \texttt{dstSubpass} and \texttt{srcStageMask} includes a \texttt{framebuffer-space stage}, \texttt{dstStageMask} \textbf{must} only contain \texttt{framebuffer-space stages}

- VUID-VkSubpassDependency-srcAccessMask-00868
  Any access flag included in \texttt{srcAccessMask} \textbf{must} be supported by one of the pipeline stages in \texttt{srcStageMask}, as specified in the \texttt{table of supported access types}

- VUID-VkSubpassDependency-dstAccessMask-00869
  Any access flag included in \texttt{dstAccessMask} \textbf{must} be supported by one of the pipeline stages in \texttt{dstStageMask}, as specified in the \texttt{table of supported access types}

- VUID-VkSubpassDependency-srcSubpass-02243
  If \texttt{srcSubpass} equals \texttt{dstSubpass}, and \texttt{srcStageMask} and \texttt{dstStageMask} both include a \texttt{framebuffer-space stage}, then \texttt{dependencyFlags} \textbf{must} include \texttt{VK_DEPENDENCY_BY_REGION_BIT}

- VUID-VkSubpassDependency-dependencyFlags-02520
  If \texttt{dependencyFlags} includes \texttt{VK_DEPENDENCY_VIEW_LOCAL_BIT}, \texttt{srcSubpass} \textbf{must} not be equal to \texttt{VK_SUBPASS_EXTERNAL}

- VUID-VkSubpassDependency-dependencyFlags-02521
  If \texttt{dependencyFlags} includes \texttt{VK_DEPENDENCY_VIEW_LOCAL_BIT}, \texttt{dstSubpass} \textbf{must} not be equal to \texttt{VK_SUBPASS_EXTERNAL}

- VUID-VkSubpassDependency-srcSubpass-00872
  If \texttt{srcSubpass} equals \texttt{dstSubpass} and that subpass has more than one bit set in the \texttt{view mask}, then \texttt{dependencyFlags} \textbf{must} include \texttt{VK_DEPENDENCY_VIEW_LOCAL_BIT}

\section*{Valid Usage (Implicit)}

- VUID-VkSubpassDependency-srcStageMask-parameter
  \texttt{srcStageMask} \textbf{must} be a valid combination of \texttt{VkPipelineStageFlagBits} values

- VUID-VkSubpassDependency-dstStageMask-parameter
  \texttt{dstStageMask} \textbf{must} be a valid combination of \texttt{VkPipelineStageFlagBits} values

- VUID-VkSubpassDependency-srcAccessMask-parameter
  \texttt{srcAccessMask} \textbf{must} be a valid combination of \texttt{VkAccessFlagBits} values

- VUID-VkSubpassDependency-dstAccessMask-parameter
  \texttt{dstAccessMask} \textbf{must} be a valid combination of \texttt{VkAccessFlagBits} values
dependencyFlags must be a valid combination of VkDependencyFlagBits values

When multiview is enabled, the execution of the multiple views of one subpass may not occur simultaneously or even back-to-back, and rather may be interleaved with the execution of other subpasses. The load and store operations apply to attachments on a per-view basis. For example, an attachment using VK_ATTACHMENT_LOAD_OP_CLEAR will have each view cleared on first use, but the first use of one view may be temporally distant from the first use of another view.

Note
A good mental model for multiview is to think of a multiview subpass as if it were a collection of individual (per-view) subpasses that are logically grouped together and described as a single multiview subpass in the API. Similarly, a multiview attachment can be thought of like several individual attachments that happen to be layers in a single image. A view-local dependency between two multiview subpasses acts like a set of one-to-one dependencies between corresponding pairs of per-view subpasses. A view-global dependency between two multiview subpasses acts like a set of $N \times M$ dependencies between all pairs of per-view subpasses in the source and destination. Thus, it is a more compact representation which also makes clear the commonality and reuse that is present between views in a subpass. This interpretation motivates the answers to questions like “when does the load op apply” - it is on the first use of each view of an attachment, as if each view was a separate attachment.

The content of each view follows the description in attachment content behavior. In particular, if an attachment is preserved, all views within the attachment are preserved.

If there is no subpass dependency from VK_SUBPASS_EXTERNAL to the first subpass that uses an attachment, then an implicit subpass dependency exists from VK_SUBPASS_EXTERNAL to the first subpass it is used in. The implicit subpass dependency only exists if there exists an automatic layout transition away from initialLayout. The subpass dependency operates as if defined with the following parameters:

```cpp
VkSubpassDependency implicitDependency = {
    .srcSubpass = VK_SUBPASS_EXTERNAL;
    .dstSubpass = firstSubpass; // First subpass attachment is used in
    .srcStageMask = VK_PIPELINE_STAGE_NONE;
    .dstStageMask = VK_PIPELINE_STAGE_ALL_COMMANDS_BIT;
    .srcAccessMask = 0;
    .dstAccessMask = VK_ACCESS_INPUT_ATTACHMENT_READ_BIT |
                     VK_ACCESS_COLOR_ATTACHMENT_READ_BIT |
                     VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
                     VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT |
                     VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
    .dependencyFlags = 0;
};
```
Similarly, if there is no subpass dependency from the last subpass that uses an attachment to VK_SUBPASS_EXTERNAL, then an implicit subpass dependency exists from the last subpass it is used in to VK_SUBPASS_EXTERNAL. The implicit subpass dependency only exists if there exists an automatic layout transition into finalLayout. The subpass dependency operates as if defined with the following parameters:

```cpp
VkSubpassDependency implicitDependency = {
    .srcSubpass = lastSubpass; // Last subpass attachment is used in
    .dstSubpass = VK_SUBPASS_EXTERNAL;
    .srcStageMask = VK_PIPELINE_STAGE_ALL_COMMANDS_BIT;
    .dstStageMask = VK_PIPELINE_STAGE_NONE;
    .srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
                      VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
    .dstAccessMask = 0;
    .dependencyFlags = 0;
};
```

As subpasses may overlap or execute out of order with regards to other subpasses unless a subpass dependency chain describes otherwise, the layout transitions required between subpasses cannot be known to an application. Instead, an application provides the layout that each attachment must be in at the start and end of a render pass, and the layout it must be in during each subpass it is used in. The implementation then must execute layout transitions between subpasses in order to guarantee that the images are in the layouts required by each subpass, and in the final layout at the end of the render pass.

Automatic layout transitions apply to the entire image subresource attached to the framebuffer. If multiview is not enabled and the attachment is a view of a 1D or 2D image, the automatic layout transitions apply to the number of layers specified by VkFramebufferCreateInfo::layers. If multiview is enabled and the attachment is a view of a 1D or 2D image, the automatic layout transitions apply to the layers corresponding to views which are used by some subpass in the render pass, even if that subpass does not reference the given attachment. If the attachment view is a 2D or 2D array view of a 3D image, even if the attachment view only refers to a subset of the slices of the selected mip level of the 3D image, automatic layout transitions apply to the entire subresource referenced which is the entire mip level in this case.

Automatic layout transitions away from the layout used in a subpass happen-after the availability operations for all dependencies with that subpass as the srcSubpass.

Automatic layout transitions into the layout used in a subpass happen-before the visibility operations for all dependencies with that subpass as the dstSubpass.

Automatic layout transitions away from initialLayout happen-after the availability operations for all dependencies with a srcSubpass equal to VK_SUBPASS_EXTERNAL, where dstSubpass uses the attachment that will be transitioned. For attachments created with VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT, automatic layout transitions away from initialLayout happen-after the availability operations for all dependencies with a srcSubpass equal to VK_SUBPASS_EXTERNAL, where dstSubpass uses any aliased attachment.

Automatic layout transitions into finalLayout happen-before the visibility operations for all...
dependencies with a dstSubpass equal to VK_SUBPASS_EXTERNAL, where srcSubpass uses the attachment that will be transitioned. For attachments created with VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT, automatic layout transitions into finalLayout happen-before the visibility operations for all dependencies with a dstSubpass equal to VK_SUBPASS_EXTERNAL, where srcSubpass uses any aliased attachment.

If two subpasses use the same attachment, and both subpasses use the attachment in a read-only layout, no subpass dependency needs to be specified between those subpasses. If an implementation treats those layouts separately, it must insert an implicit subpass dependency between those subpasses to separate the uses in each layout. The subpass dependency operates as if defined with the following parameters:

```c
// Used for input attachments
VkPipelineStageFlags inputAttachmentStages = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
VkAccessFlags inputAttachmentDstAccess = VK_ACCESS_INPUT_ATTACHMENT_READ_BIT;

// Used for depth/stencil attachments
VkPipelineStageFlags depthStencilAttachmentStages =
    VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT |
    VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT;
VkAccessFlags depthStencilAttachmentDstAccess =
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT;

VkSubpassDependency implicitDependency = {
    .srcSubpass = firstSubpass;
    .dstSubpass = secondSubpass;
    .srcStageMask = inputAttachmentStages | depthStencilAttachmentStages;
    .dstStageMask = inputAttachmentStages | depthStencilAttachmentStages;
    .srcAccessMask = 0;
    .dstAccessMask = inputAttachmentDstAccess | depthStencilAttachmentDstAccess;
    .dependencyFlags = 0;
};
```

A subpass may access the same subresource for both a color or depth/stencil attachment and as an input resource in one of the following ways:

- As an input attachment

In these situations, writes via the color or depth/stencil attachment are not automatically made visible to reads via the input resource, causing a feedback loop, except in any of the following conditions:

- If the color components or depth/stencil components read are mutually exclusive with the components written by the color or depth/stencil attachments, then there is no feedback loop. This requires the graphics pipelines used by the subpass to disable writes to color components that are read as inputs via the colorWriteMask, and to disable writes to depth/stencil components that are read as inputs via depthWriteEnable or stencilTestEnable.

- If the attachment is used as an input attachment and depth/stencil attachment only, and the depth/stencil attachment is not written to.
Rendering within a subpass containing a feedback loop creates a data race, except in the following cases:

- If a memory dependency is inserted between when the attachment is written and when it is subsequently read by later fragments. Pipeline barriers expressing a subpass self-dependency are the only way to achieve this, and one must be inserted every time a fragment will read values at a particular sample (x, y, layer, sample) coordinate, if those values have been written since the most recent pipeline barrier; or since the start of the subpass, if there have been no pipeline barriers since the start of the subpass.

An attachment must not be used as both a depth/stencil attachment and a color attachment.

A more extensible version of render pass creation is also defined below.

To create a render pass, call:

```c
// Provided by VK_VERSION_1_2
VkResult vkCreateRenderPass2(
    VkDevice device,
    const VkRenderPassCreateInfo2* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkRenderPass* pRenderPass);
```

- `device` is the logical device that creates the render pass.
- `pCreateInfo` is a pointer to a `VkRenderPassCreateInfo2` structure describing the parameters of the render pass.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pRenderPass` is a pointer to a `VkRenderPass` handle in which the resulting render pass object is returned.

This command is functionally identical to `vkCreateRenderPass`, but includes extensible sub-structures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

**Valid Usage (Implicit)**

- VUID-vkCreateRenderPass2-device-parameter
  `device` must be a valid `VkDevice` handle
- VUID-vkCreateRenderPass2-pCreateInfo-parameter
  `pCreateInfo` must be a valid pointer to a valid `VkRenderPassCreateInfo2` structure
- VUID-vkCreateRenderPass2-pAllocator-parameter
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- VUID-vkCreateRenderPass2-pRenderPass-parameter
  `pRenderPass` must be a valid pointer to a `VkRenderPass` handle
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkRenderPassCreateInfo2` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkRenderPassCreateInfo2 {
    VkStructureType sType;
    const void* pNext;
    VkRenderPassCreateFlags flags;
    uint32_t attachmentCount;
    const VkAttachmentDescription2* pAttachments;
    uint32_t subpassCount;
    const VkSubpassDescription2* pSubpasses;
    uint32_t dependencyCount;
    const VkSubpassDependency2* pDependencies;
    uint32_t correlatedViewMaskCount;
    const uint32_t* pCorrelatedViewMasks;
} VkRenderPassCreateInfo2;
```

• `sType` is the type of this structure.
• `pNext` is `NULL` or a pointer to a structure extending this structure.
• `flags` is reserved for future use.
• `attachmentCount` is the number of attachments used by this render pass.
• `pAttachments` is a pointer to an array of `attachmentCount` `VkAttachmentDescription2` structures describing the attachments used by the render pass.
• `subpassCount` is the number of subpasses to create.
• `pSubpasses` is a pointer to an array of `subpassCount` `VkSubpassDescription2` structures describing each subpass.
• `dependencyCount` is the number of dependencies between pairs of subpasses.
• `pDependencies` is a pointer to an array of `dependencyCount` `VkSubpassDependency2` structures describing dependencies between pairs of subpasses.
• `correlatedViewMaskCount` is the number of correlation masks.
• `pCorrelatedViewMasks` is a pointer to an array of view masks indicating sets of views that may be more efficient to render concurrently.

Parameters defined by this structure with the same name as those in `VkRenderPassCreateInfo` have
the identical effect to those parameters; the child structures are variants of those used in `VkRenderPassCreateInfo` which add `sType` and `pNext` parameters, allowing them to be extended.

If the `VkSubpassDescription2::viewMask` member of any element of `pSubpasses` is not zero, multiview functionality is considered to be enabled for this render pass.

correlatedViewMaskCount and `pCorrelatedViewMasks` have the same effect as `VkRenderPassMultiviewCreateInfo::correlationMaskCount` and `VkRenderPassMultiviewCreateInfo::pCorrelationMasks`, respectively.

### Valid Usage

- **VUID-VkRenderPassCreateInfo2-None-03049**
  If any two subpasses operate on attachments with overlapping ranges of the same `VkDeviceMemory` object, and at least one subpass writes to that area of `VkDeviceMemory`, a subpass dependency *must* be included (either directly or via some intermediate subpasses) between them.

- **VUID-VkRenderPassCreateInfo2-attachment-03050**
  If the attachment member of any element of `pInputAttachments`, `pColorAttachments`, `pResolveAttachments` or `pDepthStencilAttachment`, or the attachment indexed by any element of `pPreserveAttachments` in any given element of `pSubpasses` is bound to a range of a `VkDeviceMemory` object that overlaps with any other attachment in any subpass (including the same subpass), the `VkAttachmentDescription2` structures describing them *must* include `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT` in flags.

- **VUID-VkRenderPassCreateInfo2-attachment-03051**
  If the attachment member of any element of `pInputAttachments`, `pColorAttachments`, `pResolveAttachments` or `pDepthStencilAttachment`, or any element of `pPreserveAttachments` in any given element of `pSubpasses` is not `VK_ATTACHMENT_UNUSED`, then it *must* be less than `attachmentCount`.

- **VUID-VkRenderPassCreateInfo2-pSubpasses-06473**
  If the `pSubpasses` `pNext` chain includes a `VkSubpassDescriptionDepthStencilResolve` structure and the `pDepthStencilResolveAttachment` member is not `NULL` and does not have the value `VK_ATTACHMENT_UNUSED`, then attachment *must* be less than `attachmentCount`.

- **VUID-VkRenderPassCreateInfo2-pAttachments-02522**
  For any member of `pAttachments` with a `loadOp` equal to `VK_ATTACHMENT_LOAD_OP_CLEAR`, the first use of that attachment *must* not specify a layout equal to `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`.

- **VUID-VkRenderPassCreateInfo2-pAttachments-02523**
  For any member of `pAttachments` with a `stencilLoadOp` equal to `VK_ATTACHMENT_LOAD_OP_CLEAR`, the first use of that attachment *must* not specify a layout equal to `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`.

310
For any element of `pDependencies`, if the `srcSubpass` is not `VK_SUBPASS_EXTERNAL`, all stage flags included in the `srcStageMask` member of that dependency must be a pipeline stage supported by the pipeline identified by the `pipelineBindPoint` member of the source subpass.

For any element of `pDependencies`, if the `dstSubpass` is not `VK_SUBPASS_EXTERNAL`, all stage flags included in the `dstStageMask` member of that dependency must be a pipeline stage supported by the pipeline identified by the `pipelineBindPoint` member of the destination subpass.

The set of bits included in any element of `pCorrelatedViewMasks` must not overlap with the set of bits included in any other element of `pCorrelatedViewMasks`.

If the `VkSubpassDescription2::viewMask` member of all elements of `pSubpasses` is 0, `correlatedViewMaskCount` must be 0.

The `VkSubpassDescription2::viewMask` member of all elements of `pSubpasses` must either all be 0, or all not be 0.

If the `VkSubpassDescription2::viewMask` member of all elements of `pSubpasses` is 0, the `dependencyFlags` member of any element of `pDependencies` must not include `VK_DEPENDENCY_VIEW_LOCAL_BIT`.

For any element of `pDependencies` where its `srcSubpass` member equals its `dstSubpass` member, if the `viewMask` member of the corresponding element of `pSubpasses` includes more than one bit, its `dependencyFlags` member must include `VK_DEPENDENCY_VIEW_LOCAL_BIT`.

If the `attachment` member of any element of the `pInputAttachments` member of any element of `pSubpasses` is not `VK_ATTACHMENT_UNUSED`, the `aspectMask` member of that element of `pInputAttachments` must only include aspects that are present in images of the format specified by the element of `pAttachments` specified by `attachment`.

The `srcSubpass` member of each element of `pDependencies` must be less than `subpassCount`.

The `dstSubpass` member of each element of `pDependencies` must be less than `subpassCount`.

If the `attachment` member of the `pDepthStencilAttachment` member of an element of `pSubpasses` is not `VK_ATTACHMENT_UNUSED`, the `layout` member of that same structure is either `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, and the `pNext` chain of that structure does not include a `VkAttachmentReferenceStencilLayout` structure, then the element of `pAttachments` with an index equal to `attachment` must not have a `format` that includes both depth and stencil.
aspects

- VUID-VkRenderPassCreateInfo2-attachment-06245
  If the attachment member of the pDepthStencilAttachment member of an element of pSubpasses is not VK_ATTACHMENT_UNUSED and the layout member of that same structure is either VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL, then the element of pAttachments with an index equal to attachment must have a format that includes only a stencil aspect

- VUID-VkRenderPassCreateInfo2-attachment-06246
  If the attachment member of the pDepthStencilAttachment member of an element of pSubpasses is not VK_ATTACHMENT_UNUSED and the layout member of that same structure is either VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, then the element of pAttachments with an index equal to attachment must not have a format that includes only a stencil aspect

Valid Usage (Implicit)

- VUID-VkRenderPassCreateInfo2-sType-sType
  sType must be VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO_2

- VUID-VkRenderPassCreateInfo2-pNext-pNext
  pNext must be NULL

- VUID-VkRenderPassCreateInfo2-flags-zerobitmask
  flags must be 0

- VUID-VkRenderPassCreateInfo2-pAttachments-parameter
  If attachmentCount is not 0, pAttachments must be a valid pointer to an array of attachmentCount valid VkAttachmentDescription2 structures

- VUID-VkRenderPassCreateInfo2-pSubpasses-parameter
  pSubpasses must be a valid pointer to an array of subpassCount valid VkSubpassDescription2 structures

- VUID-VkRenderPassCreateInfo2-pDependencies-parameter
  If dependencyCount is not 0, pDependencies must be a valid pointer to an array of dependencyCount valid VkSubpassDependency2 structures

- VUID-VkRenderPassCreateInfo2-pCorrelatedViewMasks-parameter
  If correlatedViewMaskCount is not 0, pCorrelatedViewMasks must be a valid pointer to an array of correlatedViewMaskCount uint32_t values

- VUID-VkRenderPassCreateInfo2-subpassCount-arraylength
  subpassCount must be greater than 0

The VkAttachmentDescription2 structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkAttachmentDescription2 {
    VkStructureType                sType;
```
const void* pNext;  
VkAttachmentDescriptionFlags flags;  
VkFormat format;  
VkSampleCountFlagBits samples;  
VkAttachmentLoadOp loadOp;  
VkAttachmentStoreOp storeOp;  
VkAttachmentLoadOp stencilLoadOp;  
VkAttachmentStoreOp stencilStoreOp;  
VkImageLayout initialLayout;  
VkImageLayout finalLayout;

}\ VkAttachmentDescription2;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is a bitmask of **VkAttachmentDescriptionFlagBits** specifying additional properties of the attachment.
- **format** is a **VkFormat** value specifying the format of the image that will be used for the attachment.
- **samples** is a **VkSampleCountFlagBits** value specifying the number of samples of the image.
- **loadOp** is a **VkAttachmentLoadOp** value specifying how the contents of color and depth components of the attachment are treated at the beginning of the subpass where it is first used.
- **storeOp** is a **VkAttachmentStoreOp** value specifying how the contents of color and depth components of the attachment are treated at the end of the subpass where it is last used.
- **stencilLoadOp** is a **VkAttachmentLoadOp** value specifying how the contents of stencil components of the attachment are treated at the beginning of the subpass where it is first used.
- **stencilStoreOp** is a **VkAttachmentStoreOp** value specifying how the contents of stencil components of the attachment are treated at the end of the last subpass where it is used.
- **initialLayout** is the layout the attachment image subresource will be in when a render pass instance begins.
- **finalLayout** is the layout the attachment image subresource will be transitioned to when a render pass instance ends.

Parameters defined by this structure with the same name as those in **VkAttachmentDescription** have the identical effect to those parameters.

If the **separateDepthStencilLayouts** feature is enabled, and **format** is a depth/stencil format, **initialLayout** and **finalLayout** can be set to a layout that only specifies the layout of the depth aspect.

If the **pNext** chain includes a **VkAttachmentDescriptionStencilLayout** structure, then the **stencilInitialLayout** and **stencilFinalLayout** members specify the initial and final layouts of the stencil aspect of a depth/stencil format, and **initialLayout** and **finalLayout** only apply to the depth aspect. For depth-only formats, the **VkAttachmentDescriptionStencilLayout** structure is ignored. For stencil-only formats, the initial and final layouts of the stencil aspect are taken from the **VkAttachmentDescriptionStencilLayout** structure if present, or **initialLayout** and **finalLayout** if not
present.

If \texttt{format} is a depth/stencil format, and either \texttt{initialLayout} or \texttt{finalLayout} does not specify a layout for the stencil aspect, then the application \textbf{must} specify the initial and final layouts of the stencil aspect by including a \texttt{VkAttachmentDescriptionStencilLayout} structure in the \texttt{pNext} chain.

**Valid Usage**

- VUID-VkAttachmentDescription2-format-06698  
  \texttt{format} \textbf{must} not be \texttt{VK_FORMAT_UNDEFINED}

- VUID-VkAttachmentDescription2-format-06699  
  If \texttt{format} includes a color or depth aspect and \texttt{loadOp} is \texttt{VK_ATTACHMENT_LOAD_OP_LOAD}, then \texttt{initialLayout} \textbf{must} not be \texttt{VK_IMAGE_LAYOUT_UNDEFINED}

- VUID-VkAttachmentDescription2-finalLayout-00843  
  \texttt{finalLayout} \textbf{must} not be \texttt{VK_IMAGE_LAYOUT_UNDEFINED} or \texttt{VK_IMAGE_LAYOUT_PREINITIALIZED}

- VUID-VkAttachmentDescription2-format-03280  
  If \texttt{format} is a color format, \texttt{initialLayout} \textbf{must} not be \texttt{VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL} or \texttt{VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL}

- VUID-VkAttachmentDescription2-format-03281  
  If \texttt{format} is a depth/stencil format, \texttt{initialLayout} \textbf{must} not be \texttt{VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL}

- VUID-VkAttachmentDescription2-format-03282  
  If \texttt{format} is a color format, \texttt{finalLayout} \textbf{must} not be \texttt{VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL} or \texttt{VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL}

- VUID-VkAttachmentDescription2-format-03283  
  If \texttt{format} is a depth/stencil format, \texttt{finalLayout} \textbf{must} not be \texttt{VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL}

- VUID-VkAttachmentDescription2-format-06487  
  If \texttt{format} is a color format, \texttt{initialLayout} \textbf{must} not be \texttt{VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL} or \texttt{VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL}

- VUID-VkAttachmentDescription2-format-06488  
  If \texttt{format} is a color format, \texttt{finalLayout} \textbf{must} not be \texttt{VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL} or \texttt{VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL}

- VUID-VkAttachmentDescription2-separateDepthStencilLayouts-03284  
  If the \texttt{separateDepthStencilLayouts} feature is not enabled, \texttt{initialLayout} \textbf{must} not be \texttt{VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL}, \texttt{VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL}, \texttt{VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL}, or \texttt{VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL}

- VUID-VkAttachmentDescription2-separateDepthStencilLayouts-03285  
  If the \texttt{separateDepthStencilLayouts} feature is not enabled, \texttt{finalLayout} \textbf{must} not be
If format is a color format, initialLayout must not be
VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL,
VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, or
VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL.

If format is a color format, finalLayout must not be
VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL,
VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, or
VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL.

If format is a color format which includes both depth and stencil aspects,
initialLayout must not be VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL or
VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL.

If format is a color format which includes both depth and stencil aspects,
finalLayout must not be VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL or
VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL.

If format is a depth/stencil format which includes only the depth aspect,
initialLayout must not be VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL or
VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL.

If format is a depth/stencil format which includes only the depth aspect,
finalLayout must not be VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL or
VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL.

If the pNext chain does not include a VkAttachmentDescriptionStencilLayout structure,
format includes a stencil aspect, and stencilLoadOp is VK_ATTACHMENT_LOAD_OP_LOAD, then
initialLayout must not be VK_IMAGE_LAYOUT_UNDEFINED.

If the pNext chain includes a VkAttachmentDescriptionStencilLayout structure,
format includes a stencil aspect, and stencilLoadOp is VK_ATTACHMENT_LOAD_OP_LOAD, then
VkAttachmentDescriptionStencilLayout::stencilInitialLayout must not be
VK_IMAGE_LAYOUT_UNDEFINED.

If format is a depth/stencil format which includes both depth and stencil aspects, and
initialLayout is VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL or
VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, the pNext chain must include a
VkAttachmentDescriptionStencilLayout structure.

If format is a depth/stencil format which includes both depth and stencil aspects,
initialLayout must not be VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL or
VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL.
If `format` is a depth/stencil format which includes both depth and stencil aspects, and `finalLayout` is `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, the `pNext` chain must include a `VkAttachmentDescriptionStencilLayout` structure.

- **VUID-VkAttachmentDescription2-format-06247**
  If the `pNext` chain does not include a `VkAttachmentDescriptionStencilLayout` structure and `format` only includes a stencil aspect, `initialLayout` must not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`.

- **VUID-VkAttachmentDescription2-format-06248**
  If the `pNext` chain does not include a `VkAttachmentDescriptionStencilLayout` structure and `format` only includes a stencil aspect, `finalLayout` must not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`.

### Valid Usage (Implicit)

- **VUID-VkAttachmentDescription2-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_2`.

- **VUID-VkAttachmentDescription2-pNext-pNext**
  `pNext` must be `NULL` or a pointer to a valid instance of `VkAttachmentDescriptionStencilLayout`.

- **VUID-VkAttachmentDescription2-sType-unique**
  The `sType` value of each struct in the `pNext` chain must be unique.

- **VUID-VkAttachmentDescription2-flags-parameter**
  `flags` must be a valid combination of `VkAttachmentDescriptionFlagBits` values.

- **VUID-VkAttachmentDescription2-format-parameter**
  `format` must be a valid `VkFormat` value.

- **VUID-VkAttachmentDescription2-samples-parameter**
  `samples` must be a valid `VkSampleCountFlagBits` value.

- **VUID-VkAttachmentDescription2-loadOp-parameter**
  `loadOp` must be a valid `VkAttachmentLoadOp` value.

- **VUID-VkAttachmentDescription2-storeOp-parameter**
  `storeOp` must be a valid `VkAttachmentStoreOp` value.

- **VUID-VkAttachmentDescription2-stencilLoadOp-parameter**
  `stencilLoadOp` must be a valid `VkAttachmentLoadOp` value.

- **VUID-VkAttachmentDescription2-stencilStoreOp-parameter**
  `stencilStoreOp` must be a valid `VkAttachmentStoreOp` value.

- **VUID-VkAttachmentDescription2-initialLayout-parameter**
  `initialLayout` must be a valid `VkImageLayout` value.

- **VUID-VkAttachmentDescription2-finalLayout-parameter**
  `finalLayout` must be a valid `VkImageLayout` value.
The `VkAttachmentDescriptionStencilLayout` structure is defined as:

// Provided by VK_VERSION_1_2
typedef struct VkAttachmentDescriptionStencilLayout {
    VkStructureType sType;
    void* pNext;
    VkImageLayout stencilInitialLayout;
    VkImageLayout stencilFinalLayout;
} VkAttachmentDescriptionStencilLayout;

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `stencilInitialLayout` is the layout the stencil aspect of the attachment image subresource will be in when a render pass instance begins.
- `stencilFinalLayout` is the layout the stencil aspect of the attachment image subresource will be transitioned to when a render pass instance ends.

Valid Usage

- VUID-VkAttachmentDescriptionStencilLayout-stencilInitialLayout-03308
  `stencilInitialLayout` must not be `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`.

- VUID-VkAttachmentDescriptionStencilLayout-stencilFinalLayout-03309
  `stencilFinalLayout` must not be `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`.

- VUID-VkAttachmentDescriptionStencilLayout-stencilFinalLayout-03310
  `stencilFinalLayout` must not be `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`.

Valid Usage (Implicit)

- VUID-VkAttachmentDescriptionStencilLayout-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_STENCIL_LAYOUT`.

- VUID-VkAttachmentDescriptionStencilLayout-stencilInitialLayout-parameter
  `stencilInitialLayout` must be a valid `VkImageLayout` value.
The VkSubpassDescription2 structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkSubpassDescription2 {
    VkStructureType sType;
    const void* pNext;
    VkSubpassDescriptionFlags flags;
    VkPipelineBindPoint pipelineBindPoint;
    uint32_t viewMask;
    uint32_t inputAttachmentCount;
    const VkAttachmentReference2* pInputAttachments;
    uint32_t colorAttachmentCount;
    const VkAttachmentReference2* pColorAttachments;
    const VkAttachmentReference2* pResolveAttachments;
    const VkAttachmentReference2* pDepthStencilAttachment;
    uint32_t preserveAttachmentCount;
    const uint32_t* pPreserveAttachments;
} VkSubpassDescription2;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is a bitmask of VkSubpassDescriptionFlagBits specifying usage of the subpass.
- **pipelineBindPoint** is a VkPipelineBindPoint value specifying the pipeline type supported for this subpass.
- **viewMask** is a bitfield of view indices describing which views rendering is broadcast to in this subpass, when multiview is enabled.
- **inputAttachmentCount** is the number of input attachments.
- **pInputAttachments** is a pointer to an array of VkAttachmentReference2 structures defining the input attachments for this subpass and their layouts.
- **colorAttachmentCount** is the number of color attachments.
- **pColorAttachments** is a pointer to an array of colorAttachmentCount VkAttachmentReference2 structures defining the color attachments for this subpass and their layouts.
- **pResolveAttachments** is NULL or a pointer to an array of colorAttachmentCount VkAttachmentReference2 structures defining the resolve attachments for this subpass and their layouts.
- **pDepthStencilAttachment** is a pointer to a VkAttachmentReference2 structure specifying the depth/stencil attachment for this subpass and its layout.
- **preserveAttachmentCount** is the number of preserved attachments.
- **pPreserveAttachments** is a pointer to an array of preserveAttachmentCount render pass attachment indices identifying attachments that are not used by this subpass, but whose contents must be
preserved throughout the subpass.

Parameters defined by this structure with the same name as those in VkSubpassDescription have the identical effect to those parameters.

viewMask has the same effect for the described subpass as VkRenderPassMultiviewCreateInfo::pViewMasks has on each corresponding subpass.

Valid Usage

- VUID-VkSubpassDescription2-attachment-06912
  If the attachment member of an element of pInputAttachments is not VK_ATTACHMENT_UNUSED, its layout member must not be VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL.

- VUID-VkSubpassDescription2-attachment-06913
  If the attachment member of an element of pColorAttachments is not VK_ATTACHMENT_UNUSED, its layout member must not be VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL.

- VUID-VkSubpassDescription2-attachment-06914
  If the attachment member of an element of pResolveAttachments is not VK_ATTACHMENT_UNUSED, its layout member must not be VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL.

- VUID-VkSubpassDescription2-attachment-06915
  If the attachment member of pDepthStencilAttachment is not VK_ATTACHMENT_UNUSED, ts layout member must not be VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL.

- VUID-VkSubpassDescription2-attachment-06916
  If the attachment member of an element of pColorAttachments is not VK_ATTACHMENT_UNUSED, its layout member must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL.

- VUID-VkSubpassDescription2-attachment-06917
  If the attachment member of an element of pResolveAttachments is not VK_ATTACHMENT_UNUSED, its layout member must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL.

- VUID-VkSubpassDescription2-attachment-06918
  If the attachment member of an element of pInputAttachments is not VK_ATTACHMENT_UNUSED, its layout member must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL.

- VUID-VkSubpassDescription2-attachment-06919
  If the attachment member of an element of pColorAttachments is not VK_ATTACHMENT_UNUSED, its layout member must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL.
If the attachment member of an element of pResolveAttachments is not VK_ATTACHMENT_UNUSED, its layout member must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL.

If the attachment member of pDepthStencilAttachment is not VK_ATTACHMENT_UNUSED and its pNext chain includes a VkAttachmentDescriptionStencilLayout structure, the layout member of pDepthStencilAttachment must not be VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL.

pipelineBindPoint must be VK_PIPELINE_BIND_POINT_GRAPHICS.

colorAttachmentCount must be less than or equal to VkPhysicalDeviceLimits::maxColorAttachments.

If the first use of an attachment in this render pass is as an input attachment, and the attachment is not also used as a color or depth/stencil attachment in the same subpass, then loadOp must not be VK_ATTACHMENT_LOAD_OP_CLEAR.

If pResolveAttachments is not NULL, for each resolve attachment that does not have the value VK_ATTACHMENT_UNUSED, the corresponding color attachment must not have the value VK_ATTACHMENT_UNUSED.

If pResolveAttachments is not NULL, for each resolve attachment that is not VK_ATTACHMENT_UNUSED, the corresponding color attachment must not have a sample count of VK_SAMPLE_COUNT_1_BIT.

If pResolveAttachments is not NULL, each resolve attachment that is not VK_ATTACHMENT_UNUSED must have a sample count of VK_SAMPLE_COUNT_1_BIT.

Any given element of pResolveAttachments must have the same VkFormat as its corresponding color attachment.

All attachments in pInputAttachments that are not VK_ATTACHMENT_UNUSED must have image formats whose potential format features contain at least VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT or VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT.

All attachments in pColorAttachments that are not VK_ATTACHMENT_UNUSED must have image...
formats whose potential format features contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT

- VUID-VkSubpassDescription2-pResolveAttachments-02899
  All attachments in pResolveAttachments that are not VK_ATTACHMENT_UNUSED must have image formats whose potential format features contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT

- VUID-VkSubpassDescription2-pDepthStencilAttachment-02900
  If pDepthStencilAttachment is not NULL and the attachment is not VK_ATTACHMENT_UNUSED then it must have an image format whose potential format features contain VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT

- VUID-VkSubpassDescription2-multisampledRenderToSingleSampled-06872
  If none of the VK_AMD_mixed_attachment_samples extension, the VK_NV_framebuffer_mixed_samples extension, or the multisampledRenderToSingleSampled feature are enabled, all attachments in pDepthStencilAttachment or pColorAttachments that are not VK_ATTACHMENT_UNUSED must have the same sample count

- VUID-VkSubpassDescription2-attachment-03073
  Each element of pPreserveAttachments must not be VK_ATTACHMENT_UNUSED

- VUID-VkSubpassDescription2-pPreserveAttachments-03074
  Any given element of pPreserveAttachments must not also be an element of any other member of the subpass description

- VUID-VkSubpassDescription2-layout-02528
  If any attachment is used by more than one VkAttachmentReference2 member, then each use must use the same layout

- VUID-VkSubpassDescription2-attachment-02799
  If the attachment member of any element of pInputAttachments is not VK_ATTACHMENT_UNUSED, then the aspectMask member must be a valid combination of VkImageAspectFlagBits

- VUID-VkSubpassDescription2-attachment-02800
  If the attachment member of any element of pInputAttachments is not VK_ATTACHMENT_UNUSED, then the aspectMask member must not be 0

- VUID-VkSubpassDescription2-attachment-02801
  If the attachment member of any element of pInputAttachments is not VK_ATTACHMENT_UNUSED, then the aspectMask member must not include VK_IMAGE_ASPECT_METADATA_BIT

- VUID-VkSubpassDescription2-pDepthStencilAttachment-04440
  An attachment must not be used in both pDepthStencilAttachment and pColorAttachments

- VUID-VkSubpassDescription2-multiview-06558
  If the multiview feature is not enabled, viewMask must be 0

- VUID-VkSubpassDescription2-viewMask-06706
  The index of the most significant bit in viewMask must be less than maxMultiviewViewCount

---

**Valid Usage (Implicit)**

- VUID-VkSubpassDescription2-sType-sType
  sType must be VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_2
• VUID-VkSubpassDescription2-pNext-pNext
  `pNext` must be `NULL` or a pointer to a valid instance of `VkSubpassDescriptionDepthStencilResolve`

• VUID-VkSubpassDescription2-sType-unique
  The `sType` value of each struct in the `pNext` chain must be unique

• VUID-VkSubpassDescription2-flags-zerobitmask
  `flags` must be 0

• VUID-VkSubpassDescription2-pipelineBindPoint-parameter
  `pipelineBindPoint` must be a valid `VkPipelineBindPoint` value

• VUID-VkSubpassDescription2-pInputAttachments-parameter
  If `inputAttachmentCount` is not 0, `pInputAttachments` must be a valid pointer to an array of `VkAttachmentReference2` structures

• VUID-VkSubpassDescription2-pColorAttachments-parameter
  If `colorAttachmentCount` is not 0, `pColorAttachments` must be a valid pointer to an array of `VkAttachmentReference2` structures

• VUID-VkSubpassDescription2-pResolveAttachments-parameter
  If `colorAttachmentCount` is not 0, and `pResolveAttachments` is not NULL, `pResolveAttachments` must be a valid pointer to an array of `colorAttachmentCount` valid `VkAttachmentReference2` structures

• VUID-VkSubpassDescription2-pDepthStencilAttachment-parameter
  If `pDepthStencilAttachment` is not NULL, `pDepthStencilAttachment` must be a valid pointer to a valid `VkAttachmentReference2` structure

• VUID-VkSubpassDescription2-pPreserveAttachments-parameter
  If `preserveAttachmentCount` is not 0, `pPreserveAttachments` must be a valid pointer to an array of `preserveAttachmentCount` uint32_t values

If the `pNext` chain of `VkSubpassDescription2` includes a `VkSubpassDescriptionDepthStencilResolve` structure, then that structure describes multisample resolve operations for the depth/stencil attachment in a subpass.

The `VkSubpassDescriptionDepthStencilResolve` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkSubpassDescriptionDepthStencilResolve {
    VkStructureType              sType;
    const void*                  pNext;
    VkResolveModeFlagBits        depthResolveMode;
    VkResolveModeFlagBits        stencilResolveMode;
    const VkAttachmentReference2* pDepthStencilResolveAttachment;
} VkSubpassDescriptionDepthStencilResolve;
```

• `sType` is the type of this structure.
• `pNext` is `NULL` or a pointer to a structure extending this structure.
• depthResolveMode is a VkResolveModeFlagBits value describing the depth resolve mode.
• stencilResolveMode is a VkResolveModeFlagBits value describing the stencil resolve mode.
• pDepthStencilResolveAttachment is NULL or a pointer to a VkAttachmentReference2 structure defining the depth/stencil resolve attachment for this subpass and its layout.

If pDepthStencilResolveAttachment is NULL, or if its attachment index is VK_ATTACHMENT_UNUSED, it indicates that no depth/stencil resolve attachment will be used in the subpass.

Valid Usage

• VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-03177
  If pDepthStencilResolveAttachment is not NULL and does not have the value VK_ATTACHMENT_UNUSED, pDepthStencilAttachment must not be NULL or have the value VK_ATTACHMENT_UNUSED

• VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-03179
  If pDepthStencilResolveAttachment is not NULL and does not have the value VK_ATTACHMENT_UNUSED, pDepthStencilAttachment must not have a sample count of VK_SAMPLE_COUNT_1_BIT

• VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-03180
  If pDepthStencilResolveAttachment is not NULL and does not have the value VK_ATTACHMENT_UNUSED, pDepthStencilResolveAttachment must have a sample count of VK_SAMPLE_COUNT_1_BIT

• VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-02651
  If pDepthStencilResolveAttachment is not NULL and does not have the value VK_ATTACHMENT_UNUSED then it must have an image format whose potential format features contain VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT

• VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-03181
  If pDepthStencilResolveAttachment is not NULL and does not have the value VK_ATTACHMENT_UNUSED and VkFormat of pDepthStencilResolveAttachment has a depth component, then the VkFormat of pDepthStencilAttachment must have a depth component with the same number of bits and numerical type

• VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-03182
  If pDepthStencilResolveAttachment is not NULL and does not have the value VK_ATTACHMENT_UNUSED, and VkFormat of pDepthStencilResolveAttachment has a stencil component, then the VkFormat of pDepthStencilAttachment must have a stencil component with the same number of bits and numerical type

• VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-03178
  If pDepthStencilResolveAttachment is not NULL and does not have the value VK_ATTACHMENT_UNUSED, depthResolveMode and stencilResolveMode must not both be VK_RESOLVE_MODE_NONE

• VUID-VkSubpassDescriptionDepthStencilResolve-depthResolveMode-03183
  If pDepthStencilResolveAttachment is not NULL and does not have the value VK_ATTACHMENT_UNUSED and the VkFormat of pDepthStencilResolveAttachment has a depth component, then the value of depthResolveMode must be one of the bits set in
If \( pDepthStencilResolveAttachment \) is not NULL and does not have the value \( VK_ATTACHMENT_UNUSED \) and the \( VkFormat \) of \( pDepthStencilResolveAttachment \) has a stencil component, then the value of \( stencilResolveMode \) must be one of the bits set in \( VkPhysicalDeviceDepthStencilResolveProperties::supportedStencilResolveModes \) or \( VK_RESOLVE_MODE_NONE \).

If \( pDepthStencilResolveAttachment \) is not NULL and does not have the value \( VK_ATTACHMENT_UNUSED \), the \( VkFormat \) of \( pDepthStencilResolveAttachment \) has both depth and stencil components, \( VkPhysicalDeviceDepthStencilResolveProperties::independentResolve \) is \( VK_FALSE \), and \( VkPhysicalDeviceDepthStencilResolveProperties::independentResolveNone \) is \( VK_FALSE \), then the values of \( depthResolveMode \) and \( stencilResolveMode \) must be identical.

If \( pDepthStencilResolveAttachment \) is not NULL and does not have the value \( VK_ATTACHMENT_UNUSED \), the \( VkFormat \) of \( pDepthStencilResolveAttachment \) has both depth and stencil components, \( VkPhysicalDeviceDepthStencilResolveProperties::independentResolve \) is \( VK_FALSE \) and \( VkPhysicalDeviceDepthStencilResolveProperties::independentResolveNone \) is \( VK_TRUE \), then the values of \( depthResolveMode \) and \( stencilResolveMode \) must be identical or one of them must be \( VK_RESOLVE_MODE_NONE \).

Valid Usage (Implicit)

- \( VUID-VkSubpassDescriptionDepthStencilResolve-sType-sType \) must be \( VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_DEPTH_STENCIL_RESOLVE \).
- \( VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-parameter \) If \( pDepthStencilResolveAttachment \) is not NULL, \( pDepthStencilResolveAttachment \) must be a valid pointer to a valid \( VkAttachmentReference2 \) structure.

Possible values of \( VkSubpassDescriptionDepthStencilResolve::depthResolveMode \) and \( stencilResolveMode \), specifying the depth and stencil resolve modes, are:

```c
// Provided by VK_VERSION_1_2
typedef enum VkResolveModeFlagBits {
    VK_RESOLVE_MODE_NONE = 0,
    VK_RESOLVE_MODE_SAMPLE_ZERO_BIT = 0x00000001,
    VK_RESOLVE_MODE_AVERAGE_BIT = 0x00000002,
    VK_RESOLVE_MODE_MIN_BIT = 0x00000004,
    VK_RESOLVE_MODE_MAX_BIT = 0x00000008,
} VkResolveModeFlagBits;
```

- \( VK_RESOLVE_MODE_NONE \) indicates that no resolve operation is done.
• **VK_RESOLVE_MODE_SAMPLE_ZERO_BIT** indicates that result of the resolve operation is equal to the value of sample 0.

• **VK_RESOLVE_MODE_AVERAGE_BIT** indicates that result of the resolve operation is the average of the sample values.

• **VK_RESOLVE_MODE_MIN_BIT** indicates that result of the resolve operation is the minimum of the sample values.

• **VK_RESOLVE_MODE_MAX_BIT** indicates that result of the resolve operation is the maximum of the sample values.

```c
// Provided by VK_VERSION_1_2
typedef VkFlags VkResolveModeFlags;
```

`VkResolveModeFlags` is a bitmask type for setting a mask of zero or more `VkResolveModeFlagBits`.

The `VkAttachmentReference2` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkAttachmentReference2 {
    VkStructureType sType;
    const void* pNext;
    uint32_t attachment;
    VkImageLayout layout;
    VkImageAspectFlags aspectMask;
} VkAttachmentReference2;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **attachment** is either an integer value identifying an attachment at the corresponding index in `VkRenderPassCreateInfo2::pAttachments`, or `VK_ATTACHMENT_UNUSED` to signify that this attachment is not used.
- **layout** is a `VkImageLayout` value specifying the layout the attachment uses during the subpass.
- **aspectMask** is a mask of which aspect(s) can be accessed within the specified subpass as an input attachment.

Parameters defined by this structure with the same name as those in `VkAttachmentReference` have the identical effect to those parameters.

`aspectMask` is ignored when this structure is used to describe anything other than an input attachment reference.

If the `separateDepthStencilLayouts` feature is enabled, and `attachment` has a depth/stencil format, `layout` can be set to a layout that only specifies the layout of the depth aspect.

If `layout` only specifies the layout of the depth aspect of the attachment, the layout of the stencil aspect is specified by the `stencilLayout` member of a `VkAttachmentReferenceStencilLayout`
structure included in the \texttt{pNext} chain. Otherwise, \texttt{layout} describes the layout for all relevant image aspects.

### Valid Usage

- VUID-VkAttachmentReference2-layout-03077
  If \texttt{attachment} is not \texttt{VK_ATTACHMENT_UNUSED}, \texttt{layout} must not be \texttt{VK_IMAGE_LAYOUT_UNDEFINED}, \texttt{VK_IMAGE_LAYOUT_PREINITIALIZED}, or \texttt{VK_IMAGE_LAYOUT_PRESENT_SRC_KHR}.

- VUID-VkAttachmentReference2-separateDepthStencilLayouts-03313
  If the \texttt{separateDepthStencilLayouts} feature is not enabled, and \texttt{attachment} is not \texttt{VK_ATTACHMENT_UNUSED}, \texttt{layout} must not be \texttt{VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL}, \texttt{VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL}, \texttt{VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL}, or \texttt{VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL}.

### Valid Usage (Implicit)

- VUID-VkAttachmentReference2-sType-sType
  \texttt{sType} must be \texttt{VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_2}.

- VUID-VkAttachmentReference2-pNext-pNext
  \texttt{pNext} must be \texttt{NULL} or a pointer to a valid instance of \texttt{VkAttachmentReferenceStencilLayout}.

- VUID-VkAttachmentReference2-sType-unique
  The \texttt{sType} value of each struct in the \texttt{pNext} chain must be unique.

- VUID-VkAttachmentReference2-layout-parameter
  \texttt{layout} must be a valid \texttt{VkImageLayout} value.

The \texttt{VkAttachmentReferenceStencilLayout} structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkAttachmentReferenceStencilLayout {
    VkStructureType sType;
    void* pNext;
    VkImageLayout stencilLayout;
} VkAttachmentReferenceStencilLayout;
```

- \texttt{sType} is the type of this structure.
- \texttt{pNext} is \texttt{NULL} or a pointer to a structure extending this structure.
- \texttt{stencilLayout} is a \texttt{VkImageLayout} value specifying the layout the stencil aspect of the attachment uses during the subpass.

### Valid Usage

- VUID-VkAttachmentReferenceStencilLayout-stencilLayout-03318
stencilLayout must not be VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_PREINITIALIZED, VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_PRESENT_SRC_KHR

Valid Usage (Implicit)

- VUID-VkAttachmentReferenceStencilLayout-sType-sType
  sType must be VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_STENCIL_LAYOUT

- VUID-VkAttachmentReferenceStencilLayout-stencilLayout-parameter
  stencilLayout must be a valid VkImageLayout value

The VkSubpassDependency2 structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkSubpassDependency2 {
    VkStructureType          sType;
    const void*              pNext;
    uint32_t                  srcSubpass;
    uint32_t                  dstSubpass;
    VkPipelineStageFlags     srcStageMask;
    VkPipelineStageFlags     dstStageMask;
    VkAccessFlags            srcAccessMask;
    VkAccessFlags            dstAccessMask;
    VkDependencyFlags        dependencyFlags;
    int32_t                   viewOffset;
} VkSubpassDependency2;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **srcSubpass** is the subpass index of the first subpass in the dependency, or VK_SUBPASS_EXTERNAL.
- **dstSubpass** is the subpass index of the second subpass in the dependency, or VK_SUBPASS_EXTERNAL.
- **srcStageMask** is a bitmask of VkPipelineStageFlagBits specifying the source stage mask.
- **dstStageMask** is a bitmask of VkPipelineStageFlagBits specifying the destination stage mask
- **srcAccessMask** is a bitmask of VkAccessFlagBits specifying a source access mask.
- **dstAccessMask** is a bitmask of VkAccessFlagBits specifying a destination access mask.
- **dependencyFlags** is a bitmask of VkDependencyFlagBits.
• **viewOffset** controls which views in the source subpass the views in the destination subpass depend on.

Parameters defined by this structure with the same name as those in **VkSubpassDependency** have the identical effect to those parameters.

**viewOffset** has the same effect for the described subpass dependency as **VkRenderPassMultiviewCreateInfo::pViewOffsets** has on each corresponding subpass dependency.

If a **VkMemoryBarrier2** is included in the **pNext** chain, **srcStageMask**, **dstStageMask**, **srcAccessMask**, and **dstAccessMask** parameters are ignored. The synchronization and access scopes instead are defined by the parameters of **VkMemoryBarrier2**.

---

### Valid Usage

- **VUID-VkSubpassDependency2-srcStageMask-04090**
  If the **geometryShader** feature is not enabled, **srcStageMask** must not contain **VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT**

- **VUID-VkSubpassDependency2-srcStageMask-04091**
  If the **tessellationShader** feature is not enabled, **srcStageMask** must not contain **VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT** or **VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT**

- **VUID-VkSubpassDependency2-srcStageMask-03937**
  If the **synchronization2** feature is not enabled, **srcStageMask** must not be 0

- **VUID-VkSubpassDependency2-dstStageMask-04090**
  If the **geometryShader** feature is not enabled, **dstStageMask** must not contain **VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT**

- **VUID-VkSubpassDependency2-dstStageMask-04091**
  If the **tessellationShader** feature is not enabled, **dstStageMask** must not contain **VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT** or **VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT**

- **VUID-VkSubpassDependency2-dstStageMask-03937**
  If the **synchronization2** feature is not enabled, **dstStageMask** must not be 0

- **VUID-VkSubpassDependency2-srcSubpass-03084**
  **srcSubpass** must be less than or equal to **dstSubpass**, unless one of them is **VK_SUBPASS_EXTERNAL**, to avoid cyclic dependencies and ensure a valid execution order

- **VUID-VkSubpassDependency2-srcSubpass-03085**
  **srcSubpass** and **dstSubpass** must not both be equal to **VK_SUBPASS_EXTERNAL**

- **VUID-VkSubpassDependency2-srcSubpass-06810**
  If **srcSubpass** is equal to **dstSubpass** and **srcStageMask** includes a framebuffer-space stage, **dstStageMask** must only contain framebuffer-space stages

- **VUID-VkSubpassDependency2-srcAccessMask-03088**
  Any access flag included in **srcAccessMask** must be supported by one of the pipeline stages in **srcStageMask**, as specified in the table of supported access types
Any access flag included in `dstAccessMask` must be supported by one of the pipeline stages in `dstStageMask`, as specified in the table of supported access types.

If `dependencyFlags` includes `VK_DEPENDENCY_VIEW_LOCAL_BIT`, `srcSubpass` must not be equal to `VK_SUBPASS_EXTERNAL`.

If `dependencyFlags` includes `VK_DEPENDENCY_VIEW_LOCAL_BIT`, `dstSubpass` must not be equal to `VK_SUBPASS_EXTERNAL`.

If `srcSubpass` equals `dstSubpass`, and `srcStageMask` and `dstStageMask` both include a framebuffer-space stage, then `dependencyFlags` must include `VK_DEPENDENCY_BY_REGION_BIT`.

If `viewOffset` is not equal to 0, `srcSubpass` must not be equal to `dstSubpass`.

If `dependencyFlags` does not include `VK_DEPENDENCY_VIEW_LOCAL_BIT`, `viewOffset` must be 0.

Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_SUBPASS_DEPENDENCY_2`.
- `pNext` must be `NULL` or a pointer to a valid instance of `VkMemoryBarrier2`.
- The `sType` value of each struct in the `pNext` chain must be unique.
- `srcStageMask` must be a valid combination of `VkPipelineStageFlagBits` values.
- `dstStageMask` must be a valid combination of `VkPipelineStageFlagBits` values.
- `srcAccessMask` must be a valid combination of `VkAccessFlagBits` values.
- `dstAccessMask` must be a valid combination of `VkAccessFlagBits` values.
- `dependencyFlags` must be a valid combination of `VkDependencyFlagBits` values.

To destroy a render pass, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyRenderPass(
    VkDevice device,
```
VkRenderPass renderPass,
const VkAllocationCallbacks* pAllocator);

- *device* is the logical device that destroys the render pass.
- *renderPass* is the handle of the render pass to destroy.
- *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage

- VUID-vkDestroyRenderPass-renderPass-00873
  All submitted commands that refer to *renderPass* must have completed execution

- VUID-vkDestroyRenderPass-renderPass-00874
  If VkAllocationCallbacks were provided when *renderPass* was created, a compatible set of callbacks must be provided here

- VUID-vkDestroyRenderPass-renderPass-00875
  If no VkAllocationCallbacks were provided when *renderPass* was created, *pAllocator* must be NULL

### Valid Usage (Implicit)

- VUID-vkDestroyRenderPass-device-parameter
  *device* must be a valid VkDevice handle

- VUID-vkDestroyRenderPass-renderPass-parameter
  If *renderPass* is not VK_NULL_HANDLE, *renderPass* must be a valid VkRenderPass handle

- VUID-vkDestroyRenderPass-pAllocator-parameter
  If *pAllocator* is not NULL, *pAllocator* must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkDestroyRenderPass-renderPass-parent
  If *renderPass* is a valid handle, it must have been created, allocated, or retrieved from *device*

### Host Synchronization

- Host access to *renderPass* must be externally synchronized

**8.2. Render Pass Compatibility**

Framebuffers and graphics pipelines are created based on a specific render pass object. They must only be used with that render pass object, or one compatible with it.

Two attachment references are compatible if they have matching format and sample count, or are
both `VK_ATTACHMENT_UNUSED` or the pointer that would contain the reference is `NULL`.

Two arrays of attachment references are compatible if all corresponding pairs of attachments are compatible. If the arrays are of different lengths, attachment references not present in the smaller array are treated as `VK_ATTACHMENT_UNUSED`.

Two render passes are compatible if their corresponding color, input, resolve, and depth/stencil attachment references are compatible and if they are otherwise identical except for:

- Initial and final image layout in attachment descriptions
- Load and store operations in attachment descriptions
- Image layout in attachment references

As an additional special case, if two render passes have a single subpass, the resolve attachment reference compatibility requirements are ignored.

A framebuffer is compatible with a render pass if it was created using the same render pass or a compatible render pass.

### 8.3. Framebuffers

Render passes operate in conjunction with *framebuffers*. Framebuffers represent a collection of specific memory attachments that a render pass instance uses.

Framebuffers are represented by `VkFramebuffer` handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkFramebuffer)
```

To create a framebuffer, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateFramebuffer(
    VkDevice device,
    const VkFramebufferCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkFramebuffer* pFramebuffer);
```

- `device` is the logical device that creates the framebuffer.
- `pCreateInfo` is a pointer to a `VkFramebufferCreateInfo` structure describing additional information about framebuffer creation.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pFramebuffer` is a pointer to a `VkFramebuffer` handle in which the resulting framebuffer object is returned.
Valid Usage

- VUID-vkCreateFramebuffer-pCreateInfo-02777
  If `pCreateInfo->flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, and `attachmentCount` is not 0, each element of `pCreateInfo->pAttachments` must have been created on `device`

Valid Usage (Implicit)

- VUID-vkCreateFramebuffer-device-parameter
  `device` must be a valid `VkDevice` handle

- VUID-vkCreateFramebuffer-pCreateInfo-parameter
  `pCreateInfo` must be a valid pointer to a valid `VkFramebufferCreateInfo` structure

- VUID-vkCreateFramebuffer-pAllocator-parameter
  If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- VUID-vkCreateFramebuffer-pFramebuffer-parameter
  `pFramebuffer` must be a valid pointer to a `VkFramebuffer` handle

Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkFramebufferCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkFramebufferCreateInfo {
    VkStructureType           sType;
    const void*                pNext;
    VkFramebufferCreateFlags  flags;
    VkRenderPass               renderPass;
    uint32_t                   attachmentCount;
    const VkImageView*         pAttachments;
    uint32_t                   width;
    uint32_t                   height;
    uint32_t                   layers;
} VkFramebufferCreateInfo;
```
• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **flags** is a bitmask of **VkFramebufferCreateFlagBits**

• **renderPass** is a render pass defining what render passes the framebuffer will be compatible with. See **Render Pass Compatibility** for details.

• **attachmentCount** is the number of attachments.

• **pAttachments** is a pointer to an array of **VkImageView** handles, each of which will be used as the corresponding attachment in a render pass instance. If **flags** includes **VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT**, this parameter is ignored.

• **width**, **height** and **layers** define the dimensions of the framebuffer. If the render pass uses multiview, then **layers must** be one and each attachment requires a number of layers that is greater than the maximum bit index set in the view mask in the subpasses in which it is used.

Applications **must** ensure that all non-attachment writes to memory backing image subresources that are used as attachments in a render pass instance happen-before or happen-after the render pass instance. If an image subresource is written during a render pass instance by anything other than load operations, store operations, and layout transitions, applications **must** ensure that all non-attachment reads from memory backing that image subresource happen-before or happen-after the render pass instance. For depth/stencil images, the aspects are not treated independently for the above guarantees - writes to either aspect **must** be synchronized with accesses to the other aspect.

**Note**
An image subresource can be used as read-only as both an attachment and a non-attachment during a render pass instance, but care must still be taken to avoid data races with load/store operations and layout transitions. The simplest way to achieve this is to keep the non-attachment and attachment accesses within the same subpass, or to avoid layout transitions and load/store operations that perform writes.

It is legal for a subpass to use no color or depth/stencil attachments, either because it has no attachment references or because all of them are **VK_ATTACHMENT_UNUSED**. This kind of subpass can use shader side effects such as image stores and atomics to produce an output. In this case, the subpass continues to use the **width**, **height**, and **layers** of the framebuffer to define the dimensions of the rendering area, and the **rasterizationSamples** from each pipeline's **VkPipelineMultisampleStateCreateInfo** to define the number of samples used in rasterization; however, if **VkPhysicalDeviceFeatures::variableMultisampleRate** is **VK_FALSE**, then all pipelines to be bound with the subpass **must** have the same value for **VkPipelineMultisampleStateCreateInfo::rasterizationSamples**.

### Valid Usage

- **VUID-VkFramebufferCreateInfo-attachmentCount-00876**
  **attachmentCount must** be equal to the attachment count specified in **renderPass**

- **VUID-VkFramebufferCreateInfo-flags-02778**
If flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT and attachmentCount is not 0, pAttachments must be a valid pointer to an array of attachmentCount valid VkImageView handles.

• VUID-VkFramebufferCreateInfo-pAttachments-00877
  If flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of pAttachments that is used as a color attachment or resolve attachment by renderPass must have been created with a usage value including VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT.

• VUID-VkFramebufferCreateInfo-pAttachments-02633
  If flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of pAttachments that is used as a depth/stencil attachment by renderPass must have been created with a usage value including VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT.

• VUID-VkFramebufferCreateInfo-pAttachments-02634
  If flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of pAttachments that is used as a depth/stencil resolve attachment by renderPass must have been created with a usage value including VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT.

• VUID-VkFramebufferCreateInfo-pAttachments-00879
  If renderpass is not VK_NULL_HANDLE, flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of pAttachments that is used as an input attachment by renderPass must have been created with a usage value including VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT.

• VUID-VkFramebufferCreateInfo-pAttachments-00880
  If flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of pAttachments must have been created with a VkFormat value that matches the VkFormat specified by the corresponding VkAttachmentDescription in renderPass.

• VUID-VkFramebufferCreateInfo-pAttachments-00881
  If flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of pAttachments must have been created with a samples value that matches the samples value specified by the corresponding VkAttachmentDescription in renderPass.

• VUID-VkFramebufferCreateInfo-flags-04533
  If flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of pAttachments that is used as an input, color, resolve, or depth/stencil attachment by renderPass must have been created with a VkImageCreateInfo::extent.width greater than or equal to width.

• VUID-VkFramebufferCreateInfo-flags-04534
  If flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of pAttachments that is used as an input, color, resolve, or depth/stencil attachment by renderPass must have been created with a VkImageCreateInfo::extent.height greater than or equal to height.

• VUID-VkFramebufferCreateInfo-flags-04535
  If flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of pAttachments that is used as an input, color, resolve, or depth/stencil attachment by renderPass must have been created with a VkImageViewCreateInfo::subresourceRange.layerCount greater than or equal to layers.

• VUID-VkFramebufferCreateInfo-renderPass-04536
If `renderPass` was specified with non-zero view masks, each element of `pAttachments` that is used as an input, color, resolve, or depth/stencil attachment by `renderPass` must have a layerCount greater than the index of the most significant bit set in any of those view masks.

- **VUID-VkFramebufferCreateInfo-pAttachments-00883**
  If flags does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of `pAttachments` must only specify a single mip level.

- **VUID-VkFramebufferCreateInfo-pAttachments-00884**
  If flags does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of `pAttachments` must have been created with the identity swizzle.

- **VUID-VkFramebufferCreateInfo-width-00885**
  width must be greater than 0

- **VUID-VkFramebufferCreateInfo-width-00886**
  width must be less than or equal to `maxFramebufferWidth`

- **VUID-VkFramebufferCreateInfo-height-00887**
  height must be greater than 0

- **VUID-VkFramebufferCreateInfo-height-00888**
  height must be less than or equal to `maxFramebufferHeight`

- **VUID-VkFramebufferCreateInfo-layers-00889**
  layers must be greater than 0

- **VUID-VkFramebufferCreateInfo-layers-00890**
  layers must be less than or equal to `maxFramebufferLayers`

- **VUID-VkFramebufferCreateInfo-renderPass-02531**
  If `renderPass` was specified with non-zero view masks, layers must be 1

- **VUID-VkFramebufferCreateInfo-pAttachments-00891**
  If flags does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of `pAttachments` that is a 2D or 2D array image view taken from a 3D image must not be a depth/stencil format.

- **VUID-VkFramebufferCreateInfo-flags-03189**
  If the imagelessFramebuffer feature is not enabled, flags must not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`

- **VUID-VkFramebufferCreateInfo-flags-03190**
  If flags includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, the pNext chain must include a `VkFramebufferAttachmentsCreateInfo` structure

- **VUID-VkFramebufferCreateInfo-flags-03191**
  If flags includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, the attachmentImageInfoCount member of a `VkFramebufferAttachmentsCreateInfo` structure in the pNext chain must be equal to either zero or `attachmentCount`

- **VUID-VkFramebufferCreateInfo-flags-04541**
  If flags includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, the width member of any element of the `pAttachmentImageInfos` member of a `VkFramebufferAttachmentsCreateInfo` structure in the pNext chain that is used as an input, color, resolve or depth/stencil attachment in `renderPass` must be greater than or equal to width.
If `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, the `height` member of any element of the `pAttachmentImageInfos` member of a `VkFramebufferAttachmentsCreateInfo` structure in the `pNext` chain that is used as an input, color, resolve or depth/stencil attachment in `renderPass` must be greater than or equal to `height`.

If multiview is enabled for `renderPass` and `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, the `layerCount` member of any element of the `pAttachmentImageInfos` member of a `VkFramebufferAttachmentsCreateInfo` structure included in the `pNext` chain used as an input, color, resolve, or depth/stencil attachment in `renderPass` must be greater than the maximum bit index set in the view mask in the subpasses in which it is used in `renderPass`.

If multiview is not enabled for `renderPass` and `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, the `layerCount` member of any element of the `pAttachmentImageInfos` member of a `VkFramebufferAttachmentsCreateInfo` structure included in the `pNext` chain used as an input, color, resolve, or depth/stencil attachment in `renderPass` must be greater than or equal to `layers`.

If `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, the `usage` member of any element of the `pAttachmentImageInfos` member of a `VkFramebufferAttachmentsCreateInfo` structure included in the `pNext` chain that refers to an attachment used as a color attachment or resolve attachment by `renderPass` must include `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`.

If `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, the `usage` member of any element of the `pAttachmentImageInfos` member of a `VkFramebufferAttachmentsCreateInfo` structure included in the `pNext` chain that refers to an attachment used as a depth/stencil attachment by `renderPass` must include `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`.

If `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, the `usage` member of any element of the `pAttachmentImageInfos` member of a `VkFramebufferAttachmentsCreateInfo` structure included in the `pNext` chain that refers to an attachment used as an input attachment by `renderPass` must include `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`.

If `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, at least one element of the `pViewFormats` member of any element of the `pAttachmentImageInfos` member of a `VkFramebufferAttachmentsCreateInfo` structure included in the `pNext` chain must be equal to the corresponding value of `VkAttachmentDescription::format` used to create `renderPass`.

If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of `pAttachments` must have been created with `VkImageViewCreateInfo::viewType` not equal to `VK_IMAGE_VIEW_TYPE_3D`.
Valid Usage (Implicit)

- **VUID-VkFramebufferCreateInfo-sType-sType**
  *sType must be VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO*

- **VUID-VkFramebufferCreateInfo-pNext-pNext**
  *pNext must be NULL or a pointer to a valid instance of VkFramebufferAttachmentsCreateInfo*

- **VUID-VkFramebufferCreateInfo-sType-unique**
  The *sType value of each struct in the pNext chain must be unique*

- **VUID-VkFramebufferCreateInfo-flags-parameter**
  *flags must be a valid combination of VkFramebufferCreateFlagBits values*

- **VUID-VkFramebufferCreateInfo-renderPass-parameter**
  *renderPass must be a valid VkRenderPass handle*

- **VUID-VkFramebufferCreateInfo-commonparent**
  Both of renderPass, and the elements of pAttachments that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same VkDevice

The VkFramebufferAttachmentsCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkFramebufferAttachmentsCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t attachmentImageInfoCount;
    const VkFramebufferAttachmentImageInfo* pAttachmentImageInfos;
} VkFramebufferAttachmentsCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **attachmentImageInfoCount** is the number of attachments being described.
- **pAttachmentImageInfos** is a pointer to an array of VkFramebufferAttachmentImageInfo structures, each structure describing a number of parameters of the corresponding attachment in a render pass instance.

Valid Usage (Implicit)

- **VUID-VkFramebufferAttachmentsCreateInfo-sType-sType**
  *sType must be VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENTS_CREATE_INFO*

- **VUID-VkFramebufferAttachmentsCreateInfo-attachmentImageInfoCount-attachmentImageInfoCount**
  If attachmentImageInfoCount is not 0, pAttachmentImageInfos must be a valid pointer to an array of attachmentImageInfoCount valid VkFramebufferAttachmentImageInfo structures
The `VkFramebufferAttachmentImageInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkFramebufferAttachmentImageInfo {
    VkStructureType    sType;
    const void*        pNext;
    VkImageCreateFlags flags;
    VkImageUsageFlags  usage;
    uint32_t           width;
    uint32_t           height;
    uint32_t           layerCount;
    uint32_t           viewFormatCount;
    const VkFormat*    pViewFormats;
} VkFramebufferAttachmentImageInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is a bitmask of `VkImageCreateFlagBits`, matching the value of `VkImageCreateInfo::flags` used to create an image that will be used with this framebuffer.
- **usage** is a bitmask of `VkImageUsageFlagBits`, matching the value of `VkImageCreateInfo::usage` used to create an image used with this framebuffer.
- **width** is the width of the image view used for rendering.
- **height** is the height of the image view used for rendering.
- **layerCount** is the number of array layers of the image view used for rendering.
- **viewFormatCount** is the number of entries in the `pViewFormats` array, matching the value of `VkImageFormatListCreateInfo::viewFormatCount` used to create an image used with this framebuffer.
- **pViewFormats** is a pointer to an array of `VkFormat` values specifying all of the formats which can be used when creating views of the image, matching the value of `VkImageFormatListCreateInfo::pViewFormats` used to create an image used with this framebuffer.

Images that can be used with the framebuffer when beginning a render pass, as specified by `VkRenderPassAttachmentBeginInfo`, must be created with parameters that are identical to those specified here.

**Valid Usage (Implicit)**

- VUID-VkFramebufferAttachmentImageInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENT_IMAGE_INFO`
- VUID-VkFramebufferAttachmentImageInfo-pNext-pNext
  `pNext` must be NULL
- VUID-VkFramebufferAttachmentImageInfo-flags-parameter
  `flags` must be a valid combination of `VkImageCreateFlagBits` values
- VUID-VkFramebufferAttachmentImageInfo-usage-parameter
  usage must be a valid combination of VkImageUsageFlagBits values
- VUID-VkFramebufferAttachmentImageInfo-usage-requiredbitmask
  usage must not be 0
- VUID-VkFramebufferAttachmentImageInfo-pViewFormats-parameter
  If viewFormatCount is not 0, pViewFormats must be a valid pointer to an array of
  viewFormatCount valid VkFormat values

Bits which can be set in VkFramebufferCreateInfo::flags, specifying options for framebuffers, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkFramebufferCreateFlagBits {
    // Provided by VK_VERSION_1_2
    VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT = 0x00000001,
} VkFramebufferCreateFlagBits;
```

- VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT specifies that image views are not specified, and only
  attachment compatibility information will be provided via a
  VkFramebufferAttachmentImageInfo structure.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkFramebufferCreateFlags;
```

VkFramebufferCreateFlags is a bitmask type for setting a mask of zero or more
VkFramebufferCreateFlagBits.

To destroy a framebuffer, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyFramebuffer(
    VkDevice                      device,        
    VkFramebuffer               framebuffer,    
    const VkAllocationCallbacks* pAllocator);    
```

- device is the logical device that destroys the framebuffer.
- framebuffer is the handle of the framebuffer to destroy.
- pAllocator controls host memory allocation as described in the Memory Allocation chapter.

**Valid Usage**

- VUID-vkDestroyFramebuffer-framebuffer-00892
  All submitted commands that refer to framebuffer must have completed execution
- VUID-vkDestroyFramebuffer-framebuffer-00893
If \texttt{VkAllocationCallbacks} were provided when \texttt{framebuffer} was created, a compatible set of callbacks \textbf{must} be provided here

- VUID-vkDestroyFramebuffer-framebuffer-00894
  If no \texttt{VkAllocationCallbacks} were provided when \texttt{framebuffer} was created, \texttt{pAllocator} \textbf{must} be \texttt{NULL}

\section*{Valid Usage (Implicit)}

- VUID-vkDestroyFramebuffer-device-parameter
  \textit{device} \textbf{must} be a valid \texttt{VkDevice} handle

- VUID-vkDestroyFramebuffer-framebuffer-parameter
  If \texttt{framebuffer} is not \texttt{VK_NULL_HANDLE}, \texttt{framebuffer} \textbf{must} be a valid \texttt{VkFramebuffer} handle

- VUID-vkDestroyFramebuffer-pAllocator-parameter
  If \texttt{pAllocator} is not \texttt{NULL}, \texttt{pAllocator} \textbf{must} be a valid pointer to a valid \texttt{VkAllocationCallbacks} structure

- VUID-vkDestroyFramebuffer-framebuffer-parent
  If \texttt{framebuffer} is a valid handle, it \textbf{must} have been created, allocated, or retrieved from \texttt{device}

\section*{Host Synchronization}

- Host access to \texttt{framebuffer} \textbf{must} be externally synchronized

\subsection*{8.4. Render Pass Commands}

An application records the commands for a render pass instance one subpass at a time, by beginning a render pass instance, iterating over the subpasses to record commands for that subpass, and then ending the render pass instance.

To begin a render pass instance, call:

\begin{verbatim}
// Provided by VK_VERSION_1_0
void vkCmdBeginRenderPass(
    VkCommandBuffer commandBuffer,
    const VkRenderPassBeginInfo* pRenderPassBegin,
    VkSubpassContents contents);
\end{verbatim}

- \texttt{commandBuffer} is the command buffer in which to record the command.

- \texttt{pRenderPassBegin} is a pointer to a \texttt{VkRenderPassBeginInfo} structure specifying the render pass to begin an instance of, and the framebuffer the instance uses.

- \texttt{contents} is a \texttt{VkSubpassContents} value specifying how the commands in the first subpass will be
After beginning a render pass instance, the command buffer is ready to record the commands for the first subpass of that render pass.

Valid Usage

• VUID-vkCmdBeginRenderPass-initialLayout-00895
  If any of the initialLayout or finalLayout member of the VkAttachmentDescription structures or the layout member of the VkAttachmentReference structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL then the corresponding attachment image view of the framebuffer specified in the framebuffer member of pRenderPassBegin must have been created with a usage value including VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT

• VUID-vkCmdBeginRenderPass-initialLayout-01758
  If any of the initialLayout or finalLayout member of the VkAttachmentDescription structures or the layout member of the VkAttachmentReference structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL then the corresponding attachment image view of the framebuffer specified in the framebuffer member of pRenderPassBegin must have been created with a usage value including VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT

• VUID-vkCmdBeginRenderPass-initialLayout-02842
  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_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL then the corresponding attachment image view of the framebuffer specified in the framebuffer member of pRenderPassBegin must have been created with a usage value including VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT

• VUID-vkCmdBeginRenderPass-stencilInitialLayout-02843
  If any of the stencilInitialLayout or stencilFinalLayout member of the VkAttachmentDescriptionStencilLayout structures or the stencilLayout member of the VkAttachmentReferenceStencilLayout structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL then the corresponding attachment image view of the framebuffer specified in the framebuffer member of pRenderPassBegin must have been created with a usage value including VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT

• VUID-vkCmdBeginRenderPass-initialLayout-00897
If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` must have been created with a `usage` value including `VK_IMAGE_USAGE_SAMPLED_BIT` or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`

• VUID-vkCmdBeginRenderPass-initialLayout-00898
If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` must have been created with a `usage` value including `VK_IMAGE_USAGE_TRANSFER_SRC_BIT`

• VUID-vkCmdBeginRenderPass-initialLayout-00899
If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` must have been created with a `usage` value including `VK_IMAGE_USAGE_TRANSFER_DST_BIT`

• VUID-vkCmdBeginRenderPass-initialLayout-00900
If the `initialLayout` member of any of the `VkAttachmentDescription` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is not `VK_IMAGE_LAYOUT_UNDEFINED`, then each such `initialLayout` must be equal to the current layout of the corresponding attachment image subresource of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin`

• VUID-vkCmdBeginRenderPass-srcStageMask-06451
The `srcStageMask` members of any element of the `pDependencies` member of `VkRenderPassCreateInfo` used to create `renderPass` must be supported by the capabilities of the queue family identified by the `queueFamilyIndex` member of the `VkCommandPoolCreateInfo` used to create the command pool which `commandBuffer` was allocated from

• VUID-vkCmdBeginRenderPass-dstStageMask-06452
The `dstStageMask` members of any element of the `pDependencies` member of `VkRenderPassCreateInfo` used to create `renderPass` must be supported by the capabilities of the queue family identified by the `queueFamilyIndex` member of the `VkCommandPoolCreateInfo` used to create the command pool which `commandBuffer` was allocated from

• VUID-vkCmdBeginRenderPass-framebuffer-02532
For any attachment in `framebuffer` that is used by `renderPass` and is bound to memory locations that are also bound to another attachment used by `renderPass`, and if at least one of those uses causes either attachment to be written to, both attachments must have had the `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT` set
Valid Usage (Implicit)

- VUID-vkCmdBeginRenderPass-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle
- VUID-vkCmdBeginRenderPass-pRenderPassBegin-parameter
  pRenderPassBegin must be a valid pointer to a valid VkRenderPassBeginInfo structure
- VUID-vkCmdBeginRenderPass-contents-parameter
  contents must be a valid VkSubpassContents value
- VUID-vkCmdBeginRenderPass-commandBuffer-recording
  commandBuffer must be in the recording state
- VUID-vkCmdBeginRenderPass-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations
- VUID-vkCmdBeginRenderPass-renderpass
  This command must only be called outside of a render pass instance
- VUID-vkCmdBeginRenderPass-bufferlevel
  commandBuffer must be a primary VkCommandBuffer

Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td>Action State Synchronization</td>
</tr>
</tbody>
</table>

Alternatively to begin a render pass, call:

```c
// Provided by VK_VERSION_1_2
void vkCmdBeginRenderPass2(
    VkCommandBuffer commandBuffer,
    const VkRenderPassBeginInfo* pRenderPassBegin,
    const VkSubpassBeginInfo* pSubpassBeginInfo);
```
• `commandBuffer` is the command buffer in which to record the command.

• `pRenderPassBegin` is a pointer to a `VkRenderPassBeginInfo` structure specifying the render pass to begin an instance of, and the framebuffer the instance uses.

• `pSubpassBeginInfo` is a pointer to a `VkSubpassBeginInfo` structure containing information about the subpass which is about to begin rendering.

After beginning a render pass instance, the command buffer is ready to record the commands for the first subpass of that render pass.

### Valid Usage

- **VUID-vkCmdBeginRenderPass2-framebuffer-02779**
  Both the `framebuffer` and `renderPass` members of `pRenderPassBegin` must have been created on the same `VkDevice` that `commandBuffer` was allocated on.

- **VUID-vkCmdBeginRenderPass2-initialLayout-03094**
  If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` must have been created with a usage value including `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`.

- **VUID-vkCmdBeginRenderPass2-initialLayout-03096**
  If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`, or `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` must have been created with a usage value including `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`.

- **VUID-vkCmdBeginRenderPass2-initialLayout-02844**
  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_ATTACHMENT_OPTIMAL`, or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL`, or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL` then the corresponding depth attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` must have been created with a usage value including `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`.

- **VUID-vkCmdBeginRenderPass2-stencilInitialLayout-02845**
  If any of the `stencilInitialLayout` or `stencilFinalLayout` member of the `VkAttachmentDescriptionStencilLayout` structures or the `stencilLayout` member of the
VkAttachmentReferenceStencilLayout structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL then the corresponding attachment image view of the framebuffer specified in the framebuffer member of pRenderPassBegin must have been created with a usage value including VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT

• VUID-vkCmdBeginRenderPass2-initialLayout-03097
  If any of the initialLayout or finalLayout member of the VkAttachmentDescription structures or the layout member of the VkAttachmentReference structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL then the corresponding attachment image view of the framebuffer specified in the framebuffer member of pRenderPassBegin must have been created with a usage value including VK_IMAGE_USAGE_SAMPLED_BIT or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT

• VUID-vkCmdBeginRenderPass2-initialLayout-03089
  If any of the initialLayout or finalLayout member of the VkAttachmentDescription structures or the layout member of the VkAttachmentReference structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL then the corresponding attachment image view of the framebuffer specified in the framebuffer member of pRenderPassBegin must have been created with a usage value including VK_IMAGE_USAGE_TRANSFER_SRC_BIT

• VUID-vkCmdBeginRenderPass2-initialLayout-03090
  If the initialLayout member of any of the VkAttachmentDescription structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is not VK_IMAGE_LAYOUT_UNDEFINED, then each such initialLayout must be equal to the current layout of the corresponding attachment image subresource of the framebuffer specified in the framebuffer member of pRenderPassBegin

• VUID-vkCmdBeginRenderPass2-srcStageMask-06453
  The srcStageMask members of any element of the pDependencies member of VkRenderPassCreateInfo used to create renderPass must be supported by the capabilities of the queue family identified by the queueFamilyIndex member of the VkCommandPoolCreateInfo used to create the command pool which commandBuffer was allocated from

• VUID-vkCmdBeginRenderPass2-dstStageMask-06454
  The dstStageMask members of any element of the pDependencies member of VkRenderPassCreateInfo used to create renderPass must be supported by the capabilities of the queue family identified by the queueFamilyIndex member of the
**VkCommandPoolCreateInfo** used to create the command pool which **commandBuffer** was allocated from

- VUID-vkCmdBeginRenderPass2-framebuffer-02533
  For any attachment in **framebuffer** that is used by **renderPass** and is bound to memory locations that are also bound to another attachment used by **renderPass**, and if at least one of those uses causes either attachment to be written to, both attachments **must** have had the **VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT** set

---

## Valid Usage (Implicit)

- VUID-vkCmdBeginRenderPass2-commandBuffer-parameter
  **commandBuffer** **must** be a valid **VkCommandBuffer** handle

- VUID-vkCmdBeginRenderPass2-pRenderPassBegin-parameter
  **pRenderPassBegin** **must** be a valid pointer to a valid **VkRenderPassBeginInfo** structure

- VUID-vkCmdBeginRenderPass2-pSubpassBeginInfo-parameter
  **pSubpassBeginInfo** **must** be a valid pointer to a valid **VkSubpassBeginInfo** structure

- VUID-vkCmdBeginRenderPass2-commandBuffer-recording
  **commandBuffer** **must** be in the **recording state**

- VUID-vkCmdBeginRenderPass2-commandBuffer-cmdpool
  The **VkCommandPool** that **commandBuffer** was allocated from **must** support graphics operations

- VUID-vkCmdBeginRenderPass2-renderpass
  This command **must** only be called outside of a render pass instance

- VUID-vkCmdBeginRenderPass2-bufferlevel
  **commandBuffer** **must** be a primary **VkCommandBuffer**

---

## Host Synchronization

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

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

---

## Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td>Action State Synchronization</td>
</tr>
</tbody>
</table>
The `VkRenderPassBeginInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkRenderPassBeginInfo {
    VkStructureType sType;
    const void* pNext;
    VkRenderPass renderPass;
    VkFramebuffer framebuffer;
    VkRect2D renderArea;
    uint32_t clearValueCount;
    const VkClearValue* pClearValues;
} VkRenderPassBeginInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `renderPass` is the render pass to begin an instance of.
- `framebuffer` is the framebuffer containing the attachments that are used with the render pass.
- `renderArea` is the render area that is affected by the render pass instance, and is described in more detail below.
- `clearValueCount` is the number of elements in `pClearValues`.
- `pClearValues` is a pointer to an array of `clearValueCount` `VkClearValue` structures containing clear values for each attachment, if the attachment uses a `loadOp` value of `VK_ATTACHMENT_LOAD_OP_CLEAR` or if the attachment has a depth/stencil format and uses a `stencilLoadOp` value of `VK_ATTACHMENT_LOAD_OPCLEAR`. The array is indexed by attachment number. Only elements corresponding to cleared attachments are used. Other elements of `pClearValues` are ignored.

`renderArea` is the render area that is affected by the render pass instance. The effects of attachment load, store and multisample resolve operations are restricted to the pixels whose x and y coordinates fall within the render area on all attachments. The render area extends to all layers of `framebuffer`. The application must ensure (using scissor if necessary) that all rendering is contained within the render area. The render area must be contained within the framebuffer dimensions.

When multiview is enabled, the resolve operation at the end of a subpass applies to all views in the view mask.

**Note**
There may be a performance cost for using a render area smaller than the framebuffer, unless it matches the render area granularity for the render pass.

**Valid Usage**

- VUID-VkRenderPassBeginInfo-clearValueCount-00902
  `clearValueCount` must be greater than the largest attachment index in `renderPass` specifying a `loadOp` (or `stencilLoadOp`, if the attachment has a depth/stencil format) of
VK_ATTACHMENT_LOAD_OP_CLEAR

- VUID-VkRenderPassBeginInfo-clearValueCount-04962
  If `clearValueCount` is not 0, `pClearValues` must be a valid pointer to an array of `clearValueCount` `VkClearValue` unions.

- VUID-VkRenderPassBeginInfo-renderPass-00904
  `renderPass` must be compatible with the `renderPass` member of the `VkFramebufferCreateInfo` structure specified when creating `framebuffer`.

- VUID-VkRenderPassBeginInfo-pNext-02850
  If the `pNext` chain does not contain `VkDeviceGroupRenderPassBeginInfo` or its `deviceRenderAreaCount` member is equal to 0, `renderArea.offset.x` must be greater than or equal to 0.

- VUID-VkRenderPassBeginInfo-pNext-02851
  If the `pNext` chain does not contain `VkDeviceGroupRenderPassBeginInfo` or its `deviceRenderAreaCount` member is equal to 0, `renderArea.offset.y` must be greater than or equal to 0.

- VUID-VkRenderPassBeginInfo-pNext-02852
  If the `pNext` chain does not contain `VkDeviceGroupRenderPassBeginInfo` or its `deviceRenderAreaCount` member is equal to 0, `renderArea.offset.x + renderArea.extent.width` must be less than or equal to `VkFramebufferCreateInfo::width` the `framebuffer` was created with.

- VUID-VkRenderPassBeginInfo-pNext-02853
  If the `pNext` chain does not contain `VkDeviceGroupRenderPassBeginInfo` or its `deviceRenderAreaCount` member is equal to 0, `renderArea.offset.y + renderArea.extent.height` must be less than or equal to `VkFramebufferCreateInfo::height` the `framebuffer` was created with.

- VUID-VkRenderPassBeginInfo-pNext-02856
  If the `pNext` chain contains `VkDeviceGroupRenderPassBeginInfo`, `offset.x + extent.width` of each element of `pDeviceRenderAreas` must be less than or equal to `VkFramebufferCreateInfo::width` the `framebuffer` was created with.

- VUID-VkRenderPassBeginInfo-pNext-02857
  If the `pNext` chain contains `VkDeviceGroupRenderPassBeginInfo`, `offset.y + extent.height` of each element of `pDeviceRenderAreas` must be less than or equal to `VkFramebufferCreateInfo::height` the `framebuffer` was created with.

- VUID-VkRenderPassBeginInfo-framebuffer-03207
  If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that did not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, and the `pNext` chain includes a `VkRenderPassAttachmentBeginInfo` structure, its `attachmentCount` must be zero.

- VUID-VkRenderPassBeginInfo-framebuffer-03208
  If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, the `attachmentCount` of a `VkRenderPassAttachmentBeginInfo` structure included in the `pNext` chain must be equal to the value of `VkFramebufferAttachmentsCreateInfo::attachmentImageInfoCount` used to create `framebuffer`.
If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of the `pAttachments` member of a `VkRenderPassAttachmentBeginInfo` structure included in the `pNext` chain must have been created on the same `VkDevice` as `framebuffer` and `renderPass`.

If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of the `pAttachments` member of a `VkRenderPassAttachmentBeginInfo` structure included in the `pNext` chain must be a `VkImageView` of an image created with a value of `VkImageCreateInfo::flags` equal to the `flags` member of the corresponding element of `VkFramebufferAttachmentsCreateInfo::pAttachmentImageInfos` used to create `framebuffer`.

If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of the `pAttachments` member of a `VkRenderPassAttachmentBeginInfo` structure included in the `pNext` chain must be a `VkImageView` with an inherited usage equal to the `usage` member of the corresponding element of `VkFramebufferAttachmentsCreateInfo::pAttachmentImageInfos` used to create `framebuffer`.

If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of the `pAttachments` member of a `VkRenderPassAttachmentBeginInfo` structure included in the `pNext` chain must be a `VkImageView` with a width equal to the `width` member of the corresponding element of `VkFramebufferAttachmentsCreateInfo::pAttachmentImageInfos` used to create `framebuffer`.

If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of the `pAttachments` member of a `VkRenderPassAttachmentBeginInfo` structure included in the `pNext` chain must be a `VkImageView` with a height equal to the `height` member of the corresponding element of `VkFramebufferAttachmentsCreateInfo::pAttachmentImageInfos` used to create `framebuffer`.

If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of the `pAttachments` member of a `VkRenderPassAttachmentBeginInfo` structure included in the `pNext` chain must be a `VkImageView` of an image created with a value of `VkImageViewCreateInfo::subresourceRange.layerCount` equal to the `layerCount` member of the corresponding element of `VkFramebufferAttachmentsCreateInfo::pAttachmentImageInfos` used to create `framebuffer`.

If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of the `pAttachments` member of a `VkRenderPassAttachmentBeginInfo` structure included in the `pNext` chain must be a `VkImageView` of an image created with a value of `VkImageFormatListCreateInfo::viewFormatCount` equal to the `viewFormatCount` member of the corresponding element of `VkFramebufferAttachmentsCreateInfo::pAttachmentImageInfos` used to create `framebuffer`.
VkFramebufferAttachmentsCreateInfo::pAttachmentImageInfos used to create framebuffer

• VUID-VkRenderPassBeginInfo-framebuffer-03215
  If framebuffer was created with a VkFramebufferCreateInfo::flags value that included VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of the pAttachments member of a VkRenderPassAttachmentBeginInfo structure included in the pNext chain must be a VkImageView of an image created with a set of elements in VkImageFormatListCreateInfo::pViewFormats equal to the set of elements in the pViewFormats member of the corresponding element of VkFramebufferAttachmentsCreateInfo::pAttachmentImageInfos used to create framebuffer

• VUID-VkRenderPassBeginInfo-framebuffer-03216
  If framebuffer was created with a VkFramebufferCreateInfo::flags value that included VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of the pAttachments member of a VkRenderPassAttachmentBeginInfo structure included in the pNext chain must be a VkImageView of an image created with a value of VkImageViewCreateInfo::format equal to the corresponding value of VkAttachmentDescription::format in renderPass

• VUID-VkRenderPassBeginInfo-framebuffer-03217
  If framebuffer was created with a VkFramebufferCreateInfo::flags value that included VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of the pAttachments member of a VkRenderPassAttachmentBeginInfo structure included in the pNext chain must be a VkImageView of an image created with a value of VkImageCreateInfo::samples equal to the corresponding value of VkAttachmentDescription::samples in renderPass

Valid Usage (Implicit)

• VUID-VkRenderPassBeginInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO

• VUID-VkRenderPassBeginInfo-pNext-pNext
  Each pNext member of any structure (including this one) in the pNext chain must be either NULL or a pointer to a valid instance of VkDeviceGroupRenderPassBeginInfo or VkRenderPassAttachmentBeginInfo

• VUID-VkRenderPassBeginInfo-sType-unique
  The sType value of each struct in the pNext chain must be unique

• VUID-VkRenderPassBeginInfo-renderPass-parameter
  renderPass must be a valid VkRenderPass handle

• VUID-VkRenderPassBeginInfo-framebuffer-parameter
  framebuffer must be a valid VkFramebuffer handle

• VUID-VkRenderPassBeginInfo-commonparent
  Both of framebuffer, and renderPass must have been created, allocated, or retrieved from the same VkDevice

The VkSubpassBeginInfo structure is defined as:

```cpp
// Provided by VK_VERSION_1_2
```
typedef struct VkSubpassBeginInfo {
    VkStructureType sType;
    const void* pNext;
    VkSubpassContents contents;
} VkSubpassBeginInfo;

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `contents` is a `VkSubpassContents` value specifying how the commands in the next subpass will be provided.

### Valid Usage (Implicit)

- VUID-VkSubpassBeginInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_SUBPASS_BEGIN_INFO`
- VUID-VkSubpassBeginInfo-pNext-pNext
  `pNext` must be `NULL`
- VUID-VkSubpassBeginInfo-contents-parameter
  `contents` must be a valid `VkSubpassContents` value

Possible values of `vkCmdBeginRenderPass::contents`, specifying how the commands in the first subpass will be provided, are:

```
// Provided by VK_VERSION_1_0
typedef enum VkSubpassContents {
    VK_SUBPASS_CONTENTS_INLINE = 0,
    VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS = 1,
} VkSubpassContents;
```

- `VK_SUBPASS_CONTENTS_INLINE` specifies that the contents of the subpass will be recorded inline in the primary command buffer, and secondary command buffers `must` not be executed within the subpass.
- `VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS` specifies that the contents are recorded in secondary command buffers that will be called from the primary command buffer, and `vkCmdExecuteCommands` is the only valid command on the command buffer until `vkCmdNextSubpass` or `vkCmdEndRenderPass`.

If the `pNext` chain of `VkRenderPassBeginInfo` or `VkRenderingInfo` includes a `VkDeviceGroupRenderPassBeginInfo` structure, then that structure includes a device mask and set of render areas for the render pass instance.

The `VkDeviceGroupRenderPassBeginInfo` structure is defined as:

```
// Provided by VK_VERSION_1_1
```
typedef struct VkDeviceGroupRenderPassBeginInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t deviceMask;
    uint32_t deviceRenderAreaCount;
    const VkRect2D* pDeviceRenderAreas;
} VkDeviceGroupRenderPassBeginInfo;

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- deviceMask is the device mask for the render pass instance.
- deviceRenderAreaCount is the number of elements in the pDeviceRenderAreas array.
- pDeviceRenderAreas is a pointer to an array of VkRect2D structures defining the render area for each physical device.

The deviceMask serves several purposes. It is an upper bound on the set of physical devices that can be used during the render pass instance, and the initial device mask when the render pass instance begins. In addition, commands transitioning to the next subpass in a render pass instance and commands ending the render pass instance, and, accordingly render pass attachment load, store, and resolve operations and subpass dependencies corresponding to the render pass instance, are executed on the physical devices included in the device mask provided here.

If deviceRenderAreaCount is not zero, then the elements of pDeviceRenderAreas override the value of VkRenderPassBeginInfo::renderArea, and provide a render area specific to each physical device. These render areas serve the same purpose as VkRenderPassBeginInfo::renderArea, including controlling the region of attachments that are cleared by VK_ATTACHMENT_LOAD_OP_CLEAR and that are resolved into resolve attachments.

If this structure is not present, the render pass instance’s device mask is the value of VkDeviceGroupCommandBufferBeginInfo::deviceMask. If this structure is not present or if deviceRenderAreaCount is zero, VkRenderPassBeginInfo::renderArea is used for all physical devices.

Valid Usage

- VUID-VkDeviceGroupRenderPassBeginInfo-deviceMask-00905
deviceMask must be a valid device mask value

- VUID-VkDeviceGroupRenderPassBeginInfo-deviceMask-00906
deviceMask must not be zero

- VUID-VkDeviceGroupRenderPassBeginInfo-deviceMask-00907
deviceMask must be a subset of the command buffer’s initial device mask

- VUID-VkDeviceGroupRenderPassBeginInfo-deviceRenderAreaCount-00908
deviceRenderAreaCount must either be zero or equal to the number of physical devices in the logical device

- VUID-VkDeviceGroupRenderPassBeginInfo-offset-06166
The offset.x member of any element of pDeviceRenderAreas must be greater than or equal
The offset.y member of any element of pDeviceRenderAreas must be greater than or equal to 0

The sum of the offset.x and extent.width members of any element of pDeviceRenderAreas must be less than or equal to maxFramebufferWidth

The sum of the offset.y and extent.height members of any element of pDeviceRenderAreas must be less than or equal to maxFramebufferHeight

Valid Usage (Implicit)

sType must be VK_STRUCTURE_TYPE_DEVICE_GROUP_RENDER_PASS_BEGIN_INFO

If deviceRenderAreaCount is not 0, pDeviceRenderAreas must be a valid pointer to an array of deviceRenderAreaCount VkRect2D structures

The VkRenderPassAttachmentBeginInfo structure is defined as:

```
// Provided by VK_VERSION_1_2
typedef struct VkRenderPassAttachmentBeginInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t attachmentCount;
    const VkImageView* pAttachments;
} VkRenderPassAttachmentBeginInfo;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- attachmentCount is the number of attachments.
- pAttachments is a pointer to an array of VkImageView handles, each of which will be used as the corresponding attachment in the render pass instance.

Valid Usage

Each element of pAttachments must only specify a single mip level

Each element of pAttachments must have been created with the identity swizzle

Each element of pAttachments must have been created with the identity swizzle
Each element of `pAttachments` must have been created with `VkImageViewCreateInfo::viewType` not equal to `VK_IMAGE_VIEW_TYPE_3D`.

### Valid Usage (Implicit)

- **VUID-VkRenderPassAttachmentBeginInfo-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_RENDER_PASS_ATTACHMENT_BEGIN_INFO`

- **VUID-VkRenderPassAttachmentBeginInfo-pAttachments-parameter**

  If `attachmentCount` is not 0, `pAttachments` must be a valid pointer to an array of `attachmentCount` valid `VkImageView` handles.

To query the render area granularity, call:

```c
// Provided by VK_VERSION_1_0
void vkGetRenderAreaGranularity(
    VkDevice device,
    VkRenderPass renderPass,
    VkExtent2D* pGranularity);
```

- `device` is the logical device that owns the render pass.
- `renderPass` is a handle to a render pass.
- `pGranularity` is a pointer to a `VkExtent2D` structure in which the granularity is returned.

The conditions leading to an optimal `renderArea` are:

- the `offset.x` member in `renderArea` is a multiple of the `width` member of the returned `VkExtent2D` (the horizontal granularity).

- the `offset.y` member in `renderArea` is a multiple of the `height` member of the returned `VkExtent2D` (the vertical granularity).

- either the `extent.width` member in `renderArea` is a multiple of the horizontal granularity or `offset.x+extent.width` is equal to the `width` of the `framebuffer` in the `VkRenderPassBeginInfo`.

- either the `extent.height` member in `renderArea` is a multiple of the vertical granularity or `offset.y+extent.height` is equal to the `height` of the `framebuffer` in the `VkRenderPassBeginInfo`.

Subpass dependencies are not affected by the render area, and apply to the entire image subresources attached to the framebuffer as specified in the description of automatic layout transitions. Similarly, pipeline barriers are valid even if their effect extends outside the render area.

### Valid Usage (Implicit)

- **VUID-vkGetRenderAreaGranularity-device-parameter**

  `device` must be a valid `VkDevice` handle.
To transition to the next subpass in the render pass instance after recording the commands for a subpass, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdNextSubpass(
    VkCommandBuffer commandBuffer,  
    VkSubpassContents contents);
```

- `commandBuffer` is the command buffer in which to record the command.
- `contents` specifies how the commands in the next subpass will be provided, in the same fashion as the corresponding parameter of `vkCmdBeginRenderPass`.

The subpass index for a render pass begins at zero when `vkCmdBeginRenderPass` is recorded, and increments each time `vkCmdNextSubpass` is recorded.

Moving to the next subpass automatically performs any multisample resolve operations in the subpass being ended. End-of-subpass multisample resolves are treated as color attachment writes for the purposes of synchronization. This applies to resolve operations for both color and depth/stencil attachments. That is, they are considered to execute in the `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` pipeline stage and their writes are synchronized with `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`. Synchronization between rendering within a subpass and any resolve operations at the end of the subpass occurs automatically, without need for explicit dependencies or pipeline barriers. However, if the resolve attachment is also used in a different subpass, an explicit dependency is needed.

After transitioning to the next subpass, the application can record the commands for that subpass.

### Valid Usage

- VUID-vkCmdNextSubpass-None-00909
  The current subpass index must be less than the number of subpasses in the render pass minus one

### Valid Usage (Implicit)

- VUID-vkCmdNextSubpass-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle
To transition to the next subpass in the render pass instance after recording the commands for a subpass, call:

```c
// Provided by VK_VERSION_1_2
void vkCmdNextSubpass2(
    VkCommandBuffer commandBuffer,
    const VkSubpassBeginInfo* pSubpassBeginInfo,
    const VkSubpassEndInfo* pSubpassEndInfo);
```

- `commandBuffer` is the command buffer in which to record the command.
- `pSubpassBeginInfo` is a pointer to a `VkSubpassBeginInfo` structure containing information about the subpass which is about to begin rendering.
- `pSubpassEndInfo` is a pointer to a `VkSubpassEndInfo` structure containing information about how the previous subpass will be ended.
vkCmdNextSubpass2 is semantically identical to vkCmdNextSubpass, except that it is extensible, and that contents is provided as part of an extensible structure instead of as a flat parameter.

### Valid Usage

- VUID-vkCmdNextSubpass2-None-03102
  The current subpass index must be less than the number of subpasses in the render pass minus one

### Valid Usage (Implicit)

- VUID-vkCmdNextSubpass2-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle
- VUID-vkCmdNextSubpass2-pSubpassBeginInfo-parameter
  pSubpassBeginInfo must be a valid pointer to a valid VkSubpassBeginInfo structure
- VUID-vkCmdNextSubpass2-pSubpassEndInfo-parameter
  pSubpassEndInfo must be a valid pointer to a valid VkSubpassEndInfo structure
- VUID-vkCmdNextSubpass2-commandBuffer-recording
  commandBuffer must be in the recording state
- VUID-vkCmdNextSubpass2-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations
- VUID-vkCmdNextSubpass2-renderpass
  This command must only be called inside of a render pass instance
- VUID-vkCmdNextSubpass2-bufferlevel
  commandBuffer must be a primary VkCommandBuffer

### Host Synchronization

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Action State Synchronization</td>
</tr>
</tbody>
</table>
To record a command to end a render pass instance after recording the commands for the last subpass, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdEndRenderPass(
    VkCommandBuffer commandBuffer);
```

- `commandBuffer` is the command buffer in which to end the current render pass instance.

Ending a render pass instance performs any multisample resolve operations on the final subpass.

### Valid Usage

- VUID-vkCmdEndRenderPass-None-00910
  The current subpass index **must** be equal to the number of subpasses in the render pass minus one
- VUID-vkCmdEndRenderPass-None-06170
  The current render pass instance **must** not have been begun with `vkCmdBeginRendering`
- VUID-vkCmdEndRenderPass-None-07004
  If `vkCmdBeginQuery*` was called within a subpass of the render pass, the corresponding `vkCmdEndQuery*` **must** have been called subsequently within the same subpass

### Valid Usage (Implicit)

- VUID-vkCmdEndRenderPass-commandBuffer-parameter
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- VUID-vkCmdEndRenderPass-commandBuffer-recording
  `commandBuffer` **must** be in the recording state
- VUID-vkCmdEndRenderPass-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- VUID-vkCmdEndRenderPass-renderpass
  This command **must** only be called inside of a render pass instance
- VUID-vkCmdEndRenderPass-bufferlevel
  `commandBuffer` **must** be a primary `VkCommandBuffer`

### Host Synchronization

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

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Action</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>State</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Synchronization</td>
</tr>
</tbody>
</table>

To record a command to end a render pass instance after recording the commands for the last subpass, call:

```c
// Provided by VK_VERSION_1_2
void vkCmdEndRenderPass2(
    VkCommandBuffer commandBuffer,
    const VkSubpassEndInfo* pSubpassEndInfo);
```

- `commandBuffer` is the command buffer in which to end the current render pass instance.
- `pSubpassEndInfo` is a pointer to a `VkSubpassEndInfo` structure containing information about how the last subpass will be ended.

`vkCmdEndRenderPass2` is semantically identical to `vkCmdEndRenderPass`, except that it is extensible.

Valid Usage

- VUID-vkCmdEndRenderPass2-None-03103
  The current subpass index **must** be equal to the number of subpasses in the render pass minus one

- VUID-vkCmdEndRenderPass2-None-06171
  The current render pass instance **must** not have been begun with `vkCmdBeginRendering`

- VUID-vkCmdEndRenderPass2-None-07005
  If `vkCmdBeginQuery*` was called within a subpass of the render pass, the corresponding `vkCmdEndQuery*` **must** have been called subsequently within the same subpass

Valid Usage (Implicit)

- VUID-vkCmdEndRenderPass2-commandBuffer-parameter
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle

- VUID-vkCmdEndRenderPass2-pSubpassEndInfo-parameter
  `pSubpassEndInfo` **must** be a valid pointer to a valid `VkSubpassEndInfo` structure

- VUID-vkCmdEndRenderPass2-commandBuffer-recording
  `commandBuffer` **must** be in the recording state

- VUID-vkCmdEndRenderPass2-commandBuffer-cmdpool

359
The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

- VUID-vkCmdEndRenderPass2-renderpass
  This command must only be called inside of a render pass instance

- VUID-vkCmdEndRenderPass2-bufferlevel
  `commandBuffer` must be a primary `VkCommandBuffer`

### Host Synchronization

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Action State Synchronization</td>
</tr>
</tbody>
</table>

The `VkSubpassEndInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkSubpassEndInfo {
    VkStructureType sType;
    const void*   pNext;
} VkSubpassEndInfo;
```

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

### Valid Usage (Implicit)

- VUID-VkSubpassEndInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_SUBPASS_END_INFO`

- VUID-VkSubpassEndInfo-pNext-pNext
  `pNext` must be NULL
Chapter 9. Shaders

A shader specifies programmable operations that execute for each vertex, control point, tessellated vertex, primitive, fragment, or workgroup in the corresponding stage(s) of the graphics and compute pipelines.

Graphics pipelines include vertex shader execution as a result of primitive assembly, followed, if enabled, by tessellation control and evaluation shaders operating on patches, geometry shaders, if enabled, operating on primitives, and fragment shaders, if present, operating on fragments generated by Rasterization. In this specification, vertex, tessellation control, tessellation evaluation and geometry shaders are collectively referred to as pre-rasterization shader stages and occur in the logical pipeline before rasterization. The fragment shader occurs logically after rasterization.

Only the compute shader stage is included in a compute pipeline. Compute shaders operate on compute invocations in a workgroup.

Shaders can read from input variables, and read from and write to output variables. Input and output variables can be used to transfer data between shader stages, or to allow the shader to interact with values that exist in the execution environment. Similarly, the execution environment provides constants describing capabilities.

Shader variables are associated with execution environment-provided inputs and outputs using built-in decorations in the shader. The available decorations for each stage are documented in the following subsections.

9.1. Shader Modules

Shader modules contain shader code and one or more entry points. Shaders are selected from a shader module by specifying an entry point as part of pipeline creation. The stages of a pipeline can use shaders that come from different modules. The shader code defining a shader module must be in the SPIR-V format, as described by the Vulkan Environment for SPIR-V appendix.

Shader modules are represented by VkShaderModule handles:

```
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkShaderModule)
```

To create a shader module, call:

```
// Provided by VK_VERSION_1_0
VkResult vkCreateShaderModule(
    VkDevice device,
    const VkShaderModuleCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkShaderModule* pShaderModule);
```

- `device` is the logical device that creates the shader module.
• **pCreateInfo** is a pointer to a `VkShaderModuleCreateInfo` structure.
• **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
• **pShaderModule** is a pointer to a `VkShaderModule` handle in which the resulting shader module object is returned.

Once a shader module has been created, any entry points it contains can be used in pipeline shader stages as described in Compute Pipelines and Graphics Pipelines.

### Valid Usage

- **VUID-vkCreateShaderModule-pCreateInfo-06905**
  If `pCreateInfo` is not NULL, `pCreateInfo->pNext` must be NULL

### Valid Usage (Implicit)

- **VUID-vkCreateShaderModule-device-parameter**
  `device` must be a valid `VkDevice` handle
- **VUID-vkCreateShaderModule-pCreateInfo-parameter**
  `pCreateInfo` must be a valid pointer to a valid `VkShaderModuleCreateInfo` structure
- **VUID-vkCreateShaderModule-pAllocator-parameter**
  If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- **VUID-vkCreateShaderModule-pShaderModule-parameter**
  `pShaderModule` must be a valid pointer to a `VkShaderModule` handle

### Return Codes

**Success**
- **VK_SUCCESS**

**Failure**
- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**

The `VkShaderModuleCreateInfo` structure is defined as:

```
// Provided by VK_VERSION_1_0
typedef struct VkShaderModuleCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkShaderModuleCreateFlags flags;
    size_t codeSize;
    const uint32_t* pCode;
} VkShaderModuleCreateInfo;
```
VkShaderModuleCreateInfo;

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **codeSize** is the size, in bytes, of the code pointed to by **pCode**.
- **pCode** is a pointer to code that is used to create the shader module. The type and format of the code is determined from the content of the memory addressed by **pCode**.

### Valid Usage

- VUID-VkShaderModuleCreateInfo-codeSize-01085
  - **codeSize** must be greater than 0

- VUID-VkShaderModuleCreateInfo-codeSize-01086
  - **codeSize** must be a multiple of 4

- VUID-VkShaderModuleCreateInfo-pCode-01087
  - **pCode** must point to valid SPIR-V code, formatted and packed as described by the [Khronos SPIR-V Specification](#)

- VUID-VkShaderModuleCreateInfo-pCode-01088
  - **pCode** must adhere to the validation rules described by the Validation Rules within a Module section of the SPIR-V Environment appendix

- VUID-VkShaderModuleCreateInfo-pCode-01089
  - **pCode** must declare the Shader capability for SPIR-V code

- VUID-VkShaderModuleCreateInfo-pCode-01090
  - If **pCode** declares any of the capabilities listed in the SPIR-V Environment appendix, one of the corresponding requirements must be satisfied

- VUID-VkShaderModuleCreateInfo-pCode-04146
  - **pCode** must not declare any SPIR-V extension that is not supported by the API, as described by the Extension section of the SPIR-V Environment appendix

- VUID-VkShaderModuleCreateInfo-pCode-04147
  - If **pCode** declares any of the SPIR-V extensions listed in the SPIR-V Environment appendix, one of the corresponding requirements must be satisfied

### Valid Usage (Implicit)

- VUID-VkShaderModuleCreateInfo-sType-sType
  - **sType** must be VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO

- VUID-VkShaderModuleCreateInfo-flags-zerobitmask
  - **flags** must be zero
flags must be 0

- VUID-VkShaderModuleCreateInfo-pCode-parameter
  pCode must be a valid pointer to an array of \( \text{codeSize} / 4 \) uint32_t values

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkShaderModuleCreateFlags;
```

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

To destroy a shader module, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyShaderModule(
    VkDevice device,
    VkShaderModule shaderModule,
    const VkAllocationCallbacks* pAllocator);
```

- device is the logical device that destroys the shader module.
- shaderModule is the handle of the shader module to destroy.
- pAllocator controls host memory allocation as described in the Memory Allocation chapter.

A shader module can be destroyed while pipelines created using its shaders are still in use.

### Valid Usage

- VUID-vkDestroyShaderModule-shaderModule-01092
  If VkAllocationCallbacks were provided when shaderModule was created, a compatible set of callbacks must be provided here

- VUID-vkDestroyShaderModule-shaderModule-01093
  If no VkAllocationCallbacks were provided when shaderModule was created, pAllocator must be NULL

### Valid Usage (Implicit)

- VUID-vkDestroyShaderModule-device-parameter
  device must be a valid VkDevice handle

- VUID-vkDestroyShaderModule-shaderModule-parameter
  If shaderModule is not VK_NULL_HANDLE, shaderModule must be a valid VkShaderModule handle

- VUID-vkDestroyShaderModule-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid
9.2. Shader Execution

At each stage of the pipeline, multiple invocations of a shader may execute simultaneously. Further, invocations of a single shader produced as the result of different commands may execute simultaneously. The relative execution order of invocations of the same shader type is undefined. Shader invocations may complete in a different order than that in which the primitives they originated from were drawn or dispatched by the application. However, fragment shader outputs are written to attachments in rasterization order.

The relative execution order of invocations of different shader types is largely undefined. However, when invoking a shader whose inputs are generated from a previous pipeline stage, the shader invocations from the previous stage are guaranteed to have executed far enough to generate input values for all required inputs.

9.2.1. Shader Termination

A shader invocation that is terminated has finished executing instructions.

Executing OpReturn in the entry point, or executing OpTerminateInvocation in any function will terminate an invocation. Implementations may also terminate a shader invocation when OpKill is executed in any function; otherwise it becomes a helper invocation.

In addition to the above conditions, helper invocations are terminated when all non-helper invocations in the same derivative group either terminate or become helper invocations via OpKill.

A shader stage for a given command completes execution when all invocations for that stage have terminated.

9.3. Shader Memory Access Ordering

The order in which image or buffer memory is read or written by shaders is largely undefined. For some shader types (vertex, tessellation evaluation, and in some cases, fragment), even the number of shader invocations that may perform loads and stores is undefined.

In particular, the following rules apply:

- Vertex and tessellation evaluation shaders will be invoked at least once for each unique vertex,
as defined in those sections.

• **Fragment** shaders will be invoked zero or more times, as defined in that section.

• The relative execution order of invocations of the same shader type is undefined. A store issued by a shader when working on primitive B might complete prior to a store for primitive A, even if primitive A is specified prior to primitive B. This applies even to fragment shaders; while fragment shader outputs are always written to the framebuffer in rasterization order, stores executed by fragment shader invocations are not.

• The relative execution order of invocations of different shader types is largely undefined.

**Note**
The above limitations on shader invocation order make some forms of synchronization between shader invocations within a single set of primitives unimplementable. For example, having one invocation poll memory written by another invocation assumes that the other invocation has been launched and will complete its writes in finite time.

The Memory Model appendix defines the terminology and rules for how to correctly communicate between shader invocations, such as when a write is Visible-To a read, and what constitutes a Data Race.

Applications **must** not cause a data race.

The SPIR-V **SubgroupMemory**, **CrossWorkgroupMemory**, and **AtomicCounterMemory** memory semantics are ignored. Sequentially consistent atomics and barriers are not supported and **SequentlyConsistent** is treated as **AcquireRelease**. **SequentlyConsistent should** not be used.

### 9.4. Shader Inputs and Outputs

Data is passed into and out of shaders using variables with input or output storage class, respectively. User-defined inputs and outputs are connected between stages by matching their **Location** decorations. Additionally, data can be provided by or communicated to special functions provided by the execution environment using **BuiltIn** decorations.

In many cases, the same **BuiltIn** decoration can be used in multiple shader stages with similar meaning. The specific behavior of variables decorated as **BuiltIn** is documented in the following sections.

### 9.5. Vertex Shaders

Each vertex shader invocation operates on one vertex and its associated **vertex attribute** data, and outputs one vertex and associated data. Graphics pipelines **must** include a vertex shader, and the vertex shader stage is always the first shader stage in the graphics pipeline.
9.5.1. Vertex Shader Execution

A vertex shader must be executed at least once for each vertex specified by a drawing command. If the subpass includes multiple views in its view mask, the shader may be invoked separately for each view. During execution, the shader is presented with the index of the vertex and instance for which it has been invoked. Input variables declared in the vertex shader are filled by the implementation with the values of vertex attributes associated with the invocation being executed.

If the same vertex is specified multiple times in a drawing command (e.g. by including the same index value multiple times in an index buffer) the implementation may reuse the results of vertex shading if it can statically determine that the vertex shader invocations will produce identical results.

Note

It is implementation-dependent when and if results of vertex shading are reused, and thus how many times the vertex shader will be executed. This is true also if the vertex shader contains stores or atomic operations (see vertexPipelineStoresAndAtomics).

9.6. Tessellation Control Shaders

The tessellation control shader is used to read an input patch provided by the application and to produce an output patch. Each tessellation control shader invocation operates on an input patch (after all control points in the patch are processed by a vertex shader) and its associated data, and outputs a single control point of the output patch and its associated data, and can also output additional per-patch data. The input patch is sized according to the patchControlPoints member of VkPipelineTessellationStateCreateInfo, as part of input assembly.

The size of the output patch is controlled by the OpExecutionMode OutputVertices specified in the tessellation control or tessellation evaluation shaders, which must be specified in at least one of the shaders. The size of the input and output patches must each be greater than zero and less than or equal to VkPhysicalDeviceLimits::maxTessellationPatchSize.

9.6.1. Tessellation Control Shader Execution

A tessellation control shader is invoked at least once for each output vertex in a patch. If the subpass includes multiple views in its view mask, the shader may be invoked separately for each view.

Inputs to the tessellation control shader are generated by the vertex shader. Each invocation of the tessellation control shader can read the attributes of any incoming vertices and their associated data. The invocations corresponding to a given patch execute logically in parallel, with undefined relative execution order. However, the OpControlBarrier instruction can be used to provide limited control of the execution order by synchronizing invocations within a patch, effectively dividing tessellation control shader execution into a set of phases. Tessellation control shaders will read undefined values if one invocation reads a per-vertex or per-patch output written by another invocation at any point during the same phase, or if two invocations attempt to write different values to the same per-patch output in a single phase.
9.7. Tessellation Evaluation Shaders

The Tessellation Evaluation Shader operates on an input patch of control points and their associated data, and a single input barycentric coordinate indicating the invocation's relative position within the subdivided patch, and outputs a single vertex and its associated data.

9.7.1. Tessellation Evaluation Shader Execution

A tessellation evaluation shader is invoked at least once for each unique vertex generated by the tessellator. If the subpass includes multiple views in its view mask, the shader may be invoked separately for each view.

9.8. Geometry Shaders

The geometry shader operates on a group of vertices and their associated data assembled from a single input primitive, and emits zero or more output primitives and the group of vertices and their associated data required for each output primitive.

9.8.1. Geometry Shader Execution

A geometry shader is invoked at least once for each primitive produced by the tessellation stages, or at least once for each primitive generated by primitive assembly when tessellation is not in use. A shader can request that the geometry shader runs multiple instances. A geometry shader is invoked at least once for each instance. If the subpass includes multiple views in its view mask, the shader may be invoked separately for each view.

9.9. Fragment Shaders

Fragment shaders are invoked as a fragment operation in a graphics pipeline. Each fragment shader invocation operates on a single fragment and its associated data. With few exceptions, fragment shaders do not have access to any data associated with other fragments and are considered to execute in isolation of fragment shader invocations associated with other fragments.

9.10. Compute Shaders

Compute shaders are invoked via vkCmdDispatch and vkCmdDispatchIndirect commands. In general, they have access to similar resources as shader stages executing as part of a graphics pipeline.

Compute workloads are formed from groups of work items called workgroups and processed by the compute shader in the current compute pipeline. A workgroup is a collection of shader invocations that execute the same shader, potentially in parallel. Compute shaders execute in global workgroups which are divided into a number of local workgroups with a size that can be set by assigning a value to the LocalSize or LocalSizeId execution mode or via an object decorated by the WorkgroupSize decoration. An invocation within a local workgroup can share data with other members of the local workgroup through shared variables and issue memory and control flow barriers to synchronize with other members of the local workgroup.
9.11. Interpolation decorations

Variables in the Input storage class in a fragment shader’s interface are interpolated from the values specified by the primitive being rasterized.

Note
Interpolation decorations can be present on input and output variables in pre-rasterization shaders but have no effect on the interpolation performed.

An undecorated input variable will be interpolated with perspective-correct interpolation according to the primitive type being rasterized. Lines and polygons are interpolated in the same way as the primitive’s clip coordinates. If the NoPerspective decoration is present, linear interpolation is instead used for lines and polygons. For points, as there is only a single vertex, input values are never interpolated and instead take the value written for the single vertex.

If the Flat decoration is present on an input variable, the value is not interpolated, and instead takes its value directly from the provoking vertex. Fragment shader inputs that are signed or unsigned integers, integer vectors, or any double-precision floating-point type must be decorated with Flat.

Interpolation of input variables is performed at an implementation-defined position within the fragment area being shaded. The position is further constrained as follows:

- If the Centroid decoration is used, the interpolation position used for the variable must also fall within the bounds of the primitive being rasterized.
- If the Sample decoration is used, the interpolation position used for the variable must be at the position of the sample being shaded by the current fragment shader invocation.
- If a sample count of 1 is used, the interpolation position must be at the center of the fragment area.

Note
As Centroid restricts the possible interpolation position to the covered area of the primitive, the position can be forced to vary between neighboring fragments when it otherwise would not. Derivatives calculated based on these differing locations can produce inconsistent results compared to undecorated inputs. It is recommended that input variables used in derivative calculations are not decorated with Centroid.

9.12. Static Use

A SPIR-V module declares a global object in memory using the OpVariable instruction, which results in a pointer \( x \) to that object. A specific entry point in a SPIR-V module is said to statically use that object if that entry point’s call tree contains a function containing a instruction with \( x \) as an id operand.

Static use is not used to control the behavior of variables with Input and Output storage. The effects of those variables are applied based only on whether they are present in a shader entry point’s
interface.

9.13. Scope

A scope describes a set of shader invocations, where each such set is a scope instance. Each invocation belongs to one or more scope instances, but belongs to no more than one scope instance for each scope.

The operations available between invocations in a given scope instance vary, with smaller scopes generally able to perform more operations, and with greater efficiency.


All invocations executed in a Vulkan instance fall into a single cross device scope instance.

Whilst the CrossDevice scope is defined in SPIR-V, it is disallowed in Vulkan. API synchronization commands can be used to communicate between devices.

9.13.2. Device

All invocations executed on a single device form a device scope instance.

If the vulkanMemoryModel and vulkanMemoryModelDeviceScope features are enabled, this scope is represented in SPIR-V by the Device Scope, which can be used as a Memory Scope for barrier and atomic operations.

There is no method to synchronize the execution of these invocations within SPIR-V, and this can only be done with API synchronization primitives.

Invocations executing on different devices in a device group operate in separate device scope instances.

9.13.3. Queue Family

Invocations executed by queues in a given queue family form a queue family scope instance.

This scope is identified in SPIR-V as the QueueFamily Scope if the vulkanMemoryModel feature is enabled, or if not, the Device Scope, which can be used as a Memory Scope for barrier and atomic operations.

There is no method to synchronize the execution of these invocations within SPIR-V, and this can only be done with API synchronization primitives.

Each invocation in a queue family scope instance must be in the same device scope instance.

9.13.4. Command

Any shader invocations executed as the result of a single command such as vkCmdDispatch or vkCmdDraw form a command scope instance. For indirect drawing commands with drawCount greater than one, invocations from separate draws are in separate command scope instances.
There is no specific Scope for communication across invocations in a command scope instance. As this has a clear boundary at the API level, coordination here can be performed in the API, rather than in SPIR-V.

Each invocation in a command scope instance must be in the same queue-family scope instance.

For shaders without defined workgroups, this set of invocations forms an invocation group as defined in the SPIR-V specification.

9.13.5. Primitive

Any fragment shader invocations executed as the result of rasterization of a single primitive form a primitive scope instance.

There is no specific Scope for communication across invocations in a primitive scope instance.

Any generated helper invocations are included in this scope instance.

Each invocation in a primitive scope instance must be in the same command scope instance.

Any input variables decorated with Flat are uniform within a primitive scope instance.

9.13.6. Workgroup

A local workgroup is a set of invocations that can synchronize and share data with each other using memory in the Workgroup storage class.

The Workgroup Scope can be used as both an Execution Scope and Memory Scope for barrier and atomic operations.

Each invocation in a local workgroup must be in the same command scope instance.

Only compute shaders have defined workgroups - other shader types cannot use workgroup functionality. For shaders that have defined workgroups, this set of invocations forms an invocation group as defined in the SPIR-V specification.

9.13.7. Subgroup

A subgroup (see the subsection “Control Flow” of section 2 of the SPIR-V 1.3 Revision 1 specification) is a set of invocations that can synchronize and share data with each other efficiently.

The Subgroup Scope can be used as both an Execution Scope and Memory Scope for barrier and atomic operations. Other subgroup features allow the use of group operations with subgroup scope.

For shaders that have defined workgroups, each invocation in a subgroup must be in the same local workgroup.

In other shader stages, each invocation in a subgroup must be in the same device scope instance.

Only shader stages that support subgroup operations have defined subgroups.
9.13.8. Quad

A quad scope instance is formed of four shader invocations.

In a fragment shader, each invocation in a quad scope instance is formed of invocations in neighboring framebuffer locations \((x_i, y_i)\), where:

- \(i\) is the index of the invocation within the scope instance.
- \(w\) and \(h\) are the number of pixels the fragment covers in the \(x\) and \(y\) axes.
- \(w\) and \(h\) are identical for all participating invocations.
- \((x_0) = (x_1 - w) = (x_2) = (x_3 - w)\)
- \((y_0) = (y_1) = (y_2 - h) = (y_3 - h)\)
- Each invocation has the same layer and sample indices.

In all shaders, each invocation in a quad scope instance is formed of invocations in adjacent subgroup invocation indices \((s_i)\), where:

- \(i\) is the index of the invocation within the quad scope instance.
- \((s_0) = (s_1 - 1) = (s_2 - 2) = (s_3 - 3)\)
- \(s_0\) is an integer multiple of \(4\).

Each invocation in a quad scope instance must be in the same subgroup.

In a fragment shader, each invocation in a quad scope instance must be in the same primitive scope instance.

Fragment and compute shaders have defined quad scope instances. If the quadOperationsInAllStages limit is supported, any shader stages that support subgroup operations also have defined quad scope instances.

9.13.9. Invocation

The smallest scope is a single invocation; this is represented by the Invocation Scope in SPIR-V.

Fragment shader invocations must be in a primitive scope instance.

Invocations in shaders that have defined workgroups must be in a local workgroup.

Invocations in shaders that have a defined subgroup scope must be in a subgroup.

Invocations in shaders that have a defined quad scope must be in a quad scope instance.

All invocations in all stages must be in a command scope instance.


Group operations are executed by multiple invocations within a scope instance; with each invocation involved in calculating the result. This provides a mechanism for efficient

Basic group operations include the use of \texttt{OpGroupNonUniformElect}, \texttt{OpControlBarrier}, \texttt{OpMemoryBarrier}, and atomic operations.

\texttt{OpGroupNonUniformElect} can be used to choose a single invocation to perform a task for the whole group. Only the invocation with the lowest id in the group will return \texttt{true}.

The \texttt{Memory Model} appendix defines the operation of barriers and atomics.

9.14.2. Vote Group Operations

The vote group operations allow invocations within a group to compare values across a group. The types of votes enabled are:

- Do all active group invocations agree that an expression is true?
- Do any active group invocations evaluate an expression to true?
- Do all active group invocations have the same value of an expression?

\textit{Note}

These operations are useful in combination with control flow in that they allow for developers to check whether conditions match across the group and choose potentially faster code-paths in these cases.

9.14.3. Arithmetic Group Operations

The arithmetic group operations allow invocations to perform scans and reductions across a group. The operators supported are add, mul, min, max, and, or, xor.

For reductions, every invocation in a group will obtain the cumulative result of these operators applied to all values in the group. For exclusive scans, each invocation in a group will obtain the cumulative result of these operators applied to all values in invocations with a lower index in the group. Inclusive scans are identical to exclusive scans, except the cumulative result includes the operator applied to the value in the current invocation.

The order in which these operators are applied is implementation-dependent.


The ballot group operations allow invocations to perform more complex votes across the group. The ballot functionality allows all invocations within a group to provide a boolean value and get as a result what each invocation provided as their boolean value. The broadcast functionality allows
values to be broadcast from an invocation to all other invocations within the group.

9.14.5. Shuffle Group Operations

The shuffle group operations allow invocations to read values from other invocations within a group.


The shuffle relative group operations allow invocations to read values from other invocations within the group relative to the current invocation in the group. The relative operations supported allow data to be shifted up and down through the invocations within a group.


The clustered group operations allow invocations to perform an operation among partitions of a group, such that the operation is only performed within the group invocations within a partition. The partitions for clustered group operations are consecutive power-of-two size groups of invocations and the cluster size must be known at pipeline creation time. The operations supported are add, mul, min, max, and, or, xor.

9.15. Quad Group Operations

Quad group operations (OpGroupNonUniformQuad*) are a specialized type of group operations that only operate on quad scope instances. Whilst these instructions do include a Scope parameter, this scope is always overridden; only the quad scope instance is included in its execution scope.

Fragment shaders that statically execute quad group operations must launch sufficient invocations to ensure their correct operation; additional helper invocations are launched for framebuffer locations not covered by rasterized fragments if necessary.

The index used to select participating invocations is i, as described for a quad scope instance, defined as the quad index in the SPIR-V specification.

For OpGroupNonUniformQuadBroadcast this value is equal to Index. For OpGroupNonUniformQuadSwap, it is equal to the implicit Index used by each participating invocation.

9.16. Derivative Operations

Derivative operations calculate the partial derivative for an expression P as a function of an invocation's x and y coordinates.

Derivative operations operate on a set of invocations known as a derivative group as defined in the SPIR-V specification. A derivative group is equivalent to the primitive scope instance for a fragment shader invocation.

Derivatives are calculated assuming that P is piecewise linear and continuous within the derivative group. All dynamic instances of explicit derivative instructions (OpDPdx*, OpDPdy*, and OpFwidth*) must be executed in control flow that is uniform within a derivative group. For other derivative
operations, results are undefined if a dynamic instance is executed in control flow that is not
uniform within the derivative group.

Fragment shaders that statically execute derivative operations must launch sufficient invocations
to ensure their correct operation; additional helper invocations are launched for framebuffer
locations not covered by rasterized fragments if necessary.

Derivative operations calculate their results as the difference between the result of P across
invocations in the quad. For fine derivative operations (OpDPdxFine and OpDPdyFine), the values of
DPdx(P) are calculated as

$$\text{DPdx}(P_0) = \text{DPdx}(P_1) = P_1 - P_0$$

$$\text{DPdx}(P_2) = \text{DPdx}(P_3) = P_3 - P_2$$

and the values of DPdy(P) are calculated as

$$\text{DPdy}(P_0) = \text{DPdy}(P_2) = P_2 - P_0$$

$$\text{DPdy}(P_1) = \text{DPdy}(P_3) = P_3 - P_1$$

where i is the index of each invocation as described in Quad.

Coarse derivative operations (OpDPdxCoarse and OpDPdyCoarse), calculate their results in roughly the
same manner, but may only calculate two values instead of four (one for each of DPdx and DPdy),
reusing the same result no matter the originating invocation. If an implementation does this, it
should use the fine derivative calculations described for P_0.

Note

Derivative values are calculated between fragments rather than pixels. If the
fragment shader invocations involved in the calculation cover multiple pixels,
these operations cover a wider area, resulting in larger derivative values. This in
turn will result in a coarser level of detail being selected for image sampling
operations using derivatives.

Applications may want to account for this when using multi-pixel fragments; if
pixel derivatives are desired, applications should use explicit derivative operations
and divide the results by the size of the fragment in each dimension as follows:

$$\text{DPdx}(P_n)' = \text{DPdx}(P_n) / w$$

$$\text{DPdy}(P_n)' = \text{DPdy}(P_n) / h$$
where \( w \) and \( h \) are the size of the fragments in the quad, and \( \mathrm{DPdx}(P_n)' \) and \( \mathrm{DPdy}(P_n)' \) are the pixel derivatives.

The results for \( \text{OpDPdx} \) and \( \text{OpDPdy} \) may be calculated as either fine or coarse derivatives, with implementations favouring the most efficient approach. Implementations must choose coarse or fine consistently between the two.

Executing \( \text{OpFwidthFine} \), \( \text{OpFwidthCoarse} \), or \( \text{OpFwidth} \) is equivalent to executing the corresponding \( \text{OpDPdx}^* \) and \( \text{OpDPdy}^* \) instructions, taking the absolute value of the results, and summing them.

Executing an \( \text{OpImageSampleImplicitLod} \) instruction is equivalent to executing \( \text{OpDPdx}(\text{Coordinate}) \) and \( \text{OpDPdy}(\text{Coordinate}) \), and passing the results as the \( \text{Grad} \) operands \( dx \) and \( dy \).

**Note**
It is expected that using the \( \text{ImplicitLod} \) variants of sampling functions will be substantially more efficient than using the \( \text{ExplicitLod} \) variants with explicitly generated derivatives.

### 9.17. Helper Invocations

When performing derivative or quad group operations in a fragment shader, additional invocations may be spawned in order to ensure correct results. These additional invocations are known as helper invocations and can be identified by a non-zero value in the \( \text{HelperInvocation} \) built-in. Stores and atomics performed by helper invocations must not have any effect on memory except for the \( \text{Function} \), \( \text{Private} \) and \( \text{Output} \) storage classes, and values returned by atomic instructions in helper invocations are undefined.

**Note**
While storage to \( \text{Output} \) storage class has an effect even in helper invocations, it does not mean that helper invocations have an effect on the framebuffer. \( \text{Output} \) variables in fragment shaders can be read from as well, and they behave more like \( \text{Private} \) variables for the duration of the shader invocation.

For group operations other than derivative and quad group operations, helper invocations may be treated as inactive even if they would be considered otherwise active.

Helper invocations may become permanently inactive if all invocations in a quad scope instance become helper invocations.
Chapter 10. Pipelines

The following figure shows a block diagram of the Vulkan pipelines. Some Vulkan commands specify geometric objects to be drawn or computational work to be performed, while others specify state controlling how objects are handled by the various pipeline stages, or control data transfer between memory organized as images and buffers. Commands are effectively sent through a processing pipeline, either a graphics pipeline, or a compute pipeline.

The first stage of the graphics pipeline (Input Assembler) assembles vertices to form geometric primitives such as points, lines, and triangles, based on a requested primitive topology. In the next stage (Vertex Shader) vertices can be transformed, computing positions and attributes for each vertex. If tessellation and/or geometry shaders are supported, they can then generate multiple primitives from a single input primitive, possibly changing the primitive topology or generating additional attribute data in the process.

The final resulting primitives are clipped to a clip volume in preparation for the next stage, Rasterization. The rasterizer produces a series of fragments associated with a region of the framebuffer, from a two-dimensional description of a point, line segment, or triangle. These fragments are processed by fragment operations to determine whether generated values will be written to the framebuffer. Fragment shading determines the values to be written to the framebuffer attachments. Framebuffer operations then read and write the color and depth/stencil attachments of the framebuffer for a given subpass of a render pass instance. The attachments can be used as input attachments in the fragment shader in a later subpass of the same render pass.

The compute pipeline is a separate pipeline from the graphics pipeline, which operates on one-, two-, or three-dimensional workgroups which can read from and write to buffer and image memory.

This ordering is meant only as a tool for describing Vulkan, not as a strict rule of how Vulkan is implemented, and we present it only as a means to organize the various operations of the pipelines. Actual ordering guarantees between pipeline stages are explained in detail in the synchronization chapter.
Figure 2. Block diagram of the Vulkan pipeline

Each pipeline is controlled by a monolithic object created from a description of all of the shader stages and any relevant fixed-function stages. Linking the whole pipeline together allows the optimization of shaders based on their input/outputs and eliminates expensive draw time state validation.

A pipeline object is bound to the current state using `vkCmdBindPipeline`. Any pipeline object state that is specified as `dynamic` is not applied to the current state when the pipeline object is bound, but is instead set by dynamic state setting commands.

No state, including dynamic state, is inherited from one command buffer to another.

Compute, and graphics pipelines are each represented by `VkPipeline` handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPipeline)
```

### 10.1. Compute Pipelines

Compute pipelines consist of a single static compute shader stage and the pipeline layout.

The compute pipeline represents a compute shader and is created by calling `vkCreateComputePipelines` with `module` and `pName` selecting an entry point from a shader module, where that entry point defines a valid compute shader, in the `VkPipelineShaderStageCreateInfo` structure contained within the `VkComputePipelineCreateInfo` structure.

To create compute pipelines, call:

```c
// Provided by VK_VERSION_1_0
```
VkResult vkCreateComputePipelines(
    VkDevice device,
    VkPipelineCache pipelineCache,
    uint32_t createInfoCount,
    const VkComputePipelineCreateInfo* pCreateInfos,
    const VkAllocationCallbacks* pAllocator,
    VkPipeline* pPipelines);

- `device` is the logical device that creates the compute pipelines.
- `pipelineCache` is either `VK_NULL_HANDLE`, indicating that pipeline caching is disabled; or the handle of a valid pipeline cache object, in which case use of that cache is enabled for the duration of the command.
- `createInfoCount` is the length of the `pCreateInfos` and `pPipelines` arrays.
- `pCreateInfos` is a pointer to an array of `VkComputePipelineCreateInfo` structures.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pPipelines` is a pointer to an array of `VkPipeline` handles in which the resulting compute pipeline objects are returned.

**Valid Usage**

- VUID-vkCreateComputePipelines-flags-00695
  If the `flags` member of any element of `pCreateInfos` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and the `basePipelineIndex` member of that same element is not `-1`, `basePipelineIndex` must be less than the index into `pCreateInfos` that corresponds to that element

- VUID-vkCreateComputePipelines-flags-00696
  If the `flags` member of any element of `pCreateInfos` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, the base pipeline must have been created with the `VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT` flag set

- VUID-vkCreateComputePipelines-pipelineCache-02873
  If `pipelineCache` was created with `VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT`, host access to `pipelineCache` must be externally synchronized

**Valid Usage (Implicit)**

- VUID-vkCreateComputePipelines-device-parameter
  `device` must be a valid `VkDevice` handle

- VUID-vkCreateComputePipelines-pipelineCache-parameter
  If `pipelineCache` is not `VK_NULL_HANDLE`, `pipelineCache` must be a valid `VkPipelineCache` handle

- VUID-vkCreateComputePipelines-pCreateInfos-parameter
  `pCreateInfos` must be a valid pointer to an array of `createInfoCount` valid `VkComputePipelineCreateInfo` structures
• VUID-vkCreateComputePipelines-pAllocator-parameter
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

• VUID-vkCreateComputePipelines-pPipelines-parameter
  `pPipelines` must be a valid pointer to an array of `createInfoCount` `VkPipeline` handles

• VUID-vkCreateComputePipelines-createInfoCount-arraylength
  `createInfoCount` must be greater than 0

• VUID-vkCreateComputePipelines-pipelineCache-parent
  If `pipelineCache` is a valid handle, it must have been created, allocated, or retrieved from `device`

---

**Return Codes**

**Success**
- **VK_SUCCESS**

**Failure**
- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**

The `VkComputePipelineCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkComputePipelineCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineCreateFlags flags;
    VkPipelineShaderStageCreateInfo stage;
    VkPipelineLayout layout;
    VkPipeline basePipelineHandle;
    int32_t basePipelineIndex;
} VkComputePipelineCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is a bitmask of `VkPipelineCreateFlagBits` specifying how the pipeline will be generated.
- `stage` is a `VkPipelineShaderStageCreateInfo` structure describing the compute shader.
- `layout` is the description of binding locations used by both the pipeline and descriptor sets used with the pipeline.
- `basePipelineHandle` is a pipeline to derive from
- `basePipelineIndex` is an index into the `pCreateInfos` parameter to use as a pipeline to derive from
The parameters `basePipelineHandle` and `basePipelineIndex` are described in more detail in *Pipeline Derivatives*.

### Valid Usage

- **VUID-VkComputePipelineCreateInfo-flags-00697**
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is -1, `basePipelineHandle` must be a valid handle to a compute `VkPipeline`.

- **VUID-VkComputePipelineCreateInfo-flags-00698**
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineHandle` is `VK_NULL_HANDLE`, `basePipelineIndex` must be a valid index into the calling command’s `pCreateInfo` parameter.

- **VUID-VkComputePipelineCreateInfo-flags-00699**
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is not -1, `basePipelineHandle` must be `VK_NULL_HANDLE`.

- **VUID-VkComputePipelineCreateInfo-flags-00700**
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineHandle` is not `VK_NULL_HANDLE`, `basePipelineIndex` must be -1.

- **VUID-VkComputePipelineCreateInfo-stage-00701**
  The `stage` member of `stage` must be `VK_SHADER_STAGE_COMPUTE_BIT`.

- **VUID-VkComputePipelineCreateInfo-stage-00702**
  The shader code for the entry point identified by `stage` and the rest of the state identified by this structure must adhere to the pipeline linking rules described in the *Shader Interfaces* chapter.

- **VUID-VkComputePipelineCreateInfo-layout-00703**
  `layout` must be consistent with the layout of the compute shader specified in `stage`.

- **VUID-VkComputePipelineCreateInfo-layout-01687**
  The number of resources in `layout` accessible to the compute shader stage must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageResources`.

- **VUID-VkComputePipelineCreateInfo-pipelineCreationCacheControl-02875**
  If the `pipelineCreationCacheControl` feature is not enabled, `flags` must not include `VK_PIPELINE_CREATE_FAIL_ON_PIPELINE_COMPILE_REQUIRED_BIT` or `VK_PIPELINE_CREATE_EARLY_RETURN_ON_FAILURE_BIT`.

- **VUID-VkComputePipelineCreateInfo-pipelineStageCreationFeedbackCount-06566**
  If `VkPipelineCreationFeedbackCreateInfo::pipelineStageCreationFeedbackCount` is not 0, it must be 1.

### Valid Usage (Implicit)

- **VUID-VkComputePipelineCreateInfo-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO`.

- **VUID-VkComputePipelineCreateInfo-pNext-pNext**
pNext must be NULL or a pointer to a valid instance of VkPipelineCreationFeedbackCreateInfo

- VUID-VkComputePipelineCreateInfo-sType-unique
  The sType value of each struct in the pNext chain must be unique

- VUID-VkComputePipelineCreateInfo-flags-parameter
  flags must be a valid combination of VkPipelineCreateFlagBits values

- VUID-VkComputePipelineCreateInfo-stage-parameter
  stage must be a valid VkPipelineShaderStageCreateInfo structure

- VUID-VkComputePipelineCreateInfo-layout-parameter
  layout must be a valid VkPipelineLayout handle

- VUID-VkComputePipelineCreateInfo-commonparent
  Both of basePipelineHandle, and layout that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same VkDevice

The VkPipelineShaderStageCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineShaderStageCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineShaderStageCreateFlags flags;
    VkShaderStageFlagBits stage;
    VkShaderModule module;
    const char* pName;
    const VkSpecializationInfo* pSpecializationInfo;
} VkPipelineShaderStageCreateInfo;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- flags is a bitmask of VkPipelineShaderStageCreateFlagBits specifying how the pipeline shader stage will be generated.
- stage is a VkShaderStageFlagBits value specifying a single pipeline stage.
- module is a VkShaderModule object containing the shader code for this stage.
- pName is a pointer to a null-terminated UTF-8 string specifying the entry point name of the shader for this stage.
- pSpecializationInfo is a pointer to a VkSpecializationInfo structure, as described in Specialization Constants, or NULL.

The shader code used by the pipeline is defined by module.

Valid Usage

- VUID-VkPipelineShaderStageCreateInfo-stage-00704

382
If the `geometryShader` feature is not enabled, `stage` must not be `VK_SHADER_STAGE_GEOMETRY_BIT`.

- **VUID-VkPipelineShaderStageCreateInfo-stage-00705**
  If the `tessellationShader` feature is not enabled, `stage` must not be `VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT` or `VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT`.

- **VUID-VkPipelineShaderStageCreateInfo-stage-00706**
  `stage` must not be `VK_SHADER_STAGE_ALL_GRAPHICS`, or `VK_SHADER_STAGE_ALL`.

- **VUID-VkPipelineShaderStageCreateInfo-pName-00707**
  `pName` must be the name of an `OpEntryPoint` in `module` with an execution model that matches `stage`.

- **VUID-VkPipelineShaderStageCreateInfo-maxClipDistances-00708**
  If the identified entry point includes any variable in its interface that is declared with the `ClipDistance BuiltIn` decoration, that variable must not have an array size greater than `VkPhysicalDeviceLimits::maxClipDistances`.

- **VUID-VkPipelineShaderStageCreateInfo-maxCullDistances-00709**
  If the identified entry point includes any variable in its interface that is declared with the `CullDistance BuiltIn` decoration, that variable must not have an array size greater than `VkPhysicalDeviceLimits::maxCullDistances`.

- **VUID-VkPipelineShaderStageCreateInfo-maxCombinedClipAndCullDistances-00710**
  If the identified entry point includes any variables in its interface that are declared with the `ClipDistance` or `CullDistance BuiltIn` decoration, those variables must not have array sizes which sum to more than `VkPhysicalDeviceLimits::maxCombinedClipAndCullDistances`.

- **VUID-VkPipelineShaderStageCreateInfo-maxSampleMaskWords-00711**
  If the identified entry point includes any variable in its interface that is declared with the `SampleMask BuiltIn` decoration, that variable must not have an array size greater than `VkPhysicalDeviceLimits::maxSampleMaskWords`.

- **VUID-VkPipelineShaderStageCreateInfo-stage-00712**
  If `stage` is `VK_SHADER_STAGE_VERTEX_BIT`, the identified entry point must not include any input variable in its interface that is decorated with `CullDistance`.

- **VUID-VkPipelineShaderStageCreateInfo-stage-00713**
  If `stage` is `VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT` or `VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT`, and the identified entry point has an `OpExecutionMode` instruction specifying a patch size with `OutputVertices`, the patch size must be greater than 0 and less than or equal to `VkPhysicalDeviceLimits::maxTessellationPatchSize`.

- **VUID-VkPipelineShaderStageCreateInfo-stage-00714**
  If `stage` is `VK_SHADER_STAGE_GEOMETRY_BIT`, the identified entry point must have an `OpExecutionMode` instruction specifying a maximum output vertex count that is greater than 0 and less than or equal to `VkPhysicalDeviceLimits::maxGeometryOutputVertices`.

- **VUID-VkPipelineShaderStageCreateInfo-stage-00715**
  If `stage` is `VK_SHADER_STAGE_GEOMETRY_BIT`, the identified entry point must have an `OpExecutionMode` instruction specifying an invocation count that is greater than 0 and less
than or equal to `VkPhysicalDeviceLimits::maxGeometryShaderInvocations`

- **VUID-VkPipelineShaderStageCreateInfo-stage-02596**
  If `stage` is either `VK_SHADER_STAGE_VERTEX_BIT`, `VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT`, `VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT`, or `VK_SHADER_STAGE_GEOMETRY_BIT`, and the identified entry point writes to `Layer` for any primitive, it **must** write the same value to `Layer` for all vertices of a given primitive.

- **VUID-VkPipelineShaderStageCreateInfo-stage-02597**
  If `stage` is either `VK_SHADER_STAGE_VERTEX_BIT`, `VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT`, `VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT`, or `VK_SHADER_STAGE_GEOMETRY_BIT`, and the identified entry point writes to `ViewportIndex` for any primitive, it **must** write the same value to `ViewportIndex` for all vertices of a given primitive.

- **VUID-VkPipelineShaderStageCreateInfo-stage-00718**
  If `stage` is `VK_SHADER_STAGE_FRAGMENT_BIT`, the identified entry point **must** not include any output variables in its interface decorated with `CullDistance`.

- **VUID-VkPipelineShaderStageCreateInfo-stage-06685**
  If `stage` is `VK_SHADER_STAGE_FRAGMENT_BIT`, and the identified entry point writes to `FragDepth` in any execution path, all execution paths that are not exclusive to helper invocations **must** either discard the fragment, or write or initialize the value of `FragDepth`.

- **VUID-VkPipelineShaderStageCreateInfo-flags-02784**
  If `flags` has the `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT` flag set, the `subgroupSizeControl` feature **must** be enabled.

- **VUID-VkPipelineShaderStageCreateInfo-flags-02785**
  If `flags` has the `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT` flag set, the `computeFullSubgroups` feature **must** be enabled.

- **VUID-VkPipelineShaderStageCreateInfo-pNext-02754**
  If a `VkPipelineShaderStageRequiredSubgroupSizeCreateInfo` structure is included in the `pNext` chain, `flags` **must** not have the `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT` flag set.

- **VUID-VkPipelineShaderStageCreateInfo-pNext-02755**
  If a `VkPipelineShaderStageRequiredSubgroupSizeCreateInfo` structure is included in the `pNext` chain, the `subgroupSizeControl` feature **must** be enabled, and `stage` **must** be a valid bit specified in `requiredSubgroupSizeStages`.

- **VUID-VkPipelineShaderStageCreateInfo-pNext-02756**
  If a `VkPipelineShaderStageRequiredSubgroupSizeCreateInfo` structure is included in the `pNext` chain and `stage` is `VK_SHADER_STAGE_COMPUTE_BIT`, the local workgroup size of the shader **must** be less than or equal to the product of `VkPipelineShaderStageRequiredSubgroupSizeCreateInfo::requiredSubgroupSize` and `maxComputeWorkgroupSubgroups`.

- **VUID-VkPipelineShaderStageCreateInfo-pNext-02757**
  If a `VkPipelineShaderStageRequiredSubgroupSizeCreateInfo` structure is included in the `pNext` chain, and `flags` has the `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT` flag set, the local workgroup size in the X dimension of the pipeline **must** be a multiple of `VkPipelineShaderStageRequiredSubgroupSizeCreateInfo::requiredSubgroupSize`. 


If \texttt{flags} has both the \texttt{VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT} and \texttt{VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT} flags set, the local workgroup size in the X dimension of the pipeline must be a multiple of \texttt{maxSubgroupSize}.

If \texttt{flags} has the \texttt{VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT} flag set and \texttt{flags} does not have the \texttt{VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT} flag set and no \texttt{VkPipelineShaderStageRequiredSubgroupSizeCreateInfo} structure is included in the \texttt{pNext} chain, the local workgroup size in the X dimension of the pipeline must be a multiple of \texttt{subgroupSize}.

\texttt{module} must be a valid \texttt{VkShaderModule}.

The shader code used by the pipeline must be valid as described by the Khronos SPIR-V Specification after applying the specializations provided in \texttt{pSpecializationInfo}, if any, and then converting all specialization constants into fixed constants.

---

### Valid Usage (Implicit)

- **VUID-VkPipelineShaderStageCreateInfo-sType-sType**
  \texttt{sType} must be \texttt{VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO}

- **VUID-VkPipelineShaderStageCreateInfo-pNext-pNext**
  Each \texttt{pNext} member of any structure (including this one) in the \texttt{pNext} chain must be either \texttt{NULL} or a pointer to a valid instance of \texttt{VkPipelineShaderStageRequiredSubgroupSizeCreateInfo} or \texttt{VkShaderModuleCreateInfo}

- **VUID-VkPipelineShaderStageCreateInfo-sType-unique**
  The \texttt{sType} value of each struct in the \texttt{pNext} chain must be unique

- **VUID-VkPipelineShaderStageCreateInfo-flags-parameter**
  \texttt{flags} must be a valid combination of \texttt{VkPipelineShaderStageCreateFlagBits} values

- **VUID-VkPipelineShaderStageCreateInfo-stage-parameter**
  \texttt{stage} must be a valid \texttt{VkShaderStageFlagBits} value

- **VUID-VkPipelineShaderStageCreateInfo-module-parameter**
  If \texttt{module} is not \texttt{VK_NULL_HANDLE}, \texttt{module} must be a valid \texttt{VkShaderModule} handle

- **VUID-VkPipelineShaderStageCreateInfo-pName-parameter**
  \texttt{pName} must be a null-terminated UTF-8 string

- **VUID-VkPipelineShaderStageCreateInfo-pSpecializationInfo-parameter**
  If \texttt{pSpecializationInfo} is not \texttt{NULL}, \texttt{pSpecializationInfo} must be a valid pointer to a valid \texttt{VkSpecializationInfo} structure

---

// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineShaderStageCreateFlags;
VkPipelineShaderStageCreateFlags is a bitmask type for setting a mask of zero or more VkPipelineShaderStageCreateFlagBits.

Possible values of the flags member of VkPipelineShaderStageCreateInfo specifying how a pipeline shader stage is created, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkPipelineShaderStageCreateFlagBits {
    // Provided by VK_VERSION_1_3
    VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT = 0x00000001,
    // Provided by VK_VERSION_1_3
    VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT = 0x00000002,
} VkPipelineShaderStageCreateFlagBits;
```

- **VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT** specifies that the SubgroupSize may vary in the shader stage.
- **VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT** specifies that the subgroup sizes must be launched with all invocations active in the compute stage.

**Note**

If **VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT** and **VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT** are specified and minSubgroupSize does not equal maxSubgroupSize and no required subgroup size is specified, then the only way to guarantee that the 'X' dimension of the local workgroup size is a multiple of SubgroupSize is to make it a multiple of maxSubgroupSize. Under these conditions, you are guaranteed full subgroups but not any particular subgroup size.

Bits which can be set by commands and structures, specifying one or more shader stages, are:

```c
// Provided by VK_VERSION_1_0
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;
```

- **VK_SHADER_STAGE_VERTEX_BIT** specifies the vertex stage.
- **VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT** specifies the tessellation control stage.
- **VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT** specifies the tessellation evaluation stage.
• **VK_SHADER_STAGE_GEOMETRY_BIT** specifies the geometry stage.

• **VK_SHADER_STAGE_FRAGMENT_BIT** specifies the fragment stage.

• **VK_SHADER_STAGE_COMPUTE_BIT** specifies the compute stage.

• **VK_SHADER_STAGE_ALL_GRAPHICS** is a combination of bits used as shorthand to specify all graphics stages defined above (excluding the compute stage).

• **VK_SHADER_STAGE_ALL** is a combination of bits used as shorthand to specify all shader stages supported by the device, including all additional stages which are introduced by extensions.

  **Note**

  **VK_SHADER_STAGE_ALL_GRAPHICS** only includes the original five graphics stages included in Vulkan 1.0, and not any stages added by extensions. Thus, it may not have the desired effect in all cases.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkShaderStageFlags;
```

**VkShaderStageFlags** is a bitmask type for setting a mask of zero or more **VkShaderStageFlagBits**.

The **VkPipelineShaderStageRequiredSubgroupSizeCreateInfo** structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPipelineShaderStageRequiredSubgroupSizeCreateInfo {
    VkStructureType sType;
    void* pNext;
    uint32_t requiredSubgroupSize;
} VkPipelineShaderStageRequiredSubgroupSizeCreateInfo;
```

• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **requiredSubgroupSize** is an unsigned integer value specifying the required subgroup size for the newly created pipeline shader stage.

If a **VkPipelineShaderStageRequiredSubgroupSizeCreateInfo** structure is included in the **pNext** chain of **VkPipelineShaderStageCreateInfo**, it specifies that the pipeline shader stage being compiled has a required subgroup size.

**Valid Usage**

• **VUID-VkPipelineShaderStageRequiredSubgroupSizeCreateInfo-requiredSubgroupSize-02760**

  **requiredSubgroupSize** **must** be a power-of-two integer

• **VUID-VkPipelineShaderStageRequiredSubgroupSizeCreateInfo-requiredSubgroupSize-02761**
requiredSubgroupSize must be greater or equal to minSubgroupSize

- VUID-VkPipelineShaderStageRequiredSubgroupSizeCreateInfo-requiredSubgroupSize-02762
requiredSubgroupSize must be less than or equal to maxSubgroupSize

Valid Usage (Implicit)

- VUID-VkPipelineShaderStageRequiredSubgroupSizeCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_REQUIRED_SUBGROUP_SIZE_CREATE_INFO

10.2. Graphics Pipelines

Graphics pipelines consist of multiple shader stages, multiple fixed-function pipeline stages, and a pipeline layout.

To create graphics pipelines, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateGraphicsPipelines(
    VkDevice device,                      // device,    
    VkPipelineCache pipelineCache,       // pipelineCache,   
    uint32_t createInfoCount,            // createInfoCount,    
    const VkGraphicsPipelineCreateInfo* pCreateInfos,   // pCreateInfos,    
    const VkAllocationCallbacks* pAllocator,     // pAllocator,    
    VkPipeline* pPipelines);                 // pPipelines
```

- **device** is the logical device that creates the graphics pipelines.
- **pipelineCache** is either VK_NULL_HANDLE, indicating that pipeline caching is disabled; or the handle of a valid pipeline cache object, in which case use of that cache is enabled for the duration of the command.
- **createInfoCount** is the length of the **pCreateInfos** and **pPipelines** arrays.
- **pCreateInfos** is a pointer to an array of VkGraphicsPipelineCreateInfo structures.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
- **pPipelines** is a pointer to an array of VkPipeline handles in which the resulting graphics pipeline objects are returned.

The VkGraphicsPipelineCreateInfo structure includes an array of VkPipelineShaderStageCreateInfo structures for each of the desired active shader stages, as well as creation information for all relevant fixed-function stages, and a pipeline layout.
Valid Usage

• VUID-vkCreateGraphicsPipelines-flags-00720
  If the flags member of any element of pCreateInfos contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, and the basePipelineIndex member of that same element is not -1, basePipelineIndex must be less than the index into pCreateInfos that corresponds to that element

• VUID-vkCreateGraphicsPipelines-flags-00721
  If the flags member of any element of pCreateInfos contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, the base pipeline must have been created with the VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT flag set

• VUID-vkCreateGraphicsPipelines-pipelineCache-02876
  If pipelineCache was created with VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT, host access to pipelineCache must be externally synchronized

Note
An implicit cache may be provided by the implementation or a layer. For this reason, it is still valid to set VK_PIPELINE_CREATE_FAIL_ON_PIPELINE_COMPILE_REQUIRED_BIT on flags for any element of pCreateInfos while passing VK_NULL_HANDLE for pipelineCache.

Valid Usage (Implicit)

• VUID-vkCreateGraphicsPipelines-device-parameter
device must be a valid VkDevice handle

• VUID-vkCreateGraphicsPipelines-pipelineCache-parameter
  If pipelineCache is not VK_NULL_HANDLE, pipelineCache must be a valid VkPipelineCache handle

• VUID-vkCreateGraphicsPipelines-pCreateInfos-parameter
  pCreateInfos must be a valid pointer to an array of createInfoCount valid VkGraphicsPipelineCreateInfo structures

• VUID-vkCreateGraphicsPipelines-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

• VUID-vkCreateGraphicsPipelines-pPipelines-parameter
  pPipelines must be a valid pointer to an array of createInfoCount VkPipeline handles

• VUID-vkCreateGraphicsPipelines-createInfoCount-arraylength
  createInfoCount must be greater than 0

• VUID-vkCreateGraphicsPipelines-pipelineCache-parent
  If pipelineCache is a valid handle, it must have been created, allocated, or retrieved from device
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkGraphicsPipelineCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkGraphicsPipelineCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineCreateFlags flags;
    uint32_t stageCount;
    const VkPipelineShaderStageCreateInfo* pStages;
    const VkPipelineVertexInputStateCreateInfo* pVertexInputState;
    const VkPipelineInputAssemblyStateCreateInfo* pInputAssemblyState;
    const VkPipelineTessellationStateCreateInfo* pTessellationState;
    const VkPipelineViewportStateCreateInfo* pViewportState;
    const VkPipelineRasterizationStateCreateInfo* pRasterizationState;
    const VkPipelineMultisampleStateCreateInfo* pMultisampleState;
    const VkPipelineDepthStencilStateCreateInfo* pDepthStencilState;
    const VkPipelineColorBlendStateCreateInfo* pColorBlendState;
    const VkPipelineDynamicStateCreateInfo* pDynamicState;
    VkPipelineLayout layout;
    VkRenderPass renderPass;
    uint32_t subpass;
    VkPipeline basePipelineHandle;
    int32_t basePipelineIndex;
} VkGraphicsPipelineCreateInfo;
```

• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **flags** is a bitmask of VkPipelineCreateFlagBits specifying how the pipeline will be generated.
• **stageCount** is the number of entries in the **pStages** array.
• **pStages** is a pointer to an array of **stageCount** VkPipelineShaderStageCreateInfo structures describing the set of the shader stages to be included in the graphics pipeline.
• **pVertexInputState** is a pointer to a VkPipelineVertexInputStateCreateInfo structure.
• **pInputAssemblyState** is a pointer to a VkPipelineInputAssemblyStateCreateInfo structure which determines input assembly behavior for vertex shading, as described in Drawing Commands.
• **pTessellationState** is a pointer to a VkPipelineTessellationStateCreateInfo structure defining
tessellation state used by tessellation shaders.

- **pViewportState** is a pointer to a `VkPipelineViewportStateCreateInfo` structure defining viewport state used when rasterization is enabled.

- **pRasterizationState** is a pointer to a `VkPipelineRasterizationStateCreateInfo` structure defining rasterization state.

- **pMultisampleState** is a pointer to a `VkPipelineMultisampleStateCreateInfo` structure defining multisample state used when rasterization is enabled.

- **pDepthStencilState** is a pointer to a `VkPipelineDepthStencilStateCreateInfo` structure defining depth/stencil state used when rasterization is enabled for depth or stencil attachments accessed during rendering.

- **pColorBlendState** is a pointer to a `VkPipelineColorBlendStateCreateInfo` structure defining color blend state used when rasterization is enabled for any color attachments accessed during rendering.

- **pDynamicState** is a pointer to a `VkPipelineDynamicStateCreateInfo` structure defining which properties of the pipeline state object are dynamic and can be changed independently of the pipeline state. This can be NULL, which means no state in the pipeline is considered dynamic.

- **layout** is the description of binding locations used by both the pipeline and descriptor sets used with the pipeline.

- **renderPass** is a handle to a render pass object describing the environment in which the pipeline will be used. The pipeline must only be used with a render pass instance compatible with the one provided. See Render Pass Compatibility for more information.

- **subpass** is the index of the subpass in the render pass where this pipeline will be used.

- **basePipelineHandle** is a pipeline to derive from.

- **basePipelineIndex** is an index into the `pCreateInfos` parameter to use as a pipeline to derive from.

The parameters `basePipelineHandle` and `basePipelineIndex` are described in more detail in Pipeline Derivatives.

The state required for a graphics pipeline is divided into vertex input state, pre-rasterization shader state, fragment shader state, and fragment output state.

Vertex input state is defined by:

- `VkPipelineVertexInputStateCreateInfo`
- `VkPipelineInputAssemblyStateCreateInfo`

Pre-rasterization shader state is defined by:

- `VkPipelineShaderStageCreateInfo` entries for:
  - Vertex shaders
  - Tessellation control shaders
  - Tessellation evaluation shaders
Geometry shaders
- Within the `VkPipelineLayout`, the full pipeline layout must be specified.
- `VkPipelineViewportStateCreateInfo`
- `VkPipelineRasterizationStateCreateInfo`
- `VkPipelineTessellationStateCreateInfo`
- `VkRenderPass` and `subpass` parameter
- The `viewMask` parameter of `VkPipelineRenderingCreateInfo` (formats are ignored)

Fragment shader state is defined by:
- A `VkPipelineShaderStageCreateInfo` entry for the fragment shader
- Within the `VkPipelineLayout`, the full pipeline layout must be specified.
- `VkPipelineMultisampleStateCreateInfo` if sample shading is enabled or `renderpass` is not `VK_NULL_HANDLE`
- `VkPipelineDepthStencilStateCreateInfo`
- `VkRenderPass` and `subpass` parameter
- The `viewMask` parameter of `VkPipelineRenderingCreateInfo` (formats are ignored)

Fragment output state is defined by:
- `VkPipelineColorBlendStateCreateInfo`
- `VkRenderPass` and `subpass` parameter
- `VkPipelineMultisampleStateCreateInfo`
- `VkPipelineRenderingCreateInfo`

Dynamic state values set via `pDynamicState` must be ignored if the state they correspond to is not otherwise statically set by one of the state subsets used to create the pipeline. For example, if a pipeline only included pre-rasterization shader state, then any dynamic state value corresponding to depth or stencil testing has no effect.

A complete graphics pipeline always includes pre-rasterization shader state, with other subsets included depending on that state. If the pre-rasterization shader state includes a vertex shader, then vertex input state is included in a complete graphics pipeline. If the value of `VkPipelineRasterizationStateCreateInfo::rasterizerDiscardEnable` in the pre-rasterization shader state is `VK_FALSE` or the `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE` dynamic state is enabled fragment shader state and fragment output interface state is included in a complete graphics pipeline.

Pipelines must be created with a complete set of pipeline state.

---

**Valid Usage**
- VUID-VkGraphicsPipelineCreateInfo-flags-00722
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is -1,
basePipelineHandle must be a valid handle to a graphics VkPipeline

- VUID-VkGraphicsPipelineCreateInfo-flags-00723
  If flags contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, and basePipelineHandle is VK_NULL_HANDLE, basePipelineIndex must be a valid index into the calling command’s pCreateInfos parameter

- VUID-VkGraphicsPipelineCreateInfo-flags-00724
  If flags contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, and basePipelineIndex is not -1, basePipelineHandle must be VK_NULL_HANDLE

- VUID-VkGraphicsPipelineCreateInfo-flags-00725
  If flags contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, and basePipelineHandle is not VK_NULL_HANDLE, basePipelineIndex must be -1

- VUID-VkGraphicsPipelineCreateInfo-stage-00727
  If the pipeline is being created with pre-rasterization shader state the stage member of one element of pStages must be VK_SHADER_STAGE_VERTEX_BIT

- VUID-VkGraphicsPipelineCreateInfo-pStages-00729
  If the pipeline is being created with pre-rasterization shader state and pStages includes a tessellation control shader stage, it must include a tessellation evaluation shader stage

- VUID-VkGraphicsPipelineCreateInfo-pStages-00730
  If the pipeline is being created with pre-rasterization shader state and pStages includes a tessellation evaluation shader stage, it must include a tessellation control shader stage

- VUID-VkGraphicsPipelineCreateInfo-pStages-00731
  If the pipeline is being created with pre-rasterization shader state and pStages includes a tessellation control shader stage and a tessellation evaluation shader stage, pTessellationState must be a valid pointer to a valid VkPipelineTessellationStateCreateInfo structure

- VUID-VkGraphicsPipelineCreateInfo-pStages-00732
  If the pipeline is being created with pre-rasterization shader state and pStages includes tessellation shader stages, the shader code of at least one stage must contain an OpExecutionMode instruction specifying the type of subdivision in the pipeline

- VUID-VkGraphicsPipelineCreateInfo-pStages-00733
  If the pipeline is being created with pre-rasterization shader state and pStages includes tessellation shader stages, and the shader code of both stages contain an OpExecutionMode instruction specifying the type of subdivision in the pipeline, they must both specify the same subdivision mode

- VUID-VkGraphicsPipelineCreateInfo-pStages-00734
  If the pipeline is being created with pre-rasterization shader state and pStages includes tessellation shader stages, the shader code of at least one stage must contain an OpExecutionMode instruction specifying the output patch size in the pipeline

- VUID-VkGraphicsPipelineCreateInfo-pStages-00735
  If the pipeline is being created with pre-rasterization shader state and pStages includes tessellation shader stages, and the shader code of both contain an OpExecutionMode instruction specifying the out patch size in the pipeline, they must both specify the same patch size
If the pipeline is being created with pre-rasterization shader state and pStages includes tessellation shader stages, the topology member of pInputAssembly must be VK_PRIMITIVE_TOPOLOGY_PATCH_LIST.

If the pipeline is being created with pre-rasterization shader state and the topology member of pInputAssembly is VK_PRIMITIVE_TOPOLOGY_PATCH_LIST, pStages must include tessellation shader stages.

If the pipeline is being created with a Vertex Execution Model and no TessellationEvaluation or Geometry Execution Model, and the topology member of pInputAssembly is VK_PRIMITIVE_TOPOLOGY_POINT_LIST, a PointSize decorated variable must be written to.

If the pipeline is being created with a TessellationEvaluation Execution Model, no Geometry Execution Model, uses the PointMode Execution Mode, and shaderTessellationAndGeometryPointSize is enabled, a PointSize decorated variable must be written to.

If the pipeline is being created with a TessellationEvaluation Execution Model, no Geometry Execution Model, uses the PointMode Execution Mode, and shaderTessellationAndGeometryPointSize is not enabled, a PointSize decorated variable must not be written to.

If the pipeline is being created with a Geometry Execution Model, uses the OutputPoints Execution Mode, and shaderTessellationAndGeometryPointSize is enabled, a PointSize decorated variable must be written to.

If the pipeline is being created with a Geometry Execution Model, uses the OutputPoints Execution Mode, and shaderTessellationAndGeometryPointSize is not enabled, a PointSize decorated variable must not be written to.

If the pipeline is being created with pre-rasterization shader state and pStages includes a geometry shader stage, and does not include any tessellation shader stages, its shader code must contain an OpExecutionMode instruction specifying an input primitive type that is compatible with the primitive topology specified in pInputAssembly.

If the pipeline is being created with pre-rasterization shader state and pStages includes a geometry shader stage, and also includes tessellation shader stages, its shader code must contain an OpExecutionMode instruction specifying an input primitive type that is compatible with the primitive topology that is output by the tessellation stages.

If the pipeline is being created with pre-rasterization shader state and fragment shader state, it includes both a fragment shader and a geometry shader, and the fragment shader
code reads from an input variable that is decorated with `PrimitiveId`, then the geometry shader code **must** write to a matching output variable, decorated with `PrimitiveId`, in all execution paths

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06038**
  If `renderPass` is not `VK_NULL_HANDLE` and the pipeline is being created with fragment shader state the fragment shader **must** not read from any input attachment that is defined as `VK_ATTACHMENT_UNUSED` in subpass

- **VUID-VkGraphicsPipelineCreateInfo-pStages-00742**
  If the pipeline is being created with pre-rasterization shader state and multiple pre-rasterization shader stages are included in `pStages`, the shader code for the entry points identified by those `pStages` and the rest of the state identified by this structure **must** adhere to the pipeline linking rules described in the Shader Interfaces chapter

- **VUID-VkGraphicsPipelineCreateInfo-None-04889**
  If the pipeline is being created with pre-rasterization shader state and fragment shader state, the fragment shader and last pre-rasterization shader stage and any relevant state **must** adhere to the pipeline linking rules described in the Shader Interfaces chapter

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06041**
  If `renderPass` is not `VK_NULL_HANDLE`, and the pipeline is being created with fragment output interface state, then for each color attachment in the subpass, if the potential format features of the format of the corresponding attachment description do not contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT`, then the `blendEnable` member of the corresponding element of the `pAttachments` member of `pColorBlendState` **must** be `VK_FALSE`

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06042**
  If `renderPass` is not `VK_NULL_HANDLE`, and the pipeline is being created with fragment output interface state, and the subpass uses color attachments, the `attachmentCount` member of `pColorBlendState` **must** be equal to the `colorAttachmentCount` used to create subpass

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-04130**
  If the pipeline is being created with pre-rasterization shader state, and no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_VIEWPORT` or `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT`, the `pViewports` member of `pViewportState` **must** be a valid pointer to an array of `pViewportState->viewportCount` valid `VkViewport` structures

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-04131**
  If the pipeline is being created with pre-rasterization shader state, and no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_SCISSOR` or `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT`, the `pScissors` member of `pViewportState` **must** be a valid pointer to an array of `pViewportState->scissorCount` `VkRect2D` structures

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-00749**
  If the pipeline is being created with pre-rasterization shader state, and the `wideLines` feature is not enabled, and no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_LINE_WIDTH`, the `lineWidth` member of `pRasterizationState` **must** be `1.0`

- **VUID-VkGraphicsPipelineCreateInfo-rasterizerDiscardEnable-00750**
  If the pipeline is being created with pre-rasterization shader state, and the `rasterizerDiscardEnable` member of `pRasterizationState` is `VK_FALSE`, `pViewportState` **must**
be a valid pointer to a valid \texttt{VkPipelineViewportStateCreateInfo} structure

- \textbf{VUID-VkGraphicsPipelineCreateInfo-pViewportState-04892}
  If the pipeline is being created with \textit{pre-rasterization shader state}, and the graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE} dynamic state enabled, \texttt{pViewportState} must be a valid pointer to a valid \texttt{VkPipelineViewportStateCreateInfo} structure

- \textbf{VUID-VkGraphicsPipelineCreateInfo-rasterizerDiscardEnable-00751}
  If the pipeline is being created with \textit{fragment output interface state}, \texttt{pMultisampleState} must be a valid pointer to a valid \texttt{VkPipelineMultisampleStateCreateInfo} structure

- \textbf{VUID-VkGraphicsPipelineCreateInfo-renderPass-06043}
  If \texttt{renderPass} is not \texttt{VK_NULL_HANDLE}, the pipeline is being created with \textit{fragment shader state}, and \texttt{subpass} uses a depth/stencil attachment, \texttt{pDepthStencilState} must be a valid pointer to a valid \texttt{VkPipelineDepthStencilStateCreateInfo} structure

- \textbf{VUID-VkGraphicsPipelineCreateInfo-renderPass-06044}
  If \texttt{renderPass} is not \texttt{VK_NULL_HANDLE}, the pipeline is being created with \textit{fragment output interface state}, and \texttt{subpass} uses color attachments, \texttt{pColorBlendState} must be a valid pointer to a valid \texttt{VkPipelineColorBlendStateCreateInfo} structure

- \textbf{VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-00754}
  If the pipeline is being created with \textit{pre-rasterization shader state}, the \texttt{depthBiasClamp} feature is not enabled, no element of the \texttt{pDynamicStates} member of \texttt{pDynamicState} is \texttt{VK_DYNAMIC_STATE_DEPTH_BIAS}, and the \texttt{depthBiasEnable} member of \texttt{pRasterizationState} is \texttt{VK_TRUE}, the \texttt{depthBiasClamp} member of \texttt{pRasterizationState} must be 0.0

- \textbf{VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-00755}
  If the pipeline is being created with \textit{fragment shader state}, and no element of the \texttt{pDynamicStates} member of \texttt{pDynamicState} is \texttt{VK_DYNAMIC_STATE_DEPTH_BOUNDS}, and the \texttt{depthBoundsTestEnable} member of \texttt{pDepthStencilState} is \texttt{VK_TRUE}, the \texttt{minDepthBounds} and \texttt{maxDepthBounds} members of \texttt{pDepthStencilState} must be between 0.0 and 1.0, inclusive

- \textbf{VUID-VkGraphicsPipelineCreateInfo-layout-00756}
  \texttt{layout} must be consistent with all shaders specified in \texttt{pStages}

- \textbf{VUID-VkGraphicsPipelineCreateInfo-subpass-00758}
  If the pipeline is being created with \textit{fragment output interface state}, and \texttt{subpass} does not use any color and/or depth/stencil attachments, then the \texttt{rasterizationSamples} member of \texttt{pMultisampleState} must follow the rules for a zero-attachment subpass

- \textbf{VUID-VkGraphicsPipelineCreateInfo-renderPass-06046}
  If \texttt{renderPass} is not \texttt{VK_NULL_HANDLE}, \texttt{subpass} must be a valid subpass within \texttt{renderPass}

- \textbf{VUID-VkGraphicsPipelineCreateInfo-renderPass-06047}
  If \texttt{renderPass} is not \texttt{VK_NULL_HANDLE}, the pipeline is being created with \textit{pre-rasterization shader state}, \texttt{subpass} viewMask is not 0, and \texttt{multiviewTessellationShader} is not enabled, then \texttt{pStages} must not include tessellation shaders

- \textbf{VUID-VkGraphicsPipelineCreateInfo-renderPass-06048}
  If \texttt{renderPass} is not \texttt{VK_NULL_HANDLE}, the pipeline is being created with \textit{pre-rasterization shader state}, \texttt{subpass} viewMask is not 0, and \texttt{multiviewGeometryShader} is not enabled, then \texttt{pStages} must not include a geometry shader
If `renderPass` is not `VK_NULL_HANDLE`, the pipeline is being created with pre-rasterization shader state, and `subpass` viewMask is not 0, all of the shaders in the pipeline must not write to the Layer built-in output.

If `renderPass` is not `VK_NULL_HANDLE` and the pipeline is being created with pre-rasterization shader state, and `subpass` viewMask is not 0, then all of the shaders in the pipeline must not include variables decorated with the Layer built-in decoration in their interfaces.

If `renderPass` is not `VK_NULL_HANDLE` and the pipeline is being created with pre-rasterization shader state, and `subpass` viewMask is not 0, then all of the shaders in the pipeline must not include variables decorated with the ViewMask built-in decoration in their interfaces.

Flags must not contain the `VK_PIPELINE_CREATE_DISPATCH_BASE` flag.

If the pipeline is being created with fragment shader state and an input attachment was referenced by an aspectMask at renderPass creation time, the fragment shader must only read from the aspects that were specified for that input attachment.

The number of resources in layout accessible to each shader stage that is used by the pipeline must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageResources`.

If the pipeline is being created with vertex input state, `pVertexInputState` must be a valid pointer to a valid `VkPipelineVertexInputStateCreateInfo` structure.

If the pipeline is being created with vertex input state, `pInputAssemblyState` must be a valid pointer to a valid `VkPipelineInputAssemblyStateCreateInfo` structure.

If the pipeline is being created with pre-rasterization shader state, and `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT` is included in the `pDynamicStates` array then `viewportCount` must be zero.

If the pipeline is being created with pre-rasterization shader state, and `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT` is included in the `pDynamicStates` array then `VK_DYNAMIC_STATE_VIEWPORT` must not be present.

If the pipeline is being created with pre-rasterization shader state, and `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT` is included in the `pDynamicStates` array then `VK_DYNAMIC_STATE_VIEWPORT` must not be present.

If the pipeline is being created with pre-rasterization shader state, and
• **VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT** is included in the pDynamicStates array then **VK_DYNAMIC_STATE_SCISSOR** must not be present.

• **VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-04869**
  If the extendedDynamicState2LogicOp feature is not enabled, there must be no element of the pDynamicStates member of pDynamicState set to VK_DYNAMIC_STATE_LOGIC_OP_EXT.

• **VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-04870**
  If the extendedDynamicState2PatchControlPoints feature is not enabled, there must be no element of the pDynamicStates member of pDynamicState set to VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT.

• **VUID-VkGraphicsPipelineCreateInfo-pipelineCreationCacheControl-02878**
  If the pipelineCreationCacheControl feature is not enabled, flags must not include VK_PIPELINE_CREATE_FAIL_ON_PIPELINE_COMPILE_REQUIRED_BIT or VK_PIPELINE_CREATE_EARLY_RETURN_ON_FAILURE_BIT.

• **VUID-VkGraphicsPipelineCreateInfo-None-06573**
  The pipeline must be created with a complete set of state.

• **VUID-VkGraphicsPipelineCreateInfo-renderPass-06575**
  If the pipeline is being created with pre-rasterization shader state, fragment shader state, or fragment output interface state, renderPass must be VK_NULL_HANDLE or a valid render pass object.

• **VUID-VkGraphicsPipelineCreateInfo-dynamicRendering-06576**
  If the dynamicRendering feature is not enabled and the pipeline is being created with pre-rasterization shader state, fragment shader state, or fragment output interface state, renderPass must not be VK_NULL_HANDLE.

• **VUID-VkGraphicsPipelineCreateInfo-multiview-06577**
  If the multiview feature is not enabled, the pipeline is being created with pre-rasterization shader state, fragment shader state, or fragment output interface state, and renderPass is VK_NULL_HANDLE, VkPipelineRenderingCreateInfo::viewMask must be 0.

• **VUID-VkGraphicsPipelineCreateInfo-renderPass-06578**
  If the pipeline is being created with pre-rasterization shader state, fragment shader state, or fragment output interface state, and renderPass is VK_NULL_HANDLE, each element of VkPipelineRenderingCreateInfo::pColorAttachmentFormats must be a valid VkFormat value.

• **VUID-VkGraphicsPipelineCreateInfo-renderPass-06579**
  If the pipeline is being created with fragment output interface state, and renderPass is VK_NULL_HANDLE, and VkPipelineRenderingCreateInfo::colorAttachmentCount is not 0, VkPipelineRenderingCreateInfo::pColorAttachmentFormats must be a valid pointer to an array of colorAttachmentCount valid VkFormat values.

• **VUID-VkGraphicsPipelineCreateInfo-renderPass-06580**
  If the pipeline is being created with fragment output interface state, and renderPass is VK_NULL_HANDLE, each element of VkPipelineRenderingCreateInfo::pColorAttachmentFormats must be a valid VkFormat value.

• **VUID-VkGraphicsPipelineCreateInfo-renderPass-06581**
  If the pipeline is being created with fragment output interface state, renderPass is
VK_NULL_HANDLE, and any element of VkPipelineRenderingCreateInfo::pColorAttachmentFormats is not VK_FORMAT_UNDEFINED, that format must be a format with potential format features that include VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06583**
  If the pipeline is being created with fragment output interface state, and renderPass is VK_NULL_HANDLE, VkPipelineRenderingCreateInfo::depthAttachmentFormat must be a valid VkFormat value

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06584**
  If the pipeline is being created with fragment output interface state, and renderPass is VK_NULL_HANDLE, VkPipelineRenderingCreateInfo::stencilAttachmentFormat must be a valid VkFormat value

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06585**
  If the pipeline is being created with fragment output interface state, renderPass is VK_NULL_HANDLE, and VkPipelineRenderingCreateInfo::depthAttachmentFormat is not VK_FORMAT_UNDEFINED, it must be a format with potential format features that include VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06586**
  If the pipeline is being created with fragment output interface state, renderPass is VK_NULL_HANDLE, and VkPipelineRenderingCreateInfo::stencilAttachmentFormat is not VK_FORMAT_UNDEFINED, it must be a format with potential format features that include VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06587**
  If the pipeline is being created with fragment output interface state, renderPass is VK_NULL_HANDLE, and VkPipelineRenderingCreateInfo::depthAttachmentFormat is not VK_FORMAT_UNDEFINED, it must be a format that includes a depth aspect

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06588**
  If the pipeline is being created with fragment output interface state, renderPass is VK_NULL_HANDLE, and VkPipelineRenderingCreateInfo::stencilAttachmentFormat is not VK_FORMAT_UNDEFINED, it must be a format that includes a stencil aspect

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06589**
  If the pipeline is being created with fragment output interface state, renderPass is VK_NULL_HANDLE, VkPipelineRenderingCreateInfo::depthAttachmentFormat is not VK_FORMAT_UNDEFINED, and VkPipelineRenderingCreateInfo::stencilAttachmentFormat is not VK_FORMAT_UNDEFINED, depthAttachmentFormat must equal stencilAttachmentFormat

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06053**
  If renderPass is VK_NULL_HANDLE, the pipeline is being created with fragment shader state and fragment output interface state, and either of VkPipelineRenderingCreateInfo::depthAttachmentFormat or VkPipelineRenderingCreateInfo::stencilAttachmentFormat are not VK_FORMAT_UNDEFINED, pDepthStencilState must be a valid pointer to a valid VkPipelineDepthStencilStateCreateInfo structure

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06054**
  If renderPass is VK_NULL_HANDLE, the pipeline is being created with fragment output interface state, and VkPipelineRenderingCreateInfo::colorAttachmentCount is not equal to 0, pColorBlendState must be a valid pointer to a valid
VkPipelineColorBlendStateCreateInfo structure

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06055**
  If `renderPass` is `VK_NULL_HANDLE` and the pipeline is being created with fragment output interface state, `pColorBlendState->attachmentCount` must be equal to `VkPipelineRenderingCreateInfo::colorAttachmentCount`

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06056**
  If `renderPass` is `VK_NULL_HANDLE` and the pipeline is being created with fragment shader state, the fragment shader must not read from any input attachment

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06057**
  If `renderPass` is `VK_NULL_HANDLE`, the pipeline is being created with pre-rasterization shader state, `VkPipelineRenderingCreateInfo::viewMask` is not 0, and the `multiviewTessellationShader` feature is not enabled, then `pStages` must not include tessellation shaders

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06058**
  If `renderPass` is `VK_NULL_HANDLE`, the pipeline is being created with pre-rasterization shader state, `VkPipelineRenderingCreateInfo::viewMask` is not 0, and the `multiviewGeometryShader` feature is not enabled, then `pStages` must not include a geometry shader

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-07718**
  If `renderPass` is `VK_NULL_HANDLE`, the pipeline is being created with pre-rasterization shader state, and `VkPipelineRenderingCreateInfo::viewMask` is not 0, all of the shaders in the pipeline must not write to the `Layer` built-in output

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06059**
  If `renderPass` is `VK_NULL_HANDLE`, the pipeline is being created with pre-rasterization shader state, and `VkPipelineRenderingCreateInfo::viewMask` is not 0, all of the shaders in the pipeline must not include variables decorated with the `Layer` built-in decoration in their interfaces

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-07719**
  If `renderPass` is `VK_NULL_HANDLE`, the pipeline is being created with pre-rasterization shader state, and `VkPipelineRenderingCreateInfo::viewMask` is not 0, all of the shaders in the pipeline must not include variables decorated with the `ViewIndex` built-in decoration in their interfaces

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06061**
  If the pipeline is being created with fragment shader state and `renderPass` is `VK_NULL_HANDLE`, fragment shaders in `pStages` must not include the `InputAttachment` capability

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06062**
  If the pipeline is being created with fragment output interface state and `renderPass` is `VK_NULL_HANDLE`, for each color attachment format defined by the `pColorAttachmentFormats` member of `VkPipelineRenderingCreateInfo`, if its potential format features do not contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT`, then the `blendEnable` member of the corresponding element of the `pAttachments` member of `pColorBlendState` must be `VK_FALSE`

- **VUID-VkGraphicsPipelineCreateInfo-pipelineStageCreationFeedbackCount-06594**
If \( \text{VkPipelineCreationFeedbackCreateInfo}::\text{pipelineStageCreationFeedbackCount} \) is not 0, it must be equal to \( \text{stageCount} \)

- VUID-VkGraphicsPipelineCreateInfo-pStages-06600
  \( \text{pStages} \) must be a valid pointer to an array of \( \text{stageCount} \) valid \( \text{VkPipelineShaderStageCreateInfo} \) structures

- VUID-VkGraphicsPipelineCreateInfo-pRasterizationState-06601
  \( \text{pRasterizationState} \) must be a valid pointer to a valid \( \text{VkPipelineRasterizationStateCreateInfo} \) structure

- VUID-VkGraphicsPipelineCreateInfo-layout-06602
  \( \text{layout} \) must be a valid \( \text{VkPipelineLayout} \) handle

- VUID-VkGraphicsPipelineCreateInfo-renderPass-06603
  If \( \text{renderPass} \) is not \( \text{VK_NULL_HANDLE} \), \( \text{renderPass} \) must be a valid \( \text{VkRenderPass} \) handle

- VUID-VkGraphicsPipelineCreateInfo-stageCount-06604
  \( \text{stageCount} \) must be greater than 0

- VUID-VkGraphicsPipelineCreateInfo-pStages-06894
  If the pipeline is being created with pre-rasterization shader state but not fragment shader state, elements of \( \text{pStages} \) must not have stage set to \( \text{VK_SHADER_STAGE_FRAGMENT_BIT} \)

- VUID-VkGraphicsPipelineCreateInfo-pStages-06895
  If the pipeline is being created with fragment shader state but not pre-rasterization shader state, elements of \( \text{pStages} \) must not have stage set to a shader stage which participates in pre-rasterization

- VUID-VkGraphicsPipelineCreateInfo-pStages-06896
  If the pipeline is being created with pre-rasterization shader state, all elements of \( \text{pStages} \) must have a stage set to a shader stage which participates in fragment shader state or pre-rasterization shader state

- VUID-VkGraphicsPipelineCreateInfo-stage-06897
  If the pipeline is being created with fragment shader state and/or pre-rasterization shader state, any value of stage must not be set in more than one element of \( \text{pStages} \)

Valid Usage (Implicit)

- VUID-VkGraphicsPipelineCreateInfo-sType-sType
  \( \text{sType} \) must be \( \text{VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO} \)

- VUID-VkGraphicsPipelineCreateInfo-pNext-pNext
  Each \( \text{pNext} \) member of any structure (including this one) in the \( \text{pNext} \) chain must be either \( \text{NULL} \) or a pointer to a valid instance of \( \text{VkPipelineCreationFeedbackCreateInfo} \) or \( \text{VkPipelineRenderingCreateInfo} \)

- VUID-VkGraphicsPipelineCreateInfo-sType-unique
  The \( \text{sType} \) value of each struct in the \( \text{pNext} \) chain must be unique

- VUID-VkGraphicsPipelineCreateInfo-flags-parameter
  \( \text{flags} \) must be a valid combination of \( \text{VkPipelineCreateFlagBits} \) values

- VUID-VkGraphicsPipelineCreateInfo-pDynamicState-parameter
If `pDynamicState` is not `NULL`, `pDynamicState` must be a valid pointer to a valid `VkPipelineDynamicStateCreateInfo` structure.

- VUID-VkGraphicsPipelineCreateInfo-commonparent
  Each of `basePipelineHandle`, `layout`, and `renderPass` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`.

The `VkPipelineRenderingCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPipelineRenderingCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t viewMask;
    uint32_t colorAttachmentCount;
    const VkFormat* pColorAttachmentFormats;
    VkFormat depthAttachmentFormat;
    VkFormat stencilAttachmentFormat;
} VkPipelineRenderingCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `viewMask` is the viewMask used for rendering.
- `colorAttachmentCount` is the number of entries in `pColorAttachmentFormats`.
- `pColorAttachmentFormats` is a pointer to an array of `VkFormat` values defining the format of color attachments used in this pipeline.
- `depthAttachmentFormat` is a `VkFormat` value defining the format of the depth attachment used in this pipeline.
- `stencilAttachmentFormat` is a `VkFormat` value defining the format of the stencil attachment used in this pipeline.

When a pipeline is created without a `VkRenderPass`, if this structure is present in the `pNext` chain of `VkGraphicsPipelineCreateInfo`, it specifies the view mask and format of attachments used for rendering. If this structure is not specified, and the pipeline does not include a `VkRenderPass`, `viewMask` and `colorAttachmentCount` are 0, and `depthAttachmentFormat` and `stencilAttachmentFormat` are `VK_FORMAT_UNDEFINED`. If a graphics pipeline is created with a valid `VkRenderPass`, parameters of this structure are ignored.

If `depthAttachmentFormat`, `stencilAttachmentFormat`, or any element of `pColorAttachmentFormats` is `VK_FORMAT_UNDEFINED`, it indicates that the corresponding attachment is unused within the render pass. Valid formats indicate that an attachment can be used - but it is still valid to set the attachment to `NULL` when beginning rendering.
Valid Usage (Implicit)

- **VUID-VkPipelineRenderingCreateInfo-sType-sType**
  
  the `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_RENDERING_CREATE_INFO`

Bits which **can** be set in

- `VkGraphicsPipelineCreateInfo::flags`
- `VkComputePipelineCreateInfo::flags`

specify how a pipeline is created, and are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkPipelineCreateFlagBits {
    VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT = 0x00000001,
    VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT = 0x00000002,
    VK_PIPELINE_CREATE_DERIVATIVE_BIT = 0x00000004,
} VkPipelineCreateFlagBits;
```

- **VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT** specifies that the created pipeline will not be optimized. Using this flag may reduce the time taken to create the pipeline.

- **VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT** specifies that the pipeline to be created is allowed to be the parent of a pipeline that will be created in a subsequent pipeline creation call.

- **VK_PIPELINE_CREATE_DERIVATIVE_BIT** specifies that the pipeline to be created will be a child of a previously created parent pipeline.

- **VK_PIPELINE_CREATE_VIEW_INDEX_FROM_DEVICE_INDEX_BIT** specifies that any shader input variables decorated as `ViewIndex` will be assigned values as if they were decorated as `DeviceIndex`.

- **VK_PIPELINE_CREATE_DISPATCH_BASE** specifies that a compute pipeline can be used with `vkCmdDispatchBase` with a non-zero base workgroup.

- **VK_PIPELINE_CREATE_FAIL_ON_PIPELINE_COMPILE_REQUIRED_BIT** specifies that pipeline creation will fail if a compile is required for creation of a valid `VkPipeline` object; `VK_PIPELINE_COMPILE_REQUIRED` will be returned by pipeline creation, and the `VkPipeline` will be set to `VK_NULL_HANDLE`.

- When creating multiple pipelines, **VK_PIPELINE_CREATE_EARLY_RETURN_ON_FAILURE_BIT** specifies
that control will be returned to the application if any individual pipeline returns a result which
is not \texttt{VK\_SUCCESS} rather than continuing to create additional pipelines.

It is valid to set both \texttt{VK\_PIPELINE\_CREATE\_ALLOW\_DERIVATIVES\_BIT} and
\texttt{VK\_PIPELINE\_CREATE\_DERIVATIVE\_BIT}. This allows a pipeline to be both a parent and possibly a child
in a pipeline hierarchy. See Pipeline Derivatives for more information.

\begin{verbatim}
// Provided by VK\_VERSION\_1\_0
typedef VkFlags VkPipelineCreateFlags;
\end{verbatim}

\texttt{VkPipelineCreateFlags} is a bitmask type for setting a mask of zero or more
\texttt{VkPipelineCreateFlagBits}.

The \texttt{VkPipelineDynamicStateCreateInfo} structure is defined as:

\begin{verbatim}
// Provided by VK\_VERSION\_1\_0
typedef struct VkPipelineDynamicStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineDynamicStateCreateFlags flags;
    uint32_t dynamicStateCount;
    const VkDynamicState* pDynamicStates;
} VkPipelineDynamicStateCreateInfo;
\end{verbatim}

- \texttt{sType} is the type of this structure.
- \texttt{pNext} is \texttt{NULL} or a pointer to a structure extending this structure.
- \texttt{flags} is reserved for future use.
- \texttt{dynamicStateCount} is the number of elements in the \texttt{pDynamicStates} array.
- \texttt{pDynamicStates} is a pointer to an array of \texttt{VkDynamicState} values specifying which pieces of
pipeline state will use the values from dynamic state commands rather than from pipeline state
creation information.

\textbf{Valid Usage}

- VUID-VkPipelineDynamicStateCreateInfo-pDynamicStates-01442
  Each element of \texttt{pDynamicStates} must be unique

\textbf{Valid Usage (Implicit)}

- VUID-VkPipelineDynamicStateCreateInfo-sType-sType
  \texttt{sType} must be \texttt{VK\_STRUCTURE\_TYPE\_PIPELINE\_DYNAMIC\_STATE\_CREATE\_INFO}
- VUID-VkPipelineDynamicStateCreateInfo-pNext-pNext
  \texttt{pNext} must be \texttt{NULL}
• VUID-VkPipelineDynamicStateCreateInfo-flags-zerobitmask
  flags must be 0

• VUID-VkPipelineDynamicStateCreateInfo-pDynamicStates-parameter
  If dynamicStateCount is not 0, pDynamicStates must be a valid pointer to an array of
dynamicStateCount valid VkDynamicState values

// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineDynamicStateCreateFlags;

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

The source of different pieces of dynamic state is specified by the
VkPipelineDynamicStateCreateInfo::pDynamicStates property of the currently active pipeline, each
of whose elements must be one of the values:

// Provided by VK_VERSION_1_0
typedef enum VkDynamicState {
  VK_DYNAMIC_STATE_VIEWPORT = 0,
  VK_DYNAMIC_STATE_SCISSOR = 1,
  VK_DYNAMIC_STATE_LINE_WIDTH = 2,
  VK_DYNAMIC_STATE_DEPTH_BIAS = 3,
  VK_DYNAMIC_STATE_BLEND_CONSTANTS = 4,
  VK_DYNAMIC_STATE_DEPTH_BOUNDS = 5,
  VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK = 6,
  VK_DYNAMIC_STATE_STENCIL_WRITE_MASK = 7,
  VK_DYNAMIC_STATE_STENCIL_REFERENCE = 8,
  // Provided by VK_VERSION_1_3
  VK_DYNAMIC_STATE_CULL_MODE = 1000267000,
  // Provided by VK_VERSION_1_3
  VK_DYNAMIC_STATE_FRONT_FACE = 1000267001,
  // Provided by VK_VERSION_1_3
  VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY = 1000267002,
  // Provided by VK_VERSION_1_3
  VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT = 1000267003,
  // Provided by VK_VERSION_1_3
  VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT = 1000267004,
  // Provided by VK_VERSION_1_3
  VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE = 1000267005,
  // Provided by VK_VERSION_1_3
  VK_DYNAMIC_STATE_DEPTH_TEST_ENABLE = 1000267006,
  // Provided by VK_VERSION_1_3
  VK_DYNAMIC_STATE_DEPTH_WRITE_ENABLE = 1000267007,
  // Provided by VK_VERSION_1_3
  VK_DYNAMIC_STATE_DEPTH_COMPARE_OP = 1000267008,
  // Provided by VK_VERSION_1_3
  VK_DYNAMIC_STATE_DEPTH_BOUNDS_TEST_ENABLE = 1000267009,
  // Provided by VK_VERSION_1_3
}
VK_DYNAMIC_STATE_STENCIL_TEST_ENABLE = 1000267010,
// Provided by VK_VERSION_1_3
VK_DYNAMIC_STATE_STENCIL_OP = 1000267011,
// Provided by VK_VERSION_1_3
VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE = 1000377001,
// Provided by VK_VERSION_1_3
VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE = 1000377002,
// Provided by VK_VERSION_1_3
VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE = 1000377004,
} VkDynamicState;

• VK_DYNAMIC_STATE_VIEWPORT specifies that the pViewports state in VkPipelineViewportStateCreateInfo will be ignored and must be set dynamically with vkCmdSetViewport before any drawing commands. The number of viewports used by a pipeline is still specified by the viewportCount member of VkPipelineViewportStateCreateInfo.

• VK_DYNAMIC_STATE_SCISSOR specifies that the pScissors state in VkPipelineViewportStateCreateInfo will be ignored and must be set dynamically with vkCmdSetScissor before any drawing commands. The number of scissor rectangles used by a pipeline is still specified by the scissorCount member of VkPipelineViewportStateCreateInfo.

• VK_DYNAMIC_STATE_LINE_WIDTH specifies that the lineWidth state in VkPipelineRasterizationStateCreateInfo will be ignored and must be set dynamically with vkCmdSetLineWidth before any drawing commands that generate line primitives for the rasterizer.

• VK_DYNAMIC_STATE_DEPTH_BIAS specifies that the depthBiasConstantFactor, depthBiasClamp and depthBiasSlopeFactor states in VkPipelineRasterizationStateCreateInfo will be ignored and must be set dynamically with vkCmdSetDepthBias before any draws are performed with depthBiasEnable in VkPipelineRasterizationStateCreateInfo set to VK_TRUE.

• VK_DYNAMIC_STATE_BLEND_CONSTANTS specifies that the blendConstants state in VkPipelineColorBlendStateCreateInfo will be ignored and must be set dynamically with vkCmdSetBlendConstants before any draws are performed with a pipeline state with VkPipelineColorBlendAttachmentState member blendEnable set to VK_TRUE and any of the blend functions using a constant blend color.

• VK_DYNAMIC_STATE_DEPTH_BOUNDS specifies that the minDepthBounds and maxDepthBounds states of VkPipelineDepthStencilStateCreateInfo will be ignored and must be set dynamically with vkCmdSetDepthBounds before any draws are performed with a pipeline state with VkPipelineDepthStencilStateCreateInfo member depthBoundsTestEnable set to VK_TRUE.

• VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK specifies that the compareMask state in VkPipelineDepthStencilStateCreateInfo for both front and back will be ignored and must be set dynamically with vkCmdSetStencilCompareMask before any draws are performed with a pipeline state with VkPipelineDepthStencilStateCreateInfo member stencilTestEnable set to VK_TRUE.

• VK_DYNAMIC_STATE_STENCIL_WRITE_MASK specifies that the writeMask state in VkPipelineDepthStencilStateCreateInfo for both front and back will be ignored and must be set dynamically with vkCmdSetStencilWriteMask before any draws are performed with a pipeline state with VkPipelineDepthStencilStateCreateInfo member stencilTestEnable set to VK_TRUE.
• **VK_DYNAMIC_STATE_STENCIL_REFERENCE** specifies that the reference state in `VkPipelineDepthStencilStateCreateInfo` for both front and back will be ignored and **must** be set dynamically with `vkCmdSetStencilReference` before any draws are performed with a pipeline state with `VkPipelineDepthStencilStateCreateInfo` member `stencilTestEnable` set to **VK_TRUE**.

• **VK_DYNAMIC_STATE_CULL_MODE** specifies that the cullMode state in `VkPipelineRasterizationStateCreateInfo` will be ignored and **must** be set dynamically with `vkCmdSetCullMode` before any drawing commands.

• **VK_DYNAMIC_STATE_FRONT_FACE** specifies that the frontFace state in `VkPipelineRasterizationStateCreateInfo` will be ignored and **must** be set dynamically with `vkCmdSetFrontFace` before any drawing commands.

• **VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY** specifies that the topology state in `VkPipelineInputAssemblyStateCreateInfo` only specifies the topology class, and the specific topology order and adjacency **must** be set dynamically with `vkCmdSetPrimitiveTopology` before any drawing commands.

• **VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT** specifies that the viewportCount and pViewports state in `VkPipelineViewportStateCreateInfo` will be ignored and **must** be set dynamically with `vkCmdSetViewportWithCount` before any draw call.

• **VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT** specifies that the scissorCount and pScissors state in `VkPipelineViewportStateCreateInfo` will be ignored and **must** be set dynamically with `vkCmdSetScissorWithCount` before any draw call.

• **VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE** specifies that the stride state in `VkVertexInputBindingDescription` will be ignored and **must** be set dynamically with `vkCmdBindVertexBuffers2` before any draw call.

• **VK_DYNAMIC_STATE_DEPTH_TEST_ENABLE** specifies that the depthTestEnable state in `VkPipelineDepthStencilStateCreateInfo` will be ignored and **must** be set dynamically with `vkCmdSetDepthTestEnable` before any draw call.

• **VK_DYNAMIC_STATE_DEPTH_WRITE_ENABLE** specifies that the depthWriteEnable state in `VkPipelineDepthStencilStateCreateInfo` will be ignored and **must** be set dynamically with `vkCmdSetDepthWriteEnable` before any draw call.

• **VK_DYNAMIC_STATE_DEPTH_COMPARE_OP** specifies that the depthCompareOp state in `VkPipelineDepthStencilStateCreateInfo` will be ignored and **must** be set dynamically with `vkCmdSetDepthCompareOp` before any draw call.

• **VK_DYNAMIC_STATE_DEPTH_BOUNDS_TEST_ENABLE** specifies that the depthBoundsTestEnable state in `VkPipelineDepthStencilStateCreateInfo` will be ignored and **must** be set dynamically with `vkCmdSetDepthBoundsTestEnable` before any draw call.

• **VK_DYNAMIC_STATE_STENCIL_TEST_ENABLE** specifies that the stencilTestEnable state in `VkPipelineDepthStencilStateCreateInfo` will be ignored and **must** be set dynamically with `vkCmdSetStencilTestEnable` before any draw call.

• **VK_DYNAMIC_STATE_STENCIL_OP** specifies that the failOp, passOp, depthFailOp, and compareOp states in `VkPipelineDepthStencilStateCreateInfo` for both front and back will be ignored and **must** be set dynamically with `vkCmdSetStencilOp` before any draws are performed with a pipeline state with `VkPipelineDepthStencilStateCreateInfo` member `stencilTestEnable` set to **VK_TRUE**.

• **VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE** specifies that the rasterizerDiscardEnable state in
**VkPipelineRasterizationStateCreateInfo** will be ignored and **must** be set dynamically with **vkCmdSetRasterizerDiscardEnable** before any drawing commands.

- **VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE** specifies that the **depthBiasEnable** state in **VkPipelineRasterizationStateCreateInfo** will be ignored and **must** be set dynamically with **vkCmdSetDepthBiasEnable** before any drawing commands.

- **VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE** specifies that the **primitiveRestartEnable** state in **VkPipelineInputAssemblyStateCreateInfo** will be ignored and **must** be set dynamically with **vkCmdSetPrimitiveRestartEnable** before any drawing commands.

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

If tessellation shader stages are omitted, the tessellation shading and fixed-function stages of the pipeline are skipped.

If a geometry shader is omitted, the geometry shading stage is skipped.

If a fragment shader is omitted, fragment color outputs have undefined values, and the fragment depth value is determined by **Fragment Operations** state. This **can** be useful for depth-only rendering.

Presence of a shader stage in a pipeline is indicated by including a valid **VkPipelineShaderStageCreateInfo** with **module** and **pName** selecting an entry point from a shader module, where that entry point is valid for the stage specified by **stage**.

Presence of some of the fixed-function stages in the pipeline is implicitly derived from enabled shaders and provided state. For example, the fixed-function tessellator is always present when the pipeline has valid Tessellation Control and Tessellation Evaluation shaders.

*For example:*

- Depth/stencil-only rendering in a subpass with no color attachments
  - Active Pipeline Shader Stages
    - Vertex Shader
  - Required: Fixed-Function Pipeline Stages
    - **VkPipelineVertexInputStateCreateInfo**
    - **VkPipelineInputAssemblyStateCreateInfo**
    - **VkPipelineViewportStateCreateInfo**
    - **VkPipelineRasterizationStateCreateInfo**
    - **VkPipelineMultisampleStateCreateInfo**
    - **VkPipelineDepthStencilStateCreateInfo**

- Color-only rendering in a subpass with no depth/stencil attachment
  - Active Pipeline Shader Stages
    - Vertex Shader
    - Fragment Shader
• Required: Fixed-Function Pipeline Stages
  • VkPipelineVertexInputStateCreateInfo
  • VkPipelineInputAssemblyStateCreateInfo
  • VkPipelineViewportStateCreateInfo
  • VkPipelineRasterizationStateCreateInfo
  • VkPipelineMultisampleStateCreateInfo
  • VkPipelineColorBlendStateCreateInfo

• Rendering pipeline with tessellation and geometry shaders
  • Active Pipeline Shader Stages
    • Vertex Shader
    • Tessellation Control Shader
    • Tessellation Evaluation Shader
    • Geometry Shader
    • Fragment Shader
  • Required: Fixed-Function Pipeline Stages
    • VkPipelineVertexInputStateCreateInfo
    • VkPipelineInputAssemblyStateCreateInfo
    • VkPipelineTessellationStateCreateInfo
    • VkPipelineViewportStateCreateInfo
    • VkPipelineRasterizationStateCreateInfo
    • VkPipelineMultisampleStateCreateInfo
    • VkPipelineColorBlendStateCreateInfo

10.3. Pipeline Destruction

To destroy a pipeline, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyPipeline(
    VkDevice device,
    VkPipeline pipeline,
    const VkAllocationCallbacks* pAllocator);
```

• `device` is the logical device that destroys the pipeline.
• `pipeline` is the handle of the pipeline to destroy.
• `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
Valid Usage

- VUID-vkDestroyPipeline-pipeline-00765
  All submitted commands that refer to pipeline must have completed execution

- VUID-vkDestroyPipeline-pipeline-00766
  If VkAllocationCallbacks were provided when pipeline was created, a compatible set of callbacks must be provided here

- VUID-vkDestroyPipeline-pipeline-00767
  If no VkAllocationCallbacks were provided when pipeline was created, pAllocator must be NULL

Valid Usage (Implicit)

- VUID-vkDestroyPipeline-device-parameter
  device must be a valid VkDevice handle

- VUID-vkDestroyPipeline-pipeline-parameter
  If pipeline is not VK_NULL_HANDLE, pipeline must be a valid VkPipeline handle

- VUID-vkDestroyPipeline-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkDestroyPipeline-pipeline-parent
  If pipeline is a valid handle, it must have been created, allocated, or retrieved from device

Host Synchronization

- Host access to pipeline must be externally synchronized

10.4. Multiple Pipeline Creation

Multiple pipelines can be created simultaneously by passing an array of VkGraphicsPipelineCreateInfo, or VkComputePipelineCreateInfo structures into the vkCreateGraphicsPipelines, and vkCreateComputePipelines commands, respectively. Applications can group together similar pipelines to be created in a single call, and implementations are encouraged to look for reuse opportunities within a group-create.

When an application attempts to create many pipelines in a single command, it is possible that some subset may fail creation. In that case, the corresponding entries in the pPipelines output array will be filled with VK_NULL_HANDLE values. If any pipeline fails creation despite valid arguments (for example, due to out of memory errors), the VkResult code returned by vkCreate*Pipelines will indicate why. The implementation will attempt to create all pipelines, and only return VK_NULL_HANDLE values for those that actually failed.
If creation fails for a pipeline that had `VK_PIPELINE_CREATE_EARLY_RETURN_ON_FAILURE_BIT` set, pipelines at an index in the `pPipelines` array greater than or equal to that of the failing pipeline must be set to `VK_NULL_HANDLE`.

If multiple pipelines fail to be created, the `VkResult` must be the return value of any of the pipelines which did not return `VK_SUCCESS`.

### 10.5. Pipeline Derivatives

A pipeline derivative is a child pipeline created from a parent pipeline, where the child and parent are expected to have much commonality. The goal of derivative pipelines is that they be cheaper to create using the parent as a starting point, and that it be more efficient (on either host or device) to switch/bind between children of the same parent.

A derivative pipeline is created by setting the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag in the `Vk*PipelineCreateInfo` structure. If this is set, then exactly one of `basePipelineHandle` or `basePipelineIndex` members of the structure must have a valid handle/index, and specifies the parent pipeline. If `basePipelineHandle` is used, the parent pipeline must have already been created. If `basePipelineIndex` is used, then the parent is being created in the same command. `VK_NULL_HANDLE` acts as the invalid handle for `basePipelineHandle`, and -1 is the invalid index for `basePipelineIndex`. If `basePipelineIndex` is used, the base pipeline must appear earlier in the array. The base pipeline must have been created with the `VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT` flag set.

### 10.6. Pipeline Cache

Pipeline cache objects allow the result of pipeline construction to be reused between pipelines and between runs of an application. Reuse between pipelines is achieved by passing the same pipeline cache object when creating multiple related pipelines. Reuse across runs of an application is achieved by retrieving pipeline cache contents in one run of an application, saving the contents, and using them to preinitialize a pipeline cache on a subsequent run. The contents of the pipeline cache objects are managed by the implementation. Applications can manage the host memory consumed by a pipeline cache object and control the amount of data retrieved from a pipeline cache object.

Pipeline cache objects are represented by `VkPipelineCache` handles:

```cpp
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPipelineCache)
```

### 10.6.1. Creating a Pipeline Cache

To create pipeline cache objects, call:

```cpp
// Provided by VK_VERSION_1_0
VkResult vkCreatePipelineCache(VkDevice device,
                               const VkPipelineCacheCreateInfo* createInfo, 
                               VkPipelineCache* out)
```
const VkPipelineCacheCreateInfo* pCreateInfo,
const VkAllocationCallbacks* pAllocator,
VkPipelineCache* pPipelineCache);

- **device** is the logical device that creates the pipeline cache object.
- **pCreateInfo** is a pointer to a `VkPipelineCacheCreateInfo` structure containing initial parameters for the pipeline cache object.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
- **pPipelineCache** is a pointer to a `VkPipelineCache` handle in which the resulting pipeline cache object is returned.

**Note**
Applications can track and manage the total host memory size of a pipeline cache object using the **pAllocator**. Applications can limit the amount of data retrieved from a pipeline cache object in `vkGetPipelineCacheData`. Implementations should not internally limit the total number of entries added to a pipeline cache object or the total host memory consumed.

Once created, a pipeline cache can be passed to the `vkCreateGraphicsPipelines` and `vkCreateComputePipelines` commands. If the pipeline cache passed into these commands is not `VK_NULL_HANDLE`, the implementation will query it for possible reuse opportunities and update it with new content. The use of the pipeline cache object in these commands is internally synchronized, and the same pipeline cache object can be used in multiple threads simultaneously.

If flags of **pCreateInfo** includes `VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT`, all commands that modify the returned pipeline cache object must be externally synchronized.

**Note**
Implementations should make every effort to limit any critical sections to the actual accesses to the cache, which is expected to be significantly shorter than the duration of the `vkCreate*Pipelines` commands.

---

**Valid Usage (Implicit)**

- **VUID-vkCreatePipelineCache-device-parameter**
  - **device** must be a valid `VkDevice` handle

- **VUID-vkCreatePipelineCache-pCreateInfo-parameter**
  - **pCreateInfo** must be a valid pointer to a valid `VkPipelineCacheCreateInfo` structure

- **VUID-vkCreatePipelineCache-pAllocator-parameter**
  - If **pAllocator** is not NULL, **pAllocator** must be a valid pointer to a valid `VkAllocationCallbacks` structure

- **VUID-vkCreatePipelineCache-pPipelineCache-parameter**
  - **pPipelineCache** must be a valid pointer to a `VkPipelineCache` handle
Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkPipelineCacheCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineCacheCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineCacheCreateFlags flags;
    size_t initialDataSize;
    const void* pInitialData;
} VkPipelineCacheCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `flags` is a bitmask of VkPipelineCacheCreateFlagBits specifying the behavior of the pipeline cache.
- `initialDataSize` is the number of bytes in `pInitialData`. If `initialDataSize` is zero, the pipeline cache will initially be empty.
- `pInitialData` is a pointer to previously retrieved pipeline cache data. If the pipeline cache data is incompatible (as defined below) with the device, the pipeline cache will be initially empty. If `initialDataSize` is zero, `pInitialData` is ignored.

Valid Usage

- VUID-VkPipelineCacheCreateInfo-initialDataSize-00768
  If `initialDataSize` is not 0, it must be equal to the size of `pInitialData`, as returned by `vkGetPipelineCacheData` when `pInitialData` was originally retrieved

- VUID-VkPipelineCacheCreateInfo-initialDataSize-00769
  If `initialDataSize` is not 0, `pInitialData` must have been retrieved from a previous call to `vkGetPipelineCacheData`

- VUID-VkPipelineCacheCreateInfo-pipelineCreationCacheControl-02892
  If the pipelineCreationCacheControl feature is not enabled, `flags` must not include `VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT`
Valid Usage (Implicit)

- **VUID-VkPipelineCacheCreateInfo-sType-sType**
  
  *sType must be* `VK_STRUCTURE_TYPE_PIPELINE_CACHE_CREATE_INFO`.

- **VUID-VkPipelineCacheCreateInfo-pNext-pNext**
  
  *pNext must be* `NULL`.

- **VUID-VkPipelineCacheCreateInfo-flags-parameter**
  
  *flags must be a valid combination of* `VkPipelineCacheCreateFlagBits` *values*.

- **VUID-VkPipelineCacheCreateInfo-pInitialData-parameter**
  
  If `initialDataSize` is not 0, *pInitialData must be a valid pointer to an array of initialDataSize bytes*.

---

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineCacheCreateFlags;
```

`VkPipelineCacheCreateFlags` is a bitmask type for setting a mask of zero or more `VkPipelineCacheCreateFlagBits`.

Bits which *can* be set in `VkPipelineCacheCreateInfo::flags`, specifying behavior of the pipeline cache, are:

```c
// Provided by VK_VERSION_1_3
typedef enum VkPipelineCacheCreateFlagBits {
    // Provided by VK_VERSION_1_3
    VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT = 0x00000001,
    } VkPipelineCacheCreateFlagBits;
```

- **VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT** specifies that all commands that modify the created `VkPipelineCache` will be *externally synchronized*. When set, the implementation *may* skip any unnecessary processing needed to support simultaneous modification from multiple threads where allowed.

### 10.6.2. Merging Pipeline Caches

Pipeline cache objects *can* be merged using the command:

```c
// Provided by VK_VERSION_1_0
VkResult vkMergePipelineCaches(
    VkDevice device,
    VkPipelineCache dstCache,
    uint32_t srcCacheCount,
    const VkPipelineCache* pSrcCaches);
```

- **device** is the logical device that owns the pipeline cache objects.
• `dstCache` is the handle of the pipeline cache to merge results into.
• `srcCacheCount` is the length of the `pSrcCaches` array.
• `pSrcCaches` is a pointer to an array of pipeline cache handles, which will be merged into `dstCache`. The previous contents of `dstCache` are included after the merge.

Note
The details of the merge operation are implementation-dependent, but implementations should merge the contents of the specified pipelines and prune duplicate entries.

Valid Usage
• VUID-vkMergePipelineCaches-dstCache-00770
  `dstCache` must not appear in the list of source caches

Valid Usage (Implicit)
• VUID-vkMergePipelineCaches-device-parameter
  `device` must be a valid `VkDevice` handle
• VUID-vkMergePipelineCaches-dstCache-parameter
  `dstCache` must be a valid `VkPipelineCache` handle
• VUID-vkMergePipelineCaches-pSrcCaches-parameter
  `pSrcCaches` must be a valid pointer to an array of `srcCacheCount` valid `VkPipelineCache` handles
• VUID-vkMergePipelineCaches-srcCacheCount-arraylength
  `srcCacheCount` must be greater than 0
• VUID-vkMergePipelineCaches-dstCache-parent
  `dstCache` must have been created, allocated, or retrieved from `device`
• VUID-vkMergePipelineCaches-pSrcCaches-parent
  Each element of `pSrcCaches` must have been created, allocated, or retrieved from `device`

Host Synchronization
• Host access to `dstCache` must be externally synchronized

Return Codes
Success
• `VK_SUCCESS`
10.6.3. Retrieving Pipeline Cache Data

Data can be retrieved from a pipeline cache object using the command:

```c
// Provided by VK_VERSION_1_0
VkResult vkGetPipelineCacheData(
    VkDevice device,
    VkPipelineCache pipelineCache,
    size_t* pDataSize,
    void* pData);
```

- `device` is the logical device that owns the pipeline cache.
- `pipelineCache` is the pipeline cache to retrieve data from.
- `pDataSize` is a pointer to a `size_t` value related to the amount of data in the pipeline cache, as described below.
- `pData` is either `NULL` or a pointer to a buffer.

If `pData` is `NULL`, then the maximum size of the data that can be retrieved from the pipeline cache, in bytes, is returned in `pDataSize`. Otherwise, `pDataSize` must point to a variable set by the user to the size of the buffer, in bytes, pointed to by `pData`, and on return the variable is overwritten with the amount of data actually written to `pData`. If `pDataSize` is less than the maximum size that can be retrieved by the pipeline cache, at most `pDataSize` bytes will be written to `pData`, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all of the pipeline cache was returned.

Any data written to `pData` is valid and can be provided as the `pInitialData` member of the `VkPipelineCacheCreateInfo` structure passed to `vkCreatePipelineCache`.

Two calls to `vkGetPipelineCacheData` with the same parameters must retrieve the same data unless a command that modifies the contents of the cache is called between them.

The initial bytes written to `pData` must be a header as described in the Pipeline Cache Header section.

If `pDataSize` is less than what is necessary to store this header, nothing will be written to `pData` and zero will be written to `pDataSize`.

Valid Usage (Implicit)

- `VUID-vkGetPipelineCacheData-device-parameter` `device` must be a valid `VkDevice` handle
- `VUID-vkGetPipelineCacheData-pipelineCache-parameter`
pipelineCache must be a valid VkPipelineCache handle

- VUID-vkGetPipelineCacheData-pDataSize-parameter
  pDataSize must be a valid pointer to a size_t value

- VUID-vkGetPipelineCacheData-pData-parameter
  If the value referenced by pDataSize is not 0, and pData is not NULL, pData must be a valid pointer to an array of pDataSize bytes

- VUID-vkGetPipelineCacheData-pipelineCache-parent
  pipelineCache must have been created, allocated, or retrieved from device

Return Codes

Success
- VK_SUCCESS
- VK_INCOMPLETE

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

10.6.4. Pipeline Cache Header

Applications can store the data retrieved from the pipeline cache, and use these data, possibly in a future run of the application, to populate new pipeline cache objects. The results of pipeline compiles, however, may depend on the vendor ID, device ID, driver version, and other details of the device. To enable applications to detect when previously retrieved data is incompatible with the device, the pipeline cache data must begin with a valid pipeline cache header.

Version one of the pipeline cache header is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineCacheHeaderVersionOne {
    uint32_t headerSize;
    VkPipelineCacheHeaderVersion headerVersion;
    uint32_t vendorID;
    uint32_t deviceID;
    uint8_t pipelineCacheUUID[VK_UUID_SIZE];
} VkPipelineCacheHeaderVersionOne;
```

- headerSize is the length in bytes of the pipeline cache header.
- headerVersion is a VkPipelineCacheHeaderVersion enum value specifying the version of the header. A consumer of the pipeline cache should use the cache version to interpret the remainder of the cache header.
- vendorID is the VkPhysicalDeviceProperties::vendorID of the implementation.
• deviceID is the VkPhysicalDeviceProperties::deviceID of the implementation.
• pipelineCacheUUID is the VkPhysicalDeviceProperties::pipelineCacheUUID of the implementation.

Unlike most structures declared by the Vulkan API, all fields of this structure are written with the least significant byte first, regardless of host byte-order.

The C language specification does not define the packing of structure members. This layout assumes tight structure member packing, with members laid out in the order listed in the structure, and the intended size of the structure is 32 bytes. If a compiler produces code that diverges from that pattern, applications must employ another method to set values at the correct offsets.

### Valid Usage

- VUID-VkPipelineCacheHeaderVersionOne-headerSize-04967
  `headerSize must be 32`
- VUID-VkPipelineCacheHeaderVersionOne-headerVersion-04968
  `headerVersion must be VK_PIPELINE_CACHE_HEADER_VERSION_ONE`

### Valid Usage (Implicit)

- VUID-VkPipelineCacheHeaderVersionOne-headerVersion-parameter
  `headerVersion must be a valid VkPipelineCacheHeaderVersion value`

Possible values of the `headerVersion` value of the pipeline cache header are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkPipelineCacheHeaderVersion {
    VK_PIPELINE_CACHE_HEADER_VERSION_ONE = 1,
} VkPipelineCacheHeaderVersion;
```

- `VK_PIPELINE_CACHE_HEADER_VERSION_ONE` specifies version one of the pipeline cache.

### 10.6.5. Destroying a Pipeline Cache

To destroy a pipeline cache, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyPipelineCache(  
    VkDevice device,  
    VkPipelineCache pipelineCache,  
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the pipeline cache object.
- `pipelineCache` is the handle of the pipeline cache to destroy.
• pAllocator controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage

- **VUID-vkDestroyPipelineCache-pipelineCache-00771**
  If VkAllocationCallbacks were provided when pipelineCache was created, a compatible set of callbacks **must** be provided here.

- **VUID-vkDestroyPipelineCache-pipelineCache-00772**
  If no VkAllocationCallbacks were provided when pipelineCache was created, pAllocator **must** be NULL.

### Valid Usage (Implicit)

- **VUID-vkDestroyPipelineCache-device-parameter**
  device **must** be a valid VkDevice handle.

- **VUID-vkDestroyPipelineCache-pipelineCache-parameter**
  If pipelineCache is not VK_NULL_HANDLE, pipelineCache **must** be a valid VkPipelineCache handle.

- **VUID-vkDestroyPipelineCache-pAllocator-parameter**
  If pAllocator is not NULL, pAllocator **must** be a valid pointer to a valid VkAllocationCallbacks structure.

- **VUID-vkDestroyPipelineCache-pipelineCache-parent**
  If pipelineCache is a valid handle, it **must** have been created, allocated, or retrieved from device.

### Host Synchronization

- Host access to pipelineCache **must** be externally synchronized.

## 10.7. Specialization Constants

Specialization constants are a mechanism whereby constants in a SPIR-V module **can** have their constant value specified at the time the VkPipeline is created. This allows a SPIR-V module to have constants that **can** be modified while executing an application that uses the Vulkan API.

**Note**

Specialization constants are useful to allow a compute shader to have its local workgroup size changed at runtime by the user, for example.

Each VkPipelineShaderStageCreateInfo structure contains a pSpecializationInfo member, which **can** be NULL to indicate no specialization constants, or point to a VkSpecializationInfo structure.
The `VkSpecializationInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSpecializationInfo {
    uint32_t mapEntryCount;
    const VkSpecializationMapEntry* pMapEntries;
    size_t dataSize;
    const void* pData;
} VkSpecializationInfo;
```

- `mapEntryCount` is the number of entries in the `pMapEntries` array.
- `pMapEntries` is a pointer to an array of `VkSpecializationMapEntry` structures, which map constant IDs to offsets in `pData`.
- `dataSize` is the byte size of the `pData` buffer.
- `pData` contains the actual constant values to specialize with.

### Valid Usage

- **VUID-VkSpecializationInfo-offset-00773**
  The `offset` member of each element of `pMapEntries` must be less than `dataSize`

- **VUID-VkSpecializationInfo-pMapEntries-00774**
  The `size` member of each element of `pMapEntries` must be less than or equal to `dataSize` minus `offset`

- **VUID-VkSpecializationInfo-constantID-04911**
  The `constantID` value of each element of `pMapEntries` must be unique within `pMapEntries`

### Valid Usage (Implicit)

- **VUID-VkSpecializationInfo-pMapEntries-parameter**
  If `mapEntryCount` is not 0, `pMapEntries` must be a valid pointer to an array of `mapEntryCount` valid `VkSpecializationMapEntry` structures

- **VUID-VkSpecializationInfo-pData-parameter**
  If `dataSize` is not 0, `pData` must be a valid pointer to an array of `dataSize` bytes

The `VkSpecializationMapEntry` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSpecializationMapEntry {
    uint32_t constantID;
    uint32_t offset;
    size_t size;
} VkSpecializationMapEntry;
```
• **constantID** is the ID of the specialization constant in SPIR-V.
• **offset** is the byte offset of the specialization constant value within the supplied data buffer.
• **size** is the byte size of the specialization constant value within the supplied data buffer.

If a **constantID** value is not a specialization constant ID used in the shader, that map entry does not affect the behavior of the pipeline.

### Valid Usage

- VUID-VkSpecializationMapEntry-constantID-00776
  For a **constantID** specialization constant declared in a shader, **size** must match the byte size of the **constantID**. If the specialization constant is of type **boolean**, **size** must be the byte size of **VkBool32**

In human readable SPIR-V:

```
OpDecorate %x SpecId 13;  // constantID
OpDecorate %y SpecId 42;  // offset
OpDecorate %z SpecId 3;   // size
OpDecorate %workgroupsize BuiltIn WorkgroupSize;  // match WorkgroupSize

%i32 = OpTypeInt 32 0;  // declare an unsigned 32-bit type
%uvec3 = OpTypeVector %i32 3;  // declare a 3 element vector type of unsigned 32-bit
%x = OpSpecConstant %i32 1;  // declare the .x component of WorkgroupSize
%y = OpSpecConstant %i32 1;  // declare the .y component of WorkgroupSize
%z = OpSpecConstant %i32 1;  // declare the .z component of WorkgroupSize
%workgroupsize = OpSpecConstantComposite %uvec3 %x %y %z;  // declare WorkgroupSize

From the above we have three specialization constants, one for each of the x, y & z elements of the WorkgroupSize vector.

Now to specialize the above via the specialization constants mechanism:

```c
const VkSpecializationMapEntry entries[] =
{
    {
        13, // constantID
        0 * sizeof(uint32_t),  // offset
        sizeof(uint32_t)  // size
    },
    {
        42, // constantID
        1 * sizeof(uint32_t),  // offset
        sizeof(uint32_t)  // size
    },
    {
        3, // constantID
        2 * sizeof(uint32_t),  // offset
    }
};
```
sizeof(uint32_t) // size
}

const uint32_t data[] = { 16, 8, 4 }; // our workgroup size is 16x8x4

const VkSpecializationInfo info =
{
    3, // mapEntryCount
    entries, // pMapEntries
    3 * sizeof(uint32_t), // dataSize
    data, // pData
};

Then when calling vkCreateComputePipelines, and passing the VkSpecializationInfo we defined as
the pSpecializationInfo parameter of VkPipelineShaderStageCreateInfo, we will create a compute
pipeline with the runtime specified local workgroup size.

Another example would be that an application has a SPIR-V module that has some platform-
dependent constants they wish to use.

In human readable SPIR-V:

OpDecorate %1 SpecId 0 ; decorate our signed 32-bit integer constant
OpDecorate %2 SpecId 12 ; decorate our 32-bit floating-point constant
%i32 = OpTypeInt 32 1 ; declare a signed 32-bit type
%float = OpTypeFloat 32 ; declare a 32-bit floating-point type
%i1 = OpSpecConstant %i32 -1 ; some signed 32-bit integer constant
%i2 = OpSpecConstant %float 0.5 ; some 32-bit floating-point constant

From the above we have two specialization constants, one is a signed 32-bit integer and the second
is a 32-bit floating-point value.

Now to specialize the above via the specialization constants mechanism:

struct SpecializationData {
    int32_t data0;
    float data1;
};

const VkSpecializationMapEntry entries[] =
{
    { 0, // constantID
        offsetof(SpecializationData, data0), // offset
        sizeof(SpecializationData::data0) // size
    },
    { 12, // constantID
It is legal for a SPIR-V module with specializations to be compiled into a pipeline where no specialization information was provided. SPIR-V specialization constants contain default values such that if a specialization is not provided, the default value will be used. In the examples above, it would be valid for an application to only specialize some of the specialization constants within the SPIR-V module, and let the other constants use their default values encoded within the OpSpecConstant declarations.

10.8. Pipeline Binding

Once a pipeline has been created, it can be bound to the command buffer using the command:

```c
// Provided by VK_VERSION_1_0
void vkCmdBindPipeline(
  VkCommandBuffer commandBuffer,
  VkPipelineBindPoint pipelineBindPoint,
  VkPipeline pipeline);
```

- `commandBuffer` is the command buffer that the pipeline will be bound to.
- `pipelineBindPoint` is a `VkPipelineBindPoint` value specifying to which bind point the pipeline is bound. Binding one does not disturb the others.
- `pipeline` is the pipeline to be bound.

Once bound, a pipeline binding affects subsequent commands that interact with the given pipeline type in the command buffer until a different pipeline of the same type is bound to the bind point. Commands that do not interact with the given pipeline type must not be affected by the pipeline state.

- The pipeline bound to `VK_PIPELINE_BIND_POINT_COMPUTE` controls the behavior of all dispatching commands.
- The pipeline bound to `VK_PIPELINE_BIND_POINT_GRAPHICS` controls the behavior of all drawing
Valid Usage

- **VUID-vkCmdBindPipeline-pipelineBindPoint-00777**
  If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_COMPUTE`, the `VkCommandPool` that `commandBuffer` was allocated from must support compute operations.

- **VUID-vkCmdBindPipeline-pipelineBindPoint-00778**
  If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_GRAPHICS`, the `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations.

- **VUID-vkCmdBindPipeline-pipelineBindPoint-00779**
  If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_COMPUTE`, pipeline must be a compute pipeline.

- **VUID-vkCmdBindPipeline-pipelineBindPoint-00780**
  If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_GRAPHICS`, pipeline must be a graphics pipeline.

- **VUID-vkCmdBindPipeline-pipeline-00781**
  If the `variableMultisampleRate` feature is not supported, pipeline is a graphics pipeline, the current subpass uses no attachments, and this is not the first call to this function with a graphics pipeline after transitioning to the current subpass, then the sample count specified by this pipeline must match that set in the previous pipeline.

Valid Usage (Implicit)

- **VUID-vkCmdBindPipeline-commandBuffer-parameter**
  `commandBuffer` must be a valid `VkCommandBuffer` handle.

- **VUID-vkCmdBindPipeline-pipelineBindPoint-parameter**
  `pipelineBindPoint` must be a valid `VkPipelineBindPoint` value.

- **VUID-vkCmdBindPipeline-pipeline-parameter**
  `pipeline` must be a valid `VkPipeline` handle.

- **VUID-vkCmdBindPipeline-commandBuffer-recording**
  `commandBuffer` must be in the recording state.

- **VUID-vkCmdBindPipeline-commandBuffer-cmdpool**
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations.

- **VUID-vkCmdBindPipeline-commonparent**
  Both of `commandBuffer`, and `pipeline` must have been created, allocated, or retrieved from the same `VkDevice`.

Host Synchronization

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

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

Possible values of `vkCmdBindPipeline::pipelineBindPoint`, specifying the bind point of a pipeline object, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkPipelineBindPoint {
  VK_PIPELINE_BIND_POINT_GRAPHICS = 0,
  VK_PIPELINE_BIND_POINT_COMPUTE = 1,
} VkPipelineBindPoint;
```

- **VK_PIPELINE_BIND_POINT_COMPUTE** specifies binding as a compute pipeline.
- **VK_PIPELINE_BIND_POINT_GRAPHICS** specifies binding as a graphics pipeline.

### 10.9. Dynamic State

When a pipeline object is bound, any pipeline object state that is not specified as dynamic is applied to the command buffer state. Pipeline object state that is specified as dynamic is not applied to the command buffer state at this time. Instead, dynamic state can be modified at any time and persists for the lifetime of the command buffer, or until modified by another dynamic state setting command, or made invalid by another pipeline bind with that state specified as static.

When a pipeline object is bound, the following applies to each state parameter:

- If the state is not specified as dynamic in the new pipeline object, then that command buffer state is overwritten by the state in the new pipeline object. Before any draw or dispatch call with this pipeline there must not have been any calls to any of the corresponding dynamic state setting commands after this pipeline was bound.
- If the state is specified as dynamic in the new pipeline object, then that command buffer state is not disturbed. Before any draw or dispatch call with this pipeline there must have been at least one call to each of the corresponding dynamic state setting commands. The state-setting commands must be recorded after command buffer recording was begun, or after the last command binding a pipeline object with that state specified as static, whichever was the latter.
- If the state is not included (corresponding pointer in VkGraphicsPipelineCreateInfo was NULL or was ignored) in the new pipeline object, then that command buffer state is not disturbed.
Dynamic state that does not affect the result of operations can be left undefined.

**Note**

For example, if blending is disabled by the pipeline object state then the dynamic color blend constants do not need to be specified in the command buffer, even if this state is specified as dynamic in the pipeline object.

### 10.10. Pipeline Creation Feedback

Feedback about the creation of a particular pipeline object can be obtained by adding a `VkPipelineCreationFeedbackCreateInfo` structure to the `pNext` chain of `VkGraphicsPipelineCreateInfo`, or `VkComputePipelineCreateInfo`. The `VkPipelineCreationFeedbackCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPipelineCreationFeedbackCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineCreationFeedback* pPipelineCreationFeedback;
    uint32_t pipelineStageCreationFeedbackCount;
    VkPipelineCreationFeedback* pPipelineStageCreationFeedbacks;
} VkPipelineCreationFeedbackCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `pPipelineCreationFeedback` is a pointer to a `VkPipelineCreationFeedback` structure.
- `pipelineStageCreationFeedbackCount` is the number of elements in `pPipelineStageCreationFeedbacks`.
- `pPipelineStageCreationFeedbacks` is a pointer to an array of `pipelineStageCreationFeedbackCount` `VkPipelineCreationFeedback` structures.

An implementation should write pipeline creation feedback to `pPipelineCreationFeedback` and may write pipeline stage creation feedback to `pPipelineStageCreationFeedbacks`. An implementation must set or clear the `VK_PIPELINE_CREATION_FEEDBACK_VALID_BIT` in `VkPipelineCreationFeedback::flags` for `pPipelineCreationFeedback` and every element of `pPipelineStageCreationFeedbacks`.

**Note**

One common scenario for an implementation to skip per-stage feedback is when `VK_PIPELINE_CREATION_FEEDBACK_APPLICATION_PIPELINE_CACHE_HIT_BIT` is set in `pPipelineCreationFeedback`.

When chained to `VkGraphicsPipelineCreateInfo`, the `i` element of `pPipelineStageCreationFeedbacks` corresponds to the `i` element of `VkGraphicsPipelineCreateInfo::pStages`. When chained to `VkComputePipelineCreateInfo`, the first element of `pPipelineStageCreationFeedbacks` corresponds to `VkComputePipelineCreateInfo::stage`. 

426
Valid Usage (Implicit)

- VUID-VkPipelineCreationFeedbackCreateInfo-sType-sType
  *sType* must be *VK_STRUCTURE_TYPE_PIPELINE_CREATION_FEEDBACK_CREATE_INFO*

- VUID-VkPipelineCreationFeedbackCreateInfo-pPipelineCreationFeedback-parameter
  *pPipelineCreationFeedback* must be a valid pointer to a *VkPipelineCreationFeedback* structure

- VUID-VkPipelineCreationFeedbackCreateInfo-pPipelineStageCreationFeedbacks-parameter
  If *pipelineStageCreationFeedbackCount* is not 0, *pPipelineStageCreationFeedbacks* must be a valid pointer to an array of *pipelineStageCreationFeedbackCount* *VkPipelineCreationFeedback* structures

The *VkPipelineCreationFeedback* structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPipelineCreationFeedback {
    VkPipelineCreationFeedbackFlags flags;
    uint64_t duration;
} VkPipelineCreationFeedback;
```

- *flags* is a bitmask of *VkPipelineCreationFeedbackFlagBits* providing feedback about the creation of a pipeline or of a pipeline stage.
- *duration* is the duration spent creating a pipeline or pipeline stage in nanoseconds.

If the *VK_PIPELINE_CREATION_FEEDBACK_VALID_BIT* is not set in *flags*, an implementation must not set any other bits in *flags*, and the values of all other *VkPipelineCreationFeedback* data members are undefined.

Possible values of the *flags* member of *VkPipelineCreationFeedback* are:

```c
// Provided by VK_VERSION_1_3
typedef enum VkPipelineCreationFeedbackFlagBits {
    VK_PIPELINE_CREATION_FEEDBACK_VALID_BIT = 0x00000001,
    VK_PIPELINE_CREATION_FEEDBACK_APPLICATION_PIPELINE_CACHE_HIT_BIT = 0x00000002,
    VK_PIPELINE_CREATION_FEEDBACK_BASE_PIPELINE_ACCELERATION_BIT = 0x00000004,
    VK_PIPELINE_CREATION_FEEDBACK_VALID_BIT_EXT = VK_PIPELINE_CREATION_FEEDBACK_VALID_BIT,
    VK_PIPELINE_CREATION_FEEDBACK_APPLICATION_PIPELINE_CACHE_HIT_BIT_EXT =
        VK_PIPELINE_CREATION_FEEDBACK_APPLICATION_PIPELINE_CACHE_HIT_BIT,
    VK_PIPELINE_CREATION_FEEDBACK_BASE_PIPELINE_ACCELERATION_BIT_EXT =
        VK_PIPELINE_CREATION_FEEDBACK_BASE_PIPELINE_ACCELERATION_BIT,
} VkPipelineCreationFeedbackFlagBits;
```

- *VK_PIPELINE_CREATION_FEEDBACK_VALID_BIT* indicates that the feedback information is valid.
• **VK_PIPELINE_CREATION_FEEDBACK_APPLICATION_PIPELINE_CACHE_HIT_BIT** indicates that a readily usable pipeline or pipeline stage was found in the pipelineCache specified by the application in the pipeline creation command.

An implementation **should** set the **VK_PIPELINE_CREATION_FEEDBACK_APPLICATION_PIPELINE_CACHE_HIT_BIT** bit if it was able to avoid the large majority of pipeline or pipeline stage creation work by using the pipelineCache parameter of vkCreateGraphicsPipelines, or vkCreateComputePipelines. When an implementation sets this bit for the entire pipeline, it **may** leave it unset for any stage.

**Note**
Implementations are encouraged to provide a meaningful signal to applications using this bit. The intention is to communicate to the application that the pipeline or pipeline stage was created “as fast as it gets” using the pipeline cache provided by the application. If an implementation uses an internal cache, it is discouraged from setting this bit as the feedback would be unactionable.

• **VK_PIPELINE_CREATION_FEEDBACK_BASE_PIPELINE_ACCELERATION_BIT** indicates that the base pipeline specified by the basePipelineHandle or basePipelineIndex member of the Vk*PipelineCreateInfo structure was used to accelerate the creation of the pipeline.

An implementation **should** set the **VK_PIPELINE_CREATION_FEEDBACK_BASE_PIPELINE_ACCELERATION_BIT** bit if it was able to avoid a significant amount of work by using the base pipeline.

**Note**
While “significant amount of work” is subjective, implementations are encouraged to provide a meaningful signal to applications using this bit. For example, a 1% reduction in duration may not warrant setting this bit, while a 50% reduction would.

```c
// Provided by VK_VERSION_1_3
typedef VkFlags VkPipelineCreationFeedbackFlags;
```

**VkPipelineCreationFeedbackFlags** is a bitmask type for providing zero or more **VkPipelineCreationFeedbackFlagBits**.
Chapter 11. Memory Allocation

Vulkan memory is broken up into two categories, host memory and device memory.

11.1. Host Memory

Host memory is memory needed by the Vulkan implementation for non-device-visible storage.

Note

This memory may be used to store the implementation’s representation and state of Vulkan objects.

Vulkan provides applications the opportunity to perform host memory allocations on behalf of the Vulkan implementation. If this feature is not used, the implementation will perform its own memory allocations. Since most memory allocations are off the critical path, this is not meant as a performance feature. Rather, this can be useful for certain embedded systems, for debugging purposes (e.g. putting a guard page after all host allocations), or for memory allocation logging.

 Allocators are provided by the application as a pointer to a VkAllocationCallbacks structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkAllocationCallbacks {
    void* pUserData;
    PFN_vkAllocationFunction pfnAllocation;
    PFN_vkReallocatorFunction pfnReallocator;
    PFN_vkFreeFunction pfnFree;
    PFN_vkInternalAllocationNotification pfnInternalAllocation;
    PFN_vkInternalFreeNotification pfnInternalFree;
} VkAllocationCallbacks;
```

- `pUserData` is a value to be interpreted by the implementation of the callbacks. When any of the callbacks in VkAllocationCallbacks are called, the Vulkan implementation will pass this value as the first parameter to the callback. This value can vary each time an allocator is passed into a command, even when the same object takes an allocator in multiple commands.

- `pfnAllocation` is a PFN_vkAllocationFunction pointer to an application-defined memory allocation function.

- `pfnReallocator` is a PFN_vkReallocatorFunction pointer to an application-defined memory reallocation function.

- `pfnFree` is a PFN_vkFreeFunction pointer to an application-defined memory free function.

- `pfnInternalAllocation` is a PFN_vkInternalAllocationNotification pointer to an application-defined function that is called by the implementation when the implementation makes internal allocations.

- `pfnInternalFree` is a PFN_vkInternalFreeNotification pointer to an application-defined function that is called by the implementation when the implementation frees internal allocations.
Valid Usage

- VUID-VkAllocationCallbacks-pfnAllocation-00632
  `pfnAllocation` must be a valid pointer to a valid user-defined `PFN_vkAllocationFunction`.

- VUID-VkAllocationCallbacks-pfnReallocation-00633
  `pfnReallocation` must be a valid pointer to a valid user-defined `PFN_vkReallocationFunction`.

- VUID-VkAllocationCallbacks-pfnFree-00634
  `pfnFree` must be a valid pointer to a valid user-defined `PFN_vkFreeFunction`.

- VUID-VkAllocationCallbacks-pfnInternalAllocation-00635
  If either of `pfnInternalAllocation` or `pfnInternalFree` is not NULL, both must be valid callbacks.

The type of `pfnAllocation` is:

```c
// Provided by VK_VERSION_1_0
typedef void* (VKAPI_PTR *PFN_vkAllocationFunction)(
    void* pUserData,
    size_t size,
    size_t alignment,
    VkSystemAllocationScope allocationScope);
```

- `pUserData` is the value specified for `VkAllocationCallbacks::pUserData` in the allocator specified by the application.

- `size` is the size in bytes of the requested allocation.

- `alignment` is the requested alignment of the allocation in bytes and must be a power of two.

- `allocationScope` is a `VkSystemAllocationScope` value specifying the allocation scope of the lifetime of the allocation, as described here.

If `pfnAllocation` is unable to allocate the requested memory, it must return NULL. If the allocation was successful, it must return a valid pointer to memory allocation containing at least `size` bytes, and with the pointer value being a multiple of `alignment`.

**Note**

Correct Vulkan operation cannot be assumed if the application does not follow these rules.

For example, `pfnAllocation` (or `pfnReallocator`) could cause termination of running Vulkan instance(s) on a failed allocation for debugging purposes, either directly or indirectly. In these circumstances, it cannot be assumed that any part of any affected `VkInstance` objects are going to operate correctly (even `vkDestroyInstance`), and the application must ensure it cleans up properly via other means (e.g. process termination).
If \texttt{pfnAllocation} returns \texttt{NULL}, and if the implementation is unable to continue correct processing of the current command without the requested allocation, it \textbf{must} treat this as a runtime error, and generate \texttt{VK\_ERROR\_OUT\_OF\_HOST\_MEMORY} at the appropriate time for the command in which the condition was detected, as described in \textit{Return Codes}.

If the implementation is able to continue correct processing of the current command without the requested allocation, then it \textbf{may} do so, and \textbf{must} not generate \texttt{VK\_ERROR\_OUT\_OF\_HOST\_MEMORY} as a result of this failed allocation.

The type of \texttt{pfnReallocation} is:

```c
// Provided by VK\_VERSION\_1\_0
typedef void* (VKAPI_PTR *PFN_vkReallocationFunction)(
    void* pUserData,
    void* pOriginal,
    size_t size,
    size_t alignment,
    VkSystemAllocationScope allocationScope);
```

- \texttt{pUserData} is the value specified for \texttt{VkAllocationCallbacks::pUserData} in the allocator specified by the application.
- \texttt{pOriginal} \textbf{must} be either \texttt{NULL} or a pointer previously returned by \texttt{pfnReallocation} or \texttt{pfnAllocation} of a compatible allocator.
- \texttt{size} is the size in bytes of the requested allocation.
- \texttt{alignment} is the requested alignment of the allocation in bytes and \textbf{must} be a power of two.
- \texttt{allocationScope} is a \texttt{VkSystemAllocationScope} value specifying the allocation scope of the lifetime of the allocation, as described \textit{here}.

\texttt{pfnReallocation} \textbf{must} return an allocation with enough space for \texttt{size} bytes, and the contents of the original allocation from bytes zero to min(original size, new size) - 1 \textbf{must} be preserved in the returned allocation. If \texttt{size} is larger than the old size, the contents of the additional space are undefined. If satisfying these requirements involves creating a new allocation, then the old allocation \textbf{should} be freed.

If \texttt{pOriginal} is \texttt{NULL}, then \texttt{pfnReallocation} \textbf{must} behave equivalently to a call to \texttt{PFN\_vkAllocationFunction} with the same parameter values (without \texttt{pOriginal}).

If \texttt{size} is zero, then \texttt{pfnReallocation} \textbf{must} behave equivalently to a call to \texttt{PFN\_vkFreeFunction} with the same \texttt{pUserData} parameter value, and \texttt{pMemory} equal to \texttt{pOriginal}.

If \texttt{pOriginal} is non-\texttt{NULL}, the implementation \textbf{must} ensure that \texttt{alignment} is equal to the \texttt{alignment} used to originally allocate \texttt{pOriginal}.

If this function fails and \texttt{pOriginal} is non-\texttt{NULL} the application \textbf{must} not free the old allocation.

\texttt{pfnReallocation} \textbf{must} follow the same \textit{rules for return values} as \texttt{PFN\_vkAllocationFunction}.

The type of \texttt{pfnFree} is:
typedef void (VKAPI_PTR *PFN_vkFreeFunction)(
    void* pUserData,
    void* pMemory);

- **pUserData** is the value specified for `VkAllocationCallbacks::pUserData` in the allocator specified by the application.
- **pMemory** is the allocation to be freed.

*pMemory* may be **NULL**, which the callback **must** handle safely. If *pMemory* is non-**NULL**, it **must** be a pointer previously allocated by `PFN_allocation` or `PFN_reallocation`. The application **should** free this memory.

The type of `PFNInternalAllocation` is:

typedef void (VKAPI_PTR *PFN_vkInternalAllocationNotification)(
    void* pUserData,
    size_t size,
    VkInternalAllocationType allocationType,
    VkSystemAllocationScope allocationScope);

- **pUserData** is the value specified for `VkAllocationCallbacks::pUserData` in the allocator specified by the application.
- **size** is the requested size of an allocation.
- **allocationType** is a `VkInternalAllocationType` value specifying the requested type of an allocation.
- **allocationScope** is a `VkSystemAllocationScope` value specifying the allocation scope of the lifetime of the allocation, as described [here](#).

This is a purely informational callback.

The type of `PFNInternalFree` is:

typedef void (VKAPI_PTR *PFN_vkInternalFreeNotification)(
    void* pUserData,
    size_t size,
    VkInternalAllocationType allocationType,
    VkSystemAllocationScope allocationScope);

- **pUserData** is the value specified for `VkAllocationCallbacks::pUserData` in the allocator specified by the application.
- **size** is the requested size of an allocation.
• **allocationType** is a `VkInternalAllocationType` value specifying the requested type of an allocation.

• **allocationScope** is a `VkSystemAllocationScope` value specifying the allocation scope of the lifetime of the allocation, as described here.

Each allocation has an *allocation scope* defining its lifetime and which object it is associated with. Possible values passed to the `allocationScope` parameter of the callback functions specified by `VkAllocationCallbacks`, indicating the allocation scope, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkSystemAllocationScope {
    VK_SYSTEM_ALLOCATION_SCOPE_COMMAND = 0,
    VK_SYSTEM_ALLOCATION_SCOPE_OBJECT = 1,
    VK_SYSTEM_ALLOCATION_SCOPE_CACHE = 2,
    VK_SYSTEM_ALLOCATION_SCOPE_DEVICE = 3,
    VK_SYSTEM_ALLOCATION_SCOPE_INSTANCE = 4,
} VkSystemAllocationScope;
```

• **VK_SYSTEM_ALLOCATION_SCOPE_COMMAND** specifies that the allocation is scoped to the duration of the Vulkan command.

• **VK_SYSTEM_ALLOCATION_SCOPE_OBJECT** specifies that the allocation is scoped to the lifetime of the Vulkan object that is being created or used.

• **VK_SYSTEM_ALLOCATION_SCOPE_CACHE** specifies that the allocation is scoped to the lifetime of a `VkPipelineCache` object.

• **VK_SYSTEM_ALLOCATION_SCOPE_DEVICE** specifies that the allocation is scoped to the lifetime of the Vulkan device.

• **VK_SYSTEM_ALLOCATION_SCOPE_INSTANCE** specifies that the allocation is scoped to the lifetime of the Vulkan instance.

Most Vulkan commands operate on a single object, or there is a sole object that is being created or manipulated. When an allocation uses an allocation scope of `VK_SYSTEM_ALLOCATION_SCOPE_OBJECT` or `VK_SYSTEM_ALLOCATION_SCOPE_CACHE`, the allocation is scoped to the object being created or manipulated.

When an implementation requires host memory, it will make callbacks to the application using the most specific allocator and allocation scope available:

• If an allocation is scoped to the duration of a command, the allocator will use the `VK_SYSTEM_ALLOCATION_SCOPE_COMMAND` allocation scope. The most specific allocator available is used: if the object being created or manipulated has an allocator, that object's allocator will be used, else if the parent `VkDevice` has an allocator it will be used, else if the parent `VkInstance` has an allocator it will be used. Else,

• If an allocation is associated with a `VkPipelineCache` object, the allocator will use the `VK_SYSTEM_ALLOCATION_SCOPE_CACHE` allocation scope. The most specific allocator available is used (cache, else device, else instance). Else,
• If an allocation is scoped to the lifetime of an object, that object is being created or manipulated by the command, and that object's type is not VkDevice or VkInstance, the allocator will use an allocation scope of VK_SYSTEM_ALLOCATION_SCOPE_OBJECT. The most specific allocator available is used (object, else device, else instance). Else,

• If an allocation is scoped to the lifetime of a device, the allocator will use an allocation scope of VK_SYSTEM_ALLOCATION_SCOPE_DEVICE. The most specific allocator available is used (device, else instance). Else,

• If the allocation is scoped to the lifetime of an instance and the instance has an allocator, its allocator will be used with an allocation scope of VK_SYSTEM_ALLOCATION_SCOPE_INSTANCE.

• Otherwise an implementation will allocate memory through an alternative mechanism that is unspecified.

Objects that are allocated from pools do not specify their own allocator. When an implementation requires host memory for such an object, that memory is sourced from the object's parent pool’s allocator.

The application is not expected to handle allocating memory that is intended for execution by the host due to the complexities of differing security implementations across multiple platforms. The implementation will allocate such memory internally and invoke an application provided informational callback when these internal allocations are allocated and freed. Upon allocation of executable memory, pfnInternalAllocation will be called. Upon freeing executable memory, pfnInternalFree will be called. An implementation will only call an informational callback for executable memory allocations and frees.

The allocationType parameter to the pfnInternalAllocation and pfnInternalFree functions may be one of the following values:

```c
// Provided by VK_VERSION_1_0
typedef enum VkInternalAllocationType {
    VK_INTERNAL_ALLOCATION_TYPE_EXECUTABLE = 0,
} VkInternalAllocationType;
```

• VK_INTERNAL_ALLOCATION_TYPE_EXECUTABLE specifies that the allocation is intended for execution by the host.

An implementation must only make calls into an application-provided allocator during the execution of an API command. An implementation must only make calls into an application-provided allocator from the same thread that called the provoking API command. The implementation should not synchronize calls to any of the callbacks. If synchronization is needed, the callbacks must provide it themselves. The informational callbacks are subject to the same restrictions as the allocation callbacks.

If an implementation intends to make calls through a VkAllocationCallbacks structure between the time a vkCreate* command returns and the time a corresponding vkDestroy* command begins, that implementation must save a copy of the allocator before the vkCreate* command returns. The callback functions and any data structures they rely upon must remain valid for the lifetime of the object they are associated with.
If an allocator is provided to a `vkCreate*` command, a *compatible* allocator **must** be provided to the corresponding `vkDestroy*` command. Two `VkAllocationCallbacks` structures are compatible if memory allocated with `pfnAllocation` or `pfnReallocation` in each can be freed with `pfnReallocation` or `pfnFree` in the other. An allocator **must** not be provided to a `vkDestroy*` command if an allocator was not provided to the corresponding `vkCreate*` command.

If a non-NULL allocator is used, the `pfnAllocation`, `pfnReallocation` and `pfnFree` members **must** be non-NULL and point to valid implementations of the callbacks. An application **can** choose to not provide informational callbacks by setting both `pfnInternalAllocation` and `pfnInternalFree` to NULL. `pfnInternalAllocation` and `pfnInternalFree` **must** either both be NULL or both be non-NULL.

If `pfnAllocation` or `pfnReallocation` fail, the implementation **may** fail object creation and/or generate a `VK_ERROR_OUT_OF_HOST_MEMORY` error, as appropriate.

Allocation callbacks **must** not call any Vulkan commands.

The following sets of rules define when an implementation is permitted to call the allocator callbacks.

`pfnAllocation` or `pfnReallocation` **may** be called in the following situations:

- Allocations scoped to a `VkDevice` or `VkInstance` **may** be allocated from any API command.
- Allocations scoped to a command **may** be allocated from any API command.
- Allocations scoped to a `VkPipelineCache` **may** only be allocated from:
  - `vkCreatePipelineCache`
  - `vkMergePipelineCaches` for `dstCache`
  - `vkCreateGraphicsPipelines` for `pipelineCache`
  - `vkCreateComputePipelines` for `pipelineCache`
- Allocations scoped to a `VkDescriptorPool` **may** only be allocated from:
  - any command that takes the pool as a direct argument
  - `vkAllocateDescriptorSets` for the `descriptorPool` member of its `pAllocateInfo` parameter
  - `vkCreateDescriptorPool`
- Allocations scoped to a `VkCommandPool` **may** only be allocated from:
  - any command that takes the pool as a direct argument
  - `vkCreateCommandPool`
  - `vkAllocateCommandBuffers` for the `commandPool` member of its `pAllocateInfo` parameter
  - any `vkCmd*` command whose `commandBuffer` was allocated from that `VkCommandPool`
- Allocations scoped to any other object **may** only be allocated in that object’s `vkCreate*` command.

`pfnFree`, or `pfnReallocation` with zero size, **may** be called in the following situations:

- Allocations scoped to a `VkDevice` or `VkInstance` **may** be freed from any API command.
• Allocations scoped to a command **must** be freed by any API command which allocates such memory.

• Allocations scoped to a `VkPipelineCache` **may** be freed from `vkDestroyPipelineCache`.

• Allocations scoped to a `VkDescriptorPool` **may** be freed from:
  ◦ any command that takes the pool as a direct argument

• Allocations scoped to a `VkCommandPool` **may** be freed from:
  ◦ any command that takes the pool as a direct argument
  ◦ `vkResetCommandBuffer` whose `commandBuffer` was allocated from that `VkCommandPool`

• Allocations scoped to any other object **may** be freed in that object’s `vkDestroy*` command.

• Any command that allocates host memory **may** also free host memory of the same scope.

### 11.2. Device Memory

*Device memory* is memory that is visible to the device — for example the contents of the image or buffer objects, which can be natively used by the device.

#### 11.2.1. Device Memory Properties

Memory properties of a physical device describe the memory heaps and memory types available.

To query memory properties, call:

```c
// Provided by VK_VERSION_1_0
void vkGetPhysicalDeviceMemoryProperties(
    VkPhysicalDevice physicalDevice,
    VkPhysicalDeviceMemoryProperties* pMemoryProperties);
```

• `physicalDevice` is the handle to the device to query.

• `pMemoryProperties` is a pointer to a `VkPhysicalDeviceMemoryProperties` structure in which the properties are returned.

#### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceMemoryProperties-physicalDevice-parameter `physicalDevice` **must** be a valid `VkPhysicalDevice` handle

- VUID-vkGetPhysicalDeviceMemoryProperties-pMemoryProperties-parameter `pMemoryProperties` **must** be a valid pointer to a `VkPhysicalDeviceMemoryProperties` structure

The `VkPhysicalDeviceMemoryProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_0
```
typedef struct VkPhysicalDeviceMemoryProperties {
    uint32_t memoryTypeCount;
    VkMemoryType memoryTypes[VK_MAX_MEMORY_TYPES];
    uint32_t memoryHeapCount;
    VkMemoryHeap memoryHeaps[VK_MAX_MEMORY_HEAPS];
} VkPhysicalDeviceMemoryProperties;

- `memoryTypeCount` is the number of valid elements in the `memoryTypes` array.
- `memoryTypes` is an array of `VK_MAX_MEMORY_TYPES` `VkMemoryType` structures describing the memory types that can be used to access memory allocated from the heaps specified by `memoryHeaps`.
- `memoryHeapCount` is the number of valid elements in the `memoryHeaps` array.
- `memoryHeaps` is an array of `VK_MAX_MEMORY_HEAPS` `VkMemoryHeap` structures describing the memory heaps from which memory can be allocated.

The `VkPhysicalDeviceMemoryProperties` structure describes a number of memory heaps as well as a number of memory types that can be used to access memory allocated in those heaps. Each heap describes a memory resource of a particular size, and each memory type describes a set of memory properties (e.g. host cached vs. uncached) that can be used with a given memory heap. Allocations using a particular memory type will consume resources from the heap indicated by that memory type's heap index. More than one memory type may share each heap, and the heaps and memory types provide a mechanism to advertise an accurate size of the physical memory resources while allowing the memory to be used with a variety of different properties.

The number of memory heaps is given by `memoryHeapCount` and is less than or equal to `VK_MAX_MEMORY_HEAPS`. Each heap is described by an element of the `memoryHeaps` array as a `VkMemoryHeap` structure. The number of memory types available across all memory heaps is given by `memoryTypeCount` and is less than or equal to `VK_MAX_MEMORY_TYPES`. Each memory type is described by an element of the `memoryTypes` array as a `VkMemoryType` structure.

At least one heap must include `VK_MEMORY_HEAP_DEVICE_LOCAL_BIT` in `VkMemoryHeap::flags`. If there are multiple heaps that all have similar performance characteristics, they may all include `VK_MEMORY_HEAP_DEVICE_LOCAL_BIT`. In a unified memory architecture (UMA) system there is often only a single memory heap which is considered to be equally “local” to the host and to the device, and such an implementation must advertise the heap as device-local.

Each memory type returned by `vkGetPhysicalDeviceMemoryProperties` must have its `propertyFlags` set to one of the following values:

- 0
- `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` | `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`
- `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` | `VK_MEMORY_PROPERTY_HOST_CACHED_BIT`
- `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` | `VK_MEMORY_PROPERTY_HOST_CACHED_BIT` |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT
• VK_MEMORY_PROPERTYDEVICE_LOCAL_BIT
• VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT | VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORYPROPERTY_HOST_COHERENT_BIT
• VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT | VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_CACHED_BIT
• VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT | VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_CACHED_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT
• VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT | VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_CACHED_BIT | VK_MEMORY_PROPERTY_HOST_CACHED_BIT
• VK_MEMORY_PROPERTYDEVICE_LOCAL_BIT | VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT
• VK_MEMORY_PROPERTY_PROTECTED_BIT
• VK_MEMORY_PROPERTY_PROTECTED_BIT | VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT

There must be at least one memory type with both the VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT and VK_MEMORY_PROPERTY_HOST_COHERENT_BIT bits set in its propertyFlags. There must be at least one memory type with the VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT bit set in its propertyFlags.

For each pair of elements X and Y returned in memoryTypes, X must be placed at a lower index position than Y if:

• the set of bit flags returned in the propertyFlags member of X is a strict subset of the set of bit flags returned in the propertyFlags member of Y; or
• the propertyFlags members of X and Y are equal, and X belongs to a memory heap with greater performance (as determined in an implementation-specific manner)

**Note**

There is no ordering requirement between X and Y elements for the case their propertyFlags members are not in a subset relation. That potentially allows more than one possible way to order the same set of memory types. Notice that the list of all allowed memory property flag combinations is written in a valid order. But if instead VK_MEMORY_PROPERTYDEVICE_LOCAL_BIT was before VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, the list would still be in a valid order.

This ordering requirement enables applications to use a simple search loop to select the desired memory type along the lines of:

```c
// Find a memory in 'memoryTypeBitsRequirement' that includes all of 'requiredProperties'
int32_t findProperties(const VkPhysicalDeviceMemoryProperties* pMemoryProperties,
                         uint32_t memoryTypeBitsRequirement,
```
VkMemoryPropertyFlags requiredProperties) {
    const uint32_t memoryCount = pMemoryProperties->memoryTypeCount;
    for (uint32_t memoryIndex = 0; memoryIndex < memoryCount; ++memoryIndex) {
        const uint32_t memoryTypeBits = (1 << memoryIndex);
        const bool isRequiredMemoryType = memoryTypeBitsRequirement & memoryTypeBits;
        const VkMemoryPropertyFlags properties = pMemoryProperties->memoryTypes[memoryIndex].propertyFlags;
        const bool hasRequiredProperties = (properties & requiredProperties) == requiredProperties;
        if (isRequiredMemoryType && hasRequiredProperties)
            return static_cast<int32_t>(memoryIndex);
    }
    // failed to find memory type
    return -1;
}

// Try to find an optimal memory type, or if it does not exist try fallback memory type
// 'device' is the VkDevice
// 'image' is the VkImage that requires memory to be bound
// 'memoryProperties' properties as returned by vkGetPhysicalDeviceMemoryProperties
// 'requiredProperties' are the property flags that must be present
// 'optimalProperties' are the property flags that are preferred by the application
VkMemoryRequirements memoryRequirements;
vkGetImageMemoryRequirements(device, image, &memoryRequirements);
int32_t memoryType = findProperties(&memoryProperties, memoryRequirements.memoryTypeBits, optimalProperties);
if (memoryType == -1) // not found; try fallback properties
    memoryType = findProperties(&memoryProperties, memoryRequirements.memoryTypeBits, requiredProperties);

VK_MAX_MEMORY_TYPES is the length of an array of VkMemoryType structures describing memory types, as returned in VkPhysicalDeviceMemoryProperties::memoryTypes.

#define VK_MAX_MEMORY_TYPES 32U

VK_MAX_MEMORY_HEAPS is the length of an array of VkMemoryHeap structures describing memory heaps, as returned in VkPhysicalDeviceMemoryProperties::memoryHeaps.

#define VK_MAX_MEMORY_HEAPS 16U

To query memory properties, call:
```c
// Provided by VK_VERSION_1_1
void vkGetPhysicalDeviceMemoryProperties2(
    VkPhysicalDevice physicalDevice,
    VkPhysicalDeviceMemoryProperties2* pMemoryProperties);
```

- **physicalDevice** is the handle to the device to query.
- **pMemoryProperties** is a pointer to a `VkPhysicalDeviceMemoryProperties2` structure in which the properties are returned.

`vkGetPhysicalDeviceMemoryProperties2` behaves similarly to `vkGetPhysicalDeviceMemoryProperties`, with the ability to return extended information in a `pNext` chain of output structures.

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceMemoryProperties2-physicalDevice-parameter
  `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceMemoryProperties2-pMemoryProperties-parameter
  `pMemoryProperties` must be a valid pointer to a `VkPhysicalDeviceMemoryProperties2` structure

The `VkPhysicalDeviceMemoryProperties2` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceMemoryProperties2 {
    VkStructureType sType;
    void* pNext;
    VkPhysicalDeviceMemoryProperties memoryProperties;
} VkPhysicalDeviceMemoryProperties2;
```

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

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceMemoryProperties2-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PROPERTIES_2`
- VUID-VkPhysicalDeviceMemoryProperties2-pNext-pNext
  `pNext` must be NULL

The `VkMemoryHeap` structure is defined as:
typedef struct VkMemoryHeap {
  VkDeviceSize size;
  VkMemoryHeapFlags flags;
} VkMemoryHeap;

- `size` is the total memory size in bytes in the heap.
- `flags` is a bitmask of `VkMemoryHeapFlagBits` specifying attribute flags for the heap.

Bits which may be set in `VkMemoryHeap::flags`, indicating attribute flags for the heap, are:

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

- `VK_MEMORY_HEAP_DEVICE_LOCAL_BIT` specifies that the heap corresponds to device-local memory. Device-local memory may have different performance characteristics than host-local memory, and may support different memory property flags.
- `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` specifies that in a logical device representing more than one physical device, there is a per-physical device instance of the heap memory. By default, an allocation from such a heap will be replicated to each physical device's instance of the heap.

```
typedef VkFlags VkMemoryHeapFlags;
```

`VkMemoryHeapFlags` is a bitmask type for setting a mask of zero or more `VkMemoryHeapFlagBits`.

The `VkMemoryType` structure is defined as:

```cpp
typedef struct VkMemoryType {
  VkMemoryPropertyFlags propertyFlags;
  uint32_t heapIndex;
} VkMemoryType;
```

- `heapIndex` describes which memory heap this memory type corresponds to, and must be less than `memoryHeapCount` from the `VkPhysicalDeviceMemoryProperties` structure.
- `propertyFlags` is a bitmask of `VkMemoryPropertyFlagBits` of properties for this memory type.

Bits which may be set in `VkMemoryType::propertyFlags`, indicating properties of a memory type, are:
typedef enum VkMemoryPropertyFlagBits {
    VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT = 0x00000001,
    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT = 0x00000002,
    VK_MEMORY_PROPERTY_HOST_COHERENT_BIT = 0x00000004,
    VK_MEMORY_PROPERTY_HOST_CACHED_BIT = 0x00000008,
    VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT = 0x00000010,
    VK_MEMORY_PROPERTY_PROTECTED_BIT = 0x00000020,
}VkMemoryPropertyFlagBits;

- **VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT** bit specifies that memory allocated with this type is the most efficient for device access. This property will be set if and only if the memory type belongs to a heap with the **VK_MEMORY_HEAP_DEVICE_LOCAL_BIT** set.

- **VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT** bit specifies that memory allocated with this type can be mapped for host access using **vkMapMemory**.

- **VK_MEMORY_PROPERTY_HOST_COHERENT_BIT** bit specifies that the host cache management commands **vkFlushMappedMemoryRanges** and **vkInvalidateMappedMemoryRanges** are not needed to flush host writes to the device or make device writes visible to the host, respectively.

- **VK_MEMORY_PROPERTY_HOST_CACHED_BIT** bit specifies that memory allocated with this type is cached on the host. Host memory accesses to uncached memory are slower than to cached memory, however uncached memory is always host coherent.

- **VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT** bit specifies that the memory type only allows device access to the memory. Memory types must not have both **VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT** and **VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT** set. Additionally, the object's backing memory may be provided by the implementation lazily as specified in Lazily Allocated Memory.

- **VK_MEMORY_PROPERTY_PROTECTED_BIT** bit specifies that the memory type only allows device access to the memory, and allows protected queue operations to access the memory. Memory types must not have **VK_MEMORY_PROPERTY_PROTECTED_BIT** set and any of **VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT** set, or **VK_MEMORY_PROPERTY_HOST_COHERENT_BIT** set, or **VK_MEMORY_PROPERTY_HOST_CACHED_BIT** set.

// Provided by VK_VERSION_1_0
typedef VkFlags VkMemoryPropertyFlags;

**VkMemoryPropertyFlags** is a bitmask type for setting a mask of zero or more **VkMemoryPropertyFlagBits**.

### 11.2.2. Device Memory Objects

A Vulkan device operates on data in device memory via memory objects that are represented in the API by a **VkDeviceMemory** handle:
11.2.3. Device Memory Allocation

To allocate memory objects, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkAllocateMemory(
    VkDevice device,
    const VkMemoryAllocateInfo* pAllocateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkDeviceMemory* pMemory);
```

- `device` is the logical device that owns the memory.
- `pAllocateInfo` is a pointer to a `VkMemoryAllocateInfo` structure describing parameters of the allocation. A successfully returned allocation must use the requested parameters — no substitution is permitted by the implementation.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pMemory` is a pointer to a `VkDeviceMemory` handle in which information about the allocated memory is returned.

Allocations returned by `vkAllocateMemory` are guaranteed to meet any alignment requirement of the implementation. For example, if an implementation requires 128 byte alignment for images and 64 byte alignment for buffers, the device memory returned through this mechanism would be 128-byte aligned. This ensures that applications can correctly suballocate objects of different types (with potentially different alignment requirements) in the same memory object.

When memory is allocated, its contents are undefined with the following constraint:

- The contents of unprotected memory must not be a function of the contents of data protected memory objects, even if those memory objects were previously freed.

```
Note
The contents of memory allocated by one application should not be a function of data from protected memory objects of another application, even if those memory objects were previously freed.
```

The maximum number of valid memory allocations that can exist simultaneously within a `VkDevice` may be restricted by implementation- or platform-dependent limits. The `maxMemoryAllocationCount` feature describes the number of allocations that can exist simultaneously before encountering these internal limits.

```
Note
For historical reasons, if maxMemoryAllocationCount is exceeded, some
```
implementations may return \texttt{VK\_ERROR\_TOO\_MANY\_OBJECTS}. Exceeding this limit will result in undefined behavior, and an application should not rely on the use of the returned error code in order to identify when the limit is reached.

\begin{center}
\textbf{Note}
\end{center}

Many protected memory implementations involve complex hardware and system software support, and often have additional and much lower limits on the number of simultaneous protected memory allocations (from memory types with the \texttt{VK\_MEMORY\_PROPERTY\_PROTECTED\_BIT} property) than for non-protected memory allocations. These limits can be system-wide, and depend on a variety of factors outside of the Vulkan implementation, so they cannot be queried in Vulkan. Applications \textbf{should} use as few allocations as possible from such memory types by suballocating aggressively, and be prepared for allocation failure even when there is apparently plenty of capacity remaining in the memory heap. As a guideline, the Vulkan conformance test suite requires that at least 80 minimum-size allocations can exist concurrently when no other uses of protected memory are active in the system.

Some platforms \textbf{may} have a limit on the maximum size of a single allocation. For example, certain systems \textbf{may} fail to create allocations with a size greater than or equal to 4GB. Such a limit is implementation-dependent, and if such a failure occurs then the error \texttt{VK\_ERROR\_OUT\_OF\_DEVICE\_MEMORY} \textbf{must} be returned.

\begin{center}
\textbf{Valid Usage}
\end{center}

- \textbf{VUID-vkAllocateMemory-pAllocateInfo-01713} \hfill \\
\texttt{pAllocateInfo->allocationSize} must be less than or equal to \texttt{VkPhysicalDeviceMemoryProperties::memoryHeaps[memindex].size} where \texttt{memindex = VkPhysicalDeviceMemoryProperties::memoryTypes[pAllocateInfo->memoryTypeIndex].heapIndex} as returned by \texttt{vkGetPhysicalDeviceMemoryProperties} for the \texttt{VkPhysicalDevice} that \texttt{device} was created from

- \textbf{VUID-vkAllocateMemory-pAllocateInfo-01714} \hfill \\
\texttt{pAllocateInfo->memoryTypeIndex} must be less than \texttt{VkPhysicalDeviceMemoryProperties::memoryTypeCount} as returned by \texttt{vkGetPhysicalDeviceMemoryProperties} for the \texttt{VkPhysicalDevice} that \texttt{device} was created from

- \textbf{VUID-vkAllocateMemory-maxMemoryAllocationCount-04101} \hfill \\
There must be less than \texttt{VkPhysicalDeviceLimits::maxMemoryAllocationCount} device memory allocations currently allocated on the device

\begin{center}
\textbf{Valid Usage (Implicit)}
\end{center}

- \textbf{VUID-vkAllocateMemory-device-parameter} \hfill \\
\texttt{device} must be a valid \texttt{VkDevice} handle

- \textbf{VUID-vkAllocateMemory-pAllocateInfo-parameter} \hfill \\
\texttt{pAllocateInfo} must be a valid pointer to a valid \texttt{VkMemoryAllocateInfo} structure
If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure.

- **VUID-vkAllocateMemory-pAllocator-parameter**
  
  - **pMemory** must be a valid pointer to a `VkDeviceMemory` handle.

## Return Codes

### Success
- `VK_SUCCESS`

### Failure
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INVALID_EXTERNAL_HANDLE`

The `VkMemoryAllocateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkMemoryAllocateInfo {
    VkStructureType sType;
    const void* pNext;
    VkDeviceSize allocationSize;
    uint32_t memoryTypeIndex;
} VkMemoryAllocateInfo;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `allocationSize` is the size of the allocation in bytes.
- `memoryTypeIndex` is an index identifying a memory type from the `memoryTypes` array of the `VkPhysicalDeviceMemoryProperties` structure.

The internal data of an allocated device memory object must include a reference to implementation-specific resources, referred to as the memory object's `payload`.

## Valid Usage

- **VUID-VkMemoryAllocateInfo-None-06657**
  
  - The parameters must not define more than one import operation.

- **VUID-VkMemoryAllocateInfo-allocationSize-00638**
  
  - `allocationSize` must be greater than 0.

- **VUID-VkMemoryAllocateInfo-memoryTypeIndex-01872**
  
  - If the protectedMemory feature is not enabled, the `VkMemoryAllocateInfo::memoryTypeIndex`
must not indicate a memory type that reports \texttt{VK_MEMORY_PROPERTY_PROTECTED_BIT}

- **VUID-VkMemoryAllocateInfo-opaqueCaptureAddress-03329**
  If \texttt{VkMemoryOpaqueCaptureAddressAllocateInfo::opaqueCaptureAddress} is not zero, \texttt{VkMemoryAllocateFlagsInfo::flags} must include \texttt{VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT}

- **VUID-VkMemoryAllocateInfo-flags-03330**
  If \texttt{VkMemoryAllocateFlagsInfo::flags} includes \texttt{VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT}, the \texttt{bufferDeviceAddressCaptureReplay} feature must be enabled

- **VUID-VkMemoryAllocateInfo-flags-03331**
  If \texttt{VkMemoryAllocateFlagsInfo::flags} includes \texttt{VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT}, the \texttt{bufferDeviceAddress} feature must be enabled

- **VUID-VkMemoryAllocateInfo-opaqueCaptureAddress-03333**
  If the parameters define an import operation, \texttt{VkMemoryOpaqueCaptureAddressAllocateInfo::opaqueCaptureAddress} must be zero

---

**Valid Usage (Implicit)**

- **VUID-VkMemoryAllocateInfo-sType-sType**
  \texttt{sType} must be \texttt{VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO}

- **VUID-VkMemoryAllocateInfo-pNext-pNext**
  Each \texttt{pNext} member of any structure (including this one) in the \texttt{pNext} chain must be either \texttt{NULL} or a pointer to a valid instance of \texttt{VkExportMemoryAllocateInfo}, \texttt{VkMemoryAllocateFlagsInfo}, \texttt{VkMemoryDedicatedAllocateInfo}, or \texttt{VkMemoryOpaqueCaptureAddressAllocateInfo}

- **VUID-VkMemoryAllocateInfo-sType-unique**
  The \texttt{sType} value of each struct in the \texttt{pNext} chain must be unique

If the \texttt{pNext} chain includes a \texttt{VkMemoryDedicatedAllocateInfo} structure, then that structure includes a handle of the sole buffer or image resource that the memory can be bound to.

The \texttt{VkMemoryDedicatedAllocateInfo} structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VK_VERSION_1_1
    VkMemoryDedicatedAllocateInfo {
        VkStructureType sType;
        const void* pNext;
        VkImage image;
        VkBuffer buffer;
    } VkMemoryDedicatedAllocateInfo;
```

- \texttt{sType} is the type of this structure.
- \texttt{pNext} is \texttt{NULL} or a pointer to a structure extending this structure.
• **image** is `VK_NULL_HANDLE` or a handle of an image which this memory will be bound to.
• **buffer** is `VK_NULL_HANDLE` or a handle of a buffer which this memory will be bound to.

**Valid Usage**

- VUID-VkMemoryDedicatedAllocateInfo-image-01432
  At least one of **image** and **buffer** must be `VK_NULL_HANDLE`
- VUID-VkMemoryDedicatedAllocateInfo-image-01433
  If **image** is not `VK_NULL_HANDLE`, **VkMemoryAllocateInfo::allocationSize** must equal the **VkMemoryRequirements::size** of the image
- VUID-VkMemoryDedicatedAllocateInfo-image-01434
  If **image** is not `VK_NULL_HANDLE`, **image** must have been created without **VK_IMAGE_CREATE_SPARSE_BINDING_BIT** set in **VkImageCreateInfo::flags**
- VUID-VkMemoryDedicatedAllocateInfo-buffer-01435
  If **buffer** is not `VK_NULL_HANDLE`, **VkMemoryAllocateInfo::allocationSize** must equal the **VkMemoryRequirements::size** of the buffer
- VUID-VkMemoryDedicatedAllocateInfo-buffer-01436
  If **buffer** is not `VK_NULL_HANDLE`, **buffer** must have been created without **VK_BUFFER_CREATE_SPARSE_BINDING_BIT** set in **VkBufferCreateInfo::flags**

**Valid Usage (Implicit)**

- VUID-VkMemoryDedicatedAllocateInfo-sType-sType
  *sType* must be `VK_STRUCTURE_TYPE_MEMORY_DEDICATED_ALLOCATE_INFO`
- VUID-VkMemoryDedicatedAllocateInfo-image-parameter
  If **image** is not `VK_NULL_HANDLE`, **image** must be a valid **VkImage** handle
- VUID-VkMemoryDedicatedAllocateInfo-buffer-parameter
  If **buffer** is not `VK_NULL_HANDLE`, **buffer** must be a valid **VkBuffer** handle
- VUID-VkMemoryDedicatedAllocateInfo-commonparent
  Both of **buffer**, and **image** that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same **VkDevice**

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

The **VkExportMemoryAllocateInfo** structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkExportMemoryAllocateInfo {
    VkStructureType sType;
    const void* pNext;
    VkExternalMemoryHandleTypeFlags handleTypes;
};
```
sType is the type of this structure.

pNext is NULL or a pointer to a structure extending this structure.

handleTypes is zero or a bitmask of VkExternalMemoryHandleTypeFlagBits specifying one or more memory handle types the application can export from the resulting allocation. The application can request multiple handle types for the same allocation.

Valid Usage

- VUID-VkExportMemoryAllocateInfo-handleTypes-00656
  The bits in handleTypes must be supported and compatible, as reported by VkExternalImageFormatProperties or VkExternalBufferProperties

Valid Usage (Implicit)

- VUID-VkExportMemoryAllocateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO

- VUID-VkExportMemoryAllocateInfo-handleTypes-parameter
  handleTypes must be a valid combination of VkExternalMemoryHandleTypeFlagBits values

11.2.4. Device Group Memory Allocations

If the pNext chain of VkMemoryAllocateInfo includes a VkMemoryAllocateFlagsInfo structure, then that structure includes flags and a device mask controlling how many instances of the memory will be allocated.

The VkMemoryAllocateFlagsInfo structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkMemoryAllocateFlagsInfo {
    VkStructureType sType;
    const void* pNext;
    VkMemoryAllocateFlags flags;
    uint32_t deviceMask;
} VkMemoryAllocateFlagsInfo;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- flags is a bitmask of VkMemoryAllocateFlagBits controlling the allocation.
- deviceMask is a mask of physical devices in the logical device, indicating that memory must be allocated on each device in the mask, if VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT is set in flags.
If \texttt{VK\_MEMORY\_ALLOCATE\_DEVICE\_MASK\_BIT} is not set, the number of instances allocated depends on whether \texttt{VK\_MEMORY\_HEAP\_MULTI\_INSTANCE\_BIT} is set in the memory heap. If \texttt{VK\_MEMORY\_HEAP\_MULTI\_INSTANCE\_BIT} is set, then memory is allocated for every physical device in the logical device (as if \texttt{deviceMask} has bits set for all device indices). If \texttt{VK\_MEMORY\_HEAP\_MULTI\_INSTANCE\_BIT} is not set, then a single instance of memory is allocated (as if \texttt{deviceMask} is set to one).

On some implementations, allocations from a multi-instance heap \textbf{may} consume memory on all physical devices even if the \texttt{deviceMask} excludes some devices. If \texttt{VkPhysicalDeviceGroupProperties::subsetAllocation} is \texttt{VK\_TRUE}, then memory is only consumed for the devices in the device mask.

\begin{itemize}
  \item \textbf{Valid Usage}
  \begin{itemize}
    \item \textbf{VUID-VkMemoryAllocateFlagsInfo-deviceMask-00675}
      If \texttt{VK\_MEMORY\_ALLOCATE\_DEVICE\_MASK\_BIT} is set, \texttt{deviceMask} must be a valid device mask
    \item \textbf{VUID-VkMemoryAllocateFlagsInfo-deviceMask-00676}
      If \texttt{VK\_MEMORY\_ALLOCATE\_DEVICE\_MASK\_BIT} is set, \texttt{deviceMask} must not be zero
  \end{itemize}
  \end{itemize}

\begin{itemize}
  \item \textbf{Valid Usage (Implicit)}
  \begin{itemize}
    \item \textbf{VUID-VkMemoryAllocateFlagsInfo-sType-sType}
      \texttt{sType} must be \texttt{VK\_STRUCTURE\_TYPE\_MEMORY\_ALLOCATE\_FLAGS\_INFO}
    \item \textbf{VUID-VkMemoryAllocateFlagsInfo-flags-parameter}
      \texttt{flags} must be a valid combination of \texttt{VkMemoryAllocateFlagBits} values
  \end{itemize}
\end{itemize}

Bits which \textbf{can} be set in \texttt{VkMemoryAllocateFlagsInfo::flags}, controlling device memory allocation, are:

\begin{verbatim}
// Provided by VK\_VERSION\_1\_1
typedef enum VkMemoryAllocateFlagBits {
    VK\_MEMORY\_ALLOCATE\_DEVICE\_MASK\_BIT = 0x00000001,
    // Provided by VK\_VERSION\_1\_2
    VK\_MEMORY\_ALLOCATE\_DEVICE\_ADDRESS\_BIT = 0x00000002,
    // Provided by VK\_VERSION\_1\_2
    VK\_MEMORY\_ALLOCATE\_DEVICE\_ADDRESS\_CAPTURE\_REPLAY\_BIT = 0x00000004,
} VkMemoryAllocateFlagBits;
\end{verbatim}

- \texttt{VK\_MEMORY\_ALLOCATE\_DEVICE\_MASK\_BIT} specifies that memory will be allocated for the devices in
VkMemoryAllocateFlagsInfo::deviceMask.

- **VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT** specifies that the memory can be attached to a buffer object created with the **VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT** bit set in usage, and that the memory handle can be used to retrieve an opaque address via vkGetDeviceMemoryOpaqueCaptureAddress.

- **VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT** specifies that the memory's address can be saved and reused on a subsequent run (e.g. for trace capture and replay), see VkBufferOpaqueCaptureAddressCreateInfo for more detail.

```
// Provided by VK_VERSION_1_1
typedef VkFlags VkMemoryAllocateFlags;
```

VkMemoryAllocateFlags is a bitmask type for setting a mask of zero or more VkMemoryAllocateFlagBits.

### 11.2.5. Opaque Capture Address Allocation

To request a specific device address for a memory allocation, add a VkMemoryOpaqueCaptureAddressAllocateInfo structure to the pNext chain of the VkMemoryAllocateInfo structure. The VkMemoryOpaqueCaptureAddressAllocateInfo structure is defined as:

```
// Provided by VK_VERSION_1_2
typedef struct VkMemoryOpaqueCaptureAddressAllocateInfo {
    VkStructureType sType;
    const void* pNext;
    uint64_t opaqueCaptureAddress;
} VkMemoryOpaqueCaptureAddressAllocateInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **opaqueCaptureAddress** is the opaque capture address requested for the memory allocation.

If **opaqueCaptureAddress** is zero, no specific address is requested.

If **opaqueCaptureAddress** is not zero, it should be an address retrieved from vkGetDeviceMemoryOpaqueCaptureAddress on an identically created memory allocation on the same implementation.

**Note**

In most cases, it is expected that a non-zero **opaqueAddress** is an address retrieved from vkGetDeviceMemoryOpaqueCaptureAddress on an identically created memory allocation. If this is not the case, it is likely that **VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS** errors will occur.
This is, however, not a strict requirement because trace capture/replay tools may need to adjust memory allocation parameters for imported memory.

If this structure is not present, it is as if `opaqueCaptureAddress` is zero.

### Valid Usage (Implicit)

- VUID-VkMemoryOpaqueCaptureAddressAllocateInfo-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_MEMORY_OPAQUE_CAPTURE_ADDRESS_ALLOCATE_INFO`.

### 11.2.6. Freeing Device Memory

To free a memory object, call:

```c
// Provided by VK_VERSION_1_0
void vkFreeMemory(
    VkDevice device,       // Provided by VK_VERSION_1_0
    VkDeviceMemory memory,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that owns the memory.
- `memory` is the `VkDeviceMemory` object to be freed.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

Before freeing a memory object, an application must ensure the memory object is no longer in use by the device — for example by command buffers in the `pending state`. Memory can be freed whilst still bound to resources, but those resources must not be used afterwards. Freeing a memory object releases the reference it held, if any, to its payload. If there are still any bound images or buffers, the memory object's payload may not be immediately released by the implementation, but must be released by the time all bound images and buffers have been destroyed. Once all references to a payload are released, it is returned to the heap from which it was allocated.

How memory objects are bound to Images and Buffers is described in detail in the Resource Memory Association section.

If a memory object is mapped at the time it is freed, it is implicitly unmapped.

**Note**

As described below, host writes are not implicitly flushed when the memory object is unmapped, but the implementation must guarantee that writes that have not been flushed do not affect any other memory.

### Valid Usage

- VUID-vkFreeMemory-memory-00677
All submitted commands that refer to memory (via images or buffers) must have completed execution.

Valid Usage (Implicit)

- VUID-vkFreeMemory-device-parameter
  
  device must be a valid VkDevice handle.

- VUID-vkFreeMemory-memory-parameter
  
  If memory is not VK_NULL_HANDLE, memory must be a valid VkDeviceMemory handle.

- VUID-vkFreeMemory-pAllocator-parameter
  
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure.

- VUID-vkFreeMemory-memory-parent
  
  If memory is a valid handle, it must have been created, allocated, or retrieved from device.

Host Synchronization

- Host access to memory must be externally synchronized.

11.2.7. Host Access to Device Memory Objects

Memory objects created with vkAllocateMemory are not directly host accessible.

Memory objects created with the memory property VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT are considered mappable. Memory objects must be mappable in order to be successfully mapped on the host.

To retrieve a host virtual address pointer to a region of a mappable memory object, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkMapMemory(
    VkDevice device,
    VkDeviceMemory memory,
    VkDeviceSize offset,
    VkDeviceSize size,
    VkMemoryMapFlags flags,
    void** ppData);
```

- device is the logical device that owns the memory.

- memory is the VkDeviceMemory object to be mapped.

- offset is a zero-based byte offset from the beginning of the memory object.

- size is the size of the memory range to map, or VK_WHOLE_SIZE to map from offset to the end of the allocation.
• flags is reserved for future use.
• ppData is a pointer to a void* variable in which a host-accessible pointer to the beginning of the mapped range is returned. This pointer minus offset must be aligned to at least VkPhysicalDeviceLimits::minMemoryMapAlignment.

After a successful call to vkMapMemory the memory object memory is considered to be currently host mapped.

_valid usage:

valid usage

- VUID-vkMapMemory-memory-00678
  memory must not be currently host mapped
- VUID-vkMapMemory-offset-00679
  offset must be less than the size of memory
- VUID-vkMapMemory-size-00680
If size is not equal to VK_WHOLE_SIZE, size must be greater than 0

- VUID-vkMapMemory-size-00681
  If size is not equal to VK_WHOLE_SIZE, size must be less than or equal to the size of the memory minus offset

- VUID-vkMapMemory-memory-00682
  memory must have been created with a memory type that reports VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT

Valid Usage (Implicit)

- VUID-vkMapMemory-device-parameter
device must be a valid VkDevice handle

- VUID-vkMapMemory-memory-parameter
memory must be a valid VkDeviceMemory handle

- VUID-vkMapMemory-flags-zerobitmask
flags must be 0

- VUID-vkMapMemory-ppData-parameter
ppData must be a valid pointer to a pointer value

- VUID-vkMapMemory-memory-parent
memory must have been created, allocated, or retrieved from device

Host Synchronization

- Host access to memory must be externally synchronized

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_MEMORY_MAPFailed

// Provided by VK_VERSION_1_0
typedef VkFlags VkMemoryMapFlags;

VkMemoryMapFlags is a bitmask type for setting a mask, but is currently reserved for future use.
Two commands are provided to enable applications to work with non-coherent memory allocations: `vkFlushMappedMemoryRanges` and `vkInvalidateMappedMemoryRanges`.

**Note**

If the memory object was created with the `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` set, `vkFlushMappedMemoryRanges` and `vkInvalidateMappedMemoryRanges` are unnecessary and **may** have a performance cost. However, **availability and visibility operations** still need to be managed on the device. See the description of **host access types** for more information.

To flush ranges of non-coherent memory from the host caches, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkFlushMappedMemoryRanges(
    VkDevice device,
    uint32_t memoryRangeCount,
    const VkMappedMemoryRange* pMemoryRanges);
```

- **device** is the logical device that owns the memory ranges.
- **memoryRangeCount** is the length of the **pMemoryRanges** array.
- **pMemoryRanges** is a pointer to an array of `VkMappedMemoryRange` structures describing the memory ranges to flush.

`vkFlushMappedMemoryRanges` guarantees that host writes to the memory ranges described by **pMemoryRanges** are made available to the host memory domain, such that they **can** be made available to the device memory domain via **memory domain operations** using the `VK_ACCESS_HOST_WRITE_BIT` access type.

Within each range described by **pMemoryRanges**, each set of `nonCoherentAtomSize` bytes in that range is flushed if any byte in that set has been written by the host since it was first host mapped, or the last time it was flushed. If **pMemoryRanges** includes sets of `nonCoherentAtomSize` bytes where no bytes have been written by the host, those bytes **must** not be flushed.

Unmapping non-coherent memory does not implicitly flush the host mapped memory, and host writes that have not been flushed **may** not ever be visible to the device. However, implementations **must** ensure that writes that have not been flushed do not become visible to any other memory.

**Note**

The above guarantee avoids a potential memory corruption in scenarios where host writes to a mapped memory object have not been flushed before the memory is unmapped (or freed), and the virtual address range is subsequently reused for a different mapping (or memory allocation).

**Valid Usage (Implicit)**

- VUID-vkFlushMappedMemoryRanges-device-parameter
device must be a valid VkDevice handle

- VUID-vkFlushMappedMemoryRanges-pMemoryRanges-parameter
  pMemoryRanges must be a valid pointer to an array of memoryRangeCount valid VkMappedMemoryRange structures

- VUID-vkFlushMappedMemoryRanges-memoryRangeCount-arraylength
  memoryRangeCount must be greater than 0

### Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

To invalidate ranges of non-coherent memory from the host caches, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkInvalidateMappedMemoryRanges(
    VkDevice device,
    uint32_t memoryRangeCount,
    const VkMappedMemoryRange* pMemoryRanges);
```

- **device** is the logical device that owns the memory ranges.
- **memoryRangeCount** is the length of the **pMemoryRanges** array.
- **pMemoryRanges** is a pointer to an array of VkMappedMemoryRange structures describing the memory ranges to invalidate.

vkInvalidateMappedMemoryRanges guarantees that device writes to the memory ranges described by **pMemoryRanges**, which have been made available to the host memory domain using the **VK_ACCESS_HOST_WRITE_BIT** and **VK_ACCESS_HOST_READ_BIT** access types, are made visible to the host. If a range of non-coherent memory is written by the host and then invalidated without first being flushed, its contents are undefined.

Within each range described by **pMemoryRanges**, each set of nonCoherentAtomSize bytes in that range is invalidated if any byte in that set has been written by the device since it was first host mapped, or the last time it was invalidated.

**Note**
Mapping non-coherent memory does not implicitly invalidate that memory.
Valid Usage (Implicit)

- VUID-vkInvalidateMappedMemoryRanges-device-parameter
  
  `device` **must** be a valid `VkDevice` handle

- VUID-vkInvalidateMappedMemoryRanges-pMemoryRanges-parameter
  
  `pMemoryRanges` **must** be a valid pointer to an array of `memoryRangeCount` valid `VkMappedMemoryRange` structures

- VUID-vkInvalidateMappedMemoryRanges-memoryRangeCount-arraylength
  
  `memoryRangeCount` **must** be greater than 0

Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkMappedMemoryRange` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkMappedMemoryRange {
    VkStructureType       sType;
    const void*            pNext;
    VkDeviceMemory         memory;
    VkDeviceSize           offset;
    VkDeviceSize           size;
} VkMappedMemoryRange;
```

- `sType` is the type of this structure.
- `pNext` is **NULL** or a pointer to a structure extending this structure.
- `memory` is the memory object to which this range belongs.
- `offset` is the zero-based byte offset from the beginning of the memory object.
- `size` is either the size of range, or `VK_WHOLE_SIZE` to affect the range from `offset` to the end of the current mapping of the allocation.

Valid Usage

- VUID-VkMappedMemoryRange-memory-memory-00684
  
  `memory` **must** be currently host mapped

- VUID-VkMappedMemoryRange-size-size-00685
If `size` is not equal to `VK_WHOLE_SIZE`, `offset` and `size` must specify a range contained within the currently mapped range of `memory`.

- VUID-VkMappedMemoryRange-size-00686
  If `size` is equal to `VK_WHOLE_SIZE`, `offset` must be within the currently mapped range of `memory`.

- VUID-VkMappedMemoryRange-offset-00687
  `offset` must be a multiple of `VkPhysicalDeviceLimits::nonCoherentAtomSize`.

- VUID-VkMappedMemoryRange-size-01389
  If `size` is equal to `VK_WHOLE_SIZE`, the end of the current mapping of `memory` must either be a multiple of `VkPhysicalDeviceLimits::nonCoherentAtomSize` bytes from the beginning of the memory object, or be equal to the end of the memory object.

- VUID-VkMappedMemoryRange-size-01390
  If `size` is not equal to `VK_WHOLE_SIZE`, `size` must either be a multiple of `VkPhysicalDeviceLimits::nonCoherentAtomSize`, or `offset` plus `size` must equal the size of `memory`.

### Valid Usage (Implicit)

- VUID-VkMappedMemoryRange-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE`.

- VUID-VkMappedMemoryRange-pNext-pNext
  `pNext` must be `NULL`.

- VUID-VkMappedMemoryRange-memory-parameter
  `memory` must be a valid `VkDeviceMemory` handle.

To unmapped a memory object once host access to it is no longer needed by the application, call:

```c
// Provided by VK_VERSION_1_0
void vkUnmapMemory(
    VkDevice device,          // device is the logical device that owns the memory.
    VkDeviceMemory memory);  // memory is the memory object to be unmapped.
```

### Valid Usage

- VUID-vkUnmapMemory-memory-00689
  `memory` must be currently host mapped.
Valid Usage (Implicit)

- VUID-vkUnmapMemory-device-parameter
  device must be a valid VkDevice handle

- VUID-vkUnmapMemory-memory-parameter
  memory must be a valid VkDeviceMemory handle

- VUID-vkUnmapMemory-memory-parent
  memory must have been created, allocated, or retrieved from device

Host Synchronization

- Host access to memory must be externally synchronized

11.2.8. Lazily Allocated Memory

If the memory object is allocated from a heap with the VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT bit set, that object’s backing memory may be provided by the implementation lazily. The actual committed size of the memory may initially be as small as zero (or as large as the requested size), and monotonically increases as additional memory is needed.

A memory type with this flag set is only allowed to be bound to a VkImage whose usage flags include VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT.

Note

Using lazily allocated memory objects for framebuffer attachments that are not needed once a render pass instance has completed may allow some implementations to never allocate memory for such attachments.

To determine the amount of lazily-allocated memory that is currently committed for a memory object, call:

```c
// Provided by VK_VERSION_1_0
void vkGetDeviceMemoryCommitment(
    VkDevice device,
    VkDeviceMemory memory,
    VkDeviceSize* pCommittedMemoryInBytes);
```

- device is the logical device that owns the memory.
- memory is the memory object being queried.
- pCommittedMemoryInBytes is a pointer to a VkDeviceSize value in which the number of bytes currently committed is returned, on success.

The implementation may update the commitment at any time, and the value returned by this query
The implementation guarantees to allocate any committed memory from the heapIndex indicated by the memory type that the memory object was created with.

### Valid Usage

- **VUID-vkGetDeviceMemoryCommitment-memory-00690**  
  *memory* must have been created with a memory type that reports `VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT`

### Valid Usage (Implicit)

- **VUID-vkGetDeviceMemoryCommitment-device-parameter**  
  *device* must be a valid `VkDevice` handle

- **VUID-vkGetDeviceMemoryCommitment-memory-parameter**  
  *memory* must be a valid `VkDeviceMemory` handle

- **VUID-vkGetDeviceMemoryCommitment-pCommittedMemoryInBytes-parameter**  
  *pCommittedMemoryInBytes* must be a valid pointer to a `VkDeviceSize` value

- **VUID-vkGetDeviceMemoryCommitment-memory-parent**  
  *memory* must have been created, allocated, or retrieved from *device*

### 11.2.9. Protected Memory

*Protected memory* divides device memory into protected device memory and unprotected device memory.

Protected memory adds the following concepts:

- **Memory**:  
  - Unprotected device memory, which *can* be visible to the device and *can* be visible to the host  
  - Protected device memory, which *can* be visible to the device but *must* not be visible to the host

- **Resources**:  
  - Unprotected images and unprotected buffers, to which unprotected memory *can* be bound  
  - Protected images and protected buffers, to which protected memory *can* be bound

- **Command buffers**:  
  - Unprotected command buffers, which *can* be submitted to a device queue to execute unprotected queue operations  
  - Protected command buffers, which *can* be submitted to a protected-capable device queue to execute protected queue operations
• Device queues:
  ◦ Unprotected device queues, to which unprotected command buffers can be submitted
  ◦ Protected-capable device queues, to which unprotected command buffers or protected command buffers can be submitted

• Queue submissions
  ◦ Unprotected queue submissions, through which unprotected command buffers can be submitted
  ◦ Protected queue submissions, through which protected command buffers can be submitted

• Queue operations
  ◦ Unprotected queue operations
  ◦ Protected queue operations

Protected Memory Access Rules

If VkPhysicalDeviceProtectedMemoryProperties::protectedNoFault is VK_FALSE, applications must not perform any of the following operations:

• Write to unprotected memory within protected queue operations.
• Access protected memory within protected queue operations other than in framebuffer-space pipeline stages, the compute shader stage, or the transfer stage.
• Perform a query within protected queue operations.

If VkPhysicalDeviceProtectedMemoryProperties::protectedNoFault is VK_TRUE, these operations are valid, but reads will return undefined values, and writes will either be dropped or store undefined values.

Additionally, indirect operations must not be performed within protected queue operations.

Whether these operations are valid or not, or if any other invalid usage is performed, the implementation must guarantee that:

• Protected device memory must never be visible to the host.
• Values written to unprotected device memory must not be a function of values from protected memory.

11.2.10. Peer Memory Features

Peer memory is memory that is allocated for a given physical device and then bound to a resource and accessed by a different physical device, in a logical device that represents multiple physical devices. Some ways of reading and writing peer memory may not be supported by a device.

To determine how peer memory can be accessed, call:

// Provided by VK_VERSION_1_1
void vkGetDeviceGroupPeerMemoryFeatures(
VkDevice
uint32_t
uint32_t
uint32_t
VkPeerMemoryFeatureFlags*

device, heapIndex, localDeviceIndex, remoteDeviceIndex, pPeerMemoryFeatures);

• device is the logical device that owns the memory.
• heapIndex is the index of the memory heap from which the memory is allocated.
• localDeviceIndex is the device index of the physical device that performs the memory access.
• remoteDeviceIndex is the device index of the physical device that the memory is allocated for.
• pPeerMemoryFeatures is a pointer to a VkPeerMemoryFeatureFlags bitmask indicating which types of memory accesses are supported for the combination of heap, local, and remote devices.

Valid Usage

• VUID-vkGetDeviceGroupPeerMemoryFeatures-heapIndex-00691
  heapIndex must be less than memoryHeapCount
• VUID-vkGetDeviceGroupPeerMemoryFeatures-localDeviceIndex-00692
  localDeviceIndex must be a valid device index
• VUID-vkGetDeviceGroupPeerMemoryFeatures-remoteDeviceIndex-00693
  remoteDeviceIndex must be a valid device index
• VUID-vkGetDeviceGroupPeerMemoryFeatures-localDeviceIndex-00694
  localDeviceIndex must not equal remoteDeviceIndex

Valid Usage (Implicit)

• VUID-vkGetDeviceGroupPeerMemoryFeatures-device-parameter
  device must be a valid VkDevice handle
• VUID-vkGetDeviceGroupPeerMemoryFeatures-pPeerMemoryFeatures-parameter
  pPeerMemoryFeatures must be a valid pointer to a VkPeerMemoryFeatureFlags value

Bits which may be set in vkGetDeviceGroupPeerMemoryFeatures::pPeerMemoryFeatures, indicating supported peer memory features, are:

// Provided by VK_VERSION_1_1
typedef enum VkPeerMemoryFeatureFlagBits {
  VK_PEER_MEMORY_FEATURE_COPY_SRC_BIT = 0x00000001,
  VK_PEER_MEMORY_FEATURE_COPY_DST_BIT = 0x00000002,
  VK_PEER_MEMORY_FEATURE_GENERIC_SRC_BIT = 0x00000004,
  VK_PEER_MEMORY_FEATURE_GENERIC_DST_BIT = 0x00000008,
} VkPeerMemoryFeatureFlagBits;
• **VK_PEER_MEMORY_FEATURE_COPY_SRC_BIT** specifies that the memory can be accessed as the source of any `vkCmdCopy*` command.

• **VK_PEER_MEMORY_FEATURE_COPY_DST_BIT** specifies that the memory can be accessed as the destination of any `vkCmdCopy*` command.

• **VK_PEER_MEMORY_FEATURE_GENERIC_SRC_BIT** specifies that the memory can be read as any memory access type.

• **VK_PEER_MEMORY_FEATURE_GENERIC_DST_BIT** specifies that the memory can be written as any memory access type. Shader atomics are considered to be writes.

---

**Note**

The peer memory features of a memory heap also apply to any accesses that **may** be performed during **image layout transitions**.

**VK_PEER_MEMORY_FEATURE_COPY_DST_BIT** must be supported for all host local heaps and for at least one device-local memory heap.

If a device does not support a peer memory feature, it is still valid to use a resource that includes both local and peer memory bindings with the corresponding access type as long as only the local bindings are actually accessed. For example, an application doing split-frame rendering would use framebuffer attachments that include both local and peer memory bindings, but would scissor the rendering to only update local memory.

```c
// Provided by VK_VERSION_1_1
typedef VkFlags VkPeerMemoryFeatureFlags;
```

**VkPeerMemoryFeatureFlags** is a bitmask type for setting a mask of zero or more **VkPeerMemoryFeatureFlagBits**.

### 11.2.11. Opaque Capture Address Query

To query a 64-bit opaque capture address value from a memory object, call:

```c
// Provided by VK_VERSION_1_2
uint64_t vkGetDeviceMemoryOpaqueCaptureAddress(
    VkDevice device,
    const VkDeviceMemoryOpaqueCaptureAddressInfo* pInfo);
```

• **device** is the logical device that the memory object was allocated on.

• **pInfo** is a pointer to a **VkDeviceMemoryOpaqueCaptureAddressInfo** structure specifying the memory object to retrieve an address for.

The 64-bit return value is an opaque address representing the start of **pInfo->memory**.

If the memory object was allocated with a non-zero value of **VkMemoryOpaqueCaptureAddressAllocateInfo::opaqueCaptureAddress**, the return value **must** be the
same address.

**Note**
The expected usage for these opaque addresses is only for trace capture/replay tools to store these addresses in a trace and subsequently specify them during replay.

### Valid Usage

- VUID-vkGetDeviceMemoryOpaqueCaptureAddress-None-03334
  The `bufferDeviceAddress` feature **must** be enabled

- VUID-vkGetDeviceMemoryOpaqueCaptureAddress-device-03335
  If `device` was created with multiple physical devices, then the `bufferDeviceAddressMultiDevice` feature **must** be enabled

### Valid Usage (Implicit)

- VUID-vkGetDeviceMemoryOpaqueCaptureAddress-device-parameter
  `device` **must** be a valid `VkDevice` handle

- VUID-vkGetDeviceMemoryOpaqueCaptureAddress-pInfo-parameter
  `pNext` **must** be a valid pointer to a valid `VkDeviceMemoryOpaqueCaptureAddressInfo` structure

The `VkDeviceMemoryOpaqueCaptureAddressInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkDeviceMemoryOpaqueCaptureAddressInfo {
    VkStructureType sType;
    const void* pNext;
    VkDeviceMemory memory;
} VkDeviceMemoryOpaqueCaptureAddressInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `memory` specifies the memory whose address is being queried.

### Valid Usage

- VUID-VkDeviceMemoryOpaqueCaptureAddressInfo-memory-03336
  `memory` **must** have been allocated with `VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT`
Valid Usage (Implicit)

- VUID-VkDeviceMemoryOpaqueCaptureAddressInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_DEVICE_MEMORY_OPAQUE_CAPTURE_ADDRESS_INFO

- VUID-VkDeviceMemoryOpaqueCaptureAddressInfo-pNext-pNext
  pNext must be NULL

- VUID-VkDeviceMemoryOpaqueCaptureAddressInfo-memory-parameter
  memory must be a valid VkDeviceMemory handle
Chapter 12. Resource Creation

Vulkan supports two primary resource types: buffers and images. Resources are views of memory with associated formatting and dimensionality. Buffers provide access to raw arrays of bytes, whereas images can be multidimensional and may have associated metadata.

12.1. Buffers

Buffers represent linear arrays of data which are used for various purposes by binding them to a graphics or compute pipeline via descriptor sets or via certain commands, or by directly specifying them as parameters to certain commands.

Buffers are represented by VkBuffer handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkBuffer)
```

To create buffers, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateBuffer(
    VkDevice device,
    const VkBufferCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkBuffer* pBuffer);
```

- `device` is the logical device that creates the buffer object.
- `pCreateInfo` is a pointer to a VkBufferCreateInfo structure containing parameters affecting creation of the buffer.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pBuffer` is a pointer to a VkBuffer handle in which the resulting buffer object is returned.

**Valid Usage**

- VUID-vkCreateBuffer-flags-00911
  If the flags member of pCreateInfo includes VK_BUFFER_CREATE_SPARSE_BINDING_BIT, creating this VkBuffer must not cause the total required sparse memory for all currently valid sparse resources on the device to exceed VkPhysicalDeviceLimits::sparseAddressSpaceSize

**Valid Usage (Implicit)**

- VUID-vkCreateBuffer-device-parameter
**device** must be a valid `VkDevice` handle

- VUID-vkCreateBuffer-pCreateInfo-parameter
  **pCreateInfo** must be a valid pointer to a valid `VkBufferCreateInfo` structure

- VUID-vkCreateBuffer-pAllocator-parameter
  If **pAllocator** is not **NULL**, **pAllocator** must be a valid pointer to a valid `VkAllocationCallbacks` structure

- VUID-vkCreateBuffer-pBuffer-parameter
  **pBuffer** must be a valid pointer to a `VkBuffer` handle

**Return Codes**

**Success**
- **VK_SUCCESS**

**Failure**
- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**

The `VkBufferCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkBufferCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkBufferCreateFlags flags;
    VkDeviceSize size;
    VkBufferUsageFlags usage;
    VkSharingMode sharingMode;
    uint32_t queueFamilyIndexCount;
    const uint32_t* pQueueFamilyIndices;
} VkBufferCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **flags** is a bitmask of `VkBufferCreateFlagBits` specifying additional parameters of the buffer.
- **size** is the size in bytes of the buffer to be created.
- **usage** is a bitmask of `VkBufferUsageFlagBits` specifying allowed usages of the buffer.
- **sharingMode** is a `VkSharingMode` value specifying the sharing mode of the buffer when it will be accessed by multiple queue families.
- **queueFamilyIndexCount** is the number of entries in the `pQueueFamilyIndices` array.
- **pQueueFamilyIndices** is a pointer to an array of queue families that will access this buffer. It is ignored if **sharingMode** is not **VK_SHARING_MODE_CONCURRENT**.
Valid Usage

- **VUID-VkBufferCreateInfo-size-00912**
  - `size` must be greater than 0

- **VUID-VkBufferCreateInfo-sharingMode-00913**
  - If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, `pQueueFamilyIndices` must be a valid pointer to an array of `queueFamilyIndexCount` uint32_t values

- **VUID-VkBufferCreateInfo-sharingMode-00914**
  - If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, `queueFamilyIndexCount` must be greater than 1

- **VUID-VkBufferCreateInfo-sharingMode-01419**
  - If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, each element of `pQueueFamilyIndices` must be unique and must be less than `pQueueFamilyPropertyCount` returned by either `vkGetPhysicalDeviceQueueFamilyProperties` or `vkGetPhysicalDeviceQueueFamilyProperties2` for the `physicalDevice` that was used to create device

- **VUID-VkBufferCreateInfo-flags-00915**
  - If the `sparseBinding` feature is not enabled, `flags` must not contain `VK_BUFFER_CREATE_SPARSE_BINDING_BIT`

- **VUID-VkBufferCreateInfo-flags-00916**
  - If the `sparseResidencyBuffer` feature is not enabled, `flags` must not contain `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT`

- **VUID-VkBufferCreateInfo-flags-00917**
  - If the `sparseResidencyAliased` feature is not enabled, `flags` must not contain `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT`

- **VUID-VkBufferCreateInfo-flags-00918**
  - 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`

- **VUID-VkBufferCreateInfo-pNext-00920**
  - If the `pNext` chain includes a `VkExternalMemoryBufferCreateInfo` structure, its `handleTypes` member must only contain bits that are also in `VkExternalBufferProperties::externalMemoryProperties.compatibleHandleTypes`, as returned by `vkGetPhysicalDeviceExternalBufferProperties` with `pExternalBufferInfo->handleType` equal to any one of the handle types specified in `VkExternalMemoryBufferCreateInfo::handleTypes`

- **VUID-VkBufferCreateInfo-flags-01887**
  - If the `protectedMemory` feature is not enabled, `flags` must not contain `VK_BUFFER_CREATE_PROTECTED_BIT`

- **VUID-VkBufferCreateInfo-None-01888**
  - If any of the bits `VK_BUFFER_CREATE_SPARSE_BINDING_BIT`, `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT`, or `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT` are set, `VK_BUFFER_CREATE_PROTECTED_BIT` must not also be set
If `VkBufferOpaqueCaptureAddressCreateInfo::opaqueCaptureAddress` is not zero, `flags` must include `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT`.

If `flags` includes `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT`, the `bufferDeviceAddressCaptureReplay` feature must be enabled.

`size` must be less than or equal to `VkPhysicalDeviceMaintenance4Properties::maxBufferSize`.

### Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO`
- Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkBufferOpaqueCaptureAddressCreateInfo` or `VkExternalMemoryBufferCreateInfo`
- The `sType` value of each struct in the `pNext` chain must be unique.
- `flags` must be a valid combination of `VkBufferCreateFlagBits` values.
- `usage` must be a valid combination of `VkBufferUsageFlagBits` values.
- `usage` must not be 0.
- `sharingMode` must be a valid `VkSharingMode` value.

Bits which can be set in `VkBufferCreateInfo::usage`, specifying usage behavior of a buffer, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkBufferUsageFlagBits {
    VK_BUFFER_USAGE_TRANSFER_SRC_BIT = 0x00000001,
    VK_BUFFER_USAGE_TRANSFER_DST_BIT = 0x00000002,
    VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT = 0x00000004,
    VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT = 0x00000008,
    VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT = 0x00000010,
    VK_BUFFER_USAGE_STORAGE_BUFFER_BIT = 0x00000020,
    VK_BUFFER_USAGE_INDEX_BUFFER_BIT = 0x00000040,
    VK_BUFFER_USAGE_VERTEX_BUFFER_BIT = 0x00000080,
    VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT = 0x00000100,
} VkBufferUsageFlagBits;
// Provided by VK_VERSION_1_2
```
VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT specifies that the buffer can be used to retrieve a buffer device address via vkGetBufferDeviceAddress and use that address to access the buffer’s memory from a shader.

// Provided by VK_VERSION_1_0
typedef VkFlags VkBufferUsageFlags;

VkBufferUsageFlags is a bitmask type for setting a mask of zero or more VkBufferUsageFlagBits.

Bits which can be set in VkBufferCreateInfo::flags, specifying additional parameters of a buffer, are:

// Provided by VK_VERSION_1_0
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;
VK_BUFFER_CREATE_PROTECTED_BIT = 0x00000008,
VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT = 0x00000010,

VkBufferCreateFlagBits

• VK_BUFFER_CREATE_SPARSE_BINDING_BIT specifies that the buffer will be backed using sparse memory binding.

• VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT specifies that the buffer can be partially backed using sparse memory binding. Buffers created with this flag must also be created with the VK_BUFFER_CREATE_SPARSE_BINDING_BIT flag.

• VK_BUFFER_CREATE_SPARSE_ALIASED_BIT specifies that the buffer will be backed using sparse memory binding with memory ranges that might also simultaneously be backing another buffer (or another portion of the same buffer). Buffers created with this flag must also be created with the VK_BUFFER_CREATE_SPARSE_BINDING_BIT flag.

• VK_BUFFER_CREATE_PROTECTED_BIT specifies that the buffer is a protected buffer.

• VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT specifies that the buffer’s address can be saved and reused on a subsequent run (e.g. for trace capture and replay), see VkBufferOpaqueCaptureAddressCreateInfo for more detail.

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

typedef VkFlags VkBufferCreateFlags;

VkBufferCreateFlags is a bitmask type for setting a mask of zero or more VkBufferCreateFlagBits.

To define a set of external memory handle types that may be used as backing store for a buffer, add a VkExternalMemoryBufferCreateInfo structure to the pNext chain of the VkBufferCreateInfo structure. The VkExternalMemoryBufferCreateInfo structure is defined as:

typedef struct VkExternalMemoryBufferCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkExternalMemoryHandleTypeFlags handleTypes;
} VkExternalMemoryBufferCreateInfo;

Note

A VkExternalMemoryBufferCreateInfo structure with a non-zero handleTypes field must be included in the creation parameters for a buffer that will be bound to memory that is either exported or imported.

• sType is the type of this structure.
• `pNext` is NULL or a pointer to a structure extending this structure.

• `handleTypes` is zero or a bitmask of `VkExternalMemoryHandleTypeFlagBits` specifying one or more external memory handle types.

### Valid Usage (Implicit)

- VUID-VkExternalMemoryBufferCreateInfo-sType-sType
  - `sType` must be `VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO`
- VUID-VkExternalMemoryBufferCreateInfo-handleTypes-parameter
  - `handleTypes` must be a valid combination of `VkExternalMemoryHandleTypeFlagBits` values

To request a specific device address for a buffer, add a `VkBufferOpaqueCaptureAddressCreateInfo` structure to the `pNext` chain of the `VkBufferCreateInfo` structure. The `VkBufferOpaqueCaptureAddressCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkBufferOpaqueCaptureAddressCreateInfo {
    VkStructureType           sType;
    const void*               pNext;
    uint64_t                   opaqueCaptureAddress;
} VkBufferOpaqueCaptureAddressCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `opaqueCaptureAddress` is the opaque capture address requested for the buffer.

If `opaqueCaptureAddress` is zero, no specific address is requested.

If `opaqueCaptureAddress` is not zero, then it should be an address retrieved from `vkGetBufferOpaqueCaptureAddress` for an identically created buffer on the same implementation.

If this structure is not present, it is as if `opaqueCaptureAddress` is zero.

Apps should avoid creating buffers with app-provided addresses and implementation-provided addresses in the same process, to reduce the likelihood of `VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS` errors.

---

**Note**

The expected usage for this is that a trace capture/replay tool will add the `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT` flag to all buffers that use `VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT`, and during capture will save the queried opaque device addresses in the trace. During replay, the buffers will be created specifying the original address so any address values stored in the trace data will remain valid.
Implementations are expected to separate such buffers in the GPU address space so normal allocations will avoid using these addresses. Apps/tools should avoid mixing app-provided and implementation-provided addresses for buffers created with `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT`, to avoid address space allocation conflicts.

**Valid Usage (Implicit)**

- VUID-VkBufferOpaqueCaptureAddressCreateInfo-sType-sType
  
  `sType` **must** be `VK_STRUCTURE_TYPE_BUFFER_OPAQUE_CAPTURE_ADDRESS_CREATE_INFO`

To destroy a buffer, call:

```cpp
// Provided by VK_VERSION_1_0
void vkDestroyBuffer(
    VkDevice device,  
    VkBuffer buffer,  
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the buffer.
- `buffer` is the buffer to destroy.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

**Valid Usage**

- VUID-vkDestroyBuffer-buffer-00922
  
  All submitted commands that refer to `buffer`, either directly or via a `VkBufferView`, **must** have completed execution

- VUID-vkDestroyBuffer-buffer-00923
  
  If `VkAllocationCallbacks` were provided when `buffer` was created, a compatible set of callbacks **must** be provided here

- VUID-vkDestroyBuffer-buffer-00924
  
  If no `VkAllocationCallbacks` were provided when `buffer` was created, `pAllocator` **must** be `NULL`

**Valid Usage (Implicit)**

- VUID-vkDestroyBuffer-device-parameter
  
  `device` **must** be a valid `VkDevice` handle

- VUID-vkDestroyBuffer-buffer-parameter
  
  If `buffer` is not `VK_NULL_HANDLE`, `buffer` **must** be a valid `VkBuffer` handle

- VUID-vkDestroyBuffer-pAllocator-parameter
If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure

- VUID-vkDestroyBuffer-buffer-parent
  If `buffer` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

### Host Synchronization

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

## 12.2. Buffer Views

A **buffer view** represents a contiguous range of a buffer and a specific format to be used to interpret the data. Buffer views are used to enable shaders to access buffer contents using image operations. In order to create a valid buffer view, the buffer **must** have been created with at least one of the following usage flags:

- `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT`
- `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT`

Buffer views are represented by `VkBufferView` handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkBufferView)
```

To create a buffer view, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateBufferView(
    VkDevice device,
    const VkBufferViewCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkBufferView* pView);
```

- `device` is the logical device that creates the buffer view.
- `pCreateInfo` is a pointer to a `VkBufferViewCreateInfo` structure containing parameters to be used to create the buffer view.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pView` is a pointer to a `VkBufferView` handle in which the resulting buffer view object is returned.
Valid Usage (Implicit)

- **VUID-vkCreateBufferView-device-parameter**
  
  device must be a valid VkDevice handle

- **VUID-vkCreateBufferView-pCreateInfo-parameter**
  pCreateInfo must be a valid pointer to a valid VkBufferViewCreateInfo structure

- **VUID-vkCreateBufferView-pAllocator-parameter**
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- **VUID-vkCreateBufferView-pView-parameter**
  pView must be a valid pointer to a VkBufferView handle

Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkBufferViewCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkBufferViewCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkBufferViewCreateFlags flags;
    VkBuffer buffer;
    VkFormat format;
    VkDeviceSize offset;
    VkDeviceSize range;
} VkBufferViewCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **buffer** is a VkBuffer on which the view will be created.
- **format** is a VkFormat describing the format of the data elements in the buffer.
- **offset** is an offset in bytes from the base address of the buffer. Accesses to the buffer view from shaders use addressing that is relative to this starting offset.
- **range** is a size in bytes of the buffer view. If range is equal to VK_WHOLE_SIZE, the range from...
offset to the end of the buffer is used. If VK_WHOLE_SIZE is used and the remaining size of the buffer is not a multiple of the texel block size of format, the nearest smaller multiple is used.

### Valid Usage

- **VUID-VkBufferViewCreateInfo-offset-00925**
  
  offset must be less than the size of buffer

- **VUID-VkBufferViewCreateInfo-range-00928**
  
  If range is not equal to VK_WHOLE_SIZE, range must be greater than 0

- **VUID-VkBufferViewCreateInfo-range-00929**
  
  If range is not equal to VK_WHOLE_SIZE, range must be an integer multiple of the texel block size of format

- **VUID-VkBufferViewCreateInfo-range-00930**
  
  If range is not equal to VK_WHOLE_SIZE, the number of texel buffer elements given by (\text{\lceil}(\text{size} - \text{offset}) / \text{(texel block size)}\rceil \times \text{(texels per block)}) where size is the size of buffer, and texel block size and texels per block are as defined in the Compatible Formats table for format, must be less than or equal to VkPhysicalDeviceLimits::maxTexelBufferElements

- **VUID-VkBufferViewCreateInfo-offset-00931**
  
  If range is not equal to VK_WHOLE_SIZE, the sum of offset and range must be less than or equal to the size of buffer

- **VUID-VkBufferViewCreateInfo-range-04059**
  
  If range is equal to VK_WHOLE_SIZE, the number of texel buffer elements given by (\text{\lceil}(\text{size} - \text{offset}) / \text{(texel block size)}\rceil \times \text{(texels per block)}) where size is the size of buffer, and texel block size and texels per block are as defined in the Compatible Formats table for format, must be less than or equal to VkPhysicalDeviceLimits::maxTexelBufferElements

- **VUID-VkBufferViewCreateInfo-buffer-00932**
  
  buffer must have been created with a usage value containing at least one of VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT or VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT

- **VUID-VkBufferViewCreateInfo-buffer-00933**
  
  If buffer was created with usage containing VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT, format must be supported for uniform texel buffers, as specified by the VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT flag in VkFormatProperties::bufferFeatures returned by vkGetPhysicalDeviceFormatProperties

- **VUID-VkBufferViewCreateInfo-buffer-00934**
  
  If buffer was created with usage containing VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT, format must be supported for storage texel buffers, as specified by the VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT flag in VkFormatProperties::bufferFeatures returned by vkGetPhysicalDeviceFormatProperties

- **VUID-VkBufferViewCreateInfo-buffer-00935**
  
  If buffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

- **VUID-VkBufferViewCreateInfo-offset-02749**
  
  If the texelBufferAlignment feature is not enabled, offset must be a multiple of
VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment

- VUID-VkBufferViewCreateInfo-buffer-02750
  If the texelBufferAlignment feature is enabled and if buffer was created with usage containing VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT, offset must be a multiple of the lesser of VkPhysicalDeviceTexelBufferAlignmentProperties::storageTexelBufferOffsetAlignmentBytes or, if VkPhysicalDeviceTexelBufferAlignmentProperties::storageTexelBufferOffsetSingleTexelAlignment is VK_TRUE, the size of a texel of the requested format. If the size of a texel is a multiple of three bytes, then the size of a single component of format is used instead.

- VUID-VkBufferViewCreateInfo-buffer-02751
  If the texelBufferAlignment feature is enabled and if buffer was created with usage containing VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT, offset must be a multiple of the lesser of VkPhysicalDeviceTexelBufferAlignmentProperties::uniformTexelBufferOffsetAlignmentBytes or, if VkPhysicalDeviceTexelBufferAlignmentProperties::uniformTexelBufferOffsetSingleTexelAlignment is VK_TRUE, the size of a texel of the requested format. If the size of a texel is a multiple of three bytes, then the size of a single component of format is used instead.

Valid Usage (Implicit)

- VUID-VkBufferViewCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_BUFFER_VIEW_CREATE_INFO

- VUID-VkBufferViewCreateInfo-pNext-pNext
  pNext must be NULL

- VUID-VkBufferViewCreateInfo-flags-zerobitmask
  flags must be 0

- VUID-VkBufferViewCreateInfo-buffer-parameter
  buffer must be a valid VkBuffer handle

- VUID-VkBufferViewCreateInfo-format-parameter
  format must be a valid VkFormat value

// Provided by VK_VERSION_1_0
typedef VkFlags VkBufferViewCreateFlags;

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

To destroy a buffer view, call:

// Provided by VK_VERSION_1_0
void vkDestroyBufferView(
  VkDevice device,
  VkBufferView bufferView,
)
const VkAllocationCallbacks* pAllocator);

- **device** is the logical device that destroys the buffer view.
- **bufferView** is the buffer view to destroy.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage

- VUID-vkDestroyBufferView-bufferView-00936
  All submitted commands that refer to **bufferView** **must** have completed execution

- VUID-vkDestroyBufferView-bufferView-00937
  If **VkAllocationCallbacks** were provided when **bufferView** was created, a compatible set of callbacks **must** be provided here

- VUID-vkDestroyBufferView-bufferView-00938
  If no **VkAllocationCallbacks** were provided when **bufferView** was created, **pAllocator** **must** be **NULL**

### Valid Usage (Implicit)

- VUID-vkDestroyBufferView-device-parameter
  **device** **must** be a valid **VkDevice** handle

- VUID-vkDestroyBufferView-bufferView-parameter
  If **bufferView** is not **VK_NULL_HANDLE**, **bufferView** **must** be a valid **VkBufferView** handle

- VUID-vkDestroyBufferView-pAllocator-parameter
  If **pAllocator** is not **NULL**, **pAllocator** **must** be a valid pointer to a valid **VkAllocationCallbacks** structure

- VUID-vkDestroyBufferView-bufferView-parent
  If **bufferView** is a valid handle, it **must** have been created, allocated, or retrieved from **device**

### Host Synchronization

- Host access to **bufferView** **must** be externally synchronized

### 12.3. Images

Images represent multidimensional - up to 3 - arrays of data which **can** be used for various purposes (e.g. attachments, textures), by binding them to a graphics or compute pipeline via descriptor sets, or by directly specifying them as parameters to certain commands.

Images are represented by **VkImage** handles:
To create images, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateImage(
    VkDevice device,
    const VkImageCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkImage* pImage);
```

- `device` is the logical device that creates the image.
- `pCreateInfo` is a pointer to a `VkImageCreateInfo` structure containing parameters to be used to create the image.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pImage` is a pointer to a `VkImage` handle in which the resulting image object is returned.

### Valid Usage

- **VUID-vkCreateImage-flags-00939**
  If the `flags` member of `pCreateInfo` includes `VK_IMAGE_CREATE_SPARSE_BINDING_BIT`, creating this `VkImage` must not cause the total required sparse memory for all currently valid sparse resources on the device to exceed `VkPhysicalDeviceLimits::sparseAddressSpaceSize`

### Valid Usage (Implicit)

- **VUID-vkCreateImage-device-parameter**
  `device` must be a valid `VkDevice` handle
- **VUID-vkCreateImage-pCreateInfo-parameter**
  `pCreateInfo` must be a valid pointer to a valid `VkImageCreateInfo` structure
- **VUID-vkCreateImage-pAllocator-parameter**
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- **VUID-vkCreateImage-pImage-parameter**
  `pImage` must be a valid pointer to a `VkImage` handle

### Return Codes

**Success**
- `VK_SUCCESS`
The `VkImageCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkImageCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkImageCreateFlags flags;
    VkImageType imageType;
    VkFormat format;
    VkExtent3D extent;
    uint32_t mipLevels;
    uint32_t arrayLayers;
    VkSampleCountFlagBits samples;
    VkImageTiling tiling;
    VkImageUsageFlags usage;
    VkSharingMode sharingMode;
    uint32_t queueFamilyIndexCount;
    const uint32_t* pQueueFamilyIndices;
    VkImageLayout initialLayout;
} VkImageCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **flags** is a bitmask of `VkImageCreateFlagBits` describing additional parameters of the image.
- **imageType** is a `VkImageType` value specifying the basic dimensionality of the image. Layers in array textures do not count as a dimension for the purposes of the image type.
- **format** is a `VkFormat` describing the format and type of the texel blocks that will be contained in the image.
- **extent** is a `VkExtent3D` describing the number of data elements in each dimension of the base level.
- **mipLevels** describes the number of levels of detail available for minified sampling of the image.
- **arrayLayers** is the number of layers in the image.
- **samples** is a `VkSampleCountFlagBits` value specifying the number of samples per texel.
- **tiling** is a `VkImageTiling` value specifying the tiling arrangement of the texel blocks in memory.
- **usage** is a bitmask of `VkImageUsageFlagBits` describing the intended usage of the image.
- **sharingMode** is a `VkSharingMode` value specifying the sharing mode of the image when it will be accessed by multiple queue families.
- **queueFamilyIndexCount** is the number of entries in the `pQueueFamilyIndices` array.
• `pQueueFamilyIndices` is a pointer to an array of queue families that will access this image. It is ignored if `sharingMode` is not `VK_SHARING_MODE_CONCURRENT`.

• `initialLayout` is a `VkImageLayout` value specifying the initial `VkImageLayout` of all image subresources of the image. See Image Layouts.

Images created with `tiling` equal to `VK_IMAGE_TILING_LINEAR` have further restrictions on their limits and capabilities compared to images created with `tiling` equal to `VK_IMAGE_TILING_OPTIMAL`. Creation of images with tiling `VK_IMAGE_TILING_LINEAR` may not be supported unless other parameters meet all of the constraints:

• `imageType` is `VK_IMAGE_TYPE_2D`
• `format` is not a depth/stencil format
• `mipLevels` is 1
• `arrayLayers` is 1
• `samples` is `VK_SAMPLE_COUNT_1_BIT`
• `usage` only includes `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` and/or `VK_IMAGE_USAGE_TRANSFER_DST_BIT`

Images created with one of the formats that require a sampler Y’CbCr conversion, have further restrictions on their limits and capabilities compared to images created with other formats. Creation of images with a format requiring Y’CbCr conversion may not be supported unless other parameters meet all of the constraints:

• `imageType` is `VK_IMAGE_TYPE_2D`
• `mipLevels` is 1
• `arrayLayers` is 1, unless otherwise indicated by `VkImageFormatProperties::maxArrayLayers`, as returned by `vkGetPhysicalDeviceImageFormatProperties`
• `samples` is `VK_SAMPLE_COUNT_1_BIT`

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

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

If the size of the resultant image would exceed `maxResourceSize`, then `vkCreateImage` must fail and return `VK_ERROR_OUT_OF_DEVICE_MEMORY`. This failure may occur even when all image creation parameters satisfy their valid usage requirements.

**Note**

For images created without `VK_IMAGE_CREATE_EXTENDED_USAGE_BIT` a `usage` bit is valid if it is supported for the format the image is created with.

For images created with `VK_IMAGE_CREATE_EXTENDED_USAGE_BIT` a `usage` bit is valid if it is supported for at least one of the formats a `VkImageView` created from the image can have (see Image Views for more detail).
Image Creation Limits

Valid values for some image creation parameters are limited by a numerical upper bound or by inclusion in a bitset. For example, `VkImageCreateInfo::arrayLayers` is limited by `imageCreateMaxArrayLayers`, defined below; and `VkImageCreateInfo::samples` is limited by `imageCreateSampleCounts`, also defined below.

Several limiting values are defined below, as well as assisting values from which the limiting values are derived. The limiting values are referenced by the relevant valid usage statements of `VkImageCreateInfo`.

- Let `VkBool32 imageCreateMaybeLinear` indicate if the resultant image may be linear. (The definition below is trivial because certain extensions are disabled in this build of the specification).
  - If `tiling` is `VK_IMAGE_TILING_LINEAR`, then `imageCreateMaybeLinear` is `VK_TRUE`.
  - If `tiling` is `VK_IMAGE_TILING_OPTIMAL`, then `imageCreateMaybeLinear` is `VK_FALSE`.

- Let `VkFormatFeatureFlags imageCreateFormatFeatures` be the set of valid format features available during image creation.
  - If `tiling` is `VK_IMAGE_TILING_LINEAR`, then `imageCreateFormatFeatures` is the value of `VkFormatProperties::linearTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` with parameter `format` equal to `VkImageCreateInfo::format`.
  - If `tiling` is `VK_IMAGE_TILING_OPTIMAL`, then `imageCreateFormatFeatures` is the value of `VkFormatProperties::optimalTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` with parameter `format` equal to `VkImageCreateInfo::format`.

- Let `VkImageFormatProperties2 imageCreateImageFormatPropertiesList[]` be the list of structures obtained by calling `vkGetPhysicalDeviceImageFormatProperties2`, possibly multiple times, as follows:
  - The parameters `VkPhysicalDeviceImageFormatInfo2::format`, `imageType`, `tiling`, `usage`, and `flags must` be equal to those in `VkImageCreateInfo`.
  - If `VkImageCreateInfo::pNext` contains a `VkExternalMemoryImageCreateInfo` structure whose `handleTypes` is not `0`, then `VkPhysicalDeviceImageFormatInfo2::pNext must` contain a `VkPhysicalDeviceExternalImageFormatInfo` structure whose `handleType` is not `0`; and `vkGetPhysicalDeviceImageFormatProperties2` must be called for each handle type in `VkExternalMemoryImageCreateInfo::handleTypes`, successively setting `VkPhysicalDeviceExternalImageFormatInfo::handleType` on each call.
  - If `VkImageCreateInfo::pNext` contains no `VkExternalMemoryImageCreateInfo` structure, or contains a structure whose `handleTypes` is `0`, then `VkPhysicalDeviceImageFormatInfo2::pNext must` either contain no `VkPhysicalDeviceExternalImageFormatInfo` structure, or contain a structure whose `handleType` is `0`.
  - If any call to `vkGetPhysicalDeviceImageFormatProperties2` returns an error, then `imageCreateImageFormatPropertiesList` is defined to be the empty list.
• Let uint32_t imageCreateMaxMipLevels be the minimum value of VkImageFormatProperties::maxMipLevels in imageCreateImageFormatPropertiesList. The value is undefined if imageCreateImageFormatPropertiesList is empty.

• Let uint32_t imageCreateMaxArrayLayers be the minimum value of VkImageFormatProperties::maxArrayLayers in imageCreateImageFormatPropertiesList. The value is undefined if imageCreateImageFormatPropertiesList is empty.

• Let VkExtent3D imageCreateMaxExtent be the component-wise minimum over all VkImageFormatProperties::maxExtent values in imageCreateImageFormatPropertiesList. The value is undefined if imageCreateImageFormatPropertiesList is empty.

• Let VkSampleCountFlags imageCreateSampleCounts be the intersection of each VkImageFormatProperties::sampleCounts in imageCreateImageFormatPropertiesList. The value is undefined if imageCreateImageFormatPropertiesList is empty.

Valid Usage

• VUID-VkImageCreateInfo-imageCreateMaxMipLevels-02251
  Each of the following values (as described in Image Creation Limits) must not be undefined: imageCreateMaxMipLevels, imageCreateMaxArrayLayers, imageCreateMaxExtent, and imageCreateSampleCounts

• VUID-VkImageCreateInfo-sharingMode-00941
  If sharingMode is VK_SHARING_MODE_CONCURRENT, pQueueFamilyIndices must be a valid pointer to an array of queueFamilyIndexCount uint32_t values

• VUID-VkImageCreateInfo-sharingMode-00942
  If sharingMode is VK_SHARING_MODE_CONCURRENT, queueFamilyIndexCount must be greater than 1

• VUID-VkImageCreateInfo-sharingMode-01420
  If sharingMode is VK_SHARING_MODE_CONCURRENT, each element of pQueueFamilyIndices must be unique and must be less than pQueueFamilyPropertyCount returned by either vkGetPhysicalDeviceQueueFamilyProperties or vkGetPhysicalDeviceQueueFamilyProperties2 for the physicalDevice that was used to create device

• VUID-VkImageCreateInfo-format-00943
  format must not be VK_FORMAT_UNDEFINED

• VUID-VkImageCreateInfo-extent-00944
  extent.width must be greater than 0

• VUID-VkImageCreateInfo-extent-00945
  extent.height must be greater than 0

• VUID-VkImageCreateInfo-extent-00946
  extent.depth must be greater than 0

• VUID-VkImageCreateInfo-mipLevels-00947
  mipLevels must be greater than 0

• VUID-VkImageCreateInfo-arrayLayers-00948
arrayLayers must be greater than 0

- **VUID-VkImageCreateInfo-flags-00949**
  If flags contains VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT, imageType must be VK_IMAGE_TYPE_2D

- **VUID-VkImageCreateInfo-flags-00950**
  If flags contains VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT, imageType must be VK_IMAGE_TYPE_3D

- **VUID-VkImageCreateInfo-extent-02252**
  extent.width must be less than or equal to imageCreateMaxExtent.width (as defined in Image Creation Limits)

- **VUID-VkImageCreateInfo-extent-02253**
  extent.height must be less than or equal to imageCreateMaxExtent.height (as defined in Image Creation Limits)

- **VUID-VkImageCreateInfo-extent-02254**
  extent.depth must be less than or equal to imageCreateMaxExtent.depth (as defined in Image Creation Limits)

- **VUID-VkImageCreateInfo-imageType-00954**
  If imageType is VK_IMAGE_TYPE_2D and flags contains VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT, extent.width and extent.height must be equal and arrayLayers must be greater than or equal to 6

- **VUID-VkImageCreateInfo-imageType-00956**
  If imageType is VK_IMAGE_TYPE_1D, both extent.height and extent.depth must be 1

- **VUID-VkImageCreateInfo-imageType-00957**
  If imageType is VK_IMAGE_TYPE_2D, extent.depth must be 1

- **VUID-VkImageCreateInfo-mipLevels-00958**
  mipLevels must be less than or equal to the number of levels in the complete mipmap chain based on extent.width, extent.height, and extent.depth

- **VUID-VkImageCreateInfo-mipLevels-02255**
  mipLevels must be less than or equal to imageCreateMaxMipLevels (as defined in Image Creation Limits)

- **VUID-VkImageCreateInfo-arrayLayers-02256**
  arrayLayers must be less than or equal to imageCreateMaxArrayLayers (as defined in Image Creation Limits)

- **VUID-VkImageCreateInfo-imageType-00961**
  If imageType is VK_IMAGE_TYPE_3D, arrayLayers must be 1

- **VUID-VkImageCreateInfo-samples-02257**
  If samples is not VK_SAMPLE_COUNT_1_BIT, then imageType must be VK_IMAGE_TYPE_2D, flags must not contain VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT, mipLevels must be equal to 1, and imageCreateMaybeLinear (as defined in Image Creation Limits) must be VK_FALSE,

- **VUID-VkImageCreateInfo-usage-00963**
  If usage includes VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT, then bits other than VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, and
VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT must not be set

- VUID-VkImageCreateInfo-usage-00964
  If usage includes VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT, or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT, extent.width must be less than or equal to VkPhysicalDeviceLimits::maxFramebufferWidth

- VUID-VkImageCreateInfo-usage-00965
  If usage includes VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT, or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT, extent.height must be less than or equal to VkPhysicalDeviceLimits::maxFramebufferHeight

- VUID-VkImageCreateInfo-usage-00966
  If usage includes VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT, usage must also contain at least one of VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT

- VUID-VkImageCreateInfo-samples-02258
  samples must be a bit value that is set in imageCreateSampleCounts (as defined in Image Creation Limits)

- VUID-VkImageCreateInfo-usage-00968
  If the shaderStorageImageMultisample feature is not enabled, and usage contains VK_IMAGE_USAGE_STORAGE_BIT, samples must be VK_SAMPLE_COUNT_1_BIT

- VUID-VkImageCreateInfo-flags-00969
  If the sparseBinding feature is not enabled, flags must not contain VK_IMAGE_CREATE_SPARSE_BINDING_BIT

- VUID-VkImageCreateInfo-flags-01924
  If the sparseResidencyAliased feature is not enabled, flags must not contain VK_IMAGE_CREATE_SPARSE_ALIASED_BIT

- VUID-VkImageCreateInfo-tiling-04121
  If tiling is VK_IMAGE_TILING_LINEAR, flags must not contain VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT

- VUID-VkImageCreateInfo-imageType-00970
  If imageType is VK_IMAGE_TYPE_1D, flags must not contain VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT

- VUID-VkImageCreateInfo-imageType-00971
  If the sparseResidencyImage2D feature is not enabled, and imageType is VK_IMAGE_TYPE_2D, flags must not contain VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT

- VUID-VkImageCreateInfo-imageType-00972
  If the sparseResidencyImage3D feature is not enabled, and imageType is VK_IMAGE_TYPE_3D, flags must not contain VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT

- VUID-VkImageCreateInfo-imageType-00973
  If the sparseResidency2Samples feature is not enabled, imageType is VK_IMAGE_TYPE_2D, and samples is VK_SAMPLE_COUNT_2_BIT, flags must not contain VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT
If the `sparseResidency4Samples` feature is not enabled, `imageType` is `VK_IMAGE_TYPE_2D`, and `samples` is `VK_SAMPLE_COUNT_4_BIT`, `flags` must not contain `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.

If the `sparseResidency8Samples` feature is not enabled, `imageType` is `VK_IMAGE_TYPE_2D`, and `samples` is `VK_SAMPLE_COUNT_8_BIT`, `flags` must not contain `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.

If the `sparseResidency16Samples` feature is not enabled, `imageType` is `VK_IMAGE_TYPE_2D`, and `samples` is `VK_SAMPLE_COUNT_16_BIT`, `flags` must not contain `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.

If `flags` contains `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` or `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT`, it must also contain `VK_IMAGE_CREATE_SPARSE_BINDING_BIT`.

If any of the bits `VK_IMAGE_CREATE_SPARSE_BINDING_BIT`, `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`, or `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT` are set, `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT` must not also be set.

If the `protectedMemory` feature is not enabled, `flags` must not contain `VK_IMAGE_CREATE_PROTECTED_BIT`.

If any of the bits `VK_IMAGE_CREATE_SPARSE_BINDING_BIT`, `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`, or `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT` are set, `VK_IMAGE_CREATE_PROTECTED_BIT` must not also be set.

If the `pNext` chain includes a `VkExternalMemoryImageCreateInfo` structure, its `handleTypes` member must only contain bits that are also in `VkExternalImageFormatProperties::externalMemoryProperties.compatibleHandleTypes`, as returned by `vkGetPhysicalDeviceImageFormatProperties2` with `format`, `imageType`, `tiling`, `usage`, and `flags` equal to those in this structure, and with a `VkPhysicalDeviceExternalImageFormatInfo` structure included in the `pNext` chain, with a `handleType` equal to any one of the handle types specified in `VkExternalMemoryImageCreateInfo::handleTypes`.

If the logical device was created with `VkDeviceGroupDeviceCreateInfo::physicalDeviceCount` equal to 1, `flags` must not contain `VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT`.

If `flags` contains `VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT`, then `mipLevels` must be one, `arrayLayers` must be one, `imageType` must be `VK_IMAGE_TYPE_2D`, and `imageCreateMaybeLinear` (as defined in Image Creation Limits) must be `VK_FALSE`.
• VUID-VkImageCreateInfo-flags-01572
  If `flags` contains `VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT`, then `format` must be a compressed image format

• VUID-VkImageCreateInfo-flags-01573
  If `flags` contains `VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT`, then `flags` must also contain `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT`

• VUID-VkImageCreateInfo-initialLayout-00993
  `initialLayout` must be `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`

• VUID-VkImageCreateInfo-pNext-01443
  If the `pNext` chain includes a `VkExternalMemoryImageCreateInfo` or `VkExternalMemoryImageCreateInfoNV` structure whose `handleTypes` member is not 0, `initialLayout` must be `VK_IMAGE_LAYOUT_UNDEFINED`

• VUID-VkImageCreateInfo-format-06410
  If the image `format` is one of the formats that require a sampler Y'CbCr conversion, `mipLevels` must be 1

• VUID-VkImageCreateInfo-format-06411
  If the image `format` is one of the formats that require a sampler Y'CbCr conversion, `samples` must be `VK_SAMPLE_COUNT_1_BIT`

• VUID-VkImageCreateInfo-format-06412
  If the image `format` is one of the formats that require a sampler Y'CbCr conversion, `imageType` must be `VK_IMAGE_TYPE_2D`

• VUID-VkImageCreateInfo-imageCreateFormatFeatures-02260
  If `format` is a `multi-planar` format, and if `imageCreateFormatFeatures` (as defined in Image Creation Limits) does not contain `VK_FORMAT_FEATURE_DISJOINT_BIT`, then `flags` must not contain `VK_IMAGE_CREATE_DISJOINT_BIT`

• VUID-VkImageCreateInfo-format-01577
  If `format` is not a `multi-planar` format, and `flags` does not include `VK_IMAGE_CREATE_ALIAS_BIT`, `flags` must not contain `VK_IMAGE_CREATE_DISJOINT_BIT`

• VUID-VkImageCreateInfo-format-04712
  If `format` has a `_422` or `_420` suffix, `width` must be a multiple of 2

• VUID-VkImageCreateInfo-format-04713
  If `format` has a `_420` suffix, `height` must be a multiple of 2

• VUID-VkImageCreateInfo-format-02795
  If `format` is a depth-stencil format, `usage` includes `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, and the `pNext` chain includes a `VkImageStencilUsageCreateInfo` structure, then its `VkImageStencilUsageCreateInfo::stencilUsage` member must also include `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`

• VUID-VkImageCreateInfo-format-02796
  If `format` is a depth-stencil format, `usage` does not include `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, and the `pNext` chain includes a `VkImageStencilUsageCreateInfo` structure, then its `VkImageStencilUsageCreateInfo::stencilUsage` member must also not include `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`
• **VUID-VkImageCreateInfo-format-02797**
  If `format` is a depth-stencil format, `usage` includes `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`, and the `pNext` chain includes a `VkImageStencilUsageCreateInfo` structure, then its `VkImageStencilUsageCreateInfo::stencilUsage` member **must** also include `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`.

• **VUID-VkImageCreateInfo-format-02798**
  If `format` is a depth-stencil format, `usage` does not include `VK_IMAGEUSAGE_TRANSIENT_ATTACHMENT_BIT`, and the `pNext` chain includes a `VkImageStencilUsageCreateInfo` structure, then its `VkImageStencilUsageCreateInfo::stencilUsage` member **must** also not include `VK_IMAGEUSAGE_TRANSIENT_ATTACHMENT_BIT`.

• **VUID-VkImageCreateInfo-Format-02536**
  If `Format` is a depth-stencil format and the `pNext` chain includes a `VkImageStencilUsageCreateInfo` structure with its `stencilUsage` member including `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`, `extent.width` **must** be less than or equal to `VkPhysicalDeviceLimits::maxFramebufferWidth`.

• **VUID-VkImageCreateInfo-format-02537**
  If `format` is a depth-stencil format and the `pNext` chain includes a `VkImageStencilUsageCreateInfo` structure with its `stencilUsage` member including `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`, `extent.height` **must** be less than or equal to `VkPhysicalDeviceLimits::maxFramebufferHeight`.

• **VUID-VkImageCreateInfo-format-02538**
  If the `shaderStorageImageMultisample` feature is not enabled, `format` is a depth-stencil format and the `pNext` chain includes a `VkImageStencilUsageCreateInfo` structure with its `stencilUsage` including `VK_IMAGE_USAGE_STORAGE_BIT`, `samples` **must** be `VK_SAMPLE_COUNT_1_BIT`.

• **VUID-VkImageCreateInfo-pNext-06722**
  If a `VkImageFormatListCreateInfo` structure was included in the `pNext` chain and `VkImageFormatListCreateInfo::viewFormatCount` is not zero, then each format in `VkImageFormatListCreateInfo::pViewFormats` **must** either be compatible with the `format` as described in the compatibility table or, if `flags` contains `VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT`, be an uncompressed format that is size-compatible with `format`.

• **VUID-VkImageCreateInfo-flags-04738**
  If `flags` does not contain `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` and the `pNext` chain includes a `VkImageFormatListCreateInfo` structure, then `VkImageFormatListCreateInfo::viewFormatCount` **must** be 0 or 1.

---

**Valid Usage (Implicit)**

• **VUID-VkImageCreateInfo-sType-sType**
  `sType` **must** be `VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO`.

• **VUID-VkImageCreateInfo-pNext-pNext**
  Each `pNext` member of any structure (including this one) in the `pNext` chain **must** be either `NULL` or a pointer to a valid instance of `VkExternalMemoryImageCreateInfo`,
The `VkImageStencilUsageCreateInfo` structure is defined as:

```
// Provided by VK_VERSION_1_2
typedef struct VkImageStencilUsageCreateInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkImageUsageFlags         stencilUsage;
} VkImageStencilUsageCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `stencilUsage` is a bitmask of `VkImageUsageFlagBits` describing the intended usage of the stencil aspect of the image.

If the `pNext` chain of `VkImageCreateInfo` includes a `VkImageStencilUsageCreateInfo` structure, then that structure includes the usage flags specific to the stencil aspect of the image for an image with a depth-stencil format.

This structure specifies image usages which only apply to the stencil aspect of a depth/stencil format image. When this structure is included in the `pNext` chain of `VkImageCreateInfo`, the stencil
aspect of the image must only be used as specified by stencilUsage. When this structure is not included in the pNext chain of VkImageCreateInfo, the stencil aspect of an image must only be used as specified by VkImageCreateInfo::usage. Use of other aspects of an image are unaffected by this structure.

This structure can also be included in the pNext chain of VkPhysicalDeviceImageFormatInfo2 to query additional capabilities specific to image creation parameter combinations including a separate set of usage flags for the stencil aspect of the image using vkGetPhysicalDeviceImageFormatProperties2. When this structure is not included in the pNext chain of VkPhysicalDeviceImageFormatInfo2 then the implicit value of stencilUsage matches that of VkPhysicalDeviceImageFormatInfo2::usage.

**Valid Usage**

- VUID-VkImageStencilUsageCreateInfo-stencilUsage-02539
  If stencilUsage includes VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT, it must not include bits other than VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT

**Valid Usage (Implicit)**

- VUID-VkImageStencilUsageCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_IMAGE_STENCIL_USAGE_CREATE_INFO
- VUID-VkImageStencilUsageCreateInfo-stencilUsage-parameter
  stencilUsage must be a valid combination of VkImageUsageFlagBits values
- VUID-VkImageStencilUsageCreateInfo-stencilUsage-requiredbitmask
  stencilUsage must not be 0

To define a set of external memory handle types that may be used as backing store for an image, add a VkExternalMemoryImageCreateInfo structure to the pNext chain of the VkImageCreateInfo structure. The VkExternalMemoryImageCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkExternalMemoryImageCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkExternalMemoryHandleTypeFlags handleTypes;
} VkExternalMemoryImageCreateInfo;
```

**Note**

A VkExternalMemoryImageCreateInfo structure with a non-zero handleTypes field must be included in the creation parameters for an image that will be bound to memory that is either exported or imported.
• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **handleTypes** is zero or a bitmask of **VkExternalMemoryHandleTypeFlagBits** specifying one or more external memory handle types.

### Valid Usage (Implicit)

- VUID-VkExternalMemoryImageCreateInfo-sType-sType
  sType **must** be **VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO**
- VUID-VkExternalMemoryImageCreateInfo-handleTypes-parameter
  handleTypes **must** be a valid combination of **VkExternalMemoryHandleTypeFlagBits** values

If the **pNext** chain of **VkImageCreateInfo** includes a **VkImageFormatListCreateInfo** structure, then that structure contains a list of all formats that **can** be used when creating views of this image.

The **VkImageFormatListCreateInfo** structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkImageFormatListCreateInfo {
    VkStructureType     sType;
    const void*         pNext;
    uint32_t            viewFormatCount;
    const VkFormat*     pViewFormats;
} VkImageFormatListCreateInfo;
```

• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **viewFormatCount** is the number of entries in the **pViewFormats** array.
• **pViewFormats** is a pointer to an array of **VkFormat** values specifying all formats which **can** be used when creating views of this image.

If **viewFormatCount** is zero, **pViewFormats** is ignored and the image is created as if the **VkImageFormatListCreateInfo** structure were not included in the **pNext** chain of **VkImageCreateInfo**.

### Valid Usage (Implicit)

- VUID-VkImageFormatListCreateInfo-sType-sType
  sType **must** be **VK_STRUCTURE_TYPE_IMAGE_FORMAT_LIST_CREATE_INFO**
- VUID-VkImageFormatListCreateInfo-pViewFormats-parameter
  If **viewFormatCount** is not 0, **pViewFormats** **must** be a valid pointer to an array of **viewFormatCount** valid **VkFormat** values
Bits which can be set in

- VkImageViewUsageCreateInfo::usage
- VkImageStencilUsageCreateInfo::stencilUsage
- VkImageCreateInfo::usage

specify intended usage of an image, and are:

```c
// Provided by VK_VERSION_1_0
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;
```

- `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` specifies that the image can be used as the source of a transfer command.
- `VK_IMAGE_USAGE_TRANSFER_DST_BIT` specifies that the image can be used as the destination of a transfer command.
- `VK_IMAGE_USAGE_SAMPLED_BIT` specifies that the image can be used to create a VkImageView suitable for occupying a VkDescriptorSet slot either of type `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, and be sampled by a shader.
- `VK_IMAGE_USAGE_STORAGE_BIT` specifies that the image can be used to create a VkImageView suitable for occupying a VkDescriptorSet slot of type `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`.
- `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` specifies that the image can be used to create a VkImageView suitable for use as a color or resolve attachment in a VkFramebuffer.
- `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT` specifies that the image can be used to create a VkImageView suitable for use as a depth/stencil or depth/stencil resolve attachment in a VkFramebuffer.
- `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT` specifies that implementations may support using memory allocations with the `VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT` to back an image with this usage. This bit can be set for any image that can be used to create a VkImageView suitable for use as a color, resolve, depth/stencil, or input attachment.
- `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT` specifies that the image can be used to create a VkImageView suitable for occupying VkDescriptorSet slot of type `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`; be read from a shader as an input attachment; and be used as an input attachment in a framebuffer.

```c
// Provided by VK_VERSION_1_0
```


typedef VkFlags VkImageUsageFlags;

VkImageUsageFlags is a bitmask type for setting a mask of zero or more VkImageUsageFlagBits.

When creating a VkImageView one of the following VkImageUsageFlagBits must be set:

- VK_IMAGE_USAGE_SAMPLED_BIT
- VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
- VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
- VK_IMAGE_USAGE_STORAGE_BIT
- VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT
- VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT

Bits which can be set in VkImageCreateInfo::flags, specifying additional parameters of an image, are:

```cpp
// Provided by VK_VERSION_1_0
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,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_ALIAS_BIT = 0x00000400,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT = 0x00000040,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT = 0x00000020,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT = 0x00000080,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_EXTENDED_USAGE_BIT = 0x00000100,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_PROTECTED_BIT = 0x00000800,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_DISJOINT_BIT = 0x00000200,
} VkImageCreateFlagBits;
```

- VK_IMAGE_CREATE_SPARSE_BINDING_BIT specifies that the image will be backed using sparse memory binding.
- VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT specifies that the image can be partially backed using sparse memory binding. Images created with this flag must also be created with the VK_IMAGE_CREATE_SPARSE_BINDING_BIT flag.
- VK_IMAGE_CREATE_SPARSE_ALIASED_BIT specifies that the image will be backed using sparse memory binding with memory ranges that might also simultaneously be backing another image
Images created with this flag must also be created with the \texttt{VK\_IMAGE\_CREATE\_SPARSE\_BINDING\_BIT} flag.

- \texttt{VK\_IMAGE\_CREATE\_Mutable\_Format\_BIT} specifies that the image can be used to create a \texttt{VkImageView} with a different format from the image. For \textit{multi-planar} formats, \texttt{VK\_IMAGE\_CREATE\_Mutable\_Format\_BIT} specifies that a \texttt{VkImageView} can be created of a \textit{plane} of the image.

- \texttt{VK\_IMAGE\_CREATE\_Cube\_Compatible\_BIT} specifies that the image can be used to create a \texttt{VkImageView} of type \texttt{VK\_IMAGE\_VIEW\_TYPE\_Cube} or \texttt{VK\_IMAGE\_VIEW\_TYPE\_Cube\_Array}.

- \texttt{VK\_IMAGE\_CREATE\_2D\_Array\_Compatible\_BIT} specifies that the image can be used to create a \texttt{VkImageView} of type \texttt{VK\_IMAGE\_VIEW\_TYPE\_2D} or \texttt{VK\_IMAGE\_VIEW\_TYPE\_2D\_Array}.

- \texttt{VK\_IMAGE\_CREATE\_Protected\_BIT} specifies that the image is a protected image.

- \texttt{VK\_IMAGE\_CREATE\_Split\_Instance\_Bind\_Regions\_BIT} specifies that the image can be used with a non-zero value of the \texttt{splitInstanceBindRegionCount} member of a \texttt{VkBindImageMemoryDeviceGroupInfo} structure passed into \texttt{vkBindImageMemory2}. This flag also has the effect of making the image use the standard sparse image block dimensions.

- \texttt{VK\_IMAGE\_CREATE\_Block\_Texel\_View\_Compatible\_BIT} specifies that the image having a compressed format can be used to create a \texttt{VkImageView} with an uncompressed format where each texel in the image view corresponds to a compressed texel block of the image.

- \texttt{VK\_IMAGE\_CREATE\_Extended\_Usage\_BIT} specifies that the image can be created with usage flags that are not supported for the format the image is created with but are supported for at least one format a \texttt{VkImageView} created from the image can have.

- \texttt{VK\_IMAGE\_CREATE\_Disjoint\_BIT} specifies that an image with a \textit{multi-planar format} must have each plane separately bound to memory, rather than having a single memory binding for the whole image; the presence of this bit distinguishes a \textit{disjoint image} from an image without this bit set.

- \texttt{VK\_IMAGE\_CREATE\_Alias\_BIT} specifies that two images created with the same creation parameters and aliased to the same memory can interpret the contents of the memory consistently with each other, subject to the rules described in the \textit{Memory Aliasing} section. This flag further specifies that each plane of a \textit{disjoint} image can share an in-memory non-linear representation with single-plane images, and that a single-plane image can share an in-memory non-linear representation with a plane of a multi-planar disjoint image, according to the rules in \textit{Compatible formats of planes of multi-planar formats}. If the \texttt{pNext} chain includes a \texttt{VkExternalMemoryImageCreateInfo} structure whose \texttt{handleTypes} member is not \texttt{0}, it is as if \texttt{VK\_IMAGE\_CREATE\_Alias\_BIT} is set.

See \textit{Sparse Resource Features} and \textit{Sparse Physical Device Features} for more details.

```c
// Provided by VK\_VERSION\_1\_0
typedef VkFlags VkImageCreateFlags;
```

\texttt{VkImageCreateFlags} is a bitmask type for setting a mask of zero or more \texttt{VkImageCreateFlagBits}.

Possible values of \texttt{VkImageCreateInfo::imageType}, specifying the basic dimensionality of an image, are:
typedef enum VkImageType {
    VK_IMAGE_TYPE_1D = 0,
    VK_IMAGE_TYPE_2D = 1,
    VK_IMAGE_TYPE_3D = 2,
} VkImageType;

- VK_IMAGE_TYPE_1D specifies a one-dimensional image.
- VK_IMAGE_TYPE_2D specifies a two-dimensional image.
- VK_IMAGE_TYPE_3D specifies a three-dimensional image.

Possible values of VkImageCreateInfo::tiling, specifying the tiling arrangement of texel blocks in an image, are:

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

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

To query the memory layout of an image subresource, call:

void vkGetImageSubresourceLayout(
    VkDevice device,   
    VkImage image,     
    const VkImageSubresource* pSubresource,   
    VkSubresourceLayout* pLayout);

- device is the logical device that owns the image.
- image is the image whose layout is being queried.
- pSubresource is a pointer to a VkImageSubresource structure selecting a specific image subresource from the image.
- pLayout is a pointer to a VkSubresourceLayout structure in which the layout is returned.

The image must be linear. The returned layout is valid for host access.

If the image’s format is a multi-planar format, then vkGetImageSubresourceLayout describes one plane of the image.
vkGetImageSubresourceLayout is invariant for the lifetime of a single image.

### Valid Usage

- **VUID-vkGetImageSubresourceLayout-image-00996**
  - Image must have been created with tiling equal to `VK_IMAGE_TILING_LINEAR`

- **VUID-vkGetImageSubresourceLayout-aspectMask-00997**
  - The aspectMask member of pSubresource must only have a single bit set

- **VUID-vkGetImageSubresourceLayout-mipLevel-01716**
  - The mipLevel member of pSubresource must be less than the mipLevels specified in VkImageCreateInfo when image was created

- **VUID-vkGetImageSubresourceLayout-arrayLayer-01717**
  - The arrayLayer member of pSubresource must be less than the arrayLayers specified in VkImageCreateInfo when image was created

- **VUID-vkGetImageSubresourceLayout-format-04461**
  - If format is a color format, the aspectMask member of pSubresource must be `VK_IMAGE_ASPECT_COLOR_BIT`

- **VUID-vkGetImageSubresourceLayout-format-04462**
  - If format has a depth component, the aspectMask member of pSubresource must contain `VK_IMAGE_ASPECT_DEPTH_BIT`

- **VUID-vkGetImageSubresourceLayout-format-04463**
  - If format has a stencil component, the aspectMask member of pSubresource must contain `VK_IMAGE_ASPECT_STENCIL_BIT`

- **VUID-vkGetImageSubresourceLayout-format-04464**
  - If format does not contain a stencil or depth component, the aspectMask member of pSubresource must not contain `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT`

- **VUID-vkGetImageSubresourceLayout-format-01581**
  - If the tiling of the image is `VK_IMAGE_TILING_LINEAR` and its format is a multi-planar format with two planes, the aspectMask member of pSubresource must be `VK_IMAGE_ASPECT_PLANE_0_BIT` or `VK_IMAGE_ASPECT_PLANE_1_BIT`

- **VUID-vkGetImageSubresourceLayout-format-01582**
  - If the tiling of the image is `VK_IMAGE_TILING_LINEAR` and its format is a multi-planar format with three planes, the aspectMask member of pSubresource must be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT` or `VK_IMAGE_ASPECT_PLANE_2_BIT`

### Valid Usage (Implicit)

- **VUID-vkGetImageSubresourceLayout-device-parameter**
  - Device must be a valid VkDevice handle

- **VUID-vkGetImageSubresourceLayout-image-parameter**
  - Image must be a valid VkImage handle

- **VUID-vkGetImageSubresourceLayout-pSubresource-parameter**

---

496
pSubresource must be a valid pointer to a valid VkImageSubresource structure

- VUID-vkGetImageSubresourceLayout-pLayout-parameter
  pLayout must be a valid pointer to a VkSubresourceLayout structure
- VUID-vkGetImageSubresourceLayout-image-parent
  image must have been created, allocated, or retrieved from device

The VkImageSubresource structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkImageSubresource {
    VkImageAspectFlags aspectMask;
    uint32_t mipLevel;
    uint32_t arrayLayer;
} VkImageSubresource;
```

- aspectMask is a VkImageAspectFlags value selecting the image aspect.
- mipLevel selects the mipmap level.
- arrayLayer selects the array layer.

Valid Usage (Implicit)

- VUID-VkImageSubresource-aspectMask-parameter
  aspectMask must be a valid combination of VkImageAspectFlagBits values
- VUID-VkImageSubresource-aspectMask-requiredbitmask
  aspectMask must not be 0

Information about the layout of the image subresource is returned in a VkSubresourceLayout structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSubresourceLayout {
    VkDeviceSize offset;
    VkDeviceSize size;
    VkDeviceSize rowPitch;
    VkDeviceSize arrayPitch;
    VkDeviceSize depthPitch;
} VkSubresourceLayout;
```

- offset is the byte offset from the start of the image or the plane where the image subresource begins.
- size is the size in bytes of the image subresource. size includes any extra memory that is required based on rowPitch.
- rowPitch describes the number of bytes between each row of texels in an image.
arrayPitch describes the number of bytes between each array layer of an image.

depthPitch describes the number of bytes between each slice of 3D image.

If the image is linear, then rowPitch, arrayPitch and depthPitch describe the layout of the image subresource in linear memory. For uncompressed formats, rowPitch is the number of bytes between texels with the same x coordinate in adjacent rows (y coordinates differ by one). arrayPitch is the number of bytes between texels with the same x and y coordinate in adjacent array layers of the image (array layer values differ by one). depthPitch is the number of bytes between texels with the same x and y coordinate in adjacent slices of a 3D image (z coordinates differ by one). Expressed as an addressing formula, the starting byte of a texel in the image subresource has address:

```
// (x,y,z,layer) are in texel coordinates
address(x,y,z,layer) = layer*arrayPitch + z*depthPitch + y*rowPitch + x*elementSize + offset
```

For compressed formats, the rowPitch is the number of bytes between compressed texel blocks in adjacent rows. arrayPitch is the number of bytes between compressed texel blocks in adjacent array layers. depthPitch is the number of bytes between compressed texel blocks in adjacent slices of a 3D image.

```
// (x,y,z,layer) are in compressed texel block coordinates
address(x,y,z,layer) = layer*arrayPitch + z*depthPitch + y*rowPitch + x
*compressedTexelBlockByteSize + offset;
```

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

If the image has a single-plane color format, then the aspectMask member of VkImageSubresource must be VK_IMAGE_ASPECT_COLOR_BIT.

If the image has a depth/stencil format, then aspectMask must be either VK_IMAGE_ASPECT_DEPTH_BIT or VK_IMAGE_ASPECT_STENCIL_BIT. On implementations that store depth and stencil aspects separately, querying each of these image subresource layouts will return a different offset and size representing the region of memory used for that aspect. On implementations that store depth and stencil aspects interleaved, the same offset and size are returned and represent the interleaved memory allocation.

If the image has a multi-planar format, then the aspectMask member of VkImageSubresource must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or (for 3-plane formats only) VK_IMAGE_ASPECT_PLANE_2_BIT. Querying each of these image subresource layouts will return a different offset and size representing the region of memory used for that plane. If the image is disjoint, then the offset is relative to the base address of the plane. If the image is non-disjoint, then the offset is relative to the base address of the image.

To destroy an image, call:

```
// Provided by VK_VERSION_1_0
```
```c
void vkDestroyImage(
    VkDevice device,  
    VkImage image,   
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the image.
- `image` is the image to destroy.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage

- **VUID-vkDestroyImage-image-01000**
  All submitted commands that refer to `image`, either directly or via a `VkImageView`, must have completed execution

- **VUID-vkDestroyImage-image-01001**
  If `VkAllocationCallbacks` were provided when `image` was created, a compatible set of callbacks must be provided here

- **VUID-vkDestroyImage-image-01002**
  If no `VkAllocationCallbacks` were provided when `image` was created, `pAllocator` must be `NULL`

### Valid Usage (Implicit)

- **VUID-vkDestroyImage-device-parameter**
  `device` must be a valid `VkDevice` handle

- **VUID-vkDestroyImage-image-parameter**
  If `image` is not `VK_NULL_HANDLE`, `image` must be a valid `VkImage` handle

- **VUID-vkDestroyImage-pAllocator-parameter**
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- **VUID-vkDestroyImage-image-parent**
  If `image` is a valid handle, it must have been created, allocated, or retrieved from `device`

### Host Synchronization

- Host access to `image` must be externally synchronized

### 12.3.1. Image Format Features

Valid uses of a `VkImage` may depend on the image’s `format features`, defined below. Such constraints are documented in the affected valid usage statement.
• If the image was created with `VK_IMAGE_TILING_LINEAR`, then its set of `format features` is the value of `VkFormatProperties::linearTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` on the same `format` as `VkImageCreateInfo::format`.

• If the image was created with `VK_IMAGE_TILING_OPTIMAL`, then its set of `format features` is the value of `VkFormatProperties::optimalTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` on the same `format` as `VkImageCreateInfo::format`.

### 12.3.2. Image Miplevel Sizing

A complete mipmap chain is the full set of miplevels, from the largest mipmap provided, down to the minimum mipmap size.

#### Conventional Images

For conventional images, the dimensions of each successive mipmap, \( n+1 \), are:

\[
\begin{align*}
    \text{width}_{n+1} &= \max(\lfloor \text{width}_n / 2 \rfloor, 1) \\
    \text{height}_{n+1} &= \max(\lfloor \text{height}_n / 2 \rfloor, 1) \\
    \text{depth}_{n+1} &= \max(\lfloor \text{depth}_n / 2 \rfloor, 1)
\end{align*}
\]

where \( \text{width}_n, \text{height}_n, \) and \( \text{depth}_n \) are the dimensions of the next larger mipmap, \( n \).

The minimum mipmap size is:

- 1 for one-dimensional images,
- 1x1 for two-dimensional images, and
- 1x1x1 for three-dimensional images.

The number of levels in a complete mipmap chain is:

\[
\lceil \log_2(\max(\text{width}_0, \text{height}_0, \text{depth}_0)) \rceil + 1
\]

where \( \text{width}_0, \text{height}_0, \) and \( \text{depth}_0 \) are the dimensions of the largest (most detailed) mipmap, \( 0 \).

### 12.4. Image Layouts

Images are stored in implementation-dependent opaque layouts in memory. Each layout has limitations on what kinds of operations are supported for image subresources using the layout. At any given time, the data representing an image subresource in memory exists in a particular layout which is determined by the most recent layout transition that was performed on that image subresource. Applications have control over which layout each image subresource uses, and can transition an image subresource from one layout to another. Transitions can happen with an image...
memory barrier, included as part of a `vkCmdPipelineBarrier` or a `vkCmdWaitEvents` command buffer command (see Image Memory Barriers), or as part of a subpass dependency within a render pass (see VkSubpassDependency).

Image layout is per-image subresource. Separate image subresources of the same image can be in different layouts at the same time, with the exception that depth and stencil aspects of a given image subresource can only be in different layouts if the `separateDepthStencilLayouts` feature is enabled.

Note

Each layout may offer optimal performance for a specific usage of image memory. For example, an image with a layout of `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` may provide optimal performance for use as a color attachment, but be unsupported for use in transfer commands. Applications can transition an image subresource from one layout to another in order to achieve optimal performance when the image subresource is used for multiple kinds of operations. After initialization, applications need not use any layout other than the general layout, though this may produce suboptimal performance on some implementations.

Upon creation, all image subresources of an image are initially in the same layout, where that layout is selected by the `VkImageCreateInfo::initialLayout` member. The `initialLayout` must be either `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`. If it is `VK_IMAGE_LAYOUT_PREINITIALIZED`, then the image data can be preinitialized by the host while using this layout, and the transition away from this layout will preserve that data. If it is `VK_IMAGE_LAYOUT_UNDEFINED`, then the contents of the data are considered to be undefined, and the transition away from this layout is not guaranteed to preserve that data. For either of these initial layouts, any image subresources must be transitioned to another layout before they are accessed by the device.

Host access to image memory is only well-defined for linear images and for image subresources of those images which are currently in either the `VK_IMAGE_LAYOUT_PREINITIALIZED` or `VK_IMAGE_LAYOUT_GENERAL` layout. Calling `vkGetImageSubresourceLayout` for a linear image returns a subresource layout mapping that is valid for either of those image layouts.

The set of image layouts consists of:

```c
// Provided by VK_VERSION_1_0
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,
} // Provided by VK_VERSION_1_1
```
The type(s) of device access supported by each layout are:

- **VK_IMAGE_LAYOUT_UNDEFINED** specifies that the layout is unknown. Image memory cannot be transitioned into this layout. This layout can be used as the initialLayout member of VkImageCreateInfo. This layout can be used in place of the current image layout in a layout transition, but doing so will cause the contents of the image's memory to be undefined.

- **VK_IMAGE_LAYOUT_PREINITIALIZED** specifies that an image's memory is in a defined layout and can be populated by data, but that it has not yet been initialized by the driver. Image memory cannot be transitioned into this layout. This layout can be used as the initialLayout member of VkImageCreateInfo. This layout is intended to be used as the initial layout for an image whose contents are written by the host, and hence the data can be written to memory immediately, without first executing a layout transition. Currently, VK_IMAGE_LAYOUT_PREINITIALIZED is only useful with linear images because there is not a standard layout defined for VK_IMAGE_TILING_OPTIMAL images.

- **VK_IMAGE_LAYOUT_GENERAL** supports all types of device access.

- **VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL** specifies a layout that must only be used with attachment accesses in the graphics pipeline.

- **VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL** specifies a layout allowing read only access as an attachment, or in shaders as a sampled image, combined image/sampler, or input attachment.

- **VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL** must only be used as a color or resolve attachment in a VkFramebuffer. This layout is valid only for image subresources of images created with the VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT usage bit enabled.

- **VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL** specifies a layout for both the depth and stencil aspects of a depth/stencil format image allowing read and write access as a depth/stencil attachment. It is equivalent to VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL and VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL.

- **VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL** specifies a layout for both the depth and stencil aspects of a depth/stencil format image allowing read only access as a depth/stencil attachment or in shaders as a sampled image, combined image/sampler, or input attachment.
is equivalent to \texttt{VK\_IMAGE\_LAYOUT\_DEPTH\_READ\_ONLY\_OPTIMAL} and \texttt{VK\_IMAGE\_LAYOUT\_STENCIL\_READ\_ONLY\_OPTIMAL}.

- \texttt{VK\_IMAGE\_LAYOUT\_DEPTH\_READ\_ONLY\_STENCIL\_ATTACHMENT\_OPTIMAL} specifies a layout for depth/stencil format images allowing read and write access to the stencil aspect as a stencil attachment, and read only access to the depth aspect as a depth attachment or in shaders as a sampled image, combined image/sampler, or input attachment. It is equivalent to \texttt{VK\_IMAGE\_LAYOUT\_DEPTH\_READ\_ONLY\_OPTIMAL} and \texttt{VK\_IMAGE\_LAYOUT\_STENCIL\_ATTACHMENT\_OPTIMAL}.

- \texttt{VK\_IMAGE\_LAYOUT\_DEPTH\_ATTACHMENT\_STENCIL\_READ\_ONLY\_OPTIMAL} specifies a layout for depth/stencil format images allowing read and write access to the depth aspect as a depth attachment, and read only access to the stencil aspect as a stencil attachment or in shaders as a sampled image, combined image/sampler, or input attachment. It is equivalent to \texttt{VK\_IMAGE\_LAYOUT\_DEPTH\_ATTACHMENT\_OPTIMAL} and \texttt{VK\_IMAGE\_LAYOUT\_STENCIL\_READ\_ONLY\_OPTIMAL}.

- \texttt{VK\_IMAGE\_LAYOUT\_DEPTH\_ATTACHMENT\_OPTIMAL} specifies a layout for the depth aspect of a depth/stencil format image allowing read access as a depth attachment.

- \texttt{VK\_IMAGE\_LAYOUT\_DEPTH\_READ\_ONLY\_OPTIMAL} specifies a layout for the depth aspect of a depth/stencil format image allowing read-only access as a depth attachment or in shaders as a sampled image, combined image/sampler, or input attachment.

- \texttt{VK\_IMAGE\_LAYOUT\_STENCIL\_ATTACHMENT\_OPTIMAL} specifies a layout for the stencil aspect of a depth/stencil format image allowing read and write access as a stencil attachment.

- \texttt{VK\_IMAGE\_LAYOUT\_STENCIL\_READ\_ONLY\_OPTIMAL} specifies a layout for the stencil aspect of a depth/stencil format image allowing read-only access as a stencil attachment or in shaders as a sampled image, combined image/sampler, or input attachment.

- \texttt{VK\_IMAGE\_LAYOUT\_SHADER\_READ\_ONLY\_OPTIMAL} specifies a layout allowing read-only access in a shader as a sampled image, combined image/sampler, or input attachment. This layout is valid only for image subresources of images created with the \texttt{VK\_IMAGE\_USAGE\_SAMPLED\_BIT} or \texttt{VK\_IMAGE\_USAGE\_INPUT\_ATTACHMENT\_BIT} usage bits enabled.

- \texttt{VK\_IMAGE\_LAYOUT\_TRANSFER\_SRC\_OPTIMAL} must only be used as a source image of a transfer command (see the definition of \texttt{VK\_PIPELINE\_STAGE\_TRANSFER\_BIT}). This layout is valid only for image subresources of images created with the \texttt{VK\_IMAGE\_USAGE\_TRANSFER\_SRC\_BIT} usage bit enabled.

- \texttt{VK\_IMAGE\_LAYOUT\_TRANSFER\_DST\_OPTIMAL} must only be used as a destination image of a transfer command. This layout is valid only for image subresources of images created with the \texttt{VK\_IMAGE\_USAGE\_TRANSFER\_DST\_BIT} usage bit enabled.

The layout of each image subresource is not a state of the image subresource itself, but is rather a property of how the data in memory is organized, and thus for each mechanism of accessing an image in the API the application must specify a parameter or structure member that indicates which image layout the image subresource(s) are considered to be in when the image will be accessed. For transfer commands, this is a parameter to the command (see \texttt{Clear Commands} and \texttt{Copy Commands}). For use as a framebuffer attachment, this is a member in the substructures of the \texttt{VkRenderPassCreateInfo} (see \texttt{Render Pass}). For use in a descriptor set, this is a member in the \texttt{VkDescriptorImageInfo} structure (see \texttt{Descriptor Set Updates}).
12.4.1. Image Layout Matching Rules

At the time that any command buffer command accessing an image executes on any queue, the layouts of the image subresources that are accessed must all match exactly the layout specified via the API controlling those accesses, except in case of accesses to an image with a depth/stencil format performed through descriptors referring to only a single aspect of the image, where the following relaxed matching rules apply:

- Descriptors referring just to the depth aspect of a depth/stencil image only need to match in the image layout of the depth aspect, thus `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL` and `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL` are considered to match.

- Descriptors referring just to the stencil aspect of a depth/stencil image only need to match in the image layout of the stencil aspect, thus `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL` and `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL` are considered to match.

When performing a layout transition on an image subresource, the old layout value must either equal the current layout of the image subresource (at the time the transition executes), or else be `VK_IMAGE_LAYOUT_UNDEFINED` (implying that the contents of the image subresource need not be preserved). The new layout used in a transition must not be `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`.

12.5. Image Views

Image objects are not directly accessed by pipeline shaders for reading or writing image data. Instead, image views representing contiguous ranges of the image subresources and containing additional metadata are used for that purpose. Views must be created on images of compatible types, and must represent a valid subset of image subresources.

Image views are represented by `VkImageView` handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkImageView)
```

`VK_REMAINING_ARRAY_LAYERS` is a special constant value used for image views to indicate that all remaining array layers in an image after the base layer should be included in the view.

```c
#define VK_REMAINING_ARRAY_LAYERS (~0U)
```

`VK_REMAINING_MIP_LEVELS` is a special constant value used for image views to indicate that all remaining mipmap levels in an image after the base level should be included in the view.

```c
#define VK_REMAINING_MIP_LEVELS (~0U)
```

The types of image views that can be created are:
typedef enum VkImageViewType {
    VK_IMAGE_VIEW_TYPE_1D = 0,
    VK_IMAGE_VIEW_TYPE_2D = 1,
    VK_IMAGE_VIEW_TYPE_3D = 2,
    VK_IMAGE_VIEW_TYPE_CUBE = 3,
    VK_IMAGE_VIEW_TYPE_1D_ARRAY = 4,
    VK_IMAGE_VIEW_TYPE_2D_ARRAY = 5,
    VK_IMAGE_VIEW_TYPE_CUBE_ARRAY = 6,
} VkImageViewType;

To create an image view, call:

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

- `device` is the logical device that creates the image view.
- `pCreateInfo` is a pointer to a `VkImageViewCreateInfo` structure containing parameters to be used to create the image view.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pView` is a pointer to a `VkImageView` handle in which the resulting image view object is returned.

**Valid Usage (Implicit)**

- VUID-vkCreateImageView-device-parameter
  - `device must` be a valid `VkDevice` handle
- VUID-vkCreateImageView-pCreateInfo-parameter
  - `pCreateInfo must` be a valid pointer to a valid `VkImageViewCreateInfo` structure
- VUID-vkCreateImageView-pAllocator-parameter
  - If `pAllocator` is not `NULL`, `pAllocator must` be a valid pointer to a valid `VkAllocationCallbacks` structure
- VUID-vkCreateImageView-pView-parameter
  - `pView must` be a valid pointer to a `VkImageView` handle

**Return Codes**

**Success**
- `VK_SUCCESS`
Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkImageViewCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkImageViewCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkImageViewCreateFlags flags;
    VkImage image;
    VkImageViewType viewType;
    VkFormat format;
    VkComponentMapping components;
    VkImageSubresourceRange subresourceRange;
} VkImageViewCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is a bitmask of `VkImageViewCreateFlagBits` specifying additional parameters of the image view.
- `image` is a `VkImage` on which the view will be created.
- `viewType` is a `VkImageViewType` value specifying the type of the image view.
- `format` is a `VkFormat` specifying the format and type used to interpret texel blocks of the image.
- `components` is a `VkComponentMapping` structure specifying a remapping of color components (or of depth or stencil components after they have been converted into color components).
- `subresourceRange` is a `VkImageSubresourceRange` structure selecting the set of mipmap levels and array layers to be accessible to the view.

Some of the `image` creation parameters are inherited by the view. In particular, image view creation inherits the implicit parameter `usage` specifying the allowed usages of the image view that, by default, takes the value of the corresponding `usage` parameter specified in `VkImageCreateInfo` at image creation time. The implicit `usage` can be overridden by adding a `VkImageViewUsageCreateInfo` structure to the `pNext` chain, but the view usage must be a subset of the image usage. If `image` has a depth-stencil format and was created with a `VkImageStencilUsageCreateInfo` structure included in the `pNext` chain of `VkImageCreateInfo`, the usage is calculated based on the `subresource.aspectMask` provided:

- If `aspectMask` includes only `VK_IMAGE_ASPECT_STENCIL_BIT`, the implicit `usage` is equal to `VkImageStencilUsageCreateInfo::stencilUsage`.
- If `aspectMask` includes only `VK_IMAGE_ASPECT_DEPTH_BIT`, the implicit `usage` is equal to `VkImageCreateInfo::usage`.

506
• If both aspects are included in aspectMask, the implicit usage is equal to the intersection of VkImageCreateInfo::usage and VkImageStencilUsageCreateInfo::stencilUsage.

If image was created with the VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT flag, and if the format of the image is not multi-planar, format can be different from the image’s format, but if image was created without the VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT flag and they are not equal they must be compatible. Image format compatibility is defined in the Format Compatibility Classes section. Views of compatible formats will have the same mapping between texel coordinates and memory locations irrespective of the format, with only the interpretation of the bit pattern changing.

Note

Values intended to be used with one view format may not be exactly preserved when written or read through a different format. For example, an integer value that happens to have the bit pattern of a floating point denorm or NaN may be flushed or canonicalized when written or read through a view with a floating point format. Similarly, a value written through a signed normalized format that has a bit pattern exactly equal to \(-2^b\) may be changed to \(-2^b + 1\) as described in Conversion from Normalized Fixed-Point to Floating-Point.

If image was created with the VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT flag, format must be compatible with the image’s format as described above; or must be an uncompressed format, in which case it must be size-compatible with the image’s format. In this case, the resulting image view’s texel dimensions equal the dimensions of the selected mip level divided by the compressed texel block size and rounded up.

The VkComponentMapping components member describes a remapping from components of the image to components of the vector returned by shader image instructions. This remapping must be the identity swizzle for storage image descriptors, input attachment descriptors, framebuffer attachments, and any VkImageView used with a combined image sampler that enables sampler Y’CbCr conversion.

If the image view is to be used with a sampler which supports sampler Y’C_bC_r conversion, an identically defined object of type VkSamplerYcbcrConversion to that used to create the sampler must be passed to vkCreateImageView in a VkSamplerYcbcrConversionInfo included in the pNext chain of VkImageViewCreateInfo. Conversely, if a VkSamplerYcbcrConversion object is passed to vkCreateImageView, an identically defined VkSamplerYcbcrConversion object must be used when sampling the image.

If the image has a multi-planar format and subresourceRange.aspectMask is VK_IMAGE_ASPECT_COLOR_BIT, then the format must be identical to the image format, and the sampler to be used with the image view must enable sampler Y’C_bC_r conversion.

If image was created with the VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT and the image has a multi-planar format, and if subresourceRange.aspectMask is VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or VK_IMAGE_ASPECT_PLANE_2_BIT, format must be compatible with the corresponding plane of the image, and the sampler to be used with the image view must not enable sampler Y’C_bC_r conversion. The width and height of the single-plane image view must be derived from the multi-planar image’s dimensions in the manner listed for plane compatibility for the plane.
Any view of an image plane will have the same mapping between texel coordinates and memory locations as used by the components of the color aspect, subject to the formulae relating texel coordinates to lower-resolution planes as described in Chroma Reconstruction. That is, if an R or B plane has a reduced resolution relative to the G plane of the multi-planar image, the image view operates using the \((u_{\text{plane}}, v_{\text{plane}})\) unnormalized coordinates of the reduced-resolution plane, and these coordinates access the same memory locations as the \((u_{\text{color}}, v_{\text{color}})\) unnormalized coordinates of the color aspect for which chroma reconstruction operations operate on the same \((u_{\text{plane}}, v_{\text{plane}})\) or \((i_{\text{plane}}, j_{\text{plane}})\) coordinates.

Table 7. Image type and image view type compatibility requirements

<table>
<thead>
<tr>
<th>Image View Type</th>
<th>Compatible Image Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_IMAGE_VIEW_TYPE_1D</td>
<td>VK_IMAGE_TYPE_1D</td>
</tr>
<tr>
<td>VK_IMAGE_VIEW_TYPE_1D_ARRAY</td>
<td>VK_IMAGE_TYPE_1D</td>
</tr>
<tr>
<td>VK_IMAGE_VIEW_TYPE_2D</td>
<td>VK_IMAGE_TYPE_2D, VK_IMAGE_TYPE_3D</td>
</tr>
<tr>
<td>VK_IMAGE_VIEW_TYPE_2D_ARRAY</td>
<td>VK_IMAGE_TYPE_2D, VK_IMAGE_TYPE_3D</td>
</tr>
<tr>
<td>VK_IMAGE_VIEW_TYPE_CUBE</td>
<td>VK_IMAGE_TYPE_2D</td>
</tr>
<tr>
<td>VK_IMAGE_VIEW_TYPE_CUBE_ARRAY</td>
<td>VK_IMAGE_TYPE_2D</td>
</tr>
<tr>
<td>VK_IMAGE_VIEW_TYPE_3D</td>
<td>VK_IMAGE_TYPE_3D</td>
</tr>
</tbody>
</table>

Valid Usage

- **VUID-VkImageViewCreateInfo-image-01003**
  If \texttt{image} was not created with \texttt{VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT} then \texttt{viewType} must not be \texttt{VK_IMAGE_VIEW_TYPE_CUBE} or \texttt{VK_IMAGE_VIEW_TYPE_CUBE_ARRAY}

- **VUID-VkImageViewCreateInfo-viewType-01004**
  If the \texttt{imageCubeArray} feature is not enabled, \texttt{viewType} must not be \texttt{VK_IMAGE_VIEW_TYPE_CUBE_ARRAY}

- **VUID-VkImageViewCreateInfo-image-06723**
  If \texttt{image} was created with \texttt{VK_IMAGE_TYPE_3D} but without \texttt{VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT} set then \texttt{viewType} must not be \texttt{VK_IMAGE_VIEW_TYPE_2D_ARRAY}

- **VUID-VkImageViewCreateInfo-image-06727**
  If \texttt{image} was created with \texttt{VK_IMAGE_TYPE_3D} but without \texttt{VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT} set then \texttt{viewType} must not be \texttt{VK_IMAGE_VIEW_TYPE_2D}

- **VUID-VkImageViewCreateInfo-image-04970**
  If \texttt{image} was created with \texttt{VK_IMAGE_TYPE_3D} and \texttt{viewType} is \texttt{VK_IMAGE_VIEW_TYPE_2D} or \texttt{VK_IMAGE_VIEW_TYPE_2D_ARRAY} then \texttt{subresourceRange.levelCount} must be 1

- **VUID-VkImageViewCreateInfo-image-04971**
  If \texttt{image} was created with \texttt{VK_IMAGE_TYPE_3D} and \texttt{viewType} is \texttt{VK_IMAGE_VIEW_TYPE_2D} or \texttt{VK_IMAGE_VIEW_TYPE_2D_ARRAY} then \texttt{VkImageCreateInfo::flags} must not contain any of \texttt{VK_IMAGE_CREATE_SPARSE_BINDING_BIT}, \texttt{VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT}, and \texttt{VK_IMAGE_CREATE_SPARSE_ALIASED_BIT}
If `image` was created with a `samples` value not equal to `VK_SAMPLE_COUNT_1_BIT` then `viewType` must be either `VK_IMAGE_VIEW_TYPE_2D` or `VK_IMAGE_VIEW_TYPE_2D_ARRAY`.

`image` must have been created with a `usage` value containing at least one of the usages defined in the valid image usage list for image views.

The format features of the resultant image view must contain at least one bit.

If `usage` contains `VK_IMAGE_USAGE_SAMPLED_BIT`, then the format features of the resultant image view must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`.

If `usage` contains `VK_IMAGE_USAGE_STORAGE_BIT`, then the image view's format features must contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT`.

If `usage` contains `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`, then the image view's format features must contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`.

If `usage` contains `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, then the image view's format features must contain `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`.

If `usage` contains `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`, then the image view's format features must contain at least one of `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT` or `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`.

`subresourceRange.baseMipLevel` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `image` was created.

If `subresourceRange.layerCount` is not `VK_REMAINING_ARRAY_LAYERS`, `subresourceRange.baseMipLevel + subresourceRange.layerCount` must be less than or equal to the `mipLevels` specified in `VkImageCreateInfo` when `image` was created.

If `image` is not a 3D image created with `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT` set, or `viewType` is not `VK_IMAGE_VIEW_TYPE_2D` or `VK_IMAGE_VIEW_TYPE_2D_ARRAY`, `subresourceRange.baseArrayLayer` must be less than the `arrayLayers` specified in `VkImageCreateInfo` when `image` was created.

If `subresourceRange.layerCount` is not `VK_REMAINING_ARRAY_LAYERS`, `image` is not a 3D image created with `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT` set, or `viewType` is not `VK_IMAGEVIEW_TYPE_2D` or `VK_IMAGEVIEW_TYPE_2D_ARRAY`, `subresourceRange.layerCount` must be non-zero and `subresourceRange.baseArrayLayer + subresourceRange.layerCount` must be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `image` was created.
• VUID-VkImageViewCreateInfo-image-02724
  If image is a 3D image created with VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT set, and
  viewType is VK_IMAGE_VIEW_TYPE_2D or VK_IMAGE_VIEW_TYPE_2D_ARRAY,
  subresourceRange.baseArrayLayer must be less than the depth computed from baseMipLevel
  and extent.depth specified in VkImageCreateInfo when image was created, according to
  the formula defined in Image Miplevel Sizing

• VUID-VkImageViewCreateInfo-subresourceRange-02725
  If subresourceRange.layerCount is not VK_REMAINING_ARRAY_LAYERS, image is a 3D image
  created with VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT set, and
  viewType is VK_IMAGE_VIEW_TYPE_2D or VK_IMAGE_VIEW_TYPE_2D_ARRAY,
  subresourceRange.layerCount must be non-zero and subresourceRange.baseArrayLayer + subresourceRange.layerCount must
  be less than or equal to the depth computed from baseMipLevel and extent.depth specified in
  VkImageCreateInfo when image was created, according to the formula defined in Image
  Miplevel Sizing

• VUID-VkImageViewCreateInfo-image-01761
  If image was created with the VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT flag, but without the
  VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT flag, and if the format of the image is not
  a multi-planar format, format must be compatible with the format used to create image, as
  defined in Format Compatibility Classes

• VUID-VkImageViewCreateInfo-image-01583
  If image was created with the VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT flag,
  format must be compatible with, or must be an uncompressed format that is size-
  compatible with, the format used to create image

• VUID-VkImageViewCreateInfo-image-07072
  If image was created with the VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT flag and
  format is a non-compressed format, the levelCount and layerCount members of
  subresourceRange must both be 1

• VUID-VkImageViewCreateInfo-pNext-01585
  If a VkImageFormatListCreateInfo structure was included in the pNext chain of the
  VkImageCreateInfo structure used when creating image and
  VkImageFormatListCreateInfo:viewFormatCount is not zero then format must be one of the
  formats in VkImageFormatListCreateInfo::pViewFormats

• VUID-VkImageViewCreateInfo-image-01586
  If image was created with the VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT flag, if the format of the
  image is a multi-planar format, and if subresourceRange.aspectMask is one of
  VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or
  VK_IMAGE_ASPECT_PLANE_2_BIT, then format must be compatible with the VkFormat for the
  plane of the image format indicated by subresourceRange.aspectMask, as defined in
  Compatible formats of planes of multi-planar formats

• VUID-VkImageViewCreateInfo-image-01762
  If image was not created with the VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT flag, or if the format
  of the image is a multi-planar format and if subresourceRange.aspectMask is
  VK_IMAGE_ASPECT_COLOR_BIT, format must be identical to the format used to create image

• VUID-VkImageViewCreateInfo-format-06415
  If the image view requires a sampler Y′C_bC_r conversion and usage contains
VK_IMAGE_USAGE_SAMPLED_BIT, then the pNext chain must include a VkSamplerYcbcrConversionInfo structure with a conversion value other than VK_NULL_HANDLE.

- VUID-VkImageViewCreateInfo-format-04714
  If format has a _422 or _420 suffix then image must have been created with a width that is a multiple of 2.

- VUID-VkImageViewCreateInfo-format-04715
  If format has a _420 suffix then image must have been created with a height that is a multiple of 2.

- VUID-VkImageViewCreateInfo-pNext-01970
  If the pNext chain includes a VkSamplerYcbcrConversionInfo structure with a conversion value other than VK_NULL_HANDLE, all members of components must have the identity swizzle.

- VUID-VkImageViewCreateInfo-pNext-06658
  If the pNext chain includes a VkSamplerYcbcrConversionInfo structure with a conversion value other than VK_NULL_HANDLE, format must be the same used in VkSamplerYcbcrConversionCreateInfo::format.

- VUID-VkImageViewCreateInfo-image-01020
  If image is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object.

- VUID-VkImageViewCreateInfo-subResourceRange-01021
  viewType must be compatible with the type of image as shown in the view type compatibility table.

- VUID-VkImageViewCreateInfo-pNext-02662
  If the pNext chain includes a VkImageViewUsageCreateInfo structure, and image was not created with a VkImageStencilUsageCreateInfo structure included in the pNext chain of VkImageCreateInfo, its usage member must not include any bits that were not set in the usage member of the VkImageCreateInfo structure used to create image.

- VUID-VkImageViewCreateInfo-pNext-02663
  If the pNext chain includes a VkImageViewUsageCreateInfo structure, image was created with a VkImageStencilUsageCreateInfo structure included in the pNext chain of VkImageCreateInfo, and subresourceRange.aspectMask includes VK_IMAGE_ASPECT_STENCIL_BIT, the usage member of the VkImageViewUsageCreateInfo structure must not include any bits that were not set in the usage member of the VkImageCreateInfo structure used to create image.

- VUID-VkImageViewCreateInfo-pNext-02664
  If the pNext chain includes a VkImageViewUsageCreateInfo structure, image was created with a VkImageStencilUsageCreateInfo structure included in the pNext chain of VkImageCreateInfo, and subresourceRange.aspectMask includes bits other than VK_IMAGE_ASPECT_STENCIL_BIT, the usage member of the VkImageViewUsageCreateInfo structure must not include any bits that were not set in the usage member of the VkImageCreateInfo structure used to create image.

- VUID-VkImageViewCreateInfo-imageViewType-04973
  If viewType is VK_IMAGE_VIEW_TYPE_1D, VK_IMAGE_VIEW_TYPE_2D, or VK_IMAGE_VIEW_TYPE_3D; and...
subresourceRange.layerCount is not VK_REMAINING_ARRAY_LAYERS, then subresourceRange.layerCount must be 1

- VUID-VkImageViewCreateInfo-imageViewType-04974
  If viewType is VK_IMAGE_VIEW_TYPE_1D, VK_IMAGE_VIEW_TYPE_2D, or VK_IMAGE_VIEW_TYPE_3D; and subresourceRange.layerCount is VK_REMAINING_ARRAY_LAYERS, then the remaining number of layers must be 1

- VUID-VkImageViewCreateInfo-viewType-02960
  If viewType is VK_IMAGE_VIEW_TYPE_CUBE and subresourceRange.layerCount is not VK_REMAINING_ARRAY_LAYERS, subresourceRange.layerCount must be 6

- VUID-VkImageViewCreateInfo-viewType-02961
  If viewType is VK_IMAGE_VIEW_TYPE_CUBE and subresourceRange.layerCount is not VK_REMAINING_ARRAY_LAYERS, subresourceRange.layerCount must be a multiple of 6

- VUID-VkImageViewCreateInfo-viewType-02962
  If viewType is VK_IMAGE_VIEW_TYPE_CUBE and subresourceRange.layerCount is VK_REMAINING_ARRAY_LAYERS, the remaining number of layers must be 6

- VUID-VkImageViewCreateInfo-viewType-02963
  If viewType is VK_IMAGE_VIEW_TYPE_CUBE_ARRAY and subresourceRange.layerCount is VK_REMAINING_ARRAY_LAYERS, the remaining number of layers must be a multiple of 6

Valid Usage (Implicit)

- VUID-VkImageViewCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO

- VUID-VkImageViewCreateInfo-pNext-pNext
  Each pNext member of any structure (including this one) in the pNext chain must be either NULL or a pointer to a valid instance of VkImageViewUsageCreateInfo or VkSamplerYcbcrConversionInfo

- VUID-VkImageViewCreateInfo-sType-unique
  The sType value of each struct in the pNext chain must be unique

- VUID-VkImageViewCreateInfo-flags-zerobitmask
  flags must be 0

- VUID-VkImageViewCreateInfo-image-parameter
  image must be a valid VkImage handle

- VUID-VkImageViewCreateInfo-viewType-parameter
  viewType must be a valid VkImageViewType value

- VUID-VkImageViewCreateInfo-format-parameter
  format must be a valid VkFormat value

- VUID-VkImageViewCreateInfo-components-parameter
  components must be a valid VkComponentMapping structure

- VUID-VkImageViewCreateInfo-subresourceRange-parameter
  subresourceRange must be a valid VkImageSubresourceRange structure
Bits which can be set in `VkImageViewCreateInfo::flags`, specifying additional parameters of an image view, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkImageViewCreateFlagBits {
} VkImageViewCreateFlagBits;
```

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkImageViewCreateFlags;
```

`VkImageViewCreateFlags` is a bitmask type for setting a mask of zero or more `VkImageViewCreateFlagBits`.

The set of usages for the created image view can be restricted compared to the parent image's usage flags by adding a `VkImageViewUsageCreateInfo` structure to the `pNext` chain of `VkImageViewCreateInfo`.

The `VkImageViewUsageCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkImageViewUsageCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkImageUsageFlags usage;
} VkImageViewUsageCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `usage` is a bitmask of `VkImageUsageFlagBits` specifying allowed usages of the image view.

When this structure is chained to `VkImageViewCreateInfo` the `usage` field overrides the implicit usage parameter inherited from image creation time and its value is used instead for the purposes of determining the valid usage conditions of `VkImageViewCreateInfo`.

### Valid Usage (Implicit)

- **VUID-VkImageViewUsageCreateInfo-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_IMAGE_VIEW_USAGE_CREATE_INFO`

- **VUID-VkImageViewUsageCreateInfo-usage-parameter**
  
  `usage` must be a valid combination of `VkImageUsageFlagBits` values

- **VUID-VkImageViewUsageCreateInfo-usage-requiredbitsetmask**
  
  `usage` must not be `0`

The `VkImageSubresourceRange` structure is defined as:
```c
typedef struct VkImageSubresourceRange {
    VkImageAspectFlags aspectMask;
    uint32_t baseMipLevel;
    uint32_t levelCount;
    uint32_t baseArrayLayer;
    uint32_t layerCount;
} VkImageSubresourceRange;
```

- `aspectMask` is a bitmask of `VkImageAspectFlagBits` specifying which aspect(s) of the image are included in the view.
- `baseMipLevel` is the first mipmap level accessible to the view.
- `levelCount` is the number of mipmap levels (starting from `baseMipLevel`) accessible to the view.
- `baseArrayLayer` is the first array layer accessible to the view.
- `layerCount` is the number of array layers (starting from `baseArrayLayer`) accessible to the view.

The number of mipmap levels and array layers must be a subset of the image subresources in the image. If an application wants to use all mip levels or layers in an image after the `baseMipLevel` or `baseArrayLayer`, it can set `levelCount` and `layerCount` to the special values `VK_REMAINING_MIP_LEVELS` and `VK_REMAINING_ARRAY_LAYERS` without knowing the exact number of mip levels or layers.

For cube and cube array image views, the layers of the image view starting at `baseArrayLayer` correspond to faces in the order +X, -X, +Y, -Y, +Z, -Z. For cube arrays, each set of six sequential layers is a single cube, so the number of cube maps in a cube map array view is `layerCount / 6`, and image array layer (`baseArrayLayer + i`) is face index `(i mod 6)` of cube `i / 6`. If the number of layers in the view, whether set explicitly in `layerCount` or implied by `VK_REMAINING_ARRAY_LAYERS`, is not a multiple of 6, the last cube map in the array must not be accessed.

`aspectMask` must be only `VK_IMAGE_ASPECT_COLOR_BIT`, `VK_IMAGE_ASPECT_DEPTH_BIT`, or `VK_IMAGE_ASPECT_STENCIL_BIT` if `format` is a color, depth-only or stencil-only format, respectively, except if `format` is a multi-planar format. If using a depth/stencil format with both depth and stencil components, `aspectMask` must include at least one of `VK_IMAGE_ASPECT_DEPTH_BIT` and `VK_IMAGE_ASPECT_STENCIL_BIT`, and can include both.

When the `VkImageSubresourceRange` structure is used to select a subset of the slices of a 3D image's mip level in order to create a 2D or 2D array image view of a 3D image created with `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT`, `baseArrayLayer` and `layerCount` specify the first slice index and the number of slices to include in the created image view. Such an image view can be used as a framebuffer attachment that refers only to the specified range of slices of the selected mip level. However, any layout transitions performed on such an attachment view during a render pass instance still apply to the entire subresource referenced which includes all the slices of the selected mip level.

When using an image view of a depth/stencil image to populate a descriptor set (e.g. for sampling in the shader, or for use as an input attachment), the `aspectMask` must only include one bit, which selects whether the image view is used for depth reads (i.e. using a floating-point sampler or input attachment in the shader) or stencil reads (i.e. using an unsigned integer sampler or input...
attachment in the shader). When an image view of a depth/stencil image is used as a depth/stencil framebuffer attachment, the aspectMask is ignored and both depth and stencil image subresources are used.

When creating a VkImageView, if sampler Y'CbCr conversion is enabled in the sampler, the aspectMask of a subresourceRange used by the VkImageView must be VK_IMAGE_ASPECT_COLOR_BIT.

When creating a VkImageView, if sampler Y'CbCr conversion is not enabled in the sampler and the image format is multi-planar, the image must have been created with VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT, and the aspectMask of the VkImageView's subresourceRange must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT or VK_IMAGE_ASPECT_PLANE_2_BIT.

Valid Usage

- VUID-VkImageSubresourceRange-levelCount-01720
  If levelCount is not VK_REMAINING_MIP_LEVELS, it must be greater than 0

- VUID-VkImageSubresourceRange-layerCount-01721
  If layerCount is not VK_REMAINING_ARRAY_LAYERS, it must be greater than 0

- VUID-VkImageSubresourceRange-aspectMask-01670
  If aspectMask includes VK_IMAGE_ASPECT_COLOR_BIT, then it must not include any of VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or VK_IMAGE_ASPECT_PLANE_2_BIT

Valid Usage (Implicit)

- VUID-VkImageSubresourceRange-aspectMask-parameter
  aspectMask must be a valid combination of VkImageAspectFlagBits values

- VUID-VkImageSubresourceRange-aspectMask-requiredbitsetmask
  aspectMask must not be 0

Bits which can be set in an aspect mask to specify aspects of an image for purposes such as identifying a subresource, are:

```c
// Provided by VK_VERSION_1_0
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,
// Provided by VK_VERSION_1_1
    VK_IMAGE_ASPECT_PLANE_0_BIT = 0x00000010,
// Provided by VK_VERSION_1_1
    VK_IMAGE_ASPECT_PLANE_1_BIT = 0x00000020,
// Provided by VK_VERSION_1_1
    VK_IMAGE_ASPECT_PLANE_2_BIT = 0x00000040,
// Provided by VK_VERSION_1_3
};
```
• **VK_IMAGE_ASPECT_NONE** specifies no image aspect, or the image aspect is not applicable.
• **VK_IMAGE_ASPECT_COLOR_BIT** specifies the color aspect.
• **VK_IMAGE_ASPECT_DEPTH_BIT** specifies the depth aspect.
• **VK_IMAGE_ASPECT_STENCIL_BIT** specifies the stencil aspect.
• **VK_IMAGE_ASPECT_METADATA_BIT** specifies the metadata aspect used for sparse resource operations.
• **VK_IMAGE_ASPECT_PLANE_0_BIT** specifies plane 0 of a multi-planar image format.
• **VK_IMAGE_ASPECT_PLANE_1_BIT** specifies plane 1 of a multi-planar image format.
• **VK_IMAGE_ASPECT_PLANE_2_BIT** specifies plane 2 of a multi-planar image format.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkImageAspectFlags;
```

**VkImageAspectFlags** is a bitmask type for setting a mask of zero or more **VkImageAspectFlagBits**.

The **VkComponentMapping** structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkComponentMapping {
    VkComponentSwizzle r;
    VkComponentSwizzle g;
    VkComponentSwizzle b;
    VkComponentSwizzle a;
} VkComponentMapping;
```

• **r** is a **VkComponentSwizzle** specifying the component value placed in the R component of the output vector.
• **g** is a **VkComponentSwizzle** specifying the component value placed in the G component of the output vector.
• **b** is a **VkComponentSwizzle** specifying the component value placed in the B component of the output vector.
• **a** is a **VkComponentSwizzle** specifying the component value placed in the A component of the output vector.

**Valid Usage (Implicit)**

- **VUID-VkComponentMapping-r-parameter**
  - **r** must be a valid **VkComponentSwizzle** value
- **VUID-VkComponentMapping-g-parameter**
  - **g** must be a valid **VkComponentSwizzle** value
Possible values of the members of `VkComponentMapping`, specifying the component values placed in each component of the output vector, are:

```cpp
// Provided by VK_VERSION_1_0
typedef enum VkComponentSwizzle {
    VK_COMPONENT_SWIZZLE_IDENTITY = 0,
    VK_COMPONENT_SWIZZLE_ZERO = 1,
    VK_COMPONENT_SWIZZLE_ONE = 2,
    VK_COMPONENT_SWIZZLE_R = 3,
    VK_COMPONENT_SWIZZLE_G = 4,
    VK_COMPONENT_SWIZZLE_B = 5,
    VK_COMPONENT_SWIZZLE_A = 6,
} VkComponentSwizzle;
```

- **VK_COMPONENT_SWIZZLE_IDENTITY** specifies that the component is set to the identity swizzle.
- **VK_COMPONENT_SWIZZLE_ZERO** specifies that the component is set to zero.
- **VK_COMPONENT_SWIZZLE_ONE** specifies that the component is set to either 1 or 1.0, depending on whether the type of the image view format is integer or floating-point respectively, as determined by the Format Definition section for each `VkFormat`.
- **VK_COMPONENT_SWIZZLE_R** specifies that the component is set to the value of the R component of the image.
- **VK_COMPONENT_SWIZZLE_G** specifies that the component is set to the value of the G component of the image.
- **VK_COMPONENT_SWIZZLE_B** specifies that the component is set to the value of the B component of the image.
- **VK_COMPONENT_SWIZZLE_A** specifies that the component is set to the value of the A component of the image.

Setting the identity swizzle on a component is equivalent to setting the identity mapping on that component. That is:

Table 8. Component Mappings Equivalent To `VK_COMPONENT_SWIZZLE_IDENTITY`

<table>
<thead>
<tr>
<th>Component</th>
<th>Identity Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>components.r</td>
<td>VK_COMPONENT_SWIZZLE_R</td>
</tr>
<tr>
<td>components.g</td>
<td>VK_COMPONENT_SWIZZLE_G</td>
</tr>
<tr>
<td>components.b</td>
<td>VK_COMPONENT_SWIZZLE_B</td>
</tr>
<tr>
<td>components.a</td>
<td>VK_COMPONENT_SWIZZLE_A</td>
</tr>
</tbody>
</table>
To destroy an image view, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyImageView(
    VkDevice device, 
    VkImageView imageView, 
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the image view.
- `imageView` is the image view to destroy.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage

- VUID-vkDestroyImageView-imageView-01026
  All submitted commands that refer to `imageView` must have completed execution
- VUID-vkDestroyImageView-imageView-01027
  If `VkAllocationCallbacks` were provided when `imageView` was created, a compatible set of callbacks must be provided here
- VUID-vkDestroyImageView-imageView-01028
  If no `VkAllocationCallbacks` were provided when `imageView` was created, `pAllocator` must be `NULL`

### Valid Usage (Implicit)

- VUID-vkDestroyImageView-device-parameter
  `device` must be a valid `VkDevice` handle
- VUID-vkDestroyImageView-imageView-parameter
  If `imageView` is not `VK_NULL_HANDLE`, `imageView` must be a valid `VkImageView` handle
- VUID-vkDestroyImageView-pAllocator-parameter
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- VUID-vkDestroyImageView-imageView-parent
  If `imageView` is a valid handle, it must have been created, allocated, or retrieved from `device`

### Host Synchronization

- Host access to `imageView` must be externally synchronized
12.5.1. Image View Format Features

Valid uses of a `VkImageView` may depend on the image view’s `format features`, defined below. Such constraints are documented in the affected valid usage statement.

- If Vulkan 1.3 is supported or the `VK_KHR_format_feature_flags2` extension is enabled, and `VkImageViewCreateInfo::image` was created with `VK_IMAGE_TILING_LINEAR`, then the image view’s set of `format features` is the value of `VkFormatProperties3::linearTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties2` on the same `format` as `VkImageViewCreateInfo::format`.

- If Vulkan 1.3 is not supported and the `VK_KHR_format_feature_flags2` extension is not enabled, and `VkImageViewCreateInfo::image` was created with `VK_IMAGE_TILING_LINEAR`, then the image view’s set of `format features` is the union of the value of `VkFormatProperties::linearTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` on the same `format` as `VkImageViewCreateInfo::format`, with:
  - `VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_DEPTH_COMPARISON_BIT` if the format is a depth/stencil format and the image view features also contain `VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_BIT`.
  - `VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT` if the format is one of the extended storage formats and `shaderStorageImageReadWithoutFormat` is enabled on the device.
  - `VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT` if the format is one of the extended storage formats and `shaderStorageImageWriteWithoutFormat` is enabled on the device.

- If Vulkan 1.3 is supported or the `VK_KHR_format_feature_flags2` extension is enabled, and `VkImageViewCreateInfo::image` was created with `VK_IMAGE_TILING_OPTIMAL`, then the image view’s set of `format features` is the value of `VkFormatProperties::optimalTilingFeatures` or `VkFormatProperties3::optimalTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` or `vkGetPhysicalDeviceImageFormatProperties2` on the same `format` as `VkImageViewCreateInfo::format`.

- If Vulkan 1.3 is not supported and the `VK_KHR_format_feature_flags2` extension is not enabled, and `VkImageViewCreateInfo::image` was created with `VK_IMAGE_TILING_OPTIMAL`, then the image view’s set of `format features` is the union of the value of `VkFormatProperties::optimalTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` on the same `format` as `VkImageViewCreateInfo::format`, with:
  - `VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_DEPTH_COMPARISON_BIT` if the format is a depth/stencil format and the image view features also contain `VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_BIT`.
  - `VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT` if the format is one of the extended storage formats and `shaderStorageImageReadWithoutFormat` is enabled on the device.
  - `VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT` if the format is one of the extended storage formats and `shaderStorageImageWriteWithoutFormat` is enabled on the device.

12.6. Resource Memory Association

Resources are initially created as `virtual allocations` with no backing memory. Device memory is allocated separately (see Device Memory) and then associated with the resource. This association is done differently for sparse and non-sparse resources.
Resources created with any of the sparse creation flags are considered sparse resources. Resources created without these flags are non-sparse. The details on resource memory association for sparse resources is described in Sparse Resources.

Non-sparse resources must be bound completely and contiguously to a single VkDeviceMemory object before the resource is passed as a parameter to any of the following operations:

- creating image or buffer views
- updating descriptor sets
- recording commands in a command buffer

Once bound, the memory binding is immutable for the lifetime of the resource.

In a logical device representing more than one physical device, buffer and image resources exist on all physical devices but can be bound to memory differently on each. Each such replicated resource is an instance of the resource. For sparse resources, each instance can be bound to memory arbitrarily differently. For non-sparse resources, each instance can either be bound to the local or a peer instance of the memory, or for images can be bound to rectangular regions from the local and/or peer instances. When a resource is used in a descriptor set, each physical device interprets the descriptor according to its own instance’s binding to memory.

Note

There are no new copy commands to transfer data between physical devices. Instead, an application can create a resource with a peer mapping and use it as the source or destination of a transfer command executed by a single physical device to copy the data from one physical device to another.

To determine the memory requirements for a buffer resource, call:

```c
// Provided by VK_VERSION_1_0
void vkGetBufferMemoryRequirements(
    VkDevice device,                  // Logical device that owns the buffer.
    VkBuffer buffer,                 // Buffer to query.
    VkMemoryRequirements* pMemoryRequirements);  // Pointer to a VkMemoryRequirements structure in which the memory requirements of the buffer object are returned.
```

Valid Usage (Implicit)

- VUID-vkGetBufferMemoryRequirements-device-parameter device must be a valid VkDevice handle
- VUID-vkGetBufferMemoryRequirements-buffer-parameter
buffer must be a valid VkBuffer handle

- VUID-vkGetBufferMemoryRequirements-pMemoryRequirements-parameter
  pMemoryRequirements must be a valid pointer to a VkMemoryRequirements structure
- VUID-vkGetBufferMemoryRequirements-buffer-parent
  buffer must have been created, allocated, or retrieved from device

To determine the memory requirements for an image resource which is not created with the VK_IMAGE_CREATE_DISJOINT_BIT flag set, call:

```c
// Provided by VK_VERSION_1_0
void vkGetImageMemoryRequirements(
    VkDevice device,
    VkImage image,
    VkMemoryRequirements* pMemoryRequirements);
```

- device is the logical device that owns the image.
- image is the image to query.
- pMemoryRequirements is a pointer to a VkMemoryRequirements structure in which the memory requirements of the image object are returned.

### Valid Usage

- VUID-vkGetImageMemoryRequirements-image-01588
  image must not have been created with the VK_IMAGE_CREATE_DISJOINT_BIT flag set

### Valid Usage (Implicit)

- VUID-vkGetImageMemoryRequirements-device-parameter
  device must be a valid VkDevice handle
- VUID-vkGetImageMemoryRequirements-image-parameter
  image must be a valid VkImage handle
- VUID-vkGetImageMemoryRequirements-pMemoryRequirements-parameter
  pMemoryRequirements must be a valid pointer to a VkMemoryRequirements structure
- VUID-vkGetImageMemoryRequirements-image-parent
  image must have been created, allocated, or retrieved from device

The VkMemoryRequirements structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkMemoryRequirements {
    VkDeviceSize size;
    VkDeviceSize alignment;
};
```
uint32_t memoryTypeBits;
} VkMemoryRequirements;

- `size` is the size, in bytes, of the memory allocation required for the resource.
- `alignment` is the alignment, in bytes, of the offset within the allocation required for the resource.
- `memoryTypeBits` is a bitmask and contains one bit set for every supported memory type for the resource. Bit `i` is set if and only if the memory type `i` in the `VkPhysicalDeviceMemoryProperties` structure for the physical device is supported for the resource.

The implementation guarantees certain properties about the memory requirements returned by `vkGetDeviceBufferMemoryRequirements`, `vkGetDeviceImageMemoryRequirements`, `vkGetBufferMemoryRequirements` and `vkGetImageMemoryRequirements`:

- The `memoryTypeBits` member always contains at least one bit set.
- If `buffer` is a `VkBuffer` not created with the `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` bit set, or if `image` is linear image, then the `memoryTypeBits` member always contains at least one bit set corresponding to a `VkMemoryType` with a `propertyFlags` that has both the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` bit and the `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` bit set. In other words, mappable coherent memory can always be attached to these objects.
- If `buffer` was created with `VkExternalMemoryBufferCreateInfo::handleTypes` set to 0 or `image` was created with `VkExternalMemoryImageCreateInfo::handleTypes` set to 0, the `memoryTypeBits` member always contains at least one bit set corresponding to a `VkMemoryType` with a `propertyFlags` that has the `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` bit set.
- The `memoryTypeBits` member is identical for all `VkBuffer` objects created with the same value for the `flags` and `usage` members in the `VkBufferCreateInfo` structure and the `handleTypes` member of the `VkExternalMemoryBufferCreateInfo` structure passed to `vkCreateBuffer`. Further, if `usage1` and `usage2` of type `VkBufferUsageFlags` are such that the bits set in `usage2` are a subset of the bits set in `usage1`, and they have the same `flags` and `VkExternalMemoryBufferCreateInfo::handleTypes`, then the bits set in `memoryTypeBits` returned for `usage1` must be a subset of the bits set in `memoryTypeBits` returned for `usage2`, for all values of `flags`.
- The `alignment` member is a power of two.
- The `alignment` member is identical for all `VkBuffer` objects created with the same combination of values for the `usage` and `flags` members in the `VkBufferCreateInfo` structure passed to `vkCreateBuffer`.
- If the `maintenance4` feature is enabled, then the `alignment` member is identical for all `VkImage` objects created with the same combination of values for the `flags`, `imageType`, `format`, `extent`, `mipLevels`, `arrayLayers`, `samples`, `tiling` and `usage` members in the `VkImageCreateInfo` structure passed to `vkCreateImage`.
- The `alignment` member satisfies the buffer descriptor offset alignment requirements associated with the `VkBuffer`’s `usage`:
  - If `usage` included `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT` or `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT`, `alignment` must be an integer multiple of `VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment`. 

522
If usage included VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT, alignment must be an integer multiple of VkPhysicalDeviceLimits::minUniformBufferOffsetAlignment.

If usage included VK_BUFFER_USAGE_STORAGE_BUFFER_BIT, alignment must be an integer multiple of VkPhysicalDeviceLimits::minStorageBufferOffsetAlignment.

- For images created with a color format, the memoryTypeBits member is identical for all VkImage objects created with the same combination of values for the tiling member, the VK_IMAGE_CREATE_SPARSE_BINDING_BIT bit of the flags member, the VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT bit of the flags member, handleTypes member of VkExternalMemoryImageCreateInfo, and the VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT of the usage member in the VkImageCreateInfo structure passed to vkCreateImage.

- For images created with a depth/stencil format, the memoryTypeBits member is identical for all VkImage objects created with the same combination of values for the format member, the tiling member, the VK_IMAGE_CREATE_SPARSE_BINDING_BIT bit of the flags member, the VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT bit of the flags member, handleTypes member of VkExternalMemoryImageCreateInfo, and the VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT of the usage member in the VkImageCreateInfo structure passed to vkCreateImage.

- If the memory requirements are for a VkImage, the memoryTypeBits member must not refer to a VkMemoryType with a propertyFlags that has the VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT bit set if the image did not have VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT bit set in the usage member of the VkImageCreateInfo structure passed to vkCreateImage.

- If the memory requirements are for a VkBuffer, the memoryTypeBits member must not refer to a VkMemoryType with a propertyFlags that has the VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT bit set.

Note
The implication of this requirement is that lazily allocated memory is disallowed for buffers in all cases.

- The size member is identical for all VkBuffer objects created with the same combination of creation parameters specified in VkBufferCreateInfo and its pNext chain.

- The size member is identical for all VkImage objects created with the same combination of creation parameters specified in VkImageCreateInfo and its pNext chain.

Note
This, however, does not imply that they interpret the contents of the bound memory identically with each other. That additional guarantee, however, can be explicitly requested using VK_IMAGE_CREATE_ALIAS_BIT.

- If the maintenance4 feature is enabled, these additional guarantees apply:
  - For a VkBuffer, the size memory requirement is never greater than that of another VkBuffer created with a greater or equal size specified in VkBufferCreateInfo, all other creation parameters being identical.
  - For a VkBuffer, the size memory requirement is never greater than the result of aligning VkBufferCreateInfo::size with the alignment memory requirement.
  - For a VkImage, the size memory requirement is never greater than that of another VkImage
The memory requirements returned by `vkGetDeviceBufferMemoryRequirements` are identical to those that would be returned by `vkGetBufferMemoryRequirements2` if it were called with a `VkBuffer` created with the same `VkBufferCreateInfo` values.

The memory requirements returned by `vkGetDeviceImageMemoryRequirements` are identical to those that would be returned by `vkGetImageMemoryRequirements2` if it were called with a `VkImage` created with the same `VkImageCreateInfo` values.

To determine the memory requirements for a buffer resource, call:

```c
// Provided by VK_VERSION_1_1
void vkGetBufferMemoryRequirements2(
    VkDevice device, const VkBufferMemoryRequirementsInfo2* pInfo, VkMemoryRequirements2* pMemoryRequirements);
```

- `device` is the logical device that owns the buffer.
- `pInfo` is a pointer to a `VkBufferMemoryRequirementsInfo2` structure containing parameters required for the memory requirements query.
- `pMemoryRequirements` is a pointer to a `VkMemoryRequirements2` structure in which the memory requirements of the buffer object are returned.

**Valid Usage (Implicit)**

- VUID-vkGetBufferMemoryRequirements2-device-parameter
  - `device` must be a valid `VkDevice` handle
- VUID-vkGetBufferMemoryRequirements2-pInfo-parameter
  - `pInfo` must be a valid pointer to a valid `VkBufferMemoryRequirementsInfo2` structure
- VUID-vkGetBufferMemoryRequirements2-pMemoryRequirements-parameter
  - `pMemoryRequirements` must be a valid pointer to a `VkMemoryRequirements2` structure

To determine the memory requirements for a buffer resource without creating an object, call:

```c
// Provided by VK_VERSION_1_3
void vkGetDeviceBufferMemoryRequirements(
    VkDevice device, const VkDeviceBufferMemoryRequirements* pInfo, VkMemoryRequirements2* pMemoryRequirements);
```

- `device` is the logical device intended to own the buffer.
- `pInfo` is a pointer to a `VkDeviceBufferMemoryRequirements` structure containing parameters required for the memory requirements query.
• pMemoryRequirements is a pointer to a VkMemoryRequirements2 structure in which the memory requirements of the buffer object are returned.

Valid Usage (Implicit)

• VUID-vkGetDeviceBufferMemoryRequirements-device-parameter
device must be a valid VkDevice handle

• VUID-vkGetDeviceBufferMemoryRequirements-pInfo-parameter
pInfo must be a valid pointer to a valid VkDeviceBufferMemoryRequirements structure

• VUID-vkGetDeviceBufferMemoryRequirements-pMemoryRequirements-parameter
pMemoryRequirements must be a valid pointer to a VkMemoryRequirements2 structure

The VkBufferMemoryRequirementsInfo2 structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkBufferMemoryRequirementsInfo2 {
    VkStructureType sType;
    const void* pNext;
    VkBuffer buffer;
} VkBufferMemoryRequirementsInfo2;
```

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• buffer is the buffer to query.

Valid Usage (Implicit)

• VUID-VkBufferMemoryRequirementsInfo2-sType-sType
sType must be VK_STRUCTURE_TYPE_BUFFER_MEMORY_REQUIREMENTS_INFO_2

• VUID-VkBufferMemoryRequirementsInfo2-pNext-pNext
pNext must be NULL

• VUID-VkBufferMemoryRequirementsInfo2-buffer-parameter
buffer must be a valid VkBuffer handle

The VkDeviceBufferMemoryRequirements structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkDeviceBufferMemoryRequirements {
    VkStructureType sType;
    const void* pNext;
    const VkBufferCreateInfo* pCreateInfo;
} VkDeviceBufferMemoryRequirements;
```
• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **pCreateInfo** is a pointer to a **VkBufferCreateInfo** structure containing parameters affecting creation of the buffer to query.

---

**Valid Usage (Implicit)**

- VUID-VkDeviceBufferMemoryRequirements-sType-sType
  
  **sType** **must** be **VK_STRUCTURE_TYPE_DEVICE_BUFFER_MEMORY_REQUIREMENTS**

- VUID-VkDeviceBufferMemoryRequirements-pNext-pNext
  
  **pNext** **must** be **NULL**

- VUID-VkDeviceBufferMemoryRequirements-pCreateInfo-parameter
  
  **pCreateInfo** **must** be a valid pointer to a valid **VkBufferCreateInfo** structure

To determine the memory requirements for an image resource, call:

```c
// Provided by VK_VERSION_1_1
void vkGetImageMemoryRequirements2(
    VkDevice device,
    const VkImageMemoryRequirementsInfo2* pInfo,
    VkMemoryRequirements2* pMemoryRequirements);
```

- **device** is the logical device that owns the image.
- **pInfo** is a pointer to a **VkImageMemoryRequirementsInfo2** structure containing parameters required for the memory requirements query.
- **pMemoryRequirements** is a pointer to a **VkMemoryRequirements2** structure in which the memory requirements of the image object are returned.

---

**Valid Usage (Implicit)**

- VUID-vkGetImageMemoryRequirements2-device-parameter
  
  **device** **must** be a valid **VkDevice** handle

- VUID-vkGetImageMemoryRequirements2-pInfo-parameter
  
  **pInfo** **must** be a valid pointer to a valid **VkImageMemoryRequirementsInfo2** structure

- VUID-vkGetImageMemoryRequirements2-pMemoryRequirements-parameter
  
  **pMemoryRequirements** **must** be a valid pointer to a **VkMemoryRequirements2** structure

To determine the memory requirements for an image resource without creating an object, call:

```c
// Provided by VK_VERSION_1_3
void vkGetDeviceImageMemoryRequirements(
    VkDevice device,
    
    VkImageMemoryRequirements2* pInfo,
    VkMemoryRequirements2* pMemoryRequirements);
```

---

526
const VkDeviceImageMemoryRequirements* pInfo, 
VkMemoryRequirements2* pMemoryRequirements);

- **device** is the logical device intended to own the image.
- **pInfo** is a pointer to a `VkDeviceImageMemoryRequirements` structure containing parameters required for the memory requirements query.
- **pMemoryRequirements** is a pointer to a `VkMemoryRequirements2` structure in which the memory requirements of the image object are returned.

### Valid Usage (Implicit)

- VUID-vkGetDeviceImageMemoryRequirements-device-parameter  
  `device` **must** be a valid `VkDevice` handle
- VUID-vkGetDeviceImageMemoryRequirements-pInfo-parameter  
  `pInfo` **must** be a valid pointer to a valid `VkDeviceImageMemoryRequirements` structure
- VUID-vkGetDeviceImageMemoryRequirements-pMemoryRequirements-parameter  
  `pMemoryRequirements` **must** be a valid pointer to a `VkMemoryRequirements2` structure

The `VkImageMemoryRequirementsInfo2` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkImageMemoryRequirementsInfo2 {
    VkStructureType sType;
    const void* pNext;
    VkImage image;
} VkImageMemoryRequirementsInfo2;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **image** is the image to query.

### Valid Usage

- VUID-VkImageMemoryRequirementsInfo2-image-01589  
  If `image` was created with a *multi-planar* format and the `VK_IMAGE_CREATE_DISJOINT_BIT` flag, there **must** be a `VkImagePlaneMemoryRequirementsInfo` included in the `pNext` chain of the `VkImageMemoryRequirementsInfo2` structure
- VUID-VkImageMemoryRequirementsInfo2-image-01590  
  If `image` was not created with the `VK_IMAGE_CREATE_DISJOINT_BIT` flag, there **must** not be a `VkImagePlaneMemoryRequirementsInfo` included in the `pNext` chain of the `VkImageMemoryRequirementsInfo2` structure
- VUID-VkImageMemoryRequirementsInfo2-image-01591  
  If `image` was created with a single-plane format, there **must** not be a
VkImagePlaneMemoryRequirementsInfo included in the `pNext` chain of the `VkImageMemoryRequirementsInfo2` structure

**Valid Usage (Implicit)**

- VUID-VkImageMemoryRequirementsInfo2-sType-sType
  `sType` **must** be `VK_STRUCTURE_TYPE_IMAGE_MEMORY_REQUIREMENTS_INFO_2`

- VUID-VkImageMemoryRequirementsInfo2-pNext-pNext
  `pNext` **must** be `NULL` or a pointer to a valid instance of `VkImagePlaneMemoryRequirementsInfo`

- VUID-VkImageMemoryRequirementsInfo2-sType-unique
  The `sType` value of each struct in the `pNext` chain **must** be unique

- VUID-VkImageMemoryRequirementsInfo2-image-parameter
  `image` **must** be a valid `VkImage` handle

The `VkDeviceImageMemoryRequirements` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkDeviceImageMemoryRequirements {
    VkStructureType     sType;
    const void*          pNext;
    const VkImageCreateInfo* pCreateInfo;
    VkImageAspectFlagBits planeAspect;
} VkDeviceImageMemoryRequirements;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `pCreateInfo` is a pointer to a `VkImageCreateInfo` structure containing parameters affecting creation of the image to query.
- `planeAspect` is a `VkImageAspectFlagBits` value specifying the aspect corresponding to the image plane to query. This parameter is ignored unless `pCreateInfo::flags` has `VK_IMAGE_CREATE_DISJOINT_BIT` set.

**Valid Usage**

- VUID-VkDeviceImageMemoryRequirementsKHR-pCreateInfo-06416
  The `pCreateInfo::pNext` chain **must** not contain a `VkImageSwapchainCreateInfoKHR` structure

- VUID-VkDeviceImageMemoryRequirementsKHR-pCreateInfo-06417
  If `pCreateInfo::format` specifies a *multi-planar* format and `pCreateInfo::flags` has `VK_IMAGE_CREATE_DISJOINT_BIT` set then `planeAspect` **must** not be `VK_IMAGE_ASPECT_NONE_KHR`

- VUID-VkDeviceImageMemoryRequirementsKHR-pCreateInfo-06419
  If `pCreateInfo::flags` has `VK_IMAGE_CREATE_DISJOINT_BIT` set and if the `pCreateInfo::tiling` is
VK_IMAGE_TILING_LINEAR or VK_IMAGE_TILING_OPTIMAL, then planeAspect must be a single valid format plane for the image (that is, for a two-plane image planeAspect must be VK_IMAGE_ASPECT_PLANE_0_BIT or VK_IMAGE_ASPECT_PLANE_1_BIT, and for a three-plane image planeAspect must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT or VK_IMAGE_ASPECT_PLANE_2_BIT)

Valid Usage (Implicit)

- VUID-VkDeviceImageMemoryRequirements-sType-sType sType must be VK_STRUCTURE_TYPE_DEVICE_IMAGE_MEMORY_REQUIREMENTS
- VUID-VkDeviceImageMemoryRequirements-pNext-pNext pNext must be NULL
- VUID-VkDeviceImageMemoryRequirements-pCreateInfo-parameter pCreateInfo must be a valid pointer to a valid VkImageCreateInfo structure
- VUID-VkDeviceImageMemoryRequirements-planeAspect-parameter If planeAspect is not 0, planeAspect must be a valid VkImageAspectFlagBits value

To determine the memory requirements for a plane of a disjoint image, add a VkImagePlaneMemoryRequirementsInfo structure to the pNext chain of the VkImageMemoryRequirementsInfo2 structure.

The VkImagePlaneMemoryRequirementsInfo structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkImagePlaneMemoryRequirementsInfo {
    VkStructureType sType;
    const void* pNext;
    VkImageAspectFlagBits planeAspect;
} VkImagePlaneMemoryRequirementsInfo;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- planeAspect is a VkImageAspectFlagBits value specifying the aspect corresponding to the image plane to query.

Valid Usage

- VUID-VkImagePlaneMemoryRequirementsInfo-planeAspect-02281 If the image's tiling is VK_IMAGE_TILING_LINEAR or VK_IMAGE_TILING_OPTIMAL, then planeAspect must be a single valid format plane for the image (that is, for a two-plane image planeAspect must be VK_IMAGE_ASPECT_PLANE_0_BIT or VK_IMAGE_ASPECT_PLANE_1_BIT, and for a three-plane image planeAspect must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT or VK_IMAGE_ASPECT_PLANE_2_BIT)
Valid Usage (Implicit)

- **VUID-VkImagePlaneMemoryRequirementsInfo-sType-sType**
  *sType* must be `VK_STRUCTURE_TYPE_IMAGE_PLANE_MEMORY_REQUIREMENTS_INFO`

- **VUID-VkImagePlaneMemoryRequirementsInfo-planeAspect-parameter**
  *planeAspect* must be a valid `VkImageAspectFlagBits` value

The `VkMemoryRequirements2` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkMemoryRequirements2 {
    VkStructureType sType;
    void* pNext;
    VkMemoryRequirements memoryRequirements;
} VkMemoryRequirements2;
```

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to a structure extending this structure.
- *memoryRequirements* is a `VkMemoryRequirements` structure describing the memory requirements of the resource.

Valid Usage (Implicit)

- **VUID-VkMemoryRequirements2-sType-sType**
  *sType* must be `VK_STRUCTURE_TYPE_MEMORY_REQUIREMENTS_2`

- **VUID-VkMemoryRequirements2-pNext-pNext**
  *pNext* must be `NULL` or a pointer to a valid instance of `VkMemoryDedicatedRequirements`

- **VUID-VkMemoryRequirements2-sType-unique**
  The *sType* value of each struct in the *pNext* chain must be unique

The `VkMemoryDedicatedRequirements` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkMemoryDedicatedRequirements {
    VkStructureType sType;
    void* pNext;
    VkBool32 prefersDedicatedAllocation;
    VkBool32 requiresDedicatedAllocation;
} VkMemoryDedicatedRequirements;
```

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to a structure extending this structure.
• `prefersDedicatedAllocation` specifies that the implementation would prefer a dedicated allocation for this resource. The application is still free to suballocate the resource but it may get better performance if a dedicated allocation is used.

• `requiresDedicatedAllocation` specifies that a dedicated allocation is required for this resource.

To determine the dedicated allocation requirements of a buffer or image resource, add a `VkMemoryDedicatedRequirements` structure to the `pNext` chain of the `VkMemoryRequirements2` structure passed as the `pMemoryRequirements` parameter of `vkGetBufferMemoryRequirements2` or `vkGetImageMemoryRequirements2`, respectively.

Constraints on the values returned for buffer resources are:

• `requiresDedicatedAllocation` may be `VK_TRUE` if the `pNext` chain of `VkBufferCreateInfo` for the call to `vkCreateBuffer` used to create the buffer being queried included a `VkExternalMemoryBufferCreateInfo` structure, and any of the handle types specified in `VkExternalMemoryBufferCreateInfo::handleTypes` requires dedicated allocation, as reported by `vkGetPhysicalDeviceExternalBufferProperties` in `VkExternalBufferProperties::externalMemoryProperties.externalMemoryFeatures`. Otherwise, `requiresDedicatedAllocation` will be `VK_FALSE`.

• When the implementation sets `requiresDedicatedAllocation` to `VK_TRUE`, it must also set `prefersDedicatedAllocation` to `VK_TRUE`.

• If `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` was set in `VkBufferCreateInfo::flags` when buffer was created, then both `prefersDedicatedAllocation` and `requiresDedicatedAllocation` will be `VK_FALSE`.

Constraints on the values returned for image resources are:

• `requiresDedicatedAllocation` may be `VK_TRUE` if the `pNext` chain of `VkImageCreateInfo` for the call to `vkCreateImage` used to create the image being queried included a `VkExternalMemoryImageCreateInfo` structure, and any of the handle types specified in `VkExternalMemoryImageCreateInfo::handleTypes` requires dedicated allocation, as reported by `vkGetPhysicalDeviceImageFormatProperties2` in `VkExternalImageFormatProperties::externalMemoryProperties.externalMemoryFeatures`.

• `requiresDedicatedAllocation` will otherwise be `VK_FALSE`.

• If `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` was set in `VkImageCreateInfo::flags` when image was created, then both `prefersDedicatedAllocation` and `requiresDedicatedAllocation` will be `VK_FALSE`.

Valid Usage (Implicit)

• VUID-VkMemoryDedicatedRequirements-sType-sType `sType` must be `VK_STRUCTURE_TYPE_MEMORY_DEDICATED_REQUIREMENTS`
VkBuffer
VkDeviceMemory
VkDeviceSize
buffer,
memory,
memoryOffset);

- **device** is the logical device that owns the buffer and memory.
- **buffer** is the buffer to be attached to memory.
- **memory** is a **VkDeviceMemory** object describing the device memory to attach.
- **memoryOffset** is the start offset of the region of **memory** which is to be bound to the buffer. The number of bytes returned in the **VkMemoryRequirements::size** member in **memory**, starting from **memoryOffset** bytes, will be bound to the specified buffer.

**vkBindBufferMemory** is equivalent to passing the same parameters through **VkBindBufferMemoryInfo** to **vkBindBufferMemory2**.

### Valid Usage

- VUID-vkBindBufferMemory-buffer-07459
  buffer must not have been bound to a memory object

- VUID-vkBindBufferMemory-buffer-01030
  buffer must not have been created with any sparse memory binding flags

- VUID-vkBindBufferMemory-memoryOffset-01031
  memoryOffset must be less than the size of **memory**

- VUID-vkBindBufferMemory-memory-01035
  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**

- VUID-vkBindBufferMemory-memoryOffset-01036
  memoryOffset must be an integer multiple of the **alignment** member of the **VkMemoryRequirements** structure returned from a call to **vkGetBufferMemoryRequirements** with **buffer**

- VUID-vkBindBufferMemory-size-01037
  The **size** member of the **VkMemoryRequirements** structure returned from a call to **vkGetBufferMemoryRequirements** with **buffer** must be less than or equal to the size of **memory** minus **memoryOffset**

- VUID-vkBindBufferMemory-buffer-01444
  If **buffer** requires a dedicated allocation (as reported by **vkGetBufferMemoryRequirements2** in **VkMemoryDedicatedRequirements** ::**requiresDedicatedAllocation** for **buffer**), memory must have been allocated with **VkMemoryDedicatedAllocateInfo**::**buffer** equal to **buffer**

- VUID-vkBindBufferMemory-memory-01508
  If the **VkMemoryAllocateInfo** provided when **memory** was allocated included a **VkMemoryDedicatedAllocateInfo** structure in its **pNext** chain, and **VkMemoryDedicatedAllocateInfo**::**buffer** was not **VK_NULL_HANDLE**, then **buffer** must
equal VkMemoryDedicatedAllocateInfo::buffer, and memoryOffset must be zero

- VUID-vkBindBufferMemory-None-01898
  If buffer was created with the VK_BUFFER_CREATE_PROTECTED_BIT bit set, the buffer must be bound to a memory object allocated with a memory type that reports VK_MEMORY_PROPERTY_PROTECTED_BIT

- VUID-vkBindBufferMemory-None-01899
  If buffer was created with the VK_BUFFER_CREATE_PROTECTED_BIT bit not set, the buffer must not be bound to a memory object allocated with a memory type that reports VK_MEMORY_PROPERTY_PROTECTED_BIT

- VUID-vkBindBufferMemory-memory-02726
  If the value of VkExportMemoryAllocateInfo::handleTypes used to allocate memory is not 0, it must include at least one of the handles set in VkExternalMemoryBufferCreateInfo::handleTypes when buffer was created

- VUID-vkBindBufferMemory-memory-02727
  If memory was allocated by a memory import operation, the external handle type of the imported memory must also have been set in VkExternalMemoryBufferCreateInfo::handleTypes when buffer was created

- VUID-vkBindBufferMemory-bufferDeviceAddress-03339
  If the VkPhysicalDeviceBufferDeviceAddressFeatures::bufferDeviceAddress feature is enabled and buffer was created with the VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT bit set, memory must have been allocated with the VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT bit set

Valid Usage (Implicit)

- VUID-vkBindBufferMemory-device-parameter
device must be a valid VkDevice handle

- VUID-vkBindBufferMemory-buffer-parameter
buffer must be a valid VkBuffer handle

- VUID-vkBindBufferMemory-memory-parameter
memory must be a valid VkDeviceMemory handle

- VUID-vkBindBufferMemory-buffer-parent
buffer must have been created, allocated, or retrieved from device

- VUID-vkBindBufferMemory-memory-parent
memory must have been created, allocated, or retrieved from device

Host Synchronization

- Host access to buffer must be externally synchronized
To attach memory to buffer objects for one or more buffers at a time, call:

```c
// Provided by VK_VERSION_1_1
VkResult vkBindBufferMemory2(
   VkDevice device,
    uint32_t bindInfoCount,
    const VkBindBufferMemoryInfo* pBindInfos);
```

- `device` is the logical device that owns the buffers and memory.
- `bindInfoCount` is the number of elements in `pBindInfos`.
- `pBindInfos` is a pointer to an array of `bindInfoCount` `VkBindBufferMemoryInfo` structures describing buffers and memory to bind.

On some implementations, it may be more efficient to batch memory bindings into a single command.

### Valid Usage (Implicit)

- VUID-vkBindBufferMemory2-device-parameter
  - `device` must be a valid `VkDevice` handle

- VUID-vkBindBufferMemory2-pBindInfos-parameter
  - `pBindInfos` must be a valid pointer to an array of `bindInfoCount` valid `VkBindBufferMemoryInfo` structures

- VUID-vkBindBufferMemory2-bindInfoCount-arraylength
  - `bindInfoCount` must be greater than 0
 VkBindBufferMemoryInfo contains members corresponding to the parameters of 
vkBindBufferMemory.

The VkBindBufferMemoryInfo structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkBindBufferMemoryInfo {
    VkStructureType sType;
    const void* pNext;
    VkBuffer buffer;
    VkDeviceMemory memory;
    VkDeviceSize memoryOffset;
} VkBindBufferMemoryInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **buffer** is the buffer to be attached to memory.
- **memory** is a VkDeviceMemory object describing the device memory to attach.
- **memoryOffset** is the start offset of the region of memory which is to be bound to the buffer. The number of bytes returned in the VkMemoryRequirements::size member in memory, starting from memoryOffset bytes, will be bound to the specified buffer.

### Valid Usage

- **VUID-VkBindBufferMemoryInfo-buffer-07459**
  buffer must not have been bound to a memory object

- **VUID-VkBindBufferMemoryInfo-buffer-01030**
  buffer must not have been created with any sparse memory binding flags

- **VUID-VkBindBufferMemoryInfo-memoryOffset-01031**
  memoryOffset must be less than the size of memory

- **VUID-VkBindBufferMemoryInfo-memory-01035**
  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

- **VUID-VkBindBufferMemoryInfo-memoryOffset-01036**
  memoryOffset must be an integer multiple of the alignment member of the VkMemoryRequirements structure returned from a call to vkGetBufferMemoryRequirements with buffer

- **VUID-VkBindBufferMemoryInfo-size-01037**
  The size member of the VkMemoryRequirements structure returned from a call to vkGetBufferMemoryRequirements with buffer must be less than or equal to the size of memory minus memoryOffset

- **VUID-VkBindBufferMemoryInfo-buffer-01444**
If \texttt{buffer} requires a dedicated allocation (as reported by \texttt{vkGetBufferMemoryRequirements2} in \texttt{VkMemoryDedicatedRequirements::requiresDedicatedAllocation} for \texttt{buffer}), memory \textbf{must} have been allocated with \texttt{VkMemoryDedicatedAllocateInfo::buffer} equal to \texttt{buffer}

- **VUID-VkBindBufferMemoryInfo-memory-01508**
  If the \texttt{VkMemoryAllocateInfo} provided when \texttt{memory} was allocated included a \texttt{VkMemoryDedicatedAllocateInfo} structure in its \texttt{pNext} chain, and \texttt{VkMemoryDedicatedAllocateInfo::buffer} was not \texttt{VK_NULL_HANDLE}, then \texttt{buffer} \textbf{must} equal \texttt{VkMemoryDedicatedAllocateInfo::buffer}, and \texttt{memoryOffset} \textbf{must} be zero

- **VUID-VkBindBufferMemoryInfo-None-01898**
  If \texttt{buffer} was created with the \texttt{VK_BUFFER_CREATE_PROTECTED_BIT} bit set, the buffer \textbf{must} be bound to a memory object allocated with a memory type that reports \texttt{VK_MEMORY_PROPERTY_PROTECTED_BIT}

- **VUID-VkBindBufferMemoryInfo-None-01899**
  If \texttt{buffer} was created with the \texttt{VK_BUFFER_CREATE_PROTECTED_BIT} bit not set, the buffer \textbf{must} not be bound to a memory object allocated with a memory type that reports \texttt{VK_MEMORY_PROPERTY_PROTECTED_BIT}

- **VUID-VkBindBufferMemoryInfo-memory-02726**
  If the value of \texttt{VkExportMemoryAllocateInfo::handleTypes} used to allocate \texttt{memory} is not 0, it \textbf{must} include at least one of the handles set in \texttt{VkExternalMemoryBufferCreateInfo::handleTypes} when \texttt{buffer} was created

- **VUID-VkBindBufferMemoryInfo-memory-02727**
  If \texttt{memory} was allocated by a memory import operation, the external handle type of the imported memory \textbf{must} also have been set in \texttt{VkExternalMemoryBufferCreateInfo::handleTypes} when \texttt{buffer} was created

- **VUID-VkBindBufferMemoryInfo-bufferDeviceAddress-03339**
  If the \texttt{VkPhysicalDeviceBufferDeviceAddressFeatures::bufferDeviceAddress} feature is enabled and \texttt{buffer} was created with the \texttt{VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT} bit set, \texttt{memory} \textbf{must} have been allocated with the \texttt{VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT} bit set

- **VUID-VkBindBufferMemoryInfo-pNext-01605**
  If the \texttt{pNext} chain includes a \texttt{VkBindBufferMemoryDeviceGroupInfo} structure, all instances of \texttt{memory} specified by \texttt{VkBindBufferMemoryDeviceGroupInfo::pDeviceIndices} \textbf{must} have been allocated

**Valid Usage (Implicit)**

- **VUID-VkBindBufferMemoryInfo-sType-sType**
  \texttt{sType} \textbf{must} be \texttt{VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO}

- **VUID-VkBindBufferMemoryInfo-pNext-pNext**
  \texttt{pNext} \textbf{must} be \texttt{NULL} or a pointer to a valid instance of \texttt{VkBindBufferMemoryDeviceGroupInfo}

- **VUID-VkBindBufferMemoryInfo-sType-unique**
The sType value of each struct in the pNext chain must be unique

- VUID-VkBindBufferMemoryInfo-buffer-parameter
  buffer must be a valid VkBuffer handle

- VUID-VkBindBufferMemoryInfo-memory-parameter
  memory must be a valid VkDeviceMemory handle

- VUID-VkBindBufferMemoryInfo-commonparent
  Both of buffer, and memory must have been created, allocated, or retrieved from the same VkDevice

The VkBindBufferMemoryDeviceGroupInfo structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkBindBufferMemoryDeviceGroupInfo {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t            deviceIndexCount;
    const uint32_t*     pDeviceIndices;
} VkBindBufferMemoryDeviceGroupInfo;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- deviceIndexCount is the number of elements in pDeviceIndices.
- pDeviceIndices is a pointer to an array of device indices.

If the pNext chain of VkBindBufferMemoryInfo includes a VkBindBufferMemoryDeviceGroupInfo structure, then that structure determines how memory is bound to buffers across multiple devices in a device group.

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

If deviceIndexCount is zero and memory comes from a memory heap with the VK_MEMORY_HEAP_MULTI_INSTANCE_BIT bit set, then it is as if pDeviceIndices contains consecutive indices from zero to the number of physical devices in the logical device, minus one. In other words, by default each physical device attaches to its own instance of memory.

If deviceIndexCount is zero and memory comes from a memory heap without the VK_MEMORY_HEAP_MULTI_INSTANCE_BIT bit set, then it is as if pDeviceIndices contains an array of zeros. In other words, by default each physical device attaches to instance zero.

Valid Usage

- VUID-VkBindBufferMemoryDeviceGroupInfo-deviceIndexCount-01606
deviceIndexCount must either be zero or equal to the number of physical devices in the logical device
All elements of pDeviceIndices must be valid device indices

Valid Usage (Implicit)

- VUID-VkBindBufferMemoryDeviceGroupInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_DEVICE_GROUP_INFO

- VUID-VkBindBufferMemoryDeviceGroupInfo-pDeviceIndices-parameter
  If deviceIndexCount is not 0, pDeviceIndices must be a valid pointer to an array of deviceIndexCount uint32_t values

To attach memory to a VkImage object created without the VK_IMAGE_CREATE_DISJOINT_BIT set, call:

```
// Provided by VK_VERSION_1_0
VkResult vkBindImageMemory(
    VkDevice device,          
    VkImage image,            
    VkDeviceMemory memory,    
    VkDeviceSize memoryOffset);
```

- device is the logical device that owns the image and memory.
- image is the image.
- memory is the VkDeviceMemory object describing the device memory to attach.
- memoryOffset is the start offset of the region of memory which is to be bound to the image. The number of bytes returned in the VkMemoryRequirements::size member in memory, starting from memoryOffset bytes, will be bound to the specified image.

vkBindImageMemory is equivalent to passing the same parameters through VkBindImageMemoryInfo to vkBindImageMemory2.

Valid Usage

- VUID-vkBindImageMemory-image-07460
  image must not have been bound to a memory object

- VUID-vkBindImageMemory-image-01045
  image must not have been created with any sparse memory binding flags

- VUID-vkBindImageMemory-memoryOffset-01046
  memoryOffset must be less than the size of memory

- VUID-vkBindImageMemory-image-01445
  If image requires a dedicated allocation (as reported by vkGetImageMemoryRequirements2 in VkMemoryDedicatedRequirements::requiresDedicatedAllocation for image), memory must have been created with VkMemoryDedicatedAllocateInfo::image equal to image
If the `VkMemoryAllocateInfo` provided when `memory` was allocated included a `VkMemoryDedicatedAllocateInfo` structure in its `pNext` chain, and `VkMemoryDedicatedAllocateInfo::image` was not `VK_NULL_HANDLE`, then `image` must equal `VkMemoryDedicatedAllocateInfo::image` and `memoryOffset` must be zero.

If `image` was created with the `VK_IMAGE_CREATE_PROTECTED_BIT` bit set, the image must be bound to a memory object allocated with a memory type that reports `VK_MEMORY_PROPERTY_PROTECTED_BIT`.

If `image` was created with the `VK_IMAGE_CREATE_PROTECTED_BIT` bit not set, the image must not be bound to a memory object created with a memory type that reports `VK_MEMORY_PROPERTY_PROTECTED_BIT`.

If the value of `VkExportMemoryAllocateInfo::handleTypes` used to allocate `memory` is not 0, it must include at least one of the handles set in `VkExternalMemoryImageCreateInfo::handleTypes` when `image` was created.

If `memory` was created by a memory import operation, the external handle type of the imported memory must also have been set in `VkExternalMemoryImageCreateInfo::handleTypes` when `image` was created.

`image` must not have been created with the `VK_IMAGE_CREATE_DISJOINT_BIT` set.

`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 difference of the size of `memory` and `memoryOffset` must be greater than or equal to the `size` member of the `VkMemoryRequirements` structure returned from a call to `vkGetImageMemoryRequirements` with the same `image`.

Valid Usage (Implicit)

- `device` must be a valid `VkDevice` handle.
- `image` must be a valid `VkImage` handle.
- `memory` must be a valid `VkDeviceMemory` handle.
memory must be a valid VkDeviceMemory handle

- VUID-vkBindImageMemory-image-parent
  image must have been created, allocated, or retrieved from device
- VUID-vkBindImageMemory-memory-parent
  memory must have been created, allocated, or retrieved from device

**Host Synchronization**

- Host access to image must be externally synchronized

**Return Codes**

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

To attach memory to image objects for one or more images at a time, call:

```cpp
// Provided by VK_VERSION_1_1
VkResult vkBindImageMemory2(
  VkDevice device,
  uint32_t bindInfoCount,
  const VkBindImageMemoryInfo* pBindInfos);
```

- device is the logical device that owns the images and memory.
- bindInfoCount is the number of elements in pBindInfos.
- pBindInfos is a pointer to an array of VkBindImageMemoryInfo structures, describing images and memory to bind.

On some implementations, it may be more efficient to batch memory bindings into a single command.

**Valid Usage**

- VUID-vkBindImageMemory2-pBindInfos-02858
  If any VkBindImageMemoryInfo::image was created with VK_IMAGE_CREATE_DISJOINT_BIT then all planes of VkBindImageMemoryInfo::image must be bound individually in separate pBindInfos
- VUID-vkBindImageMemory2-pBindInfos-04006
**pBindInfos** must not refer to the same image subresource more than once

### Valid Usage (Implicit)

- VUID-vkBindImageMemory2-device-parameter
  - **device** must be a valid **VkDevice** handle
- VUID-vkBindImageMemory2-pBindInfos-parameter
  - **pBindInfos** must be a valid pointer to an array of **bindInfoCount** valid **VkBindImageMemoryInfo** structures
- VUID-vkBindImageMemory2-bindInfoCount-arraylength
  - **bindInfoCount** must be greater than 0

### Return Codes

**Success**
- **VK_SUCCESS**

**Failure**
- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**

**VkBindImageMemoryInfo** contains members corresponding to the parameters of **vkBindImageMemory**.

The **VkBindImageMemoryInfo** structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkBindImageMemoryInfo {
    VkStructureType sType;
    const void* pNext;
    VkImage image;
    VkDeviceMemory memory;
    VkDeviceSize memoryOffset;
} VkBindImageMemoryInfo;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **image** is the image to be attached to memory.
- **memory** is a **VkDeviceMemory** object describing the device memory to attach.
- **memoryOffset** is the start offset of the region of **memory** which is to be bound to the image. The number of bytes returned in the **VkMemoryRequirements::size** member in **memory**, starting from **memoryOffset** bytes, will be bound to the specified image.
Valid Usage

• VUID-VkBindImageMemoryInfo-image-07460
  image must not have been bound to a memory object

• VUID-VkBindImageMemoryInfo-image-01045
  image must not have been created with any sparse memory binding flags

• VUID-VkBindImageMemoryInfo-memoryOffset-01046
  memoryOffset must be less than the size of memory

• VUID-VkBindImageMemoryInfo-image-01445
  If image requires a dedicated allocation (as reported by vkGetImageMemoryRequirements2 in VkMemoryDedicatedRequirements::requiresDedicatedAllocation for image), memory must have been created with VkMemoryDedicatedAllocateInfo::image equal to image

• VUID-VkBindImageMemoryInfo-memory-01509
  If the VkMemoryAllocateInfo provided when memory was allocated included a VkMemoryDedicatedAllocateInfo structure in its pNext chain, and VkMemoryDedicatedAllocateInfo::image was not VK_NULL_HANDLE, then image must equal VkMemoryDedicatedAllocateInfo::image and memoryOffset must be zero

• VUID-VkBindImageMemoryInfo-None-01901
  If image was created with the VK_IMAGE_CREATE_PROTECTED_BIT bit set, the image must be bound to a memory object allocated with a memory type that reports VK_MEMORY_PROPERTY_PROTECTED_BIT

• VUID-VkBindImageMemoryInfo-None-01902
  If image was created with the VK_IMAGE_CREATE_PROTECTED_BIT bit not set, the image must not be bound to a memory object created with a memory type that reports VK_MEMORY_PROPERTY_PROTECTED_BIT

• VUID-VkBindImageMemoryInfo-memory-02728
  If the value of VkExportMemoryAllocateInfo::handleTypes used to allocate memory is not 0, it must include at least one of the handles set in VkExternalMemoryImageCreateInfo::handleTypes when image was created

• VUID-VkBindImageMemoryInfo-memory-02729
  If memory was created by a memory import operation, the external handle type of the imported memory must also have been set in VkExternalMemoryImageCreateInfo::handleTypes when image was created

• VUID-VkBindImageMemoryInfo-pNext-01615
  If the pNext chain does not include a VkBindImagePlaneMemoryInfo structure, memory must have been allocated using one of the memory types allowed in the memoryTypeBits member of the VkMemoryRequirements structure returned from a call to vkGetImageMemoryRequirements2 with image

• VUID-VkBindImageMemoryInfo-pNext-01616
  If the pNext chain does not include a VkBindImagePlaneMemoryInfo structure, memoryOffset must be an integer multiple of the alignment member of the VkMemoryRequirements structure returned from a call to
vkGetImageMemoryRequirements2 with image

- VUID-VkBindImageMemoryInfo-pNext-01617
  If the pNext chain does not include a VkBindImagePlaneMemoryInfo structure, the difference of the size of memory and memoryOffset must be greater than or equal to the size member of the VkMemoryRequirements structure returned from a call to vkGetImageMemoryRequirements2 with the same image

- VUID-VkBindImageMemoryInfo-pNext-01618
  If the pNext chain includes a VkBindImagePlaneMemoryInfo structure, image must have been created with the VK_IMAGE_CREATE_DISJOINT_BIT bit set

- VUID-VkBindImageMemoryInfo-image-07736
  If image was created with the VK_IMAGE_CREATE_DISJOINT_BIT bit set, then the pNext chain must include a VkBindImagePlaneMemoryInfo structure

- VUID-VkBindImageMemoryInfo-pNext-01619
  If the pNext chain includes a VkBindImagePlaneMemoryInfo structure, memory must have been allocated using one of the memory types allowed in the memoryTypeBits member of the VkMemoryRequirements structure returned from a call to vkGetImageMemoryRequirements2 with image and where VkBindImagePlaneMemoryInfo::planeAspect corresponds to the VkImagePlaneMemoryRequirementsInfo::planeAspect in the VkImageMemoryRequirementsInfo2 structure's pNext chain

- VUID-VkBindImageMemoryInfo-pNext-01620
  If the pNext chain includes a VkBindImagePlaneMemoryInfo structure, memoryOffset must be an integer multiple of the alignment member of the VkMemoryRequirements structure returned from a call to vkGetImageMemoryRequirements2 with image and where VkBindImagePlaneMemoryInfo::planeAspect corresponds to the VkImagePlaneMemoryRequirementsInfo::planeAspect in the VkImageMemoryRequirementsInfo2 structure's pNext chain

- VUID-VkBindImageMemoryInfo-pNext-01621
  If the pNext chain includes a VkBindImagePlaneMemoryInfo structure, the difference of the size of memory and memoryOffset must be greater than or equal to the size member of the VkMemoryRequirements structure returned from a call to vkGetImageMemoryRequirements2 with the same image and where VkBindImagePlaneMemoryInfo::planeAspect corresponds to the VkImagePlaneMemoryRequirementsInfo::planeAspect in the VkImageMemoryRequirementsInfo2 structure's pNext chain

- VUID-VkBindImageMemoryInfo-memory-01625
  memory must be a valid VkDeviceMemory handle

- VUID-VkBindImageMemoryInfo-pNext-01626
  If the pNext chain includes a VkBindImageMemoryDeviceGroupInfo structure, all instances of memory specified by VkBindImageMemoryDeviceGroupInfo::pDeviceIndices must have been allocated

- VUID-VkBindImageMemoryInfo-pNext-01627
  If the pNext chain includes a VkBindImageMemoryDeviceGroupInfo structure, and VkBindImageMemoryDeviceGroupInfo::splitInstanceBindRegionCount is not zero, then
**Valid Usage (Implicit)**

- **VUID-VkBindImageMemoryInfo-sType-sType**
  sType must be `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_INFO`

- **VUID-VkBindImageMemoryInfo-pNext-pNext**
  Each pNext member of any structure (including this one) in the pNext chain must be either NULL or a pointer to a valid instance of VkBindImageMemoryDeviceGroupInfo or VkBindImagePlaneMemoryInfo

- **VUID-VkBindImageMemoryInfo-sType-unique**
  The sType value of each struct in the pNext chain must be unique

- **VUID-VkBindImageMemoryInfo-image-parameter**
  image must be a valid VkImage handle

- **VUID-VkBindImageMemoryInfo-commonparent**
  Both of image, and memory that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same VkDevice

The VkBindImageMemoryDeviceGroupInfo structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkBindImageMemoryDeviceGroupInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t deviceIndexCount;
    const uint32_t* pDeviceIndices;
    uint32_t splitInstanceBindRegionCount;
    const VkRect2D* pSplitInstanceBindRegions;
} VkBindImageMemoryDeviceGroupInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **deviceIndexCount** is the number of elements in pDeviceIndices.
• `pDeviceIndices` is a pointer to an array of device indices.

• `splitInstanceBindRegionCount` is the number of elements in `pSplitInstanceBindRegions`.

• `pSplitInstanceBindRegions` is a pointer to an array of `VkRect2D` structures describing which regions of the image are attached to each instance of memory.

If the `pNext` chain of `VkBindImageMemoryInfo` includes a `VkBindImageMemoryDeviceGroupInfo` structure, then that structure determines how memory is bound to images across multiple devices in a device group.

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

Let N be the number of physical devices in the logical device. If `splitInstanceBindRegionCount` is greater than zero, then `pSplitInstanceBindRegions` is a pointer to an array of $N^2$ rectangles, where the image region specified by the rectangle at element $i*N+j$ in resource instance $i$ is bound to the memory instance $j$. The blocks of the memory that are bound to each sparse image block region use an offset in memory, relative to `memoryOffset`, computed as if the whole image was being bound to a contiguous range of memory. In other words, horizontally adjacent image blocks use consecutive blocks of memory, vertically adjacent image blocks are separated by the number of bytes per block multiplied by the width in blocks of `image`, and the block at (0,0) corresponds to memory starting at `memoryOffset`.

If `splitInstanceBindRegionCount` and `deviceIndexCount` are zero and the memory comes from a memory heap with the `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` bit set, then it is as if `pDeviceIndices` contains consecutive indices from zero to the number of physical devices in the logical device, minus one. In other words, by default each physical device attaches to its own instance of the memory.

If `splitInstanceBindRegionCount` and `deviceIndexCount` are zero and the memory comes from a memory heap without the `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` bit set, then it is as if `pDeviceIndices` contains an array of zeros. In other words, by default each physical device attaches to instance zero.

---

**Valid Usage**

- VUID-VkBindImageMemoryDeviceGroupInfo-deviceIndexCount-01633
  At least one of `deviceIndexCount` and `splitInstanceBindRegionCount` must be zero

- VUID-VkBindImageMemoryDeviceGroupInfo-deviceIndexCount-01634
  `deviceIndexCount` must either be zero or equal to the number of physical devices in the logical device

- VUID-VkBindImageMemoryDeviceGroupInfo-pDeviceIndices-01635
  All elements of `pDeviceIndices` must be valid device indices

- VUID-VkBindImageMemoryDeviceGroupInfo-splitInstanceBindRegionCount-01636
  `splitInstanceBindRegionCount` must either be zero or equal to the number of physical devices in the logical device squared

- VUID-VkBindImageMemoryDeviceGroupInfo-pSplitInstanceBindRegions-01637
  Elements of `pSplitInstanceBindRegions` that correspond to the same instance of an image
must not overlap

- **VUID-VkBindImageMemoryDeviceGroupInfo-offset-01638**
  The offset.x member of any element of pSplitInstanceBindRegions must be a multiple of the sparse image block width (VkSparseImageFormatProperties::imageGranularity.width) of all non-metadata aspects of the image

- **VUID-VkBindImageMemoryDeviceGroupInfo-offset-01639**
  The offset.y member of any element of pSplitInstanceBindRegions must be a multiple of the sparse image block height (VkSparseImageFormatProperties::imageGranularity.height) of all non-metadata aspects of the image

- **VUID-VkBindImageMemoryDeviceGroupInfo-extent-01640**
  The extent.width member of any element of pSplitInstanceBindRegions must either be a multiple of the sparse image block width of all non-metadata aspects of the image, or else extent.width + offset.x must equal the width of the image subresource

- **VUID-VkBindImageMemoryDeviceGroupInfo-extent-01641**
  The extent.height member of any element of pSplitInstanceBindRegions must either be a multiple of the sparse image block height of all non-metadata aspects of the image, or else extent.height + offset.y must equal the height of the image subresource

**Valid Usage (Implicit)**

- **VUID-VkBindImageMemoryDeviceGroupInfo-sType-sType**
  sType must be VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_DEVICE_GROUP_INFO

- **VUID-VkBindImageMemoryDeviceGroupInfo-pDeviceIndices-parameter**
  If deviceIndexCount is not 0, pDeviceIndices must be a valid pointer to an array of deviceIndexCount uint32_t values

- **VUID-VkBindImageMemoryDeviceGroupInfo-pSplitInstanceBindRegions-parameter**
  If splitInstanceBindRegionCount is not 0, pSplitInstanceBindRegions must be a valid pointer to an array of splitInstanceBindRegionCount VkRect2D structures

In order to bind planes of a disjoint image, add a VkBindImagePlaneMemoryInfo structure to the pNext chain of VkBindImageMemoryInfo.

The VkBindImagePlaneMemoryInfo structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkBindImagePlaneMemoryInfo {
    VkStructureType sType;
    const void* pNext;
    VkImageAspectFlagBits planeAspect;
} VkBindImagePlaneMemoryInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
• planeAspect is a VkImageAspectFlagBits value specifying the aspect of the disjoint image plane to bind.

### Valid Usage

• VUID-VkBindImagePlaneMemoryInfo-planeAspect-02283
  If the image's tiling is VK_IMAGE_TILING_LINEAR or VK_IMAGE_TILING_OPTIMAL, then planeAspect must be a single valid format plane for the image (that is, for a two-plane image planeAspect must be VK_IMAGE_ASPECT_PLANE_0_BIT or VK_IMAGE_ASPECT_PLANE_1_BIT, and for a three-plane image planeAspect must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or VK_IMAGE_ASPECT_PLANE_2_BIT).

### Valid Usage (Implicit)

• VUID-VkBindImagePlaneMemoryInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_BIND_IMAGE_PLANE_MEMORY_INFO

• VUID-VkBindImagePlaneMemoryInfo-planeAspect-parameter
  planeAspect must be a valid VkImageAspectFlagBits value

### Buffer-Image Granularity

The implementation-dependent limit bufferImageGranularity specifies a page-like granularity at which linear and non-linear resources must be placed in adjacent memory locations to avoid aliasing. Two resources which do not satisfy this granularity requirement are said to alias. bufferImageGranularity is specified in bytes, and must be a power of two. Implementations which do not impose a granularity restriction may report a bufferImageGranularity value of one.

**Note**

Despite its name, bufferImageGranularity is really a granularity between “linear” and “non-linear” resources.

Given resourceA at the lower memory offset and resourceB at the higher memory offset in the same VkDeviceMemory object, where one resource is linear and the other is non-linear (as defined in the Glossary), and the following:

\[
\begin{align*}
\text{resourceA.end} & = \text{resourceA.memoryOffset} + \text{resourceA.size} - 1 \\
\text{resourceA.endPage} & = \text{resourceA.end} \&\!\& (\text{bufferImageGranularity} - 1) \\
\text{resourceB.start} & = \text{resourceB.memoryOffset} \\
\text{resourceB.startPage} & = \text{resourceB.start} \&\!\& (\text{bufferImageGranularity} - 1)
\end{align*}
\]

The following property must hold:

\[
\text{resourceA.endPage} < \text{resourceB.startPage}
\]
That is, the end of the first resource (A) and the beginning of the second resource (B) must be on separate “pages” of size bufferImageGranularity. bufferImageGranularity may be different than the physical page size of the memory heap. This restriction is only needed when a linear resource and a non-linear resource are adjacent in memory and will be used simultaneously. The memory ranges of adjacent resources can be closer than bufferImageGranularity, provided they meet the alignment requirement for the objects in question.

Sparse block size in bytes and sparse image and buffer memory alignments must all be multiples of the bufferImageGranularity. Therefore, memory bound to sparse resources naturally satisfies the bufferImageGranularity.

### 12.7. Resource Sharing Mode

Buffer and image objects are created with a sharing mode controlling how they can be accessed from queues. The supported sharing modes are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkSharingMode {
    VK_SHARING_MODE_EXCLUSIVE = 0,
    VK_SHARING_MODE_CONCURRENT = 1,
} VkSharingMode;
```

- **VK_SHARING_MODE_EXCLUSIVE** specifies that access to any range or image subresource of the object will be exclusive to a single queue family at a time.
- **VK_SHARING_MODE_CONCURRENT** specifies that concurrent access to any range or image subresource of the object from multiple queue families is supported.

**Note**

VK_SHARING_MODE_CONCURRENT may result in lower performance access to the buffer or image than VK_SHARING_MODE_EXCLUSIVE.

RANGES of buffers and image subresources of image objects created using VK_SHARING_MODE_EXCLUSIVE must only be accessed by queues in the queue family that has ownership of the resource. Upon creation, such resources are not owned by any queue family; ownership is implicitly acquired upon first use within a queue. Once a resource using VK_SHARING_MODE_EXCLUSIVE is owned by some queue family, the application must perform a queue family ownership transfer to make the memory contents of a range or image subresource accessible to a different queue family.

**Note**

Images still require a layout transition from VK_IMAGE_LAYOUT_UNDEFINED or VK_IMAGE_LAYOUT_PREINITIALIZED before being used on the first queue.

A queue family can take ownership of an image subresource or buffer range of a resource created with VK_SHARING_MODE_EXCLUSIVE, without an ownership transfer, in the same way as for a resource that was just created; however, taking ownership in this way has the effect that the contents of the image subresource or buffer range are undefined.
Ranges of buffers and image subresources of image objects created using `VK_SHARING_MODE_CONCURRENT` must only be accessed by queues from the queue families specified through the `queueFamilyIndexCount` and `pQueueFamilyIndices` members of the corresponding create info structures.

## 12.7.1. External Resource Sharing

Resources should only be accessed in the Vulkan instance that has exclusive ownership of their underlying memory. Only one Vulkan instance has exclusive ownership of a resource's underlying memory at a given time, regardless of whether the resource was created using `VK_SHARING_MODE_EXCLUSIVE` or `VK_SHARING_MODE_CONCURRENT`. Applications can transfer ownership of a resource's underlying memory only if the memory has been imported from or exported to another instance or external API using external memory handles. The semantics for transferring ownership outside of the instance are similar to those used for transferring ownership of `VK_SHARING_MODE_EXCLUSIVE` resources between queues, and is also accomplished using `VkBufferMemoryBarrier` or `VkImageMemoryBarrier` operations. To make the contents of the underlying memory accessible in the destination instance or API, applications must

1. Release exclusive ownership from the source instance or API.
2. Ensure the release operation has completed using semaphores or fences.
3. Acquire exclusive ownership in the destination instance or API.

Unlike queue ownership transfers, the destination instance or API is not specified explicitly when releasing ownership, nor is the source instance or API specified when acquiring ownership. Instead, the image or memory barrier's `dstQueueFamilyIndex` or `srcQueueFamilyIndex` parameters are set to the reserved queue family index `VK_QUEUE_FAMILY_EXTERNAL` to represent the external destination or source respectively.

Binding a resource to a memory object shared between multiple Vulkan instances or other APIs does not change the ownership of the underlying memory. The first entity to access the resource implicitly acquires ownership. An entity can also implicitly take ownership from another entity in the same way without an explicit ownership transfer. However, taking ownership in this way has the effect that the contents of the underlying memory are undefined.

Accessing a resource backed by memory that is owned by a particular instance or API has the same semantics as accessing a `VK_SHARING_MODE_EXCLUSIVE` resource, with one exception: Implementations must ensure layout transitions performed on one member of a set of identical subresources of identical images that alias the same range of an underlying memory object affect the layout of all the subresources in the set.

As a corollary, writes to any image subresources in such a set must not make the contents of memory used by other subresources in the set undefined. An application can define the content of a subresource of one image by performing device writes to an identical subresource of another image provided both images are bound to the same region of external memory. Applications may also add resources to such a set after the content of the existing set members has been defined without making the content undefined by creating a new image with the initial layout `VK_IMAGE_LAYOUT_UNDEFINED` and binding it to the same region of external memory as the existing images.
Note

Because layout transitions apply to all identical images aliasing the same region of external memory, the actual layout of the memory backing a new image as well as an existing image with defined content will not be undefined. Such an image is not usable until it acquires ownership of its memory from the existing owner. Therefore, the layout specified as part of this transition will be the true initial layout of the image. The undefined layout specified when creating it is a placeholder to simplify valid usage requirements.

12.8. Memory Aliasing

A range of a `VkDeviceMemory` allocation is aliased if it is bound to multiple resources simultaneously, as described below, via `vkBindImageMemory`, `vkBindBufferMemory`, via sparse memory bindings, or by binding the memory to resources in multiple Vulkan instances or external APIs using external memory handle export and import mechanisms.

Consider two resources, resource_A and resource_B, bound respectively to memory range_A and range_B. Let paddedRange_A and paddedRange_B be, respectively, range_A and range_B aligned to `bufferImageGranularity`. If the resources are both linear or both non-linear (as defined in the Glossary), then the resources alias the memory in the intersection of range_A and range_B. If one resource is linear and the other is non-linear, then the resources alias the memory in the intersection of paddedRange_A and paddedRange_B.

Applications can alias memory, but use of multiple aliases is subject to several constraints.

Note

Memory aliasing can be useful to reduce the total device memory footprint of an application, if some large resources are used for disjoint periods of time.

When a non-linear, non-`VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` image is bound to an aliased range, all image subresources of the image overlap the range. When a linear image is bound to an aliased range, the image subresources that (according to the image’s advertised layout) include bytes from the aliased range overlap the range. When a `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` image has sparse image blocks bound to an aliased range, only image subresources including those sparse image blocks overlap the range, and when the memory bound to the image’s mip tail overlaps an aliased range all image subresources in the mip tail overlap the range.

Buffers, and linear image subresources in either the `VK_IMAGE_LAYOUT_PREINITIALIZED` or `VK_IMAGE_LAYOUT_GENERAL` layouts, are host-accessible subresources. That is, the host has a well-defined addressing scheme to interpret the contents, and thus the layout of the data in memory can be consistently interpreted across aliases if each of those aliases is a host-accessible subresource. Non-linear images, and linear image subresources in other layouts, are not host-accessible.

If two aliases are both host-accessible, then they interpret the contents of the memory in consistent ways, and data written to one alias can be read by the other alias.

If two aliases are both images that were created with identical creation parameters, both were created with the `VK_IMAGE_CREATE_ALIAS_BIT` flag set, and both are bound identically to memory.
except for `VkBindImageMemoryDeviceGroupInfo::pDeviceIndices` and `VkBindImageMemoryDeviceGroupInfo::SplitInstanceBindRegions`, then they interpret the contents of the memory in consistent ways, and data written to one alias can be read by the other alias.

Additionally, if an individual plane of a multi-planar image and a single-plane image alias the same memory, then they also interpret the contents of the memory in consistent ways under the same conditions, but with the following modifications:

- Both must have been created with the `VK_IMAGE_CREATE_DISJOINT_BIT` flag.
- The single-plane image must have a `VkFormat` that is equivalent to that of the multi-planar image’s individual plane.
- The single-plane image and the individual plane of the multi-planar image must be bound identically to memory except for `VkBindImageMemoryDeviceGroupInfo::pDeviceIndices` and `VkBindImageMemoryDeviceGroupInfo::pSplitInstanceBindRegions`.
- The width and height of the single-plane image are derived from the multi-planar image’s dimensions in the manner listed for plane compatibility for the aliased plane.
- All other creation parameters must be identical.

Aliases created by binding the same memory to resources in multiple Vulkan instances or external APIs using external memory handle export and import mechanisms interpret the contents of the memory in consistent ways, and data written to one alias can be read by the other alias.

Otherwise, the aliases interpret the contents of the memory differently, and writes via one alias make the contents of memory partially or completely undefined to the other alias. If the first alias is a host-accessible subresource, then the bytes affected are those written by the memory operations according to its addressing scheme. If the first alias is not host-accessible, then the bytes affected are those overlapped by the image subresources that were written. If the second alias is a host-accessible subresource, the affected bytes become undefined. If the second alias is not host-accessible, all sparse image blocks (for sparse partially-resident images) or all image subresources (for non-sparse image and fully resident sparse images) that overlap the affected bytes become undefined.

If any image subresources are made undefined due to writes to an alias, then each of those image subresources must have its layout transitioned from `VK_IMAGE_LAYOUT_UNDEFINED` to a valid layout before it is used, or from `VK_IMAGE_LAYOUT_PREINITIALIZED` if the memory has been written by the host. If any sparse blocks of a sparse image have been made undefined, then only the image subresources containing them must be transitioned.

Use of an overlapping range by two aliases must be separated by a memory dependency using the appropriate access types if at least one of those uses performs writes, whether the aliases interpret memory consistently or not. If buffer or image memory barriers are used, the scope of the barrier must contain the entire range and/or set of image subresources that overlap.

If two aliasing image views are used in the same framebuffer, then the render pass must declare the attachments using the `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT`, and follow the other rules listed in that section.

Note
Memory recycled via an application suballocator (i.e. without freeing and reallocating the memory objects) is not substantially different from memory aliasing. However, a suballocator usually waits on a fence before recycling a region of memory, and signaling a fence involves sufficient implicit dependencies to satisfy all the above requirements.

12.8.1. Resource Memory Overlap

Applications can safely access a resource concurrently as long as the memory locations do not overlap as defined in Memory Location. This includes aliased resources if such aliasing is well-defined. It also includes access from different queues and/or queue families if such concurrent access is supported by the resource. Transfer commands only access memory locations specified by the range of the transfer command.

Note

The intent is that buffers (or linear images) can be accessed concurrently, even when they share cache lines, but otherwise do not access the same memory range. The concept of a device cache line size is not exposed in the memory model.
Chapter 13. Samplers

VkSampler objects represent the state of an image sampler which is used by the implementation to read image data and apply filtering and other transformations for the shader.

Samplers are represented by VkSampler handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSampler)
```

To create a sampler object, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateSampler(
    VkDevice device,
    const VkSamplerCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkSampler* pSampler);
```

- `device` is the logical device that creates the sampler.
- `pCreateInfo` is a pointer to a VkSamplerCreateInfo structure specifying the state of the sampler object.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pSampler` is a pointer to a VkSampler handle in which the resulting sampler object is returned.

**Valid Usage**

- VUID-vkCreateSampler-maxSamplerAllocationCount-04110
  There must be less than VkPhysicalDeviceLimits::maxSamplerAllocationCount VkSampler objects currently created on the device

**Valid Usage (Implicit)**

- VUID-vkCreateSampler-device-parameter
device must be a valid VkDevice handle

- VUID-vkCreateSampler-pCreateInfo-parameter
  pCreateInfo must be a valid pointer to a valid VkSamplerCreateInfo structure

- VUID-vkCreateSampler-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkCreateSampler-pSampler-parameter
  pSampler must be a valid pointer to a VkSampler handle
Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkSamplerCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSamplerCreateInfo {
    VkStructureType    sType;
    const void*        pNext;
    VkSamplerCreateFlags flags;
    VkFilter           magFilter;
    VkFilter           minFilter;
    VkSamplerMipmapMode mipmapMode;
    VkSamplerAddressMode addressModeU;
    VkSamplerAddressMode addressModeV;
    VkSamplerAddressMode addressModeW;
    float              mipLodBias;
    VkBool32           anisotropyEnable;
    float              maxAnisotropy;
    VkBool32           compareEnable;
    VkCompareOp        compareOp;
    float              minLod;
    float              maxLod;
    VkBorderColor      borderColor;
    VkBool32           unnormalizedCoordinates;
} VkSamplerCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is a bitmask of `VkSamplerCreateFlagBits` describing additional parameters of the sampler.
- `magFilter` is a `VkFilter` value specifying the magnification filter to apply to lookups.
- `minFilter` is a `VkFilter` value specifying the minification filter to apply to lookups.
- `mipmapMode` is a `VkSamplerMipmapMode` value specifying the mipmap filter to apply to lookups.
- `addressModeU` is a `VkSamplerAddressMode` value specifying the addressing mode for U coordinates outside [0,1).
- `addressModeV` is a `VkSamplerAddressMode` value specifying the addressing mode for V coordinates outside [0,1).
- `addressModeW` is a `VkSamplerAddressMode` value specifying the addressing mode for W coordinates outside [0,1).
coordinates outside \([0,1)\).

- **mipLodBias** is the bias to be added to mipmap LOD (level-of-detail) calculation and bias provided by image sampling functions in SPIR-V, as described in the Level-of-Detail Operation section.

- **anisotropyEnable** is \(VK_TRUE\) to enable anisotropic filtering, as described in the Texel Anisotropic Filtering section, or \(VK_FALSE\) otherwise.

- **maxAnisotropy** is the anisotropy value clamp used by the sampler when **anisotropyEnable** is \(VK_TRUE\). If **anisotropyEnable** is \(VK_FALSE\), **maxAnisotropy** is ignored.

- **compareEnable** is \(VK_TRUE\) to enable comparison against a reference value during lookups, or \(VK_FALSE\) otherwise.
  - Note: Some implementations will default to shader state if this member does not match.

- **compareOp** is a \(VkCompareOp\) value specifying the comparison operator to apply to fetched data before filtering as described in the Depth Compare Operation section.

- **minLod** is used to clamp the minimum of the computed LOD value.

- **maxLod** is used to clamp the maximum of the computed LOD value. To avoid clamping the maximum value, set **maxLod** to the constant \(VK_LOD_CLAMP_NONE\).

- **borderColor** is a \(VkBorderColor\) value specifying the predefined border color to use.

- **unnormalizedCoordinates** controls whether to use unnormalized or normalized texel coordinates to address texels of the image. When set to \(VK_TRUE\), the range of the image coordinates used to lookup the texel is in the range of zero to the image size in each dimension. When set to \(VK_FALSE\) the range of image coordinates is zero to one.

When **unnormalizedCoordinates** is \(VK_TRUE\), images the sampler is used with in the shader have the following requirements:

- The **viewType** must be either \(VK_IMAGE_VIEW_TYPE_1D\) or \(VK_IMAGE_VIEW_TYPE_2D\).
- The image view **must** have a single layer and a single mip level.

When **unnormalizedCoordinates** is \(VK_TRUE\), image built-in functions in the shader that use the sampler have the following requirements:

- The functions **must** not use projection.
- The functions **must** not use offsets.

---

**Mapping of OpenGL to Vulkan filter modes**

**magFilter** values of **VK_FILTER_NEAREST** and **VK_FILTER_LINEAR** directly correspond to **GL_NEAREST** and **GL_LINEAR** magnification filters. **minFilter** and **.mipmapMode** combine to correspond to the similarly named OpenGL minification filter of **GL_minFilter** (e.g. **minFilter** of **VK_FILTER_LINEAR** and **mipmapMode** of **VK_SAMPLER_MIPMAP_MODE_NEAREST** correspond to **GL_LINEAR_MIPMAP_NEAREST**).

There are no Vulkan filter modes that directly correspond to OpenGL minification filters of **GL_LINEAR** or **GL_NEAREST**, but they can be emulated using **VK_SAMPLER_MIPMAP_MODE_NEAREST**, **minLod** = 0, and **maxLod** = 0.25, and using **minFilter** = **VK_FILTER_LINEAR** or **minFilter** = **VK_FILTER_NEAREST**, respectively.
Note that using a `maxLod` of zero would cause `magnification` to always be performed, and the `magFilter` to always be used. This is valid, just not an exact match for OpenGL behavior. Clamping the maximum LOD to 0.25 allows the $\lambda$ value to be non-zero and minification to be performed, while still always rounding down to the base level. If the `minFilter` and `magFilter` are equal, then using a `maxLod` of zero also works.

The maximum number of sampler objects which **can** be simultaneously created on a device is implementation-dependent and specified by the `maxSamplerAllocationCount` member of the `VkPhysicalDeviceLimits` structure.

---

**Note**

For historical reasons, if `maxSamplerAllocationCount` is exceeded, some implementations may return `VK_ERROR_TOO_MANY_OBJECTS`. Exceeding this limit will result in undefined behavior, and an application should not rely on the use of the returned error code in order to identify when the limit is reached.

Since `VkSampler` is a non-dispatchable handle type, implementations **may** return the same handle for sampler state vectors that are identical. In such cases, all such objects would only count once against the `maxSamplerAllocationCount` limit.

---

**Valid Usage**

- **VUID-VkSamplerCreateInfo-mipLodBias-01069**
  The absolute value of `mipLodBias` **must** be less than or equal to `VkPhysicalDeviceLimits::maxSamplerLodBias`

- **VUID-VkSamplerCreateInfo-maxLod-01973**
  `maxLod` **must** be greater than or equal to `minLod`

- **VUID-VkSamplerCreateInfo-anisotropyEnable-01070**
  If the `samplerAnisotropy` feature is not enabled, `anisotropyEnable` **must** be `VK_FALSE`

- **VUID-VkSamplerCreateInfo-anisotropyEnable-01071**
  If `anisotropyEnable` is `VK_TRUE`, `maxAnisotropy` **must** be between 1.0 and `VkPhysicalDeviceLimits::maxSamplerAnisotropy`, inclusive

- **VUID-VkSamplerCreateInfo-minFilter-01645**
  If `sampler Y'CbCr conversion` is enabled and the potential format features of the sampler `Y'CbCr` conversion do not support `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT`, `minFilter` and `magFilter` **must** be equal to the sampler `Y'CbCr` conversion's `chromaFilter`

- **VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01072**
  If `unnormalizedCoordinates` is `VK_TRUE`, `minFilter` and `magFilter` **must** be equal

- **VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01073**
  If `unnormalizedCoordinates` is `VK_TRUE`, `mipmapMode` **must** be `VK_SAMPLER_MIPMAP_MODE_NEAREST`

- **VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01074**
  If `unnormalizedCoordinates` is `VK_TRUE`, `minLod` and `maxLod` **must** be zero
If `unnormalizedCoordinates` is `VK_TRUE`, `addressModeU` and `addressModeV` must each be either `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE` or `VK_SAMPLER_ADDRESS_MODE.Clamp_TO_BORDER`.

If `unnormalizedCoordinates` is `VK_TRUE`, `anisotropyEnable` must be `VK_FALSE`.

If `unnormalizedCoordinates` is `VK_TRUE`, `compareEnable` must be `VK_FALSE`.

If any of `addressModeU`, `addressModeV`, or `addressModeW` are `VK_SAMPLER_ADDRESS_MODE.Clamp_TO_BORDER`, `borderColor` must be a valid `VkBorderColor` value.

If `sampler YCgCr conversion` is enabled, `addressModeU`, `addressModeV`, and `addressModeW` must be `VK_SAMPLER_ADDRESS_MODE.Clamp_TO_EDGE`, `anisotropyEnable` must be `VK_FALSE`, and `unnormalizedCoordinates` must be `VK_FALSE`.

If `sampler YCgCr conversion` is enabled and the `pNext` chain includes a `VkSamplerReductionModeCreateInfo` structure, then the sampler reduction mode must be set to `VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE`.

If `samplerFilterMinmax` is not enabled and the `pNext` chain includes a `VkSamplerReductionModeCreateInfo` structure, then the sampler reduction mode must be set to `VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE`.

If `samplerMirrorClampToEdge` is not enabled, and if the `VK_KHR_sampler_mirror_clamp_to_edge` extension is not enabled, `addressModeU`, `addressModeV`, and `addressModeW` must not be `VK_SAMPLER_ADDRESS_MODE.MIRROR_CLAMP_TO_EDGE`.

If `compareEnable` is `VK_TRUE`, `compareOp` must be a valid `VkCompareOp` value.

If `compareEnable` is `VK_TRUE`, the `reductionMode` member of `VkSamplerReductionModeCreateInfo` must be `VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE`.

**Valid Usage (Implicit)**

- `sType` must be `VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO`.
- Each `pNext` member of any structure (including this one) in the `pNext` chain must be `NULL` or a pointer to a valid instance of `VkSamplerReductionModeCreateInfo` or `VkSamplerYcbcrConversionInfo`.
- VUID-VkSamplerCreateInfo-sType-unique
  The sType value of each struct in the pNext chain must be unique
- VUID-VkSamplerCreateInfo-flags-zerobitmask
  flags must be 0
- VUID-VkSamplerCreateInfo-magFilter-parameter
  magFilter must be a valid VkFilter value
- VUID-VkSamplerCreateInfo-minFilter-parameter
  minFilter must be a valid VkFilter value
- VUID-VkSamplerCreateInfo-mipmapMode-parameter
  mipmapMode must be a valid VkSamplerMipmapMode value
- VUID-VkSamplerCreateInfo-addressModeU-parameter
  addressModeU must be a valid VkSamplerAddressMode value
- VUID-VkSamplerCreateInfo-addressModeV-parameter
  addressModeV must be a valid VkSamplerAddressMode value
- VUID-VkSamplerCreateInfo-addressModeW-parameter
  addressModeW must be a valid VkSamplerAddressMode value

VK_LOD_CLAMP_NONE is a special constant value used for VkSamplerCreateInfo::maxLod to indicate that maximum LOD clamping should not be performed.

```c
#define VK_LOD_CLAMP_NONE 1000.0F
```

Bits which can be set in VkSamplerCreateInfo::flags, specifying additional parameters of a sampler, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkSamplerCreateFlagBits {
} VkSamplerCreateFlagBits;
```

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkSamplerCreateFlags;
```

VkSamplerCreateFlags is a bitmask type for setting a mask of zero or more VkSamplerCreateFlagBits.

The VkSamplerReductionModeCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkSamplerReductionModeCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkSamplerReductionMode reductionMode;
} VkSamplerReductionModeCreateInfo;
```
VkSamplerReductionModeCreateInfo:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **reductionMode** is a `VkSamplerReductionMode` value controlling how texture filtering combines texel values.

If the **pNext** chain of `VkSamplerCreateInfo` includes a `VkSamplerReductionModeCreateInfo` structure, then that structure includes a mode controlling how texture filtering combines texel values.

If this structure is not present, **reductionMode** is considered to be **VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE**.

### Valid Usage (Implicit)

- VUID-VkSamplerReductionModeCreateInfo-sType-sType
  - **sType** must be **VK_STRUCTURE_TYPE_SAMPLER_REDUCTION_MODE_CREATE_INFO**
- VUID-VkSamplerReductionModeCreateInfo-reductionMode-parameter
  - **reductionMode** must be a valid `VkSamplerReductionMode` value

Reduction modes are specified by `VkSamplerReductionMode`, which takes values:

```c
// Provided by VK_VERSION_1_2
typedef enum VkSamplerReductionMode {
    VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE = 0,
    VK_SAMPLER_REDUCTION_MODE_MIN = 1,
    VK_SAMPLER_REDUCTION_MODE_MAX = 2,
} VkSamplerReductionMode;
```

- **VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE** specifies that texel values are combined by computing a weighted average of values in the footprint, using weights as specified in the image operations chapter.
- **VK_SAMPLER_REDUCTION_MODE_MIN** specifies that texel values are combined by taking the component-wise minimum of values in the footprint with non-zero weights.
- **VK_SAMPLER_REDUCTION_MODE_MAX** specifies that texel values are combined by taking the component-wise maximum of values in the footprint with non-zero weights.

Possible values of the `VkSamplerCreateInfo::magFilter` and `minFilter` parameters, specifying filters used for texture lookups, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkFilter {
    VK_FILTER_NEAREST = 0,
    VK_FILTER_LINEAR = 1,
};
```
VK_FILTER_NEAREST specifies nearest filtering.

VK_FILTER_LINEAR specifies linear filtering.

These filters are described in detail in Texel Filtering.

Possible values of the VkSamplerCreateInfo::mipmapMode, specifying the mipmap mode used for texture lookups, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkSamplerMipmapMode {
    VK_SAMPLER_MIPMAP_MODE_NEAREST = 0,
    VK_SAMPLER_MIPMAP_MODE_LINEAR = 1,
} VkSamplerMipmapMode;
```

- VK_SAMPLER_MIPMAP_MODE_NEAREST specifies nearest filtering.
- VK_SAMPLER_MIPMAP_MODE_LINEAR specifies linear filtering.

These modes are described in detail in Texel Filtering.

Possible values of the VkSamplerCreateInfo::addressMode* parameters, specifying the behavior of sampling with coordinates outside the range [0,1] for the respective u, v, or w coordinate as defined in the Wrapping Operation section, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkSamplerAddressMode {
    VK_SAMPLER_ADDRESS_MODE_REPEAT = 0,
    VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT = 1,
    VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE = 2,
    VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER = 3,
    // Provided by VK_VERSION_1_2
    VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE = 4,
} VkSamplerAddressMode;
```

- VK_SAMPLER_ADDRESS_MODE_REPEAT specifies that the repeat wrap mode will be used.
- VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT specifies that the mirrored repeat wrap mode will be used.
- VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE specifies that the clamp to edge wrap mode will be used.
- VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER specifies that the clamp to border wrap mode will be used.
- VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE specifies that the mirror clamp to edge wrap mode will be used. This is only valid if samplerMirror ClampToEdge is enabled, or if the VK_KHR_sampler_mirror_clamp_to_edge extension is enabled.
Comparison operators compare a reference and a test value, and return a true (“passed”) or false (“failed”) value depending on the comparison operator chosen. The supported operators are:

```
// Provided by VK_VERSION_1_0
typedef enum VkCompareOp {
    VK_COMPARE_OP_NEVER = 0,
    VK_COMPARE_OP_LESS = 1,
    VK_COMPARE_OP_EQUAL = 2,
    VK_COMPARE_OP_LESS_OR_EQUAL = 3,
    VK_COMPARE_OP_GREATER = 4,
    VK_COMPARE_OP_NOT_EQUAL = 5,
    VK_COMPARE_OP_GREATER_OR_EQUAL = 6,
    VK_COMPARE_OP_ALWAYS = 7,
} VkCompareOp;
```

- **VK_COMPARE_OP_NEVER** specifies that the comparison always evaluates false.
- **VK_COMPARE_OP_LESS** specifies that the comparison evaluates reference < test.
- **VK_COMPARE_OP_EQUAL** specifies that the comparison evaluates reference = test.
- **VK_COMPARE_OP_LESS_OR_EQUAL** specifies that the comparison evaluates reference ≤ test.
- **VK_COMPARE_OP_GREATER** specifies that the comparison evaluates reference > test.
- **VK_COMPARE_OP_NOT_EQUAL** specifies that the comparison evaluates reference ≠ test.
- **VK_COMPARE_OP_GREATER_OR_EQUAL** specifies that the comparison evaluates reference ≥ test.
- **VK_COMPARE_OP_ALWAYS** specifies that the comparison always evaluates true.

Comparison operators are used for:

- The **Depth Compare Operation** operator for a sampler, specified by `VkSamplerCreateInfo::compareOp`.
- The stencil comparison operator for the **stencil test**, specified by `vkCmdSetStencilOp::compareOp` or `VkStencilOpState::compareOp`.
- The **Depth Comparison** operator for the **depth test**, specified by `vkCmdSetDepthCompareOp::depthCompareOp` or `VkPipelineDepthStencilStateCreateInfo::depthCompareOp`.

Each such use describes how the reference and test values for that comparison are determined.

Possible values of `VkSamplerCreateInfo::borderColor`, specifying the border color used for texture lookups, are:

```
// Provided by VK_VERSION_1_0
typedef enum VkBorderColor {
    VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK = 0,
    VK_BORDER_COLOR_INT_TRANSPARENT_BLACK = 1,
    VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK = 2,
    VK_BORDER_COLOR_INT_OPAQUE_BLACK = 3,
    VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE = 4,
} VkBorderColor;
```
• **VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK** specifies a transparent, floating-point format, black color.

• **VK_BORDER_COLOR_INT_TRANSPARENT_BLACK** specifies a transparent, integer format, black color.

• **VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK** specifies an opaque, floating-point format, black color.

• **VK_BORDER_COLOR_INT_OPAQUE_BLACK** specifies an opaque, integer format, black color.

• **VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE** specifies an opaque, floating-point format, white color.

• **VK_BORDER_COLOR_INT_OPAQUE_WHITE** specifies an opaque, integer format, white color.

These colors are described in detail in *Texel Replacement*.

To destroy a sampler, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroySampler(
    VkDevice device,
    VkSampler sampler,
    const VkAllocationCallbacks* pAllocator);
```

• **device** is the logical device that destroys the sampler.

• **sampler** is the sampler to destroy.

• **pAllocator** controls host memory allocation as described in the *Memory Allocation* chapter.

### Valid Usage

• VUID-vkDestroySampler-sampler-01082
  All submitted commands that refer to **sampler** must have completed execution

• VUID-vkDestroySampler-sampler-01083
  If **VkAllocationCallbacks** were provided when **sampler** was created, a compatible set of callbacks must be provided here

• VUID-vkDestroySampler-sampler-01084
  If no **VkAllocationCallbacks** were provided when **sampler** was created, **pAllocator** must be **NULL**

### Valid Usage (Implicit)

• VUID-vkDestroySampler-device-parameter
  **device** must be a valid **VkDevice** handle

• VUID-vkDestroySampler-sampler-parameter
  If **sampler** is not **VK_NULL_HANDLE**, **sampler** must be a valid **VkSampler** handle
13.1. Sampler Y'CbCr conversion

To create a sampler with Y'CbCr conversion enabled, add a `VkSamplerYcbcrConversionInfo` structure to the `pNext` chain of the `VkSamplerCreateInfo` structure. To create a sampler Y'CbCr conversion, the `samplerYcbcrConversion` feature must be enabled. Conversion must be fixed at pipeline creation time, through use of a combined image sampler with an immutable sampler in `VkDescriptorSetLayoutBinding`.

A `VkSamplerYcbcrConversionInfo` must be provided for samplers to be used with image views that access `VK_IMAGE_ASPECT_COLOR_BIT` if the format is one of the formats that require a sampler Y'CbCr conversion.

The `VkSamplerYcbcrConversionInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkSamplerYcbcrConversionInfo {
    VkStructureType sType;
    const void* pNext;
    VkSamplerYcbcrConversion conversion;
} VkSamplerYcbcrConversionInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `conversion` is a `VkSamplerYcbcrConversion` handle created with `vkCreateSamplerYcbcrConversion`.

Valid Usage (Implicit)

- VUID-VkSamplerYcbcrConversionInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_INFO`
- VUID-VkSamplerYcbcrConversionInfo-conversion-parameter
  `conversion` must be a valid `VkSamplerYcbcrConversion` handle
A sampler Y’CbCr conversion is an opaque representation of a device-specific sampler Y’CbCr conversion description, represented as a `VkSamplerYcbcrConversion` handle:

```c
// Provided by VK_VERSION_1_1
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSamplerYcbcrConversion)
```

To create a `VkSamplerYcbcrConversion`, call:

```c
// Provided by VK_VERSION_1_1
VkResult vkCreateSamplerYcbcrConversion(
    VkDevice device,
    const VkSamplerYcbcrConversionCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkSamplerYcbcrConversion* pYcbcrConversion);
```

- `device` is the logical device that creates the sampler Y’CbCr conversion.
- `pCreateInfo` is a pointer to a `VkSamplerYcbcrConversionCreateInfo` structure specifying the requested sampler Y’CbCr conversion.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pYcbcrConversion` is a pointer to a `VkSamplerYcbcrConversion` handle in which the resulting sampler Y’CbCr conversion is returned.

The interpretation of the configured sampler Y’CbCr conversion is described in more detail in the description of sampler Y’CbCr conversion in the Image Operations chapter.

### Valid Usage

- VUID-vkCreateSamplerYcbcrConversion-None-01648
  The `samplerYcbcrConversion` feature must be enabled

### Valid Usage (Implicit)

- VUID-vkCreateSamplerYcbcrConversion-device-parameter
device must be a valid `VkDevice` handle
- VUID-vkCreateSamplerYcbcrConversion-pCreateInfo-parameter
  `pCreateInfo` must be a valid pointer to a valid `VkSamplerYcbcrConversionCreateInfo` structure
- VUID-vkCreateSamplerYcbcrConversion-pAllocator-parameter
  If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- VUID-vkCreateSamplerYcbcrConversion-pYcbcrConversion-parameter
  `pYcbcrConversion` must be a valid pointer to a `VkSamplerYcbcrConversion` handle
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkSamplerYcbcrConversionCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkSamplerYcbcrConversionCreateInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkFormat                   format;
    VkSamplerYcbcrModelConversion ycbcrModel;
    VkSamplerYcbcrRange        ycbcrRange;
    VkComponentMapping         components;
    VkChromaLocation           xChromaOffset;
    VkChromaLocation           yChromaOffset;
    VkFilter                   chromaFilter;
    VkBool32                   forceExplicitReconstruction;
} VkSamplerYcbcrConversionCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `format` is the format of the image from which color information will be retrieved.
- `ycbcrModel` describes the color matrix for conversion between color models.
- `ycbcrRange` describes whether the encoded values have headroom and foot room, or whether the encoding uses the full numerical range.
- `components` applies a swizzle based on `VkComponentSwizzle` enums prior to range expansion and color model conversion.
- `xChromaOffset` describes the sample location associated with downsampled chroma components in the x dimension. `xChromaOffset` has no effect for formats in which chroma components are not downsampled horizontally.
- `yChromaOffset` describes the sample location associated with downsampled chroma components in the y dimension. `yChromaOffset` has no effect for formats in which the chroma components are not downsampled vertically.
- `chromaFilter` is the filter for chroma reconstruction.
- `forceExplicitReconstruction` can be used to ensure that reconstruction is done explicitly, if supported.
**Note**

Setting `forceExplicitReconstruction` to `VK_TRUE` **may** have a performance penalty on implementations where explicit reconstruction is not the default mode of operation.

If `format` supports `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT` the `forceExplicitReconstruction` value behaves as if it was set to `VK_TRUE`.

Sampler Y’C_bC_r conversion objects do not support *external format conversion* without additional extensions defining *external formats*.

---

**Valid Usage**

- **VUID-VkSamplerYcbcrConversionCreateInfo-format-04060**
  
  The format **must** represent unsigned normalized values (i.e. the format must be a `UNORM` format).

- **VUID-VkSamplerYcbcrConversionCreateInfo-format-01650**

  The potential format features of the sampler Y’C_bC_r conversion **must** support `VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT` or `VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT`.

- **VUID-VkSamplerYcbcrConversionCreateInfo-xChromaOffset-01651**

  If the potential format features of the sampler Y’C_bC_r conversion do not support `VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT`, `xChromaOffset` and `yChromaOffset` **must** not be `VK_CHROMA_LOCATION_COSITED_EVEN` if the corresponding components are downscaled.

- **VUID-VkSamplerYcbcrConversionCreateInfo-xChromaOffset-01652**

  If the potential format features of the sampler Y’C_bC_r conversion do not support `VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT`, `xChromaOffset` and `yChromaOffset` **must** not be `VK_CHROMA_LOCATION_MIDPOINT` if the corresponding components are downscaled.

- **VUID-VkSamplerYcbcrConversionCreateInfo-components-02581**

  If the format has a `_422` or `_420` suffix, then `components.g` **must** be the identity swizzle.

- **VUID-VkSamplerYcbcrConversionCreateInfo-components-02582**

  If the format has a `_422` or `_420` suffix, then `components.a` **must** be the identity swizzle, `VK_COMPONENT_SWIZZLE_ONE`, or `VK_COMPONENT_SWIZZLE_ZERO`.

- **VUID-VkSamplerYcbcrConversionCreateInfo-components-02583**

  If the format has a `_422` or `_420` suffix, then `components.r` **must** be the identity swizzle or `VK_COMPONENT_SWIZZLE_B`.

- **VUID-VkSamplerYcbcrConversionCreateInfo-components-02584**

  If the format has a `_422` or `_420` suffix, then `components.b` **must** be the identity swizzle or `VK_COMPONENT_SWIZZLE_R`.

- **VUID-VkSamplerYcbcrConversionCreateInfo-components-02585**

  If the format has a `_422` or `_420` suffix, and if either `components.r` or `components.b` is the identity swizzle, both values **must** be the identity swizzle.

- **VUID-VkSamplerYcbcrConversionCreateInfo-ycbcrModel-01655**
If $ycbcrModel$ is not $VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY$, then $components.r$, $components.g$, and $components.b$ must correspond to components of the format; that is, $components.r$, $components.g$, and $components.b$ must not be $VK_COMPONENT_SWIZZLE_ZERO$ or $VK_COMPONENT_SWIZZLE_ONE$, and must not correspond to a component containing zero or one as a consequence of conversion to RGBA.

- **VUID-VkSamplerYcbcrConversionCreateInfo-ycbcrRange-02748**
  If $ycbcrRange$ is $VK_SAMPLER_YCBCR_RANGE_ITU_NARROW$ then the R, G and B components obtained by applying the component swizzle to format must each have a bit-depth greater than or equal to 8.

- **VUID-VkSamplerYcbcrConversionCreateInfo-forceExplicitReconstruction-01656**
  If the potential format features of the sampler $Y’C_B’C_R$ conversion do not support $VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT$ forceExplicitReconstruction must be $VK_FALSE$.

- **VUID-VkSamplerYcbcrConversionCreateInfo-chromaFilter-01657**
  If the potential format features of the sampler $Y’C_B’C_R$ conversion do not support $VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT$, chromaFilter must not be $VK_FILTER_LINEAR$.

### Valid Usage (Implicit)

- **VUID-VkSamplerYcbcrConversionCreateInfo-sType-sType**
  sType must be $VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_CREATE_INFO$.

- **VUID-VkSamplerYcbcrConversionCreateInfo-pNext-pNext**
  pNext must be NULL.

- **VUID-VkSamplerYcbcrConversionCreateInfo-format-parameter**
  format must be a valid $VkFormat$ value.

- **VUID-VkSamplerYcbcrConversionCreateInfo-ycbcrModel-parameter**
  ycbcrModel must be a valid $VkSamplerYcbcrModelConversion$ value.

- **VUID-VkSamplerYcbcrConversionCreateInfo-ycbcrRange-parameter**
  ycbcrRange must be a valid $VkSamplerYcbcrRange$ value.

- **VUID-VkSamplerYcbcrConversionCreateInfo-components-parameter**
  components must be a valid $VkComponentMapping$ structure.

- **VUID-VkSamplerYcbcrConversionCreateInfo-xChromaOffset-parameter**
  xChromaOffset must be a valid $VkChromaLocation$ value.

- **VUID-VkSamplerYcbcrConversionCreateInfo-yChromaOffset-parameter**
  yChromaOffset must be a valid $VkChromaLocation$ value.

- **VUID-VkSamplerYcbcrConversionCreateInfo-chromaFilter-parameter**
  chromaFilter must be a valid $VkFilter$ value.

If chromaFilter is $VK_FILTER_NEAREST$, chroma samples are reconstructed to luma component resolution using nearest-neighbour sampling. Otherwise, chroma samples are reconstructed using interpolation. More details can be found in the description of sampler $Y’C_B’C_R$ conversion in the
Image Operations chapter.

VkSamplerYcbcrModelConversion defines the conversion from the source color model to the shader color model. Possible values are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkSamplerYcbcrModelConversion {
    VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY = 0,
    VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_IDENTITY = 1,
    VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_709 = 2,
    VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_601 = 3,
    VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_2020 = 4,
} VkSamplerYcbcrModelConversion;
```

- **VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY** specifies that the input values to the conversion are unmodified.
- **VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_IDENTITY** specifies no model conversion but the inputs are range expanded as for Y′C_bC_r.
- **VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_709** specifies the color model conversion from Y′C_bC_r to R′G′B′ defined in BT.709 and described in the “BT.709 Y′C_bC_r conversion” section of the Khronos Data Format Specification.
- **VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_601** specifies the color model conversion from Y′C_bC_r to R′G′B′ defined in BT.601 and described in the “BT.601 Y′C_bC_r conversion” section of the Khronos Data Format Specification.
- **VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_2020** specifies the color model conversion from Y′C_bC_r to R′G′B′ defined in BT.2020 and described in the “BT.2020 Y′C_bC_r conversion” section of the Khronos Data Format Specification.

In the **VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_** color models, for the input to the sampler Y′C_bC_r range expansion and model conversion:

- the Y (Y′ luma) component corresponds to the G component of an RGB image.
- the CB (C_b or “U” blue color difference) component corresponds to the B component of an RGB image.
- the CR (C_r or “V” red color difference) component corresponds to the R component of an RGB image.
- the alpha component, if present, is not modified by color model conversion.

These rules reflect the mapping of components after the component swizzle operation (controlled by VkSamplerYcbcrConversionCreateInfo::components).

**Note**

For example, an “YUVA” 32-bit format comprising four 8-bit components can be implemented as **VK_FORMAT_R8G8B8A8_UNORM** with a component mapping:
The `VkSamplerYcbcrRange` enum describes whether color components are encoded using the full range of numerical values or whether values are reserved for headroom and foot room. `VkSamplerYcbcrRange` is defined as:

```c
// Provided by VK_VERSION_1_1
typedef enum VkSamplerYcbcrRange {
    VK_SAMPLER_YCBCR_RANGE_ITU_FULL = 0,
    VK_SAMPLER_YCBCR_RANGE_ITU_NARROW = 1,
} VkSamplerYcbcrRange;
```

- `VK_SAMPLER_YCBCR_RANGE_ITU_FULL` specifies that the full range of the encoded values are valid and interpreted according to the ITU “full range” quantization rules.
- `VK_SAMPLER_YCBCR_RANGE_ITU_NARROW` specifies that headroom and foot room are reserved in the numerical range of encoded values, and the remaining values are expanded according to the ITU “narrow range” quantization rules.

The formulae for these conversions is described in the Sampler Y’CbCr Range Expansion section of the Image Operations chapter.

No range modification takes place if `ycbcrModel` is `VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY`; the `ycbcrRange` field of `VkSamplerYcbcrConversionCreateInfo` is ignored in this case.

The `VkChromaLocation` enum defines the location of downsampled chroma component samples relative to the luma samples, and is defined as:

```c
// Provided by VK_VERSION_1_1
typedef enum VkChromaLocation {
    VK_CHROMA_LOCATION_COSITED_EVEN = 0,
    VK_CHROMA_LOCATION_MIDPOINT = 1,
} VkChromaLocation;
```

- `VK_CHROMA_LOCATION_COSITED_EVEN` specifies that downsampled chroma samples are aligned with luma samples with even coordinates.
- `VK_CHROMA_LOCATION_MIDPOINT` specifies that downsampled chroma samples are located half way between each even luma sample and the nearest higher odd luma sample.

To destroy a sampler Y’C_bC_r conversion, call:

```c
// Provided by VK_VERSION_1_1
typedef enum VkChromaLocation {
    VK_CHROMA_LOCATION_COSITED_EVEN = 0,
    VK_CHROMA_LOCATION_MIDPOINT = 1,
} VkChromaLocation;
```
VkDevice device,
VkSamplerYcbcrConversion ycbcrConversion,
const VkAllocationCallbacks* pAllocator);

- **device** is the logical device that destroys the Y’C₉C₉R conversion.
- **ycbcrConversion** is the conversion to destroy.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage (Implicit)

- VUID-vkDestroySamplerYcbcrConversion-device-parameter
  
  **device** must be a valid *VkDevice* handle

- VUID-vkDestroySamplerYcbcrConversion-ycbcrConversion-parameter
  
  If **ycbcrConversion** is not **VK_NULL_HANDLE**, **ycbcrConversion** must be a valid *VkSamplerYcbcrConversion* handle

- VUID-vkDestroySamplerYcbcrConversion-pAllocator-parameter
  
  If **pAllocator** is not NULL, **pAllocator** must be a valid pointer to a valid *VkAllocationCallbacks* structure

- VUID-vkDestroySamplerYcbcrConversion-ycbcrConversion-parent
  
  If **ycbcrConversion** is a valid handle, it must have been created, allocated, or retrieved from **device**

### Host Synchronization

- Host access to **ycbcrConversion** must be externally synchronized
Chapter 14. Resource Descriptors

A descriptor is an opaque data structure representing a shader resource such as a buffer, buffer view, image view, sampler, or combined image sampler. Descriptors are organized into descriptor sets, which are bound during command recording for use in subsequent drawing commands. The arrangement of content in each descriptor set is determined by a descriptor set layout, which determines what descriptors can be stored within it. The sequence of descriptor set layouts that can be used by a pipeline is specified in a pipeline layout. Each pipeline object can use up to maxBoundDescriptorSets (see Limits) descriptor sets.

Shaders access resources via variables decorated with a descriptor set and binding number that link them to a descriptor in a descriptor set. The shader interface mapping to bound descriptor sets is described in the Shader Resource Interface section.

Shaders can also access buffers without going through descriptors by using Physical Storage Buffer Access to access them through 64-bit addresses.

14.1. Descriptor Types

There are a number of different types of descriptor supported by Vulkan, corresponding to different resources or usage. The following sections describe the API definitions of each descriptor type. The mapping of each type to SPIR-V is listed in the Shader Resource and Descriptor Type Correspondence and Shader Resource and Storage Class Correspondence tables in the Shader Interfaces chapter.

14.1.1. Storage Image

A storage image (VK_DESCRIPTOR_TYPE_STORAGE_IMAGE) is a descriptor type associated with an image resource via an image view that load, store, and atomic operations can be performed on.

Storage image loads are supported in all shader stages for image views whose format features contain VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT.

Stores to storage images are supported in compute shaders for image views whose format features contain VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT.

Atomic operations on storage images are supported in compute shaders for image views whose format features contain VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT.

When the fragmentStoresAndAtomics feature is enabled, stores and atomic operations are also supported for storage images in fragment shaders with the same set of image formats as supported in compute shaders. When the vertexPipelineStoresAndAtomics feature is enabled, stores and atomic operations are also supported in vertex, tessellation, and geometry shaders with the same set of image formats as supported in compute shaders.

The image subresources for a storage image must be in the VK_IMAGE_LAYOUT_GENERAL layout in order to access its data in a shader.
14.1.2. Sampler

A sampler descriptor (VK_DESCRIPTOR_TYPE_SAMPLER) is a descriptor type associated with a sampler object, used to control the behavior of sampling operations performed on a sampled image.

14.1.3. Sampled Image

A sampled image (VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE) is a descriptor type associated with an image resource via an image view that sampling operations can be performed on.

Shaders combine a sampled image variable and a sampler variable to perform sampling operations.

Sampled images are supported in all shader stages for image views whose format features contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT.

An image subresources for a sampled image must be in one of the following layouts:

- VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL
- VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
- VK_IMAGE_LAYOUT_GENERAL
- VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL
- VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_ATTACHMENT_OPTIMAL
- VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL
- VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL
- VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL

14.1.4. Combined Image Sampler

A combined image sampler (VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER) is a single descriptor type associated with both a sampler and an image resource, combining both a sampler and sampled image descriptor into a single descriptor.

If the descriptor refers to a sampler that performs YCbCr conversion, the sampler must only be used to sample the image in the same descriptor. Otherwise, the sampler and image in this type of descriptor can be used freely with any other samplers and images.

An image subresources for a combined image sampler must be in one of the following layouts:

- VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL
- VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
- VK_IMAGE_LAYOUT_GENERAL
- VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_ATTACHMENT_OPTIMAL
- VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL
- VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL
- VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL
- VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL
Note
On some implementations, it may be more efficient to sample from an image using a combination of sampler and sampled image that are stored together in the descriptor set in a combined descriptor.

14.1.5. Uniform Texel Buffer

A uniform texel buffer (VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER) is a descriptor type associated with a buffer resource via a buffer view that image sampling operations can be performed on.

Uniform texel buffers define a tightly-packed 1-dimensional linear array of texels, with texels going through format conversion when read in a shader in the same way as they are for an image.

Load operations from uniform texel buffers are supported in all shader stages for image formats which report support for the VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT feature bit via vkGetPhysicalDeviceFormatProperties in VkFormatProperties::bufferFeatures.

14.1.6. Storage Texel Buffer

A storage texel buffer (VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER) is a descriptor type associated with a buffer resource via a buffer view that image load, store, and atomic operations can be performed on.

Storage texel buffers define a tightly-packed 1-dimensional linear array of texels, with texels going through format conversion when read in a shader in the same way as they are for an image. Unlike uniform texel buffers, these buffers can also be written to in the same way as for storage images.

Storage texel buffer loads are supported in all shader stages for texel buffer formats which report support for the VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT feature bit via vkGetPhysicalDeviceFormatProperties in VkFormatProperties::bufferFeatures.

Stores to storage texel buffers are supported in compute shaders for texel buffer formats which report support for the VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT feature via vkGetPhysicalDeviceFormatProperties in VkFormatProperties::bufferFeatures.

Atomic operations on storage texel buffers are supported in compute shaders for texel buffer formats which report support for the VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT feature via vkGetPhysicalDeviceFormatProperties in VkFormatProperties::bufferFeatures.

When the fragmentStoresAndAtomics feature is enabled, stores and atomic operations are also supported for storage texel buffers in fragment shaders with the same set of texel buffer formats as supported in compute shaders. When the vertexPipelineStoresAndAtomics feature is enabled, stores and atomic operations are also supported in vertex, tessellation, and geometry shaders with the same set of texel buffer formats as supported in compute shaders.

14.1.7. Storage Buffer

A storage buffer (VK_DESCRIPTOR_TYPE_STORAGE_BUFFER) is a descriptor type associated with a buffer resource directly, described in a shader as a structure with various members that load, store, and
atomic operations can be performed on.

Note
Atomic operations can only be performed on members of certain types as defined in the SPIR-V environment appendix.

14.1.8. Uniform Buffer

A uniform buffer (VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER) is a descriptor type associated with a buffer resource directly, described in a shader as a structure with various members that load operations can be performed on.

14.1.9. Dynamic Uniform Buffer

A dynamic uniform buffer (VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC) is almost identical to a uniform buffer, and differs only in how the offset into the buffer is specified. The base offset calculated by the VkDescriptorBufferInfo when initially updating the descriptor set is added to a dynamic offset when binding the descriptor set.

14.1.10. Dynamic Storage Buffer

A dynamic storage buffer (VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC) is almost identical to a storage buffer, and differs only in how the offset into the buffer is specified. The base offset calculated by the VkDescriptorBufferInfo when initially updating the descriptor set is added to a dynamic offset when binding the descriptor set.

14.1.11. Inline Uniform Block

An inline uniform block (VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK) is almost identical to a uniform buffer, and differs only in taking its storage directly from the encompassing descriptor set instead of being backed by buffer memory. It is typically used to access a small set of constant data that does not require the additional flexibility provided by the indirection enabled when using a uniform buffer where the descriptor and the referenced buffer memory are decoupled. Compared to push constants, they allow reusing the same set of constant data across multiple disjoint sets of drawing and dispatching commands.

Inline uniform block descriptors cannot be aggregated into arrays. Instead, the array size specified for an inline uniform block descriptor binding specifies the binding’s capacity in bytes.

14.1.12. Input Attachment

An input attachment (VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT) is a descriptor type associated with an image resource via an image view that can be used for framebuffer local load operations in fragment shaders.

All image formats that are supported for color attachments (VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT) or depth/stencil attachments (VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT) for a given image tiling mode are also supported.
for input attachments.

An image view used as an input attachment must be in one of the following layouts:

- VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL
- VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
- VK_IMAGE_LAYOUT_GENERAL
- VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL
- VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL

14.2. Descriptor Sets

Descriptors are grouped together into descriptor set objects. A descriptor set object is an opaque object containing storage for a set of descriptors, where the types and number of descriptors is defined by a descriptor set layout. The layout object may be used to define the association of each descriptor binding with memory or other implementation resources. The layout is used both for determining the resources that need to be associated with the descriptor set, and determining the interface between shader stages and shader resources.

14.2.1. Descriptor Set Layout

A descriptor set layout object is defined by an array of zero or more descriptor bindings. Each individual descriptor binding is specified by a descriptor type, a count (array size) of the number of descriptors in the binding, a set of shader stages that can access the binding, and (if using immutable samplers) an array of sampler descriptors.

Descriptor set layout objects are represented by VkDescriptorSetLayout handles:

```
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorSetLayout)
```

To create descriptor set layout objects, call:

```
// Provided by VK_VERSION_1_0
VkResult vkCreateDescriptorSetLayout(
    VkDevice device,               // device,
    const VkDescriptorSetLayoutCreateInfo* pCreateInfo,   // pCreateInfo,
    const VkAllocationCallbacks* pAllocator,                // pAllocator,
    VkDescriptorSetLayout* pSetLayout);                     // pSetLayout;
```

- **device** is the logical device that creates the descriptor set layout.
- **pCreateInfo** is a pointer to a VkDescriptorSetLayoutCreateInfo structure specifying the state of the descriptor set layout object.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
• `pSetLayout` is a pointer to a `VkDescriptorSetLayout` handle in which the resulting descriptor set layout object is returned.

### Valid Usage (Implicit)

- **VUID-vkCreateDescriptorSetLayout-device-parameter**
  - `device` must be a valid `VkDevice` handle
- **VUID-vkCreateDescriptorSetLayout-pCreateInfo-parameter**
  - `pCreateInfo` must be a valid pointer to a valid `VkDescriptorSetLayoutCreateInfo` structure
- **VUID-vkCreateDescriptorSetLayout-pAllocator-parameter**
  - If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- **VUID-vkCreateDescriptorSetLayout-pSetLayout-parameter**
  - `pSetLayout` must be a valid pointer to a `VkDescriptorSetLayout` handle

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

Information about the descriptor set layout is passed in a `VkDescriptorSetLayoutCreateInfo` structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDescriptorSetLayoutCreateInfo {
  VkStructureType sType;
  const void* pNext;
  VkDescriptorSetLayoutCreateFlags flags;
  uint32_t bindingCount;
  const VkDescriptorSetLayoutBinding* pBindings;
} VkDescriptorSetLayoutCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is a bitmask specifying options for descriptor set layout creation.
- `bindingCount` is the number of elements in `pBindings`.
- `pBindings` is a pointer to an array of `VkDescriptorSetLayoutBinding` structures.
Valid Usage

- VUID-VkDescriptorSetLayoutCreateInfo-binding-00279
  The VkDescriptorSetLayoutBinding::binding members of the elements of the pBindings array must each have different values

- VUID-VkDescriptorSetLayoutCreateInfo-flags-03000
  If any binding has the VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT bit set, flags must include VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT

- VUID-VkDescriptorSetLayoutCreateInfo-descriptorType-03001
  If any binding has the VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT bit set, then all bindings must not have descriptorType of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC

Valid Usage (Implicit)

- VUID-VkDescriptorSetLayoutCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO

- VUID-VkDescriptorSetLayoutCreateInfo-pNext-pNext
  pNext must be NULL or a pointer to a valid instance of VkDescriptorSetLayoutBindingFlagsCreateInfo

- VUID-VkDescriptorSetLayoutCreateInfo-sType-unique
  The sType value of each struct in the pNext chain must be unique

- VUID-VkDescriptorSetLayoutCreateInfo-flags-parameter
  flags must be a valid combination of VkDescriptorSetLayoutCreateFlagBits values

- VUID-VkDescriptorSetLayoutCreateInfo-pBindings-parameter
  If bindingCount is not 0, pBindings must be a valid pointer to an array of bindingCount valid VkDescriptorSetLayoutBinding structures

Bits which can be set in VkDescriptorSetLayoutCreateInfo::flags, specifying options for descriptor set layout, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkDescriptorSetLayoutCreateFlagBits {
    // Provided by VK_VERSION_1_2
    VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT = 0x00000002,
} VkDescriptorSetLayoutCreateFlagBits;
```

- VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT specifies that descriptor sets using this layout must be allocated from a descriptor pool created with the VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT bit set. Descriptor set layouts created with this bit set have alternate limits for the maximum number of descriptors per-stage and per-pipeline layout. The non-UpdateAfterBind limits only count descriptors in sets created without this flag. The UpdateAfterBind limits count all descriptors, but the limits may be higher than the non-
UpdateAfterBind limits.

**Note**

All bits for this type are defined by extensions, and none of those extensions are enabled in this build of the specification.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkDescriptorSetLayoutCreateFlags;
```

`VkDescriptorSetLayoutCreateFlags` is a bitmask type for setting a mask of zero or more `VkDescriptorSetLayoutCreateFlagBits`.

The `VkDescriptorSetLayoutBinding` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDescriptorSetLayoutBinding {
    uint32_t binding;
    VkDescriptorType descriptorType;
    uint32_t descriptorCount;
    VkShaderStageFlags stageFlags;
    const VkSampler* pImmutableSamplers;
} VkDescriptorSetLayoutBinding;
```

- **binding** is the binding number of this entry and corresponds to a resource of the same binding number in the shader stages.
- **descriptorType** is a `VkDescriptorType` specifying which type of resource descriptors are used for this binding.
- **descriptorCount** is the number of descriptors contained in the binding, accessed in a shader as an array, except if `descriptorType` is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK` in which case `descriptorCount` is the size in bytes of the inline uniform block. If `descriptorCount` is zero this binding entry is reserved and the resource must not be accessed from any stage via this binding within any pipeline using the set layout.
- **stageFlags** member is a bitmask of `VkShaderStageFlagBits` specifying which pipeline shader stages can access a resource for this binding. `VK_SHADER_STAGE_ALL` is a shorthand specifying that all defined shader stages, including any additional stages defined by extensions, can access the resource.

If a shader stage is not included in `stageFlags`, then a resource must not be accessed from that stage via this binding within any pipeline using the set layout. Other than input attachments which are limited to the fragment shader, there are no limitations on what combinations of stages can use a descriptor binding, and in particular a binding can be used by both graphics stages and the compute stage.

- **pImmutableSamplers** affects initialization of samplers. If `descriptorType` specifies a `VK_DESCRIPTOR_TYPE_SAMPLER` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` type descriptor, then
pImmutableSamplers can be used to initialize a set of immutable samplers. Immutable samplers are permanently bound into the set layout and must not be changed; updating a VK_DESCRIPTOR_TYPE_SAMPLER descriptor with immutable samplers is not allowed and updates to a VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER descriptor with immutable samplers does not modify the samplers (the image views are updated, but the sampler updates are ignored). If pImmutableSamplers is not NULL, then it is a pointer to an array of sampler handles that will be copied into the set layout and used for the corresponding binding. Only the sampler handles are copied; the sampler objects must not be destroyed before the final use of the set layout and any descriptor pools and sets created using it. If pImmutableSamplers is NULL, then the sampler slots are dynamic and sampler handles must be bound into descriptor sets using this layout. If descriptorType is not one of these descriptor types, then pImmutableSamplers is ignored.

The above layout definition allows the descriptor bindings to be specified sparsely such that not all binding numbers between 0 and the maximum binding number need to be specified in the pBindings array. Bindings that are not specified have a descriptorCount and stageFlags of zero, and the value of descriptorType is undefined. However, all binding numbers between 0 and the maximum binding number in the VkDescriptorSetLayoutCreateInfo::pBindings array may consume memory in the descriptor set layout even if not all descriptor bindings are used, though it should not consume additional memory from the descriptor pool.

Note
The maximum binding number specified should be as compact as possible to avoid wasted memory.

Valid Usage

- VUID-VkDescriptorSetLayoutBinding-descriptorType-00282
  If descriptorType is VK_DESCRIPTOR_TYPE_SAMPLER or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, and descriptorCount is not 0 and pImmutableSamplers is not NULL, pImmutableSamplers must be a valid pointer to an array of descriptorCount valid VkSampler handles

- VUID-VkDescriptorSetLayoutBinding-descriptorType-04604
  If the inlineUniformBlock feature is not enabled, descriptorType must not be VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK

- VUID-VkDescriptorSetLayoutBinding-descriptorType-02209
  If descriptorType is VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK then descriptorCount must be a multiple of 4

- VUID-VkDescriptorSetLayoutBinding-descriptorType-02210
  If descriptorType is VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK then descriptorCount must be less than or equal to VkPhysicalDeviceInlineUniformBlockSize

- VUID-VkDescriptorSetLayoutBinding-descriptorCount-00283
  If descriptorCount is not 0, stageFlags must be a valid combination of VkShaderStageFlagBits values

- VUID-VkDescriptorSetLayoutBinding-descriptorType-01510
If `descriptorType` is `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` and `descriptorCount` is not 0, then `stageFlags` must be 0 or `VK_SHADER_STAGE_FRAGMENT_BIT`.

**Valid Usage (Implicit)**

- **VUID-VkDescriptorSetLayoutBinding-descriptorType-parameter**
  `descriptorType` must be a valid `VkDescriptorType` value.

If the `pNext` chain of a `VkDescriptorSetLayoutCreateInfo` structure includes a `VkDescriptorSetLayoutBindingFlagsCreateInfo` structure, then that structure includes an array of flags, one for each descriptor set layout binding.

The `VkDescriptorSetLayoutBindingFlagsCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkDescriptorSetLayoutBindingFlagsCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t bindingCount;
    const VkDescriptorBindingFlags* pBindingFlags;
} VkDescriptorSetLayoutBindingFlagsCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `bindingCount` is zero or the number of elements in `pBindingFlags`.
- `pBindingFlags` is a pointer to an array of `VkDescriptorBindingFlags` bitfields, one for each descriptor set layout binding.

If `bindingCount` is zero or if this structure is not included in the `pNext` chain, the `VkDescriptorSetLayoutBindingFlags` for each descriptor set layout binding is considered to be zero. Otherwise, the descriptor set layout binding at `VkDescriptorSetLayoutCreateInfo::pBindings[i]` uses the flags in `pBindingFlags[i]`.

**Valid Usage**

- **VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-bindingCount-03002**
  If `bindingCount` is not zero, `bindingCount` must equal `VkDescriptorSetLayoutCreateInfo::bindingCount`.

- **VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-pBindingFlags-03004**
  If an element of `pBindingFlags` includes `VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT`, then all other elements of `VkDescriptorSetLayoutCreateInfo::pBindings` must have a smaller value of `binding`.

- **VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingUniformBufferUpdateAfterBind-03005**
If \( \text{VkPhysicalDeviceDescriptorIndexingFeatures} :: \text{descriptorBindingUniformBufferUpdateAfterBind} \) is not enabled, all bindings with descriptor type \( \text{VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER} \) must not use \( \text{VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT} \)

- \( \text{VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingSampledImageUpdateAfterBind-03006} \)
  If \( \text{VkPhysicalDeviceDescriptorIndexingFeatures} :: \text{descriptorBindingSampledImageUpdateAfterBind} \) is not enabled, all bindings with descriptor type \( \text{VK_DESCRIPTOR_TYPE_SAMPLER}, \text{VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER}, \) or \( \text{VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE} \) must not use \( \text{VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT} \)

- \( \text{VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingStorageImageUpdateAfterBind-03007} \)
  If \( \text{VkPhysicalDeviceDescriptorIndexingFeatures} :: \text{descriptorBindingStorageImageUpdateAfterBind} \) is not enabled, all bindings with descriptor type \( \text{VK_DESCRIPTOR_TYPE_STORAGE_IMAGE} \) must not use \( \text{VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT} \)

- \( \text{VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingStorageBufferUpdateAfterBind-03008} \)
  If \( \text{VkPhysicalDeviceDescriptorIndexingFeatures} :: \text{descriptorBindingStorageBufferUpdateAfterBind} \) is not enabled, all bindings with descriptor type \( \text{VK_DESCRIPTOR_TYPE_STORAGE_BUFFER} \) must not use \( \text{VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT} \)

- \( \text{VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingUniformTexelBufferUpdateAfterBind-03009} \)
  If \( \text{VkPhysicalDeviceDescriptorIndexingFeatures} :: \text{descriptorBindingUniformTexelBufferUpdateAfterBind} \) is not enabled, all bindings with descriptor type \( \text{VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER} \) must not use \( \text{VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT} \)

- \( \text{VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingStorageTexelBufferUpdateAfterBind-03010} \)
  If \( \text{VkPhysicalDeviceDescriptorIndexingFeatures} :: \text{descriptorBindingStorageTexelBufferUpdateAfterBind} \) is not enabled, all bindings with descriptor type \( \text{VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER} \) must not use \( \text{VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT} \)

- \( \text{VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingInlineUniformBlockUpdateAfterBind-02211} \)
  If \( \text{VkPhysicalDeviceInlineUniformBlockFeatures} :: \text{descriptorBindingInlineUniformBlockUpdateAfterBind} \) is not enabled, all bindings with descriptor type \( \text{VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK} \) must not use \( \text{VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT} \)

- \( \text{VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-None-03011} \)
  All bindings with descriptor type \( \text{VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT}, \text{VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC}, \) or \( \text{VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC} \) must not use \( \text{VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT} \)
• VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingUpdateUnusedWhilePending-03012
  If VkPhysicalDeviceDescriptorIndexingFeatures
  ::descriptorBindingUpdateUnusedWhilePending is not enabled, all elements of pBindingFlags
  must not include VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT

• VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingPartiallyBound-03013
  If VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingPartiallyBound is not
  enabled, all elements of pBindingFlags must not include VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT

• VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingVariableDescriptorCount-03014
  If VkPhysicalDeviceDescriptorIndexingFeatures ::descriptorBindingVariableDescriptorCount is not enabled, all elements of pBindingFlags
  must not include VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT

• VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-pBindingFlags-03015
  If an element of pBindingFlags includes VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT, that element’s descriptorType must
  not be VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC or
  VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC

Valid Usage (Implicit)

• VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_BINDING_FLAGS_CREATE_INFO

• VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-pBindingFlags-parameter
  If bindingCount is not 0, pBindingFlags must be a valid pointer to an array of bindingCount
  valid combinations of VkDescriptorBindingFlagBits values

Bits which can be set in each element of VkDescriptorSetLayoutBindingFlagsCreateInfo
  ::pBindingFlags, specifying options for the corresponding descriptor set layout binding, are:

```c
// Provided by VK_VERSION_1_2
typedef enum VkDescriptorBindingFlagBits {
    VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT = 0x00000001,
    VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT = 0x00000002,
    VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT = 0x00000004,
    VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT = 0x00000008,
} VkDescriptorBindingFlagBits;
```

• VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT indicates that if descriptors in this binding are
  updated between when the descriptor set is bound in a command buffer and when that
  command buffer is submitted to a queue, then the submission will use the most recently set
  descriptors for this binding and the updates do not invalidate the command buffer. Descriptor
bindings created with this flag are also partially exempt from the external synchronization requirement in vkUpdateDescriptorSets. Multiple descriptors with this flag set can be updated concurrently in different threads, though the same descriptor must not be updated concurrently by two threads. Descriptors with this flag set can be updated concurrently with the set being bound to a command buffer in another thread, but not concurrently with the set being reset or freed.

- **VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT** indicates that descriptors in this binding that are not dynamically used need not contain valid descriptors at the time the descriptors are consumed. A descriptor is dynamically used if any shader invocation executes an instruction that performs any memory access using the descriptor. If a descriptor is not dynamically used, any resource referenced by the descriptor is not considered to be referenced during command execution.

- **VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT** indicates that descriptors in this binding can be updated after a command buffer has bound this descriptor set, or while a command buffer that uses this descriptor set is pending execution, as long as the descriptors that are updated are not used by those command buffers. Descriptor bindings created with this flag are also partially exempt from the external synchronization requirement in vkUpdateDescriptorSetWithTemplateKHR and vkUpdateDescriptorSets in the same way as for VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT. If VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT is also set, then descriptors can be updated as long as they are not dynamically used by any shader invocations. If VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT is not set, then descriptors can be updated as long as they are not statically used by any shader invocations.

- **VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT** indicates that this is a variable-sized descriptor binding whose size will be specified when a descriptor set is allocated using this layout. The value of descriptorCount is treated as an upper bound on the size of the binding. This must only be used for the last binding in the descriptor set layout (i.e. the binding with the largest value of binding). For the purposes of counting against limits such as maxDescriptorSet* and maxPerStageDescriptor*, the full value of descriptorCount is counted, except for descriptor bindings with a descriptor type of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK. In this case, descriptorCount specifies the upper bound on the byte size of the binding; thus it counts against the maxInlineUniformTotalSize limit instead.

---

**Note**

Note that while VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT and VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT both involve updates to descriptor sets after they are bound, VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT is a weaker requirement since it is only about descriptors that are not used, whereas VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT requires the implementation to observe updates to descriptors that are used.

```c
// Provided by VK_VERSION_1_2
typedef VkFlags VkDescriptorBindingFlags;
```

**VkDescriptorBindingFlags** is a bitmask type for setting a mask of zero or more
To query information about whether a descriptor set layout can be created, call:

```c
// Provided by VK_VERSION_1_1
void vkGetDescriptorSetLayoutSupport(
    VkDevice device,
    const VkDescriptorSetLayoutCreateInfo* pCreateInfo,
    VkDescriptorSetLayoutSupport* pSupport);
```

- `device` is the logical device that would create the descriptor set layout.
- `pCreateInfo` is a pointer to a `VkDescriptorSetLayoutCreateInfo` structure specifying the state of the descriptor set layout object.
- `pSupport` is a pointer to a `VkDescriptorSetLayoutSupport` structure, in which information about support for the descriptor set layout object is returned.

Some implementations have limitations on what fits in a descriptor set which are not easily expressible in terms of existing limits like `maxDescriptorSet`, for example if all descriptor types share a limited space in memory but each descriptor is a different size or alignment. This command returns information about whether a descriptor set satisfies this limit. If the descriptor set layout satisfies the `VkPhysicalDeviceMaintenance3Properties::maxPerSetDescriptors` limit, this command is guaranteed to return `VK_TRUE` in `VkDescriptorSetLayoutSupport::supported`. If the descriptor set layout exceeds the `VkPhysicalDeviceMaintenance3Properties::maxPerSetDescriptors` limit, whether the descriptor set layout is supported is implementation-dependent and may depend on whether the descriptor sizes and alignments cause the layout to exceed an internal limit.

This command does not consider other limits such as `maxPerStageDescriptor`, and so a descriptor set layout that is supported according to this command must still satisfy the pipeline layout limits such as `maxPerStageDescriptor` in order to be used in a pipeline layout.

**Note**

This is a `VkDevice` query rather than `VkPhysicalDevice` because the answer may depend on enabled features.

**Valid Usage (Implicit)**

- `VUID-vkGetDescriptorSetLayoutSupport-device-parameter`  
  `device` must be a valid `VkDevice` handle

- `VUID-vkGetDescriptorSetLayoutSupport-pCreateInfo-parameter`  
  `pCreateInfo` must be a valid pointer to a valid `VkDescriptorSetLayoutCreateInfo` structure

- `VUID-vkGetDescriptorSetLayoutSupport-pSupport-parameter`  
  `pSupport` must be a valid pointer to a `VkDescriptorSetLayoutSupport` structure

Information about support for the descriptor set layout is returned in a `VkDescriptorSetLayoutSupport` structure:
typedef struct VkDescriptorSetLayoutSupport {
    VkStructureType sType;
    void* pNext;
    VkBool32 supported;
} VkDescriptorSetLayoutSupport;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **supported** specifies whether the descriptor set layout can be created.

**Supported** is set to VK_TRUE if the descriptor set can be created, or else is set to VK_FALSE.

**Valid Usage (Implicit)**

- VUID-VkDescriptorSetLayoutSupport-sType-sType
  sType must be VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_SUPPORT
- VUID-VkDescriptorSetLayoutSupport-pNext-pNext
  pNext must be NULL or a pointer to a valid instance of
  VkDescriptorSetVariableDescriptorCountLayoutSupport
- VUID-VkDescriptorSetLayoutSupport-sType-unique
  The sType value of each struct in the pNext chain must be unique

If the pNext chain of a VkDescriptorSetLayoutSupport structure includes a
VkDescriptorSetVariableDescriptorCountLayoutSupport structure, then that structure returns
additional information about whether the descriptor set layout is supported.

typedef struct VkDescriptorSetVariableDescriptorCountLayoutSupport {
    VkStructureType sType;
    void* pNext;
    uint32_t maxVariableDescriptorCount;
} VkDescriptorSetVariableDescriptorCountLayoutSupport;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **maxVariableDescriptorCount** indicates the maximum number of descriptors supported in the
  highest numbered binding of the layout, if that binding is variable-sized. If the highest
  numbered binding of the layout has a descriptor type of
  VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK then maxVariableDescriptorCount indicates the
  maximum byte size supported for the binding, if that binding is variable-sized.

If the VkDescriptorSetLayoutCreateInfo structure specified in vkGetDescriptorSetLayoutSupport
::pCreateInfo includes a variable-sized descriptor, then **supported** is determined assuming the
requested size of the variable-sized descriptor, and \texttt{maxVariableDescriptorCount} is set to the maximum size of that descriptor that \textbf{can} be successfully created (which is greater than or equal to the requested size passed in). If the \texttt{VkDescriptorSetLayoutCreateInfo} structure does not include a variable-sized descriptor, or if the \texttt{VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingVariableDescriptorCount} feature is not enabled, then \texttt{maxVariableDescriptorCount} is set to zero. For the purposes of this command, a variable-sized descriptor binding with a \texttt{descriptorCount} of zero is treated as if the \texttt{descriptorCount} is one, and thus the binding is not ignored and the maximum descriptor count will be returned. If the layout is not supported, then the value written to \texttt{maxVariableDescriptorCount} is undefined.

\begin{mdframed}
\textbf{Valid Usage (Implicit)}

- \texttt{VUID-VkDescriptorSetVariableDescriptorCountLayoutSupport-sType-sType} \texttt{sType} \texttt{must} be \texttt{VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_LAYOUT_SUPPORT}

\end{mdframed}

The following examples show a shader snippet using two descriptor sets, and application code that creates corresponding descriptor set layouts.

\textit{GLSL example}

\begin{verbatim}
  // binding to a single sampled image descriptor in set 0
  // layout (set=0, binding=0) uniform texture2D mySampledImage;

  // binding to an array of sampled image descriptors in set 0
  // layout (set=0, binding=1) uniform texture2D myArrayOfSampledImages[12];

  // binding to a single uniform buffer descriptor in set 1
  // layout (set=1, binding=0) uniform myUniformBuffer
  //   { vec4 myElement[32]; }
\end{verbatim}

\textit{SPIR-V example}

\begin{verbatim}
...%1 = OpExtInstImport "GLSL.std.450"
...
  OpName %9 "mySampledImage"
  OpName %14 "myArrayOfSampledImages"
  OpName %18 "myUniformBuffer"
  OpMemberName %18 0 "myElement"
\end{verbatim}
OpName %20 ""
OpDecorate %9 DescriptorSet 0
OpDecorate %9 Binding 0
OpDecorate %14 DescriptorSet 0
OpDecorate %14 Binding 1
OpDecorate %17 ArrayStride 16
OpMemberDecorate %18 0 Offset 0
OpDecorate %18 Block
OpDecorate %20 DescriptorSet 1
OpDecorate %20 Binding 0
%2 = OpTypeVoid
%3 = OpTypeFunction %2
%6 = OpTypeFloat 32
%7 = OpTypeImage %6 2D 0 0 0 1 Unknown
%8 = OpTypePointer UniformConstant %7
%9 = OpVariable %8 UniformConstant
%10 = OpTypeInt 32 0
%11 = OpConstant %10 12
%12 = OpTypeArray %7 %11
%13 = OpTypePointer UniformConstant %12
%14 = OpVariable %13 UniformConstant
%15 = OpTypeVector %6 4
%16 = OpConstant %10 32
%17 = OpTypeArray %15 %16
%18 = OpTypeStruct %17
%19 = OpTypePointer Uniform %18
%20 = OpVariable %19 Uniform
...

API example

VkResult myResult;

const VkDescriptorSetLayoutBinding myDescriptorSetLayoutBinding[] =
{
    // binding to a single image descriptor
    {
        0, // binding
        VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, // descriptorType
        1, // descriptorCount
        VK_SHADER_STAGE_FRAGMENT_BIT, // stageFlags
        NULL // pImmutableSamplers
    },

    // binding to an array of image descriptors
    {
        1, // binding
        VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, // descriptorType
        12, // descriptorCount
        VK_SHADER_STAGE_FRAGMENT_BIT, // stageFlags
        ... // pImmutableSamplers
    }
};
// pImmutableSamplers

// binding to a single uniform buffer descriptor
{
    0, // binding
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, // descriptorType
    1, // descriptorCount
    VK_SHADER_STAGE_FRAGMENT_BIT, // stageFlags
    NULL // pImmutableSamplers
}

const VkDescriptorSetLayoutCreateInfo myDescriptorSetLayoutCreateInfo[] =
{
    // Information for first descriptor set with two descriptor bindings
    { VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO, // sType
      NULL, // pNext
      0, // flags
      2, // bindingCount
      myDescriptorSetLayoutBinding[0] // pBindings
    },

    // Information for second descriptor set with one descriptor binding
    { VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO, // sType
      NULL, // pNext
      0, // flags
      1, // bindingCount
    }
};

VkDescriptorSetLayout myDescriptorSetLayout[2];

// Create first descriptor set layout
// myResult = vkCreateDescriptorSetLayout(
//     myDevice,
//     &myDescriptorSetLayoutCreateInfo[0],
//     NULL,
//     &myDescriptorSetLayout[0]);

// Create second descriptor set layout
// myResult = vkCreateDescriptorSetLayout(
//     myDevice,
//     &myDescriptorSetLayoutCreateInfo[1],
//     NULL,
//     &myDescriptorSetLayout[1]);
To destroy a descriptor set layout, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyDescriptorSetLayout(
    VkDevice device,
    VkDescriptorSetLayout descriptorSetLayout,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the descriptor set layout.
- `descriptorSetLayout` is the descriptor set layout to destroy.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

**Valid Usage**

- VUID-vkDestroyDescriptorSetLayout-descriptorSetLayout-00284 If `VkAllocationCallbacks` were provided when `descriptorSetLayout` was created, a compatible set of callbacks must be provided here.
- VUID-vkDestroyDescriptorSetLayout-descriptorSetLayout-00285 If no `VkAllocationCallbacks` were provided when `descriptorSetLayout` was created, `pAllocator` must be NULL.

**Valid Usage (Implicit)**

- VUID-vkDestroyDescriptorSetLayout-device-parameter `device` must be a valid `VkDevice` handle.
- VUID-vkDestroyDescriptorSetLayout-descriptorSetLayout-parameter If `descriptorSetLayout` is not `VK_NULL_HANDLE`, `descriptorSetLayout` must be a valid `VkDescriptorSetLayout` handle.
- VUID-vkDestroyDescriptorSetLayout-pAllocator-parameter If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure.
- VUID-vkDestroyDescriptorSetLayout-descriptorSetLayout-parent If `descriptorSetLayout` is a valid handle, it must have been created, allocated, or retrieved from `device`.

**Host Synchronization**

- Host access to `descriptorSetLayout` must be externally synchronized.
14.2.2. Pipeline Layouts

Access to descriptor sets from a pipeline is accomplished through a pipeline layout. Zero or more descriptor set layouts and zero or more push constant ranges are combined to form a pipeline layout object describing the complete set of resources that can be accessed by a pipeline. The pipeline layout represents a sequence of descriptor sets with each having a specific layout. This sequence of layouts is used to determine the interface between shader stages and shader resources. Each pipeline is created using a pipeline layout.

Pipeline layout objects are represented by VkPipelineLayout handles:

```plaintext
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPipelineLayout)
```

To create a pipeline layout, call:

```plaintext
// Provided by VK_VERSION_1_0
VkResult vkCreatePipelineLayout(VkDevice device, const VkPipelineLayoutCreateInfo* pCreateInfo, const VkAllocationCallbacks* pAllocator, VkPipelineLayout* pPipelineLayout);
```

- `device` is the logical device that creates the pipeline layout.
- `pCreateInfo` is a pointer to a VkPipelineLayoutCreateInfo structure specifying the state of the pipeline layout object.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pPipelineLayout` is a pointer to a VkPipelineLayout handle in which the resulting pipeline layout object is returned.

**Valid Usage (Implicit)**

- VUID-vkCreatePipelineLayout-device-parameter
  - `device` must be a valid VkDevice handle
- VUID-vkCreatePipelineLayout-pCreateInfo-parameter
  - `pCreateInfo` must be a valid pointer to a valid VkPipelineLayoutCreateInfo structure
- VUID-vkCreatePipelineLayout-pAllocator-parameter
  - If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid VkAllocationCallbacks structure
- VUID-vkCreatePipelineLayout-pPipelineLayout-parameter
  - `pPipelineLayout` must be a valid pointer to a VkPipelineLayout handle
Return Codes

Success
  • VK_SUCCESS

Failure
  • VK_ERROR_OUT_OF_HOST_MEMORY
  • VK_ERROR_OUT_OF_DEVICE_MEMORY

The **VkPipelineLayoutCreateInfo** structure is defined as:

```c
// Provided by VK_VERSION_1.0
typedef struct VkPipelineLayoutCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineLayoutCreateFlags flags;
    uint32_tsetLayoutCount;
    const VkDescriptorSetLayout* pSetLayouts;
    uint32_tpushConstantRangeCount;
    const VkPushConstantRange* pPushConstantRanges;
} VkPipelineLayoutCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is a bitmask of **VkPipelineLayoutCreateFlagBits** specifying options for pipeline layout creation.
- **setLayoutCount** is the number of descriptor sets included in the pipeline layout.
- **pSetLayouts** is a pointer to an array of **VkDescriptorSetLayout** objects.
- **pushConstantRangeCount** is the number of push constant ranges included in the pipeline layout.
- **pPushConstantRanges** is a pointer to an array of **VkPushConstantRange** structures defining a set of push constant ranges for use in a single pipeline layout. In addition to descriptor set layouts, a pipeline layout also describes how many push constants can be accessed by each stage of the pipeline.

**Note**

Push constants represent a high speed path to modify constant data in pipelines that is expected to outperform memory-backed resource updates.

Valid Usage

- **VUID-VkPipelineLayoutCreateInfo-setLayoutCount-00286**
  ```
  setLayoutCount must be less than or equal to VkPhysicalDeviceLimits::maxBoundDescriptorSets
  ```
The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_SAMPLER` and `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorSamplers`.

The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` and `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorUniformBuffers`.

The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` and `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorStorageBuffers`.

The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, and `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorSampledImages`.

The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, and `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorStorageImages`.

The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorInputAttachments`.

The total number of bindings in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceInlineUniformBlockProperties::maxPerStageDescriptorInlineUniformBlocks`.

The total number of descriptors with a `descriptorType` of `VK_DESCRIPTOR_TYPE_SAMPLER` and...
VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER accessible to any given shader stage across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxPerStageDescriptorUpdateAfterBindSamplers

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03023
  The total number of descriptors with a descriptorType of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER and VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC accessible to any given shader stage across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxPerStageDescriptorUpdateAfterBindUniformBuffers

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03024
  The total number of descriptors with a descriptorType of VK_DESCRIPTOR_TYPE_STORAGE_BUFFER and VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC accessible to any given shader stage across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxPerStageDescriptorUpdateAfterBindStorageBuffers

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03025
  The total number of descriptors with a descriptorType of VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, and VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER accessible to any given shader stage across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxPerStageDescriptorUpdateAfterBindSampledImages

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03026
  The total number of descriptors with a descriptorType of VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, and VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER accessible to any given shader stage across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxPerStageDescriptorUpdateAfterBindStorageImages

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03027
  The total number of descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set with a descriptorType of VK_DESCRIPTOR_TYPE_SAMPLER and VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER accessible
across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetSamplers`.

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-03029**
  The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetUniformBuffers`.

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-03030**
  The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetUniformBuffersDynamic`.

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-03031**
  The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetStorageBuffers`.

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-03032**
  The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetStorageBuffersDynamic`.

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-03033**
  The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, and `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetSampledImages`.

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-03034**
  The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, and `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetStorageImages`.

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-03035**
  The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetInputAttachments`.
The total number of bindings in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set with a descriptorType of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceInlineUniformBlockProperties::maxDescriptorSetInlineUniformBlocks

The total number of descriptors of the type VK_DESCRIPTOR_TYPE_SAMPLER and VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindSamplers

The total number of descriptors of the type VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindUniformBuffers

The total number of descriptors of the type VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindUniformBuffersDynamic

The total number of descriptors of the type VK_DESCRIPTOR_TYPE_STORAGE_BUFFER accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindStorageBuffers

The total number of descriptors of the type VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindStorageBuffersDynamic

The total number of descriptors of the type VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, and VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindSampledImages

The total number of descriptors of the type VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, and VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindStorageImages

The total number of descriptors of the type VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT
accessible across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindInputAttachments`

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-02217**
  The total number of bindings with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK` accessible across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceInlineUniformBlockProperties::maxDescriptorSetUpdateAfterBindInlineUniformBlocks`

- **VUID-VkPipelineLayoutCreateInfo-pPushConstantRanges-00292**
  Any two elements of `pPushConstantRanges` must not include the same stage in `stageFlags`

- **VUID-VkPipelineLayoutCreateInfo-pSetLayouts-06561**
  Elements of `pSetLayouts` must be valid `VkDescriptorSetLayout` objects

---

### Valid Usage (Implicit)

- **VUID-VkPipelineLayoutCreateInfo-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO`

- **VUID-VkPipelineLayoutCreateInfo-pNext-pNext**
  `pNext` must be `NULL`

- **VUID-VkPipelineLayoutCreateInfo-flags-zerobitmask**
  `flags` must be 0

- **VUID-VkPipelineLayoutCreateInfo-pSetLayouts-parameter**
  If `setLayoutCount` is not 0, `pSetLayouts` must be a valid pointer to an array of `setLayoutCount` valid or `VK_NULL_HANDLE` `VkDescriptorSetLayout` handles

- **VUID-VkPipelineLayoutCreateInfo-pPushConstantRanges-parameter**
  If `pushConstantRangeCount` is not 0, `pPushConstantRanges` must be a valid pointer to an array of `pushConstantRangeCount` valid `VkPushConstantRange` structures

---

```c
typedef enum VkPipelineLayoutCreateFlagBits {
} VkPipelineLayoutCreateFlagBits;
```

All values for this enum are defined by extensions.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineLayoutCreateFlags;
```

`VkPipelineLayoutCreateFlags` is a bitmask type for setting a mask of `VkPipelineLayoutCreateFlagBits`.

The `VkPushConstantRange` structure is defined as:
typedef struct VkPushConstantRange {
    VkShaderStageFlags stageFlags;
    uint32_t offset;
    uint32_t size;
} VkPushConstantRange;

- **stageFlags** is a set of stage flags describing the shader stages that will access a range of push constants. If a particular stage is not included in the range, then accessing members of that range of push constants from the corresponding shader stage will return undefined values.

- **offset** and **size** are the start offset and size, respectively, consumed by the range. Both **offset** and **size** are in units of bytes and **must** be a multiple of 4. The layout of the push constant variables is specified in the shader.

### Valid Usage

- VUID-VkPushConstantRange-offset-00294
  
  **offset must** be less than **VkPhysicalDeviceLimits::maxPushConstantsSize**

- VUID-VkPushConstantRange-offset-00295
  
  **offset must** be a multiple of 4

- VUID-VkPushConstantRange-size-00296
  
  **size must** be greater than 0

- VUID-VkPushConstantRange-size-00297
  
  **size must** be a multiple of 4

- VUID-VkPushConstantRange-size-00298
  
  **size must** be less than or equal to **VkPhysicalDeviceLimits::maxPushConstantsSize** minus **offset**

### Valid Usage (Implicit)

- VUID-VkPushConstantRange-stageFlags-parameter
  
  **stageFlags must** be a valid combination of **VkShaderStageFlagBits** values

- VUID-VkPushConstantRange-stageFlags-requiredbitmask
  
  **stageFlags must** not be 0

Once created, pipeline layouts are used as part of pipeline creation (see Pipelines), as part of binding descriptor sets (see Descriptor Set Binding), and as part of setting push constants (see Push Constant Updates). Pipeline creation accepts a pipeline layout as input, and the layout may be used to map (set, binding, arrayElement) tuples to implementation resources or memory locations within a descriptor set. The assignment of implementation resources depends only on the bindings defined in the descriptor sets that comprise the pipeline layout, and not on any shader source.

All resource variables statically used in all shaders in a pipeline must be declared with a (set,
binding, arrayElement) that exists in the corresponding descriptor set layout and is of an appropriate descriptor type and includes the set of shader stages it is used by in stageFlags. The pipeline layout can include entries that are not used by a particular pipeline, or that are dead-code eliminated from any of the shaders. The pipeline layout allows the application to provide a consistent set of bindings across multiple pipeline compiles, which enables those pipelines to be compiled in a way that the implementation may cheaply switch pipelines without reprogramming the bindings.

Similarly, the push constant block declared in each shader (if present) must only place variables at offsets that are each included in a push constant range with stageFlags including the bit corresponding to the shader stage that uses it. The pipeline layout can include ranges or portions of ranges that are not used by a particular pipeline, or for which the variables have been dead-code eliminated from any of the shaders.

There is a limit on the total number of resources of each type that can be included in bindings in all descriptor set layouts in a pipeline layout as shown in Pipeline Layout Resource Limits. The “Total Resources Available” column gives the limit on the number of each type of resource that can be included in bindings in all descriptor sets in the pipeline layout. Some resource types count against multiple limits. Additionally, there are limits on the total number of each type of resource that can be used in any pipeline stage as described in Shader Resource Limits.

<table>
<thead>
<tr>
<th>Total Resources Available</th>
<th>Resource Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxDescriptorSetSamplers</td>
<td>sampler</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindSamplers</td>
<td>combined image sampler</td>
</tr>
<tr>
<td>maxDescriptorSetSampledImages</td>
<td>sampled image</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindSampledImages</td>
<td>combined image sampler</td>
</tr>
<tr>
<td>maxDescriptorSetStorageImages</td>
<td>uniform texel buffer</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindStorageImages</td>
<td>storage texel buffer</td>
</tr>
<tr>
<td>maxDescriptorSetUniformBuffers</td>
<td>uniform buffer</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindUniformBuffers</td>
<td>uniform buffer dynamic</td>
</tr>
<tr>
<td>maxDescriptorSetUniformBuffersDynamic</td>
<td>uniform buffer dynamic</td>
</tr>
<tr>
<td>maxDescriptorSetStorageBuffers</td>
<td>storage buffer</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindStorageBuffers</td>
<td>storage buffer dynamic</td>
</tr>
<tr>
<td>maxDescriptorSetStorageBuffersDynamic</td>
<td>storage buffer dynamic</td>
</tr>
<tr>
<td>maxDescriptorSetInputAttachments</td>
<td>input attachment</td>
</tr>
</tbody>
</table>

There is a limit on the total number of resources of each type that can be included in bindings in all descriptor set layouts in a pipeline layout as shown in Pipeline Layout Resource Limits. The “Total Resources Available” column gives the limit on the number of each type of resource that can be included in bindings in all descriptor sets in the pipeline layout. Some resource types count against multiple limits. Additionally, there are limits on the total number of each type of resource that can be used in any pipeline stage as described in Shader Resource Limits.
To destroy a pipeline layout, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyPipelineLayout(
    VkDevice            device,  
    VkPipelineLayout    pipelineLayout, 
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the pipeline layout.
- `pipelineLayout` is the pipeline layout to destroy.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage

- VUID-vkDestroyPipelineLayout-pipelineLayout-00299
  If `VkAllocationCallbacks` were provided when `pipelineLayout` was created, a compatible set of callbacks must be provided here.

- VUID-vkDestroyPipelineLayout-pipelineLayout-00300
  If no `VkAllocationCallbacks` were provided when `pipelineLayout` was created, `pAllocator` must be `NULL`.

- VUID-vkDestroyPipelineLayout-pipelineLayout-02004
  `pipelineLayout` must not have been passed to any `vkCmd*` command for any command buffers that are still in the recording state when `vkDestroyPipelineLayout` is called.

### Valid Usage (Implicit)

- VUID-vkDestroyPipelineLayout-device-parameter
  `device` must be a valid `VkDevice` handle.

- VUID-vkDestroyPipelineLayout-pipelineLayout-parameter
  If `pipelineLayout` is not `VK_NULL_HANDLE`, `pipelineLayout` must be a valid `VkPipelineLayout` handle.

- VUID-vkDestroyPipelineLayout-pAllocator-parameter
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure.

- VUID-vkDestroyPipelineLayout-pipelineLayout-parent
  If `pipelineLayout` is a valid handle, it must have been created, allocated, or retrieved from `device`.
Host Synchronization

- Host access to pipelineLayout must be externally synchronized

Pipeline Layout Compatibility

Two pipeline layouts are defined to be “compatible for push constants” if they were created with identical push constant ranges. Two pipeline layouts are defined to be “compatible for set N” if they were created with identically defined descriptor set layouts for sets zero through N, and if they were created with identical push constant ranges.

When binding a descriptor set (see Descriptor Set Binding) to set number N, if the previously bound descriptor sets for sets zero through N-1 were all bound using compatible pipeline layouts, then performing this binding does not disturb any of the lower numbered sets. If, additionally, the previously bound descriptor set for set N was bound using a pipeline layout compatible for set N, then the bindings in sets numbered greater than N are also not disturbed.

Similarly, when binding a pipeline, the pipeline can correctly access any previously bound descriptor sets which were bound with compatible pipeline layouts, as long as all lower numbered sets were also bound with compatible layouts.

Layout compatibility means that descriptor sets can be bound to a command buffer for use by any pipeline created with a compatible pipeline layout, and without having bound a particular pipeline first. It also means that descriptor sets can remain valid across a pipeline change, and the same resources will be accessible to the newly bound pipeline.

Implementor’s Note

A consequence of layout compatibility is that when the implementation compiles a pipeline layout and maps pipeline resources to implementation resources, the mechanism for set N should only be a function of sets [0..N].

Note

Place the least frequently changing descriptor sets near the start of the pipeline layout, and place the descriptor sets representing the most frequently changing resources near the end. When pipelines are switched, only the descriptor set bindings that have been invalidated will need to be updated and the remainder of the descriptor set bindings will remain in place.

The maximum number of descriptor sets that can be bound to a pipeline layout is queried from physical device properties (see maxBoundDescriptorSets in Limits).

API example

```cpp
const VkDescriptorSetLayout layouts[] = { layout1, layout2 };`n`const VkPushConstantRange ranges[] = `n```
14.2.3. Allocation of Descriptor Sets

A descriptor pool maintains a pool of descriptors, from which descriptor sets are allocated. Descriptor pools are externally synchronized, meaning that the application must not allocate and/or free descriptor sets from the same pool in multiple threads simultaneously.

Descriptor pools are represented by `VkDescriptorPool` handles:

```cpp
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorPool)
```

To create a descriptor pool object, call:

```cpp
// Provided by VK_VERSION_1_0
VkResult vkCreateDescriptorPool(
    VkDevice device,
```
const VkDescriptorPoolCreateInfo* pCreateInfo,  
const VkAllocationCallbacks* pAllocator,  
VkDescriptorPool* pDescriptorPool); 

• device is the logical device that creates the descriptor pool.
• pCreateInfo is a pointer to a VkDescriptorPoolCreateInfo structure specifying the state of the descriptor pool object.
• pAllocator controls host memory allocation as described in the Memory Allocation chapter.
• pDescriptorPool is a pointer to a VkDescriptorPool handle in which the resulting descriptor pool object is returned.

The created descriptor pool is returned in pDescriptorPool.

Valid Usage (Implicit)

• VUID-vkCreateDescriptorPool-device-parameter
device must be a valid VkDevice handle

• VUID-vkCreateDescriptorPool-pCreateInfo-parameter
pCreateInfo must be a valid pointer to a valid VkDescriptorPoolCreateInfo structure

• VUID-vkCreateDescriptorPool-pAllocator-parameter
If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

• VUID-vkCreateDescriptorPool-pDescriptorPool-parameter
pDescriptorPool must be a valid pointer to a VkDescriptorPool handle

Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

Additional information about the pool is passed in a VkDescriptorPoolCreateInfo structure:

// Provided by VK_VERSION_1_0
typedef struct VkDescriptorPoolCreateInfo {  
    VkStructureType sType;  
    const void* pNext;  
    VkDescriptorPoolCreateFlags flags;  
    uint32_t maxSets;  
    uint32_t poolSizeCount;
} VkDescriptorPoolCreateInfo;
• `sType` is the type of this structure.
• `pNext` is `NULL` or a pointer to a structure extending this structure.
• `flags` is a bitmask of `VkDescriptorPoolCreateFlagBits` specifying certain supported operations on the pool.
• `maxSets` is the maximum number of descriptor sets that can be allocated from the pool.
• `poolSizeCount` is the number of elements in `pPoolSizes`.
• `pPoolSizes` is a pointer to an array of `VkDescriptorPoolSize` structures, each containing a descriptor type and number of descriptors of that type to be allocated in the pool.

If multiple `VkDescriptorPoolSize` structures containing the same descriptor type appear in the `pPoolSizes` array then the pool will be created with enough storage for the total number of descriptors of each type.

Fragmentation of a descriptor pool is possible and may lead to descriptor set allocation failures. A failure due to fragmentation is defined as failing a descriptor set allocation despite the sum of all outstanding descriptor set allocations from the pool plus the requested allocation requiring no more than the total number of descriptors requested at pool creation. Implementations provide certain guarantees of when fragmentation must not cause allocation failure, as described below.

If a descriptor pool has not had any descriptor sets freed since it was created or most recently reset then fragmentation must not cause an allocation failure (note that this is always the case for a pool created without the `VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT` bit set). Additionally, if all sets allocated from the pool since it was created or most recently reset use the same number of descriptors (of each type) and the requested allocation also uses that same number of descriptors (of each type), then fragmentation must not cause an allocation failure.

If an allocation failure occurs due to fragmentation, an application can create an additional descriptor pool to perform further descriptor set allocations.

If `flags` has the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT` bit set, descriptor pool creation may fail with the error `VK_ERROR_FRAGMENTATION` if the total number of descriptors across all pools (including this one) created with this bit set exceeds `maxUpdateAfterBindDescriptorsInAllPools`, or if fragmentation of the underlying hardware resources occurs.

### Valid Usage

- VUID-VkDescriptorPoolCreateInfo-maxSets-00301
  
  `maxSets` must be greater than 0

### Valid Usage ( Implicit )

- VUID-VkDescriptorPoolCreateInfo-sType-sType

---

603
**sType** must be **VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO**

- **VUID-VkDescriptorPoolCreateInfo-pNext-pNext**
  
  `pNext` must be **NULL** or a pointer to a valid instance of `VkDescriptorPoolInlineUniformBlockCreateInfo`

- **VUID-VkDescriptorPoolCreateInfo-sType-unique**
  
  The **sType** value of each struct in the `pNext` chain must be unique.

- **VUID-VkDescriptorPoolCreateInfo-flags-parameter**
  
  `flags` must be a valid combination of `VkDescriptorPoolCreateFlagBits` values.

- **VUID-VkDescriptorPoolCreateInfo-pPoolSizes-parameter**
  
  If `poolSizeCount` is not 0, `pPoolSizes` must be a valid pointer to an array of `poolSizeCount` valid `VkDescriptorPoolSize` structures.

In order to be able to allocate descriptor sets having inline uniform block bindings the descriptor pool must be created with specifying the inline uniform block binding capacity of the descriptor pool, in addition to the total inline uniform data capacity in bytes which is specified through a `VkDescriptorPoolSize` structure with a `descriptorType` value of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK`. This can be done by adding a `VkDescriptorPoolInlineUniformBlockCreateInfo` structure to the `pNext` chain of `VkDescriptorPoolCreateInfo`.

The `VkDescriptorPoolInlineUniformBlockCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkDescriptorPoolInlineUniformBlockCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t maxInlineUniformBlockBindings;
} VkDescriptorPoolInlineUniformBlockCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **maxInlineUniformBlockBindings** is the number of inline uniform block bindings to allocate.

**Valid Usage (Implicit)**

- **VUID-VkDescriptorPoolInlineUniformBlockCreateInfo-sType-sType**
  
  `sType` must be **VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_INLINE_UNIFORM_BLOCK_CREATE_INFO**

Bits which can be set in `VkDescriptorPoolCreateInfo::flags`, enabling operations on a descriptor pool, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkDescriptorPoolCreateFlagBits {
```

---

604
VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT = 0x00000001,
// Provided by VK_VERSION_1_2
VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT = 0x00000002,
} VkDescriptorPoolCreateFlagBits;

- **VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT** specifies that descriptor sets can return their individual allocations to the pool, i.e. all of `vkAllocateDescriptorSets`, `vkFreeDescriptorSets`, and `vkResetDescriptorPool` are allowed. Otherwise, descriptor sets allocated from the pool must not be individually freed back to the pool, i.e. only `vkAllocateDescriptorSets` and `vkResetDescriptorPool` are allowed.

- **VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT** specifies that descriptor sets allocated from this pool can include bindings with the `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` bit set. It is valid to allocate descriptor sets that have bindings that do not set the `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` bit from a pool that has `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT` set.

// Provided by VK_VERSION_1_0
typedef VkFlags VkDescriptorPoolCreateFlags;

`VkDescriptorPoolCreateFlags` is a bitmask type for setting a mask of zero or more `VkDescriptorPoolCreateFlagBits`.

The `VkDescriptorPoolSize` structure is defined as:

// Provided by VK_VERSION_1_0
typedef struct VkDescriptorPoolSize {
    VkDescriptorType type;
    uint32_t descriptorCount;
} VkDescriptorPoolSize;

- **type** is the type of descriptor.
- **descriptorCount** is the number of descriptors of that type to allocate. If **type** is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK` then **descriptorCount** must be a multiple of 4.

**Valid Usage**

- VUID-VkDescriptorPoolSize-descriptorCount-00302 `descriptorCount` must be greater than 0
- VUID-VkDescriptorPoolSize-type-02218 If **type** is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK` then **descriptorCount** must be a multiple of 4
Valid Usage (Implicit)

- **VUID-VkDescriptorPoolSize-type-parameter**
  type must be a valid `VkDescriptorType` value

To destroy a descriptor pool, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyDescriptorPool(
    VkDevice device,
    VkDescriptorPool descriptorPool,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the descriptor pool.
- `descriptorPool` is the descriptor pool to destroy.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

When a pool is destroyed, all descriptor sets allocated from the pool are implicitly freed and become invalid. Descriptor sets allocated from a given pool do not need to be freed before destroying that descriptor pool.

Valid Usage

- **VUID-vkDestroyDescriptorPool-descriptorPool-00303**
  All submitted commands that refer to `descriptorPool` (via any allocated descriptor sets) must have completed execution

- **VUID-vkDestroyDescriptorPool-descriptorPool-00304**
  If `VkAllocationCallbacks` were provided when `descriptorPool` was created, a compatible set of callbacks must be provided here

- **VUID-vkDestroyDescriptorPool-descriptorPool-00305**
  If no `VkAllocationCallbacks` were provided when `descriptorPool` was created, `pAllocator` must be NULL

Valid Usage (Implicit)

- **VUID-vkDestroyDescriptorPool-device-parameter**
  `device` must be a valid `VkDevice` handle

- **VUID-vkDestroyDescriptorPool-descriptorPool-parameter**
  If `descriptorPool` is not `VK_NULL_HANDLE`, `descriptorPool` must be a valid `VkDescriptorPool` handle

- **VUID-vkDestroyDescriptorPool-pAllocator-parameter**
  If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid
Descriptor sets are allocated from descriptor pool objects, and are represented by `VkDescriptorSet` handles:

```c
// Provided by VK_VERSION_1_0
Vk_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorSet)
```

To allocate descriptor sets from a descriptor pool, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkAllocateDescriptorSets(
    VkDevice device,
    const VkDescriptorSetAllocateInfo* pAllocateInfo,
    VkDescriptorSet* pDescriptorSets);
```

- `device` is the logical device that owns the descriptor pool.
- `pAllocateInfo` is a pointer to a `VkDescriptorSetAllocateInfo` structure describing parameters of the allocation.
- `pDescriptorSets` is a pointer to an array of `VkDescriptorSet` handles in which the resulting descriptor set objects are returned.

The allocated descriptor sets are returned in `pDescriptorSets`.

When a descriptor set is allocated, the initial state is largely uninitialized and all descriptors are undefined, with the exception that samplers with a non-null `pImmutableSamplers` are initialized on allocation. Descriptors also become undefined if the underlying resource or view object is destroyed. Descriptor sets containing undefined descriptors can still be bound and used, subject to the following conditions:

- For descriptor set bindings created with the `VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT` bit set, all descriptors in that binding that are dynamically used must have been populated before the descriptor set is consumed.
- For descriptor set bindings created without the `VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT` bit set, all descriptors in that binding that are statically used must have been populated before the descriptor set is consumed.
• Descriptor bindings with descriptor type of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK` can be undefined when the descriptor set is consumed; though values in that block will be undefined.

• Entries that are not used by a pipeline can have undefined descriptors.

If a call to `vkAllocateDescriptorSets` would cause the total number of descriptor sets allocated from the pool to exceed the value of `VkDescriptorPoolCreateInfo::maxSets` used to create `pAllocateInfo->descriptorPool`, then the allocation may fail due to lack of space in the descriptor pool. Similarly, the allocation may fail due to lack of space if the call to `vkAllocateDescriptorSets` would cause the number of any given descriptor type to exceed the sum of all the `descriptorCount` members of each element of `VkDescriptorPoolCreateInfo::pPoolSizes` with a type equal to that type.

Additionally, the allocation may also fail if a call to `vkAllocateDescriptorSets` would cause the total number of inline uniform block bindings allocated from the pool to exceed the value of `VkDescriptorPoolInlineUniformBlockCreateInfo::maxInlineUniformBlockBindings` used to create the descriptor pool.

If the allocation fails due to no more space in the descriptor pool, and not because of system or device memory exhaustion, then `VK_ERROR_OUT_OF_POOL_MEMORY` must be returned.

`vkAllocateDescriptorSets` can be used to create multiple descriptor sets. If the creation of any of those descriptor sets fails, then the implementation must destroy all successfully created descriptor set objects from this command, set all entries of the `pDescriptorSets` array to `VK_NULL_HANDLE` and return the error.

---

**Valid Usage (Implicit)**

- VUID-vkAllocateDescriptorSets-device-parameter
  
  `device` must be a valid `VkDevice` handle

- VUID-vkAllocateDescriptorSets-pAllocateInfo-parameter
  
  `pAllocateInfo` must be a valid pointer to a valid `VkDescriptorSetAllocateInfo` structure

- VUID-vkAllocateDescriptorSets-pDescriptorSets-parameter
  
  `pDescriptorSets` must be a valid pointer to an array of `pAllocateInfo->descriptorSetCount` `VkDescriptorSet` handles

- VUID-vkAllocateDescriptorSets-pAllocateInfo::descriptorSetCount-arraylength
  
  `pAllocateInfo->descriptorSetCount` must be greater than 0

---

**Host Synchronization**

- Host access to `pAllocateInfo->descriptorPool` must be externally synchronized

---

**Return Codes**

**Success**

- `VK_SUCCESS`
The `VkDescriptorSetAllocateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDescriptorSetAllocateInfo {
    VkStructureType sType;
    const void* pNext;
    VkDescriptorPool descriptorPool;
    uint32_t descriptorSetCount;
    const VkDescriptorSetLayout* pSetLayouts;
} VkDescriptorSetAllocateInfo;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `descriptorPool` is the pool which the sets will be allocated from.
- `descriptorSetCount` determines the number of descriptor sets to be allocated from the pool.
- `pSetLayouts` is a pointer to an array of descriptor set layouts, with each member specifying how the corresponding descriptor set is allocated.

### Valid Usage

- VUID-VkDescriptorSetAllocateInfo-pSetLayouts-03044
  If any element of `pSetLayouts` was created with the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set, `descriptorPool` must have been created with the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT` flag set.

### Valid Usage (Implicit)

- VUID-VkDescriptorSetAllocateInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO`
- VUID-VkDescriptorSetAllocateInfo-pNext-pNext
  `pNext` must be NULL or a pointer to a valid instance of `VkDescriptorSetVariableDescriptorCountAllocateInfo`
- VUID-VkDescriptorSetAllocateInfo-sType-unique
  The `sType` value of each struct in the `pNext` chain must be unique
- VUID-VkDescriptorSetAllocateInfo-descriptorPool-parameter

---

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_FRAGMENTED_POOL`
- `VK_ERROR_OUT_OF_POOL_MEMORY`
descriptorPool must be a valid VkDescriptorPool handle

- VUID-VkDescriptorSetAllocateInfo-pSetLayouts-parameter
  pSetLayouts must be a valid pointer to an array of descriptorSetCount valid VkDescriptorSetLayout handles

- VUID-VkDescriptorSetAllocateInfo-descriptorSetCount-arraylength
descriptorSetCount must be greater than 0

- VUID-VkDescriptorSetAllocateInfo-commonparent
  Both of descriptorPool, and the elements of pSetLayouts must have been created, allocated, or retrieved from the same VkDevice

If the pNext chain of a VkDescriptorSetAllocateInfo structure includes a VkDescriptorSetVariableDescriptorCountAllocateInfo structure, then that structure includes an array of descriptor counts for variable-sized descriptor bindings, one for each descriptor set being allocated.

The VkDescriptorSetVariableDescriptorCountAllocateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkDescriptorSetVariableDescriptorCountAllocateInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t descriptorSetCount;
    const uint32_t* pDescriptorCounts;
} VkDescriptorSetVariableDescriptorCountAllocateInfo;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- descriptorSetCount is zero or the number of elements in pDescriptorCounts.
- pDescriptorCounts is a pointer to an array of descriptor counts, with each member specifying the number of descriptors in a variable-sized descriptor binding in the corresponding descriptor set being allocated.

If descriptorSetCount is zero or this structure is not included in the pNext chain, then the variable lengths are considered to be zero. Otherwise, pDescriptorCounts[i] is the number of descriptors in the variable-sized descriptor binding in the corresponding descriptor set layout. If the variable-sized descriptor binding in the corresponding descriptor set layout has a descriptor type of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK then pDescriptorCounts[i] specifies the binding's capacity in bytes. If VkDescriptorSetAllocateInfo::pSetLayouts[i] does not include a variable-sized descriptor binding, then pDescriptorCounts[i] is ignored.

**Valid Usage**

- VUID-VkDescriptorSetVariableDescriptorCountAllocateInfo-descriptorSetCount-03045
  If descriptorSetCount is not zero, descriptorSetCount must equal 610
VkDescriptorSetAllocateInfo::<code>descriptorSetCount</code>

- VUID-VkDescriptorSetVariableDescriptorCountAllocateInfo-pSetLayouts-03046
  If <code>VkDescriptorSetAllocateInfo::pSetLayouts[i]</code> has a variable-sized descriptor binding, then <code>pDescriptorCounts[i]</code> must be less than or equal to the descriptor count specified for that binding when the descriptor set layout was created.

**Valid Usage (Implicit)**

- VUID-VkDescriptorSetVariableDescriptorCountAllocateInfo-sType-sType
  <code>sType</code> must be <code>VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_ALLOCATE_INFO</code>

- VUID-VkDescriptorSetVariableDescriptorCountAllocateInfo-pDescriptorCounts-parameter
  If <code>descriptorSetCount</code> is not 0, <code>pDescriptorCounts</code> must be a valid pointer to an array of <code>descriptorSetCount uint32_t</code> values.

To free allocated descriptor sets, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkFreeDescriptorSets(  
  VkDevice device,  
  VkDescriptorPool descriptorPool,  
  uint32_t descriptorSetCount,  
  const VkDescriptorSet* pDescriptorSets);
```

- <code>device</code> is the logical device that owns the descriptor pool.
- <code>descriptorPool</code> is the descriptor pool from which the descriptor sets were allocated.
- <code>descriptorSetCount</code> is the number of elements in the <code>pDescriptorSets</code> array.
- <code>pDescriptorSets</code> is a pointer to an array of handles to <code>VkDescriptorSet</code> objects.

After calling <code>vkFreeDescriptorSets</code>, all descriptor sets in <code>pDescriptorSets</code> are invalid.

**Valid Usage**

- VUID-vkFreeDescriptorSets-pDescriptorSets-00309
  All submitted commands that refer to any element of <code>pDescriptorSets</code> must have completed execution.

- VUID-vkFreeDescriptorSets-pDescriptorSets-00310
  <code>pDescriptorSets</code> must be a valid pointer to an array of <code>descriptorSetCount VkDescriptorSet</code> handles, each element of which must either be a valid handle or <code>VK_NULL_HANDLE</code>.

- VUID-vkFreeDescriptorSets-descriptorPool-00312
  <code>descriptorPool</code> must have been created with the <code>VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT</code> flag.
Valid Usage (Implicit)

- **VUID-vkFreeDescriptorSets-device-parameter**
  
  *device must* be a valid *VkDevice* handle

- **VUID-vkFreeDescriptorSets-descriptorPool-parameter**
  
  *descriptorPool must* be a valid *VkDescriptorPool* handle

- **VUID-vkFreeDescriptorSets-descriptorSetCount-arraylength**
  
  *descriptorSetCount must* be greater than 0

- **VUID-vkFreeDescriptorSets-descriptorPool-parent**
  
  *descriptorPool must* have been created, allocated, or retrieved from *device*

- **VUID-vkFreeDescriptorSets-pDescriptorSets-parent**
  
  Each element of *pDescriptorSets* that is a valid handle *must* have been created, allocated, or retrieved from *descriptorPool*

Host Synchronization

- Host access to *descriptorPool must* be externally synchronized

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

Return Codes

**Success**

- **VK_SUCCESS**

To return all descriptor sets allocated from a given pool to the pool, rather than freeing individual descriptor sets, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkResetDescriptorPool( 
    VkDevice device, 
    VkDescriptorPool descriptorPool, 
    VkDescriptorPoolResetFlags flags);
```

- *device* is the logical device that owns the descriptor pool.

- *descriptorPool* is the descriptor pool to be reset.

- *flags* is reserved for future use.

Resetting a descriptor pool recycles all of the resources from all of the descriptor sets allocated from the descriptor pool back to the descriptor pool, and the descriptor sets are implicitly freed.
Valid Usage

- VUID-vkResetDescriptorPool-descriptorPool-00313
  All uses of descriptorPool (via any allocated descriptor sets) must have completed execution

Valid Usage (Implicit)

- VUID-vkResetDescriptorPool-device-parameter
device must be a valid VkDevice handle
- VUID-vkResetDescriptorPool-descriptorPool-parameter
descriptorPool must be a valid VkDescriptorPool handle
- VUID-vkResetDescriptorPool-flags-zerobitmask
  flags must be 0
- VUID-vkResetDescriptorPool-descriptorPool-parent
descriptorPool must have been created, allocated, or retrieved from device

Host Synchronization

- Host access to descriptorPool must be externally synchronized
- Host access to any VkDescriptorSet objects allocated from descriptorPool must be externally synchronized

Return Codes

Success
- VK_SUCCESS

// Provided by VK_VERSION_1_0
typedef VkFlags VkDescriptorPoolResetFlags;

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

14.2.4. Descriptor Set Updates

Once allocated, descriptor sets can be updated with a combination of write and copy operations. To update descriptor sets, call:

// Provided by VK_VERSION_1_0
void vkUpdateDescriptorSets(
    VkDevice device,
    uint32_t descriptorWriteCount,
    const VkWriteDescriptorSet* pDescriptorWrites,
    uint32_t descriptorCopyCount,
    const VkCopyDescriptorSet* pDescriptorCopies);

- **device** is the logical device that updates the descriptor sets.
- **descriptorWriteCount** is the number of elements in the `pDescriptorWrites` array.
- **pDescriptorWrites** is a pointer to an array of `VkWriteDescriptorSet` structures describing the descriptor sets to write to.
- **descriptorCopyCount** is the number of elements in the `pDescriptorCopies` array.
- **pDescriptorCopies** is a pointer to an array of `VkCopyDescriptorSet` structures describing the descriptor sets to copy between.

The operations described by `pDescriptor Writes` are performed first, followed by the operations described by `pDescriptorCopies`. Within each array, the operations are performed in the order they appear in the array.

Each element in the `pDescriptorWrites` array describes an operation updating the descriptor set using descriptors for resources specified in the structure.

Each element in the `pDescriptorCopies` array is a `VkCopyDescriptorSet` structure describing an operation copying descriptors between sets.

If the `dstSet` member of any element of `pDescriptorWrites` or `pDescriptorCopies` is bound, accessed, or modified by any command that was recorded to a command buffer which is currently in the recording or executable state, and any of the descriptor bindings that are updated were not created with the `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` or `VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT` bits set, that command buffer becomes invalid.

### Valid Usage

- **VUID-vkUpdateDescriptorSets-pDescriptorWrites-06236**
  For each element `i` where `pDescriptorWrites[i].descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER`, elements of the `pTexelBufferView` member of `pDescriptorWrites[i]` must have been created on device.

- **VUID-vkUpdateDescriptorSets-pDescriptorWrites-06237**
  For each element `i` where `pDescriptorWrites[i].descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`, `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER`, `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`, or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`, the `buffer` member of any element of the ` pBufferInfo` member of `pDescriptorWrites[i]` must have been created on device.

- **VUID-vkUpdateDescriptorSets-pDescriptorWrites-06238**
For each element $i$ where $p_{\text{Descriptor Writes}}[i].\text{descriptorType}$ is $\text{VK_DESCRIPTOR_TYPE_SAMPLER}$ or $\text{VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER}$, and $\text{dstSet}$ was not allocated with a layout that included immutable samplers for $\text{dstBinding}$ with $\text{descriptorType}$, the $\text{sampler}$ member of any element of the $p_{\text{ImageInfo}}$ member of $p_{\text{Descriptor Writes}}[i]$ must have been created on device.

- VUID-vkUpdateDescriptorSets-pDescriptorWrites-06239
  For each element $i$ where $p_{\text{Descriptor Writes}}[i].\text{descriptorType}$ is $\text{VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE}$, $\text{VK_DESCRIPTOR_TYPE_STORAGE_IMAGE}$, $\text{VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT}$, or $\text{VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER}$ the $\text{imageView}$ member of any element of $p_{\text{Descriptor Writes}}[i]$ must have been created on device.

- VUID-vkUpdateDescriptorSets-pDescriptorWrites-06493
  For each element $i$ where $p_{\text{Descriptor Writes}}[i].\text{descriptorType}$ is $\text{VK_DESCRIPTOR_TYPE_SAMPLER}$, $\text{VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER}$, $\text{VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE}$, $\text{VK_DESCRIPTOR_TYPE_STORAGE_IMAGE}$, or $\text{VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT}$, $p_{\text{Descriptor Writes}}[i].p_{\text{Image Info}}$ must be a valid pointer to an array of $p_{\text{Descriptor Writes}}[i].\text{descriptorCount}$ valid $\text{VkDescriptorImageInfo}$ structures.

- VUID-vkUpdateDescriptorSets-None-03047
  Descriptor bindings updated by this command which were created without the $\text{VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT}$ or $\text{VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT}$ bits set must not be used by any command that was recorded to a command buffer which is in the pending state.

- VUID-vkUpdateDescriptorSets-pDescriptorWrites-06993
  Host access to $p_{\text{Descriptor Writes}}[i].\text{dstSet}$ and $p_{\text{Descriptor Copies}}[i].\text{dstSet}$ must be externally synchronized unless explicitly denoted otherwise for specific flags.

**Valid Usage (Implicit)**

- VUID-vkUpdateDescriptorSets-device-parameter
  $\text{device}$ must be a valid $\text{VkDevice}$ handle.

- VUID-vkUpdateDescriptorSets-pDescriptorWrites-parameter
  If $\text{descriptorWriteCount}$ is not $0$, $p_{\text{Descriptor Writes}}$ must be a valid pointer to an array of $\text{descriptorWriteCount}$ valid $\text{VkWriteDescriptorSet}$ structures.

- VUID-vkUpdateDescriptorSets-pDescriptorCopies-parameter
  If $\text{descriptorCopyCount}$ is not $0$, $p_{\text{Descriptor Copies}}$ must be a valid pointer to an array of $\text{descriptorCopyCount}$ valid $\text{VkCopyDescriptorSet}$ structures.

The $\text{VkWriteDescriptorSet}$ structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkWriteDescriptorSet {
    VkStructureType sType;
    const void* pNext;
} VkWriteDescriptorSet;
```
VkDescriptorSet dstSet;
uint32_t dstBinding;
uint32_t dstArrayElement;
uint32_t descriptorCount;
VkDescriptorType descriptorType;
const VkDescriptorImageInfo* pImageInfo;
const VkDescriptorBufferInfo* pBufferInfo;
const VkBufferView* pTexelBufferView;
}

VkWriteDescriptorSet::

• sType is the type of this structure.

• pNext is NULL or a pointer to a structure extending this structure.

• dstSet is the destination descriptor set to update.

• dstBinding is the descriptor binding within that set.

• dstArrayElement is the starting element in that array. If the descriptor binding identified by dstSet and dstBinding has a descriptor type of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK then dstArrayElement specifies the starting byte offset within the binding.

• descriptorCount is the number of descriptors to update. If the descriptor binding identified by dstSet and dstBinding has a descriptor type of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK, then descriptorCount specifies the number of bytes to update. Otherwise, descriptorCount is one of
  ◦ the number of elements in pImageInfo
  ◦ the number of elements in pBufferInfo
  ◦ the number of elements in pTexelBufferView
  ◦ a value matching the dataSize member of a VkWriteDescriptorSetInlineUniformBlock structure in the pNext chain

• descriptorType is a VkDescriptorType specifying the type of each descriptor in pImageInfo, pBufferInfo, or pTexelBufferView, as described below. It must be the same type as the descriptorType specified in VkDescriptorSetLayoutBinding for dstSet at dstBinding. The type of the descriptor also controls which array the descriptors are taken from.

• pImageInfo is a pointer to an array of VkDescriptorImageInfo structures or is ignored, as described below.

• pBufferInfo is a pointer to an array of VkDescriptorBufferInfo structures or is ignored, as described below.

• pTexelBufferView is a pointer to an array of VkBufferView handles as described in the Buffer Views section or is ignored, as described below.

Only one of pImageInfo, pBufferInfo, or pTexelBufferView members is used according to the descriptor type specified in the descriptorType member of the containing VkWriteDescriptorSet structure, or none of them in case descriptorType is VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK, in which case the source data for the descriptor writes is taken from the VkWriteDescriptorSetInlineUniformBlock structure included in the pNext chain of VkWriteDescriptorSet, as specified below.
If the \texttt{dstBinding} has fewer than \texttt{descriptorCount} array elements remaining starting from \texttt{dstArrayElement}, then the remainder will be used to update the subsequent binding - \texttt{dstBinding}+1 starting at array element zero. If a binding has a \texttt{descriptorCount} of zero, it is skipped. This behavior applies recursively, with the update affecting consecutive bindings as needed to update all \texttt{descriptorCount} descriptors. Consecutive bindings \textbf{must} have identical \texttt{VkDescriptorType}, \texttt{VkShaderStageFlags}, \texttt{VkDescriptorBindingFlagBits}, and immutable samplers references.

\begin{quote}
\textbf{Note}

The same behavior applies to bindings with a descriptor type of \texttt{VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK} where \texttt{descriptorCount} specifies the number of bytes to update while \texttt{dstArrayElement} specifies the starting byte offset, thus in this case if the \texttt{dstBinding} has a smaller byte size than the sum of \texttt{dstArrayElement} and \texttt{descriptorCount}, then the remainder will be used to update the subsequent binding - \texttt{dstBinding}+1 starting at offset zero. This falls out as a special case of the above rule.
\end{quote}

\section*{Valid Usage}

\begin{itemize}
  \item VUID-VkWriteDescriptorSet-dstBinding-00315
    \texttt{dstBinding} \textbf{must} be less than or equal to the maximum value of \texttt{binding} of all \texttt{VkDescriptorSetLayoutBinding} structures specified when \texttt{dstSet}'s descriptor set layout was created
  \item VUID-VkWriteDescriptorSet-dstBinding-00316
    \texttt{dstBinding} \textbf{must} be a binding with a non-zero \texttt{descriptorCount}
  \item VUID-VkWriteDescriptorSet-descriptorCount-00317
    All consecutive bindings updated via a single \texttt{VkWriteDescriptorSet} structure, except those with a \texttt{descriptorCount} of zero, \textbf{must} have identical \texttt{descriptorType} and \texttt{stageFlags}
  \item VUID-VkWriteDescriptorSet-descriptorCount-00318
    All consecutive bindings updated via a single \texttt{VkWriteDescriptorSet} structure, except those with a \texttt{descriptorCount} of zero, \textbf{must} all either use immutable samplers or \textbf{must} all not use immutable samplers
  \item VUID-VkWriteDescriptorSet-descriptorType-00319
    \texttt{descriptorType} \textbf{must} match the type of \texttt{dstBinding} within \texttt{dstSet}
  \item VUID-VkWriteDescriptorSet-dstSet-00320
    \texttt{dstSet} \textbf{must} be a valid \texttt{VkDescriptorSet} handle
  \item VUID-VkWriteDescriptorSet-dstArrayElement-00321
    The sum of \texttt{dstArrayElement} and \texttt{descriptorCount} \textbf{must} be less than or equal to the number of array elements in the descriptor set binding specified by \texttt{dstBinding}, and all applicable consecutive bindings, as described by \texttt{consecutive binding updates}
  \item VUID-VkWriteDescriptorSet-descriptorType-02219
    If \texttt{descriptorType} is \texttt{VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK}, \texttt{dstArrayElement} \textbf{must} be an integer multiple of 4
  \item VUID-VkWriteDescriptorSet-descriptorType-02220
    If \texttt{descriptorType} is \texttt{VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK}, \texttt{descriptorCount} \textbf{must} be an
• VUID-VkWriteDescriptorSet-descriptorType-02994
  If descriptorType is VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, each element of pTexelBufferView must be either a valid VkBufferView handle or VK_NULL_HANDLE

• VUID-VkWriteDescriptorSet-descriptorType-02995
  If descriptorType is VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER and the nullDescriptor feature is not enabled, each element of pTexelBufferView must not be VK_NULL_HANDLE

• VUID-VkWriteDescriptorSet-descriptorType-00324
  If descriptorType is VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, pBufferInfo must be a valid pointer to an array of descriptorCount valid VkDescriptorBufferInfo structures

• VUID-VkWriteDescriptorSet-descriptorType-00325
  If descriptorType is VK_DESCRIPTOR_TYPE_SAMPLER or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, and dstSet was not allocated with a layout that included immutable samplers for dstBinding with descriptorType, the sampler member of each element of pImageInfo must be a valid VkSampler object

• VUID-VkWriteDescriptorSet-descriptorType-02996
  If descriptorType is VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, or VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, the imageView member of each element of pImageInfo must be either a valid VkImageView handle or VK_NULL_HANDLE

• VUID-VkWriteDescriptorSet-descriptorType-02997
  If descriptorType is VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, or VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, and the nullDescriptor feature is not enabled, the imageView member of each element of pImageInfo must not be VK_NULL_HANDLE

• VUID-VkWriteDescriptorSet-descriptorType-07683
  If descriptorType is VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT, the imageView member of each element of pImageInfo must not be VK_NULL_HANDLE

• VUID-VkWriteDescriptorSet-descriptorType-02221
  If descriptorType is VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK, the pNext chain must include a VkWriteDescriptorSetInlineUniformBlock structure whose dataSize member equals descriptorCount

• VUID-VkWriteDescriptorSet-descriptorType-00327
  If descriptorType is VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, the offset member of each element of pBufferInfo must be a multiple of VkPhysicalDeviceLimits::minUniformBufferOffsetAlignment

• VUID-VkWriteDescriptorSet-descriptorType-00328
  If descriptorType is VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, the offset member of each element of
 pBufferInfo must be a multiple of VkPhysicalDeviceLimits::minStorageBufferOffsetAlignment

- VUID-VkWriteDescriptorSet-descriptorType-00329
  If descriptorType is VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
  VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, or
  VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, and the buffer member of any element of
  pBufferInfo is the handle of a non-sparse buffer, then that buffer must be bound
  completely and contiguously to a single VkDeviceMemory object

- VUID-VkWriteDescriptorSet-descriptorType-00330
  If descriptorType is VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or
  VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, the buffer member of each element of
  pBufferInfo must have been created with VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT set

- VUID-VkWriteDescriptorSet-descriptorType-00331
  If descriptorType is VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or
  VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, the buffer member of each element of
  pBufferInfo must have been created with VK_BUFFER_USAGE_STORAGE_BUFFER_BIT set

- VUID-VkWriteDescriptorSet-descriptorType-00332
  If descriptorType is VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or
  VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, the range member of each element of
  pBufferInfo, or the effective range if range is VK_WHOLE_SIZE, must be less than or equal to
  VkPhysicalDeviceLimits::maxUniformBufferRange

- VUID-VkWriteDescriptorSet-descriptorType-00333
  If descriptorType is VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or
  VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, the range member of each element of
  pBufferInfo, or the effective range if range is VK_WHOLE_SIZE, must be less than or equal to
  VkPhysicalDeviceLimits::maxStorageBufferRange

- VUID-VkWriteDescriptorSet-descriptorType-00334
  If descriptorType is VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER, the VkBuffer that each
  element of pTexelBufferView was created from must have been created with
  VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT set

- VUID-VkWriteDescriptorSet-descriptorType-00335
  If descriptorType is VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, the VkBuffer that each
  element of pTexelBufferView was created from must have been created with
  VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT set

- VUID-VkWriteDescriptorSet-descriptorType-00336
  If descriptorType is VK_DESCRIPTOR_TYPE_STORAGE_IMAGE or
  VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT, the imageView member of each element of pImageInfo
  must have been created with the identity swizzle

- VUID-VkWriteDescriptorSet-descriptorType-00337
  If descriptorType is VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE or
  VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, the imageView member of each element of
**pImageInfo** must have been created with **VK_IMAGE_USAGE_SAMPLED_BIT** set

- **VUID-VkWriteDescriptorSet-descriptorType-04149**
  If **descriptorType** is **VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE** the **imageLayout** member of each element of **pImageInfo** must be a member of the list given in **Sampled Image**

- **VUID-VkWriteDescriptorSet-descriptorType-04150**
  If **descriptorType** is **VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER** the **imageLayout** member of each element of **pImageInfo** must be a member of the list given in **Combined Image Sampler**

- **VUID-VkWriteDescriptorSet-descriptorType-04151**
  If **descriptorType** is **VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT** the **imageLayout** member of each element of **pImageInfo** must be a member of the list given in **Input Attachment**

- **VUID-VkWriteDescriptorSet-descriptorType-04152**
  If **descriptorType** is **VK_DESCRIPTOR_TYPE_STORAGE_IMAGE** the **imageLayout** member of each element of **pImageInfo** must be a member of the list given in **Storage Image**

- **VUID-VkWriteDescriptorSet-descriptorType-00338**
  If **descriptorType** is **VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT**, the **imageView** member of each element of **pImageInfo** must have been created with **VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT** set

- **VUID-VkWriteDescriptorSet-descriptorType-00339**
  If **descriptorType** is **VK_DESCRIPTOR_TYPE_STORAGE_IMAGE**, the **imageView** member of each element of **pImageInfo** must have been created with **VK_IMAGE_USAGE_STORAGE_BIT** set

- **VUID-VkWriteDescriptorSet-descriptorType-02752**
  If **descriptorType** is **VK_DESCRIPTOR_TYPE_SAMPLER**, then **dstSet** must not have been allocated with a layout that included immutable samplers for **dstBinding**

---

**Valid Usage (Implicit)**

- **VUID-VkWriteDescriptorSet-sType-sType**
  **sType** must be **VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET**

- **VUID-VkWriteDescriptorSet-pNext-pNext**
  **pNext** must be **NULL** or a pointer to a valid instance of **VkWriteDescriptorSetInlineUniformBlock**

- **VUID-VkWriteDescriptorSet-sType-unique**
  The **sType** value of each struct in the **pNext** chain must be unique

- **VUID-VkWriteDescriptorSet-descriptorType-parameter**
  **descriptorType** must be a valid **VkDescriptorType** value

- **VUID-VkWriteDescriptorSet-descriptorCount-arraylength**
  **descriptorCount** must be greater than 0

- **VUID-VkWriteDescriptorSet-commonparent**
  Both of **dstSet**, and the elements of **pTexelBufferView** that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same **VkDevice**
The type of descriptors in a descriptor set is specified by `VkWriteDescriptorSet::descriptorType`, which **must** be one of the values:

```c
// Provided by VK_VERSION_1_0
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,
    // Provided by VK_VERSION_1_3
    VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK = 1000138000,
} VkDescriptorType;
```

- **VK_DESCRIPTOR_TYPE_SAMPLER** specifies a *sampler* descriptor.
- **VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER** specifies a *combined image sampler* descriptor.
- **VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE** specifies a *sampled image* descriptor.
- **VK_DESCRIPTOR_TYPE_STORAGE_IMAGE** specifies a *storage image* descriptor.
- **VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER** specifies a *uniform texel buffer* descriptor.
- **VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER** specifies a *storage texel buffer* descriptor.
- **VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER** specifies a *uniform buffer* descriptor.
- **VK_DESCRIPTOR_TYPE_STORAGE_BUFFER** specifies a *storage buffer* descriptor.
- **VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC** specifies a *dynamic uniform buffer* descriptor.
- **VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC** specifies a *dynamic storage buffer* descriptor.
- **VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT** specifies an *input attachment* descriptor.
- **VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK** specifies an *inline uniform block*.

When a descriptor set is updated via elements of `VkWriteDescriptorSet`, members of `pImageInfo`, `pBufferInfo` and `pTexelBufferView` are only accessed by the implementation when they correspond to descriptor type being defined - otherwise they are ignored. The members accessed are as follows for each descriptor type:

- For **VK_DESCRIPTOR_TYPE_SAMPLER**, only the `sampler` member of each element of `VkWriteDescriptorSet::pImageInfo` is accessed.
- For **VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE**, **VK_DESCRIPTOR_TYPE_STORAGE_IMAGE**, or **VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT**, only the `imageView` and `imageLayout` members of each element of `VkWriteDescriptorSet::pImageInfo` are accessed.
• For `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, all members of each element of `VkWriteDescriptorSet::pImageInfo` are accessed.

• For `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`, `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER`, `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`, or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`, all members of each element of `VkWriteDescriptorSet::pBufferInfo` are accessed.

• For `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER`, each element of `VkWriteDescriptorSet::pTexelBufferView` is accessed.

When updating descriptors with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK`, none of the `pImageInfo`, `pBufferInfo`, or `pTexelBufferView` members are accessed, instead the source data of the descriptor update operation is taken from the `VkWriteDescriptorSetInlineUniformBlock` structure in the `pNext` chain of `VkWriteDescriptorSet`.

The `VkDescriptorBufferInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDescriptorBufferInfo {
    VkBuffer buffer;
    VkDeviceSize offset;
    VkDeviceSize range;
} VkDescriptorBufferInfo;
```

• `buffer` is the buffer resource.

• `offset` is the offset in bytes from the start of `buffer`. Access to buffer memory via this descriptor uses addressing that is relative to this starting offset.

• `range` is the size in bytes that is used for this descriptor update, or `VK_WHOLE_SIZE` to use the range from `offset` to the end of the buffer.

When setting `range` to `VK_WHOLE_SIZE`, the effective range must not be larger than the maximum range for the descriptor type (`maxUniformBufferRange` or `maxStorageBufferRange`). This means that `VK_WHOLE_SIZE` is not typically useful in the common case where uniform buffer descriptors are suballocated from a buffer that is much larger than `maxUniformBufferRange`.

For `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` and `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` descriptor types, `offset` is the base offset from which the dynamic offset is applied and `range` is the static size used for all dynamic offsets.

When `range` is `VK_WHOLE_SIZE` the effective range is calculated at `vkUpdateDescriptorSets` is by taking the size of `buffer` minus the `offset`.

Valid Usage

- VUID-VkDescriptorBufferInfo-offset-00340
  `offset` must be less than the size of `buffer`
If `range` is not equal to `VK_WHOLE_SIZE`, `range` must be greater than 0.

If `range` is not equal to `VK_WHOLE_SIZE`, `range` must be less than or equal to the size of `buffer` minus `offset`.

If the `nullDescriptor` feature is not enabled, `buffer` must not be `VK_NULL_HANDLE`.

### Valid Usage (Implicit)

If `buffer` is not `VK_NULL_HANDLE`, `buffer` must be a valid `VkBuffer` handle.

The `VkDescriptorImageInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDescriptorImageInfo {
    VkSampler sampler;
    VkImageView imageView;
    VkImageLayout imageLayout;
} VkDescriptorImageInfo;
```

- `sampler` is a sampler handle, and is used in descriptor updates for types `VK_DESCRIPTOR_TYPE_SAMPLER` and `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` if the binding being updated does not use immutable samplers.

- `imageView` is an image view handle, and is used in descriptor updates for types `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, and `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`.

- `imageLayout` is the layout that the image subresources accessible from `imageView` will be in at the time this descriptor is accessed. `imageLayout` is used in descriptor updates for types `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, and `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`.

Members of `VkDescriptorImageInfo` that are not used in an update (as described above) are ignored.

### Valid Usage

- `imageView` must not be 2D image view created from a 3D image.
- `imageView` must not be a 2D array image view created from a 3D image.
- `imageView` must not be a 2D array image view created from a 3D image.

If `imageView` is created from a depth/stencil image, the `aspectMask` used to create the
**Valid Usage (Implicit)**

- VUID-VkDescriptorImageInfo-commonparent
  Both of `imageView` and `sampler` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`

If the `descriptorType` member of `VkWriteDescriptorSet` is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK` then the data to write to the descriptor set is specified through a `VkWriteDescriptorSetInlineUniformBlock` structure included in the `pNext` chain of `VkWriteDescriptorSet`.

The `VkWriteDescriptorSetInlineUniformBlock` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkWriteDescriptorSetInlineUniformBlock {
    VkStructureType sType;
    const void* pNext;
    uint32_t dataSize;
    const void* pData;
} VkWriteDescriptorSetInlineUniformBlock;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `dataSize` is the number of bytes of inline uniform block data pointed to by `pData`.
- `pData` is a pointer to `dataSize` number of bytes of data to write to the inline uniform block.

**Valid Usage**

- VUID-VkWriteDescriptorSetInlineUniformBlock-dataSize-02222
  `dataSize` must be an integer multiple of 4
Valid Usage (Implicit)

- VUID-VkWriteDescriptorSetInlineUniformBlock-sType-sType
  - sType must be VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_INLINE_UNIFORM_BLOCK
- VUID-VkWriteDescriptorSetInlineUniformBlock-pData-parameter
  - pData must be a valid pointer to an array of dataSize bytes
- VUID-VkWriteDescriptorSetInlineUniformBlock-dataSize-arraylength
  - dataSize must be greater than 0

The VkCopyDescriptorSet structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkCopyDescriptorSet {
    VkStructureType sType;
    const void* pNext;
    VkDescriptorSet srcSet;
    uint32_t srcBinding;
    uint32_t srcArrayElement;
    VkDescriptorSet dstSet;
    uint32_t dstBinding;
    uint32_t dstArrayElement;
    uint32_t descriptorCount;
} VkCopyDescriptorSet;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **srcSet**, **srcBinding**, and **srcArrayElement** are the source set, binding, and array element, respectively. If the descriptor binding identified by **srcSet** and **srcBinding** has a descriptor type of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK then **srcArrayElement** specifies the starting byte offset within the binding to copy from.
- **dstSet**, **dstBinding**, and **dstArrayElement** are the destination set, binding, and array element, respectively. If the descriptor binding identified by **dstSet** and **dstBinding** has a descriptor type of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK then **dstArrayElement** specifies the starting byte offset within the binding to copy to.
- **descriptorCount** is the number of descriptors to copy from the source to destination. If **descriptorCount** is greater than the number of remaining array elements in the source or destination binding, those affect consecutive bindings in a manner similar to VkWriteDescriptorSet above. If the descriptor binding identified by **srcSet** and **srcBinding** has a descriptor type of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK then **descriptorCount** specifies the number of bytes to copy and the remaining array elements in the source or destination binding refer to the remaining number of bytes in those.
Valid Usage

• VUID-VkCopyDescriptorSet-srcBinding-00345
  srcBinding must be a valid binding within srcSet

• VUID-VkCopyDescriptorSet-srcArrayElement-00346
  The sum of srcArrayElement and descriptorCount must be less than or equal to the number of array elements in the descriptor set binding specified by srcBinding, and all applicable consecutive bindings, as described by consecutive binding updates

• VUID-VkCopyDescriptorSet-dstBinding-00347
  dstBinding must be a valid binding within dstSet

• VUID-VkCopyDescriptorSet-dstArrayElement-00348
  The sum of dstArrayElement and descriptorCount must be less than or equal to the number of array elements in the descriptor set binding specified by dstBinding, and all applicable consecutive bindings, as described by consecutive binding updates

• VUID-VkCopyDescriptorSet-dstBinding-02632
  The type of dstBinding within dstSet must be equal to the type of srcBinding within srcSet

• VUID-VkCopyDescriptorSet-srcSet-00349
  If srcSet is equal to dstSet, then the source and destination ranges of descriptors must not overlap, where the ranges may include array elements from consecutive bindings as described by consecutive binding updates

• VUID-VkCopyDescriptorSet-srcBinding-02223
  If the descriptor type of the descriptor set binding specified by srcBinding is VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK, srcArrayElement must be an integer multiple of 4

• VUID-VkCopyDescriptorSet-dstBinding-02224
  If the descriptor type of the descriptor set binding specified by dstBinding is VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK, dstArrayElement must be an integer multiple of 4

• VUID-VkCopyDescriptorSet-srcBinding-02225
  If the descriptor type of the descriptor set binding specified by either srcBinding or dstBinding is VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK, descriptorCount must be an integer multiple of 4

• VUID-VkCopyDescriptorSet-srcSet-01918
  If srcSet’s layout was created with the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT flag set, then dstSet’s layout must also have been created with the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT flag set

• VUID-VkCopyDescriptorSet-srcSet-04886
  If srcSet’s layout was created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT flag set, then dstSet’s layout must also have been created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT flag set

• VUID-VkCopyDescriptorSet-srcSet-01920
If the descriptor pool from which srcSet was allocated was created with the VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT flag set, then the descriptor pool from which dstSet was allocated must also have been created with the VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT flag set

- VUID-VkCopyDescriptorSet-srcSet-04888
  If the descriptor pool from which srcSet was allocated was created without the VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT flag set, then the descriptor pool from which dstSet was allocated must also have been created without the VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT flag set

- VUID-VkCopyDescriptorSet-dstBinding-02753
  If the descriptor type of the descriptor set binding specified by dstBinding is VK_DESCRIPTOR_TYPE_SAMPLER, then dstSet must not have been allocated with a layout that included immutable samplers for dstBinding

**Valid Usage (Implicit)**

- VUID-VkCopyDescriptorSet-sType-sType
  sType must be VK_STRUCTURE_TYPE_COPY_DESCRIPTOR_SET

- VUID-VkCopyDescriptorSet-pNext-pNext
  pNext must be NULL

- VUID-VkCopyDescriptorSet-srcSet-parameter
  srcSet must be a valid VkDescriptorSet handle

- VUID-VkCopyDescriptorSet-dstSet-parameter
  dstSet must be a valid VkDescriptorSet handle

- VUID-VkCopyDescriptorSet-commonparent
  Both of dstSet, and srcSet must have been created, allocated, or retrieved from the same VkDevice

### 14.2.5. Descriptor Update Templates

A descriptor update template specifies a mapping from descriptor update information in host memory to descriptors in a descriptor set. It is designed to avoid passing redundant information to the driver when frequently updating the same set of descriptors in descriptor sets.

Descriptor update template objects are represented by VkDescriptorUpdateTemplate handles:

```c
// Provided by VK_VERSION_1_1
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorUpdateTemplate)
```

### 14.2.6. Descriptor Set Updates with Templates

Updating a large VkDescriptorSet array can be an expensive operation since an application must specify one VkWriteDescriptorSet structure for each descriptor or descriptor array to update, each
of which re-specifies the same state when updating the same descriptor in multiple descriptor sets. For cases when an application wishes to update the same set of descriptors in multiple descriptor sets allocated using the same `VkDescriptorSetLayout`, `vkUpdateDescriptorSetWithTemplate` can be used as a replacement for `vkUpdateDescriptorSets`.

`VkDescriptorUpdateTemplate` allows implementations to convert a set of descriptor update operations on a single descriptor set to an internal format that, in conjunction with `vkUpdateDescriptorSetWithTemplate`, can be more efficient compared to calling `vkUpdateDescriptorSets`. The descriptors themselves are not specified in the `VkDescriptorUpdateTemplate`, rather, offsets into an application provided pointer to host memory are specified, which are combined with a pointer passed to `vkUpdateDescriptorSetWithTemplate`. This allows large batches of updates to be executed without having to convert application data structures into a strictly-defined Vulkan data structure.

To create a descriptor update template, call:

```c
// Provided by VK_VERSION_1_1
VkResult vkCreateDescriptorUpdateTemplate(
    VkDevice device,
    const VkDescriptorUpdateTemplateCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkDescriptorUpdateTemplate* pDescriptorUpdateTemplate);
```

- `device` is the logical device that creates the descriptor update template.
- `pCreateInfo` is a pointer to a `VkDescriptorUpdateTemplateCreateInfo` structure specifying the set of descriptors to update with a single call to `vkUpdateDescriptorSetWithTemplate`.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pDescriptorUpdateTemplate` is a pointer to a `VkDescriptorUpdateTemplate` handle in which the resulting descriptor update template object is returned.

**Valid Usage (Implicit)**

- VUID-vkCreateDescriptorUpdateTemplate-device-parameter `device` must be a valid `VkDevice` handle
- VUID-vkCreateDescriptorUpdateTemplate-pCreateInfo-parameter `pCreateInfo` must be a valid pointer to a valid `VkDescriptorUpdateTemplateCreateInfo` structure
- VUID-vkCreateDescriptorUpdateTemplate-pAllocator-parameter If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- VUID-vkCreateDescriptorUpdateTemplate-pDescriptorUpdateTemplate-parameter `pDescriptorUpdateTemplate` must be a valid pointer to a `VkDescriptorUpdateTemplate` handle
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkDescriptorUpdateTemplateCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkDescriptorUpdateTemplateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkDescriptorUpdateTemplateCreateFlags flags;
    uint32_t descriptorUpdateEntryCount;
    const VkDescriptorUpdateTemplateEntry* pDescriptorUpdateEntries;
    VkDescriptorUpdateTemplateType templateType;
    VkDescriptorSetLayout descriptorSetLayout;
    VkPipelineBindPoint pipelineBindPoint;
    VkPipelineLayout pipelineLayout;
    uint32_t set;
} VkDescriptorUpdateTemplateCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `descriptorUpdateEntryCount` is the number of elements in the `pDescriptorUpdateEntries` array.
- `pDescriptorUpdateEntries` is a pointer to an array of `VkDescriptorUpdateTemplateEntry` structures describing the descriptors to be updated by the descriptor update template.
- `templateType` specifies the type of the descriptor update template. If set to `VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET` it can only be used to update descriptor sets with a fixed `descriptorSetLayout`.
- `descriptorSetLayout` is the descriptor set layout used to build the descriptor update template. All descriptor sets which are going to be updated through the newly created descriptor update template must be created with a layout that matches (is the same as, or defined identically to) this layout. This parameter is ignored if `templateType` is not `VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET`.
- `pipelineBindPoint` is reserved for future use and is ignored
- `pipelineLayout` is reserved for future use and is ignored
- `set` is reserved for future use and is ignored
Valid Usage

- VUID-VkDescriptorUpdateTemplateCreateInfo-templateType-00350
  If `templateType` is `VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET`, `descriptorSetLayout` must be a valid `VkDescriptorSetLayout` handle.

Valid Usage (Implicit)

- VUID-VkDescriptorUpdateTemplateCreateInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO`.

- VUID-VkDescriptorUpdateTemplateCreateInfo-pNext-pNext
  `pNext` must be `NULL`.

- VUID-VkDescriptorUpdateTemplateCreateInfo-flags-zerobitmask
  `flags` must be `0`.

- VUID-VkDescriptorUpdateTemplateCreateInfo-pDescriptorUpdateEntries-parameter
  `pDescriptorUpdateEntries` must be a valid pointer to an array of `descriptorUpdateEntryCount` valid `VkDescriptorUpdateTemplateEntry` structures.

- VUID-VkDescriptorUpdateTemplateCreateInfo-templateType-parameter
  `templateType` must be a valid `VkDescriptorUpdateTemplateType` value.

- VUID-VkDescriptorUpdateTemplateCreateInfo-descriptorUpdateEntryCount-arraylength
  `descriptorUpdateEntryCount` must be greater than `0`.

- VUID-VkDescriptorUpdateTemplateCreateInfo-commonparent
  Both of `descriptorSetLayout` and `pipelineLayout` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`.

// Provided by VK_VERSION_1_1
typedef VkFlags VkDescriptorUpdateTemplateCreateFlags;

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

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

// Provided by VK_VERSION_1_1
typedef enum VkDescriptorUpdateTemplateType {
  VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET = 0,
} VkDescriptorUpdateTemplateType;

- `VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET` specifies that the descriptor update template will be used for descriptor set updates only.
The **VkDescriptorUpdateTemplateEntry** structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkDescriptorUpdateTemplateEntry {
    uint32_t dstBinding;
    uint32_t dstArrayElement;
    uint32_t descriptorCount;
    VkDescriptorType descriptorType;
    size_t offset;
    size_t stride;
} VkDescriptorUpdateTemplateEntry;
```

- **dstBinding** is the descriptor binding to update when using this descriptor update template.
- **dstArrayElement** is the starting element in the array belonging to **dstBinding**. If the descriptor binding identified by **dstBinding** has a descriptor type of **VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK** then **dstArrayElement** specifies the starting byte offset to update.
- **descriptorCount** is the number of descriptors to update. If **descriptorCount** is greater than the number of remaining array elements in the destination binding, those affect consecutive bindings in a manner similar to **VkWriteDescriptorSet** above. If the descriptor binding identified by **dstBinding** has a descriptor type of **VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK** then **descriptorCount** specifies the number of bytes to update and the remaining array elements in the destination binding refer to the remaining number of bytes in it.
- **descriptorType** is a **VkDescriptorType** specifying the type of the descriptor.
- **offset** is the offset in bytes of the first binding in the raw data structure.
- **stride** is the stride in bytes between two consecutive array elements of the descriptor update information in the raw data structure. The actual pointer **ptr** for each array element **j** of update entry **i** is computed using the following formula:

```
const char *ptr = (const char *)pData + pDescriptorUpdateEntries[i].offset + j * pDescriptorUpdateEntries[i].stride
```

The stride is useful in case the bindings are stored in structs along with other data. If **descriptorType** is **VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK** then the value of **stride** is ignored and the stride is assumed to be 1, i.e. the descriptor update information for them is always specified as a contiguous range.

**Valid Usage**

- **VUID-VkDescriptorUpdateTemplateEntry-dstBinding-00354**
  - **dstBinding** must be a valid binding in the descriptor set layout implicitly specified when using a descriptor update template to update descriptors
- **VUID-VkDescriptorUpdateTemplateEntry-dstArrayElement-00355**
  - **dstArrayElement** and **descriptorCount** must be less than or equal to the number of array
elements in the descriptor set binding implicitly specified when using a descriptor update template to update descriptors, and all applicable consecutive bindings, as described by consecutive binding updates

- VUID-VkDescriptorUpdateTemplateEntry-descriptor-02226
  If descriptor type is VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK, dstArrayElement must be an integer multiple of 4

- VUID-VkDescriptorUpdateTemplateEntry-descriptor-02227
  If descriptor type is VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK, descriptorCount must be an integer multiple of 4

Valid Usage (Implicit)

- VUID-VkDescriptorUpdateTemplateEntry-descriptorType-parameter
  descriptorType must be a valid VkDescriptorType value

To destroy a descriptor update template, call:

```c
// Provided by VK_VERSION_1_1
void vkDestroyDescriptorUpdateTemplate(
    VkDevice device,
    VkDescriptorUpdateTemplate descriptorUpdateTemplate,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that has been used to create the descriptor update template
- `descriptorUpdateTemplate` is the descriptor update template to destroy.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- VUID-vkDestroyDescriptorUpdateTemplate-descriptorSetLayout-00356
  If VkAllocationCallbacks were provided when descriptorUpdateTemplate was created, a compatible set of callbacks must be provided here

- VUID-vkDestroyDescriptorUpdateTemplate-descriptorSetLayout-00357
  If no VkAllocationCallbacks were provided when descriptorUpdateTemplate was created, pAllocator must be NULL

Valid Usage (Implicit)

- VUID-vkDestroyDescriptorUpdateTemplate-device-parameter
  device must be a valid VkDevice handle

- VUID-vkDestroyDescriptorUpdateTemplate-descriptorUpdateTemplate-parameter
  If descriptorUpdateTemplate is not VK_NULL_HANDLE, descriptorUpdateTemplate must be a
valid `VkDescriptorUpdateTemplate` handle

- VUID-vkDestroyDescriptorUpdateTemplate-pAllocator-parameter
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- VUID-vkDestroyDescriptorUpdateTemplate-descriptorUpdateTemplate-parent
  If `descriptorUpdateTemplate` is a valid handle, it must have been created, allocated, or retrieved from `device`

### Host Synchronization

- Host access to `descriptorUpdateTemplate` must be externally synchronized

Once a `VkDescriptorUpdateTemplate` has been created, descriptor sets can be updated by calling:

```c
// Provided by VK_VERSION_1_1
void vkUpdateDescriptorSetWithTemplate(
  VkDevice device,
  VkDescriptorSet descriptorSet,
  VkDescriptorUpdateTemplate descriptorUpdateTemplate,
  const void* pData);
```

- `device` is the logical device that updates the descriptor set.
- `descriptorSet` is the descriptor set to update
- `descriptorUpdateTemplate` is a `VkDescriptorUpdateTemplate` object specifying the update mapping between `pData` and the descriptor set to update.
- `pData` is a pointer to memory containing one or more `VkDescriptorImageInfo`, `VkDescriptorBufferInfo`, or `VkBufferView` structures used to write the descriptors.

### Valid Usage

- VUID-vkUpdateDescriptorSetWithTemplate-pData-01685
  `pData` must be a valid pointer to a memory containing one or more valid instances of `VkDescriptorImageInfo`, `VkDescriptorBufferInfo`, or `VkBufferView` in a layout defined by `descriptorUpdateTemplate` when it was created with `vkCreateDescriptorUpdateTemplate`

- VUID-vkUpdateDescriptorSetWithTemplate-descriptorSet-06995
  Host access to `descriptorSet` must be externally synchronized unless explicitly denoted otherwise for specific flags

### Valid Usage (Implicit)

- VUID-vkUpdateDescriptorSetWithTemplate-device-parameter
  `device` must be a valid `VkDevice` handle
- **VUID-vkUpdateDescriptorSetWithTemplate-descriptorSet-parameter**
  descriptorSet must be a valid VkDescriptorSet handle

- **VUID-vkUpdateDescriptorSetWithTemplate-descriptorUpdateTemplate-parameter**
  descriptorUpdateTemplate must be a valid VkDescriptorUpdateTemplate handle

- **VUID-vkUpdateDescriptorSetWithTemplate-descriptorUpdateTemplate-parent**
  descriptorUpdateTemplate must have been created, allocated, or retrieved from device

**API example**

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

struct AppDataStructure {
    VkDescriptorImageInfo imageInfo;          // a single image info
    VkDescriptorBufferInfo bufferInfoArray[3]; // 3 buffer infos in an array
    AppBufferView bufferView[2];              // An application defined structure
    // ... some more application related data
};

const VkDescriptorUpdateTemplateEntry descriptorUpdateTemplateEntries[] = {
    // binding to a single image descriptor
    { 0, 0, 1, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, offsetof(AppDataStructure, imageInfo), 0 },
    // binding to an array of buffer descriptors
    { 1, 0, 0, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, offsetof(AppDataStructure, bufferInfoArray), sizeof(VkDescriptorBufferInfo) },
    // binding to an array of buffer views
    { 0, 0, 0, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, offsetof(AppDataStructure, bufferInfoArray), sizeof(VkDescriptorBufferInfo) }
};
```
2, // binding
0, // dstArrayElement
2, // descriptorCount
VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, // descriptorType
offsetof(AppDataStructure, bufferView) +
offsetof(AppBufferView, bufferView), // offset
sizeof(AppBufferView), // stride, bufferViews do not have to be compact
};

// create a descriptor update template for descriptor set updates
const VkDescriptorUpdateTemplateCreateInfo createInfo = 
{
    VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO, // sType
    NULL, // pNext
    0, // flags
    descriptorUpdateEntryCount,
    descriptorUpdateTemplateEntries, // templateType
    VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET, // descriptorSetLayout
    myLayout, // pipelineBindPoint,
    0, // ignored by given templateType
    0, // ignored by given templateType
    0, // set, ignored by given templateType
};

VkDescriptorUpdateTemplate myDescriptorUpdateTemplate;
myResult = vkCreateDescriptorUpdateTemplate(
    myDevice,
    &createInfo,
    NULL,
    &myDescriptorUpdateTemplate);

AppDataStructure appData;

// fill appData here or cache it in your engine
vkUpdateDescriptorSetWithTemplate(myDevice, myDescriptorSet, myDescriptorUpdateTemplate, &appData);

### 14.2.7. Descriptor Set Binding

To bind one or more descriptor sets to a command buffer, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdBindDescriptorSets(
```
void vkCmdBindDescriptorSets(
    VkCommandBuffer commandBuffer,
    VkPipelineBindPoint pipelineBindPoint,
    VkPipelineLayout layout,
    uint32_t firstSet,
    uint32_t descriptorSetCount,
    const VkDescriptorSet* pDescriptorSets,
    dynamicOffsetCount,
    const uint32_t* pDynamicOffsets);

- `commandBuffer` is the command buffer that the descriptor sets will be bound to.
- `pipelineBindPoint` is a `VkPipelineBindPoint` indicating the type of the pipeline that will use the descriptors. There is a separate set of bind points for each pipeline type, so binding one does not disturb the others.
- `layout` is a `VkPipelineLayout` object used to program the bindings.
- `firstSet` is the set number of the first descriptor set to be bound.
- `descriptorSetCount` is the number of elements in the `pDescriptorSets` array.
- `pDescriptorSets` is a pointer to an array of handles to `VkDescriptorSet` objects describing the descriptor sets to bind to.
- `dynamicOffsetCount` is the number of dynamic offsets in the `pDynamicOffsets` array.
- `pDynamicOffsets` is a pointer to an array of `uint32_t` values specifying dynamic offsets.

`vkCmdBindDescriptorSets` binds descriptor sets `pDescriptorSets[0..descriptorSetCount-1]` to set numbers `[firstSet..firstSet+descriptorSetCount-1]` for subsequent bound pipeline commands set by `pipelineBindPoint`. Any bindings that were previously applied via these sets are no longer valid.

Once bound, a descriptor set affects rendering of subsequent commands that interact with the given pipeline type in the command buffer until either a different set is bound to the same set number, or the set is disturbed as described in `Pipeline Layout Compatibility`.

A compatible descriptor set **must** be bound for all set numbers that any shaders in a pipeline access, at the time that a drawing or dispatching command is recorded to execute using that pipeline. However, if none of the shaders in a pipeline statically use any bindings with a particular set number, then no descriptor set need be bound for that set number, even if the pipeline layout includes a non-trivial descriptor set layout for that set number.

When consuming a descriptor, a descriptor is considered valid if the descriptor is not undefined as described by `descriptor set allocation`. A descriptor that was disturbed by `Pipeline Layout Compatibility`, or was never bound by `vkCmdBindDescriptorSets` is not considered valid. If a pipeline accesses a descriptor either statically or dynamically depending on the `VkDescriptorBindingFlagBits`, the consuming descriptor type in the pipeline **must** match the `VkDescriptorType` in `VkDescriptorSetLayoutCreateInfo` for the descriptor to be considered valid.

---

**Note**

Further validation may be carried out beyond validation for descriptor types, e.g. `Texel Input Validation`.  

---

636
If any of the sets being bound include dynamic uniform or storage buffers, then `pDynamicOffsets` includes one element for each array element in each dynamic descriptor type binding in each set. Values are taken from `pDynamicOffsets` in an order such that all entries for set N come before set N+1; within a set, entries are ordered by the binding numbers in the descriptor set layouts; and within a binding array, elements are in order. `dynamicOffsetCount` must equal the total number of dynamic descriptors in the sets being bound.

The effective offset used for dynamic uniform and storage buffer bindings is the sum of the relative offset taken from `pDynamicOffsets`, and the base address of the buffer plus base offset in the descriptor set. The range of the dynamic uniform and storage buffer bindings is the buffer range as specified in the descriptor set.

Each of the `pDescriptorSets` must be compatible with the pipeline layout specified by `layout`. The layout used to program the bindings must also be compatible with the pipeline used in subsequent bound pipeline commands with that pipeline type, as defined in the Pipeline Layout Compatibility section.

The descriptor set contents bound by a call to `vkCmdBindDescriptorSets` may be consumed at the following times:

- For descriptor bindings created with the `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` bit set, the contents may be consumed when the command buffer is submitted to a queue, or during shader execution of the resulting draws and dispatches, or any time in between. Otherwise,
- during host execution of the command, or during shader execution of the resulting draws and dispatches, or any time in between.

Thus, the contents of a descriptor set binding must not be altered (overwritten by an update command, or freed) between the first point in time that it may be consumed, and when the command completes executing on the queue.

The contents of `pDynamicOffsets` are consumed immediately during execution of `vkCmdBindDescriptorSets`. Once all pending uses have completed, it is legal to update and reuse a descriptor set.

### Valid Usage

- VUID-vkCmdBindDescriptorSets-pDescriptorSets-00358
  Each element of `pDescriptorSets` must have been allocated with a `VkDescriptorSetLayout` that matches (is the same as, or identically defined as) the `VkDescriptorSetLayout` at set n in `layout`, where n is the sum of `firstSet` and the index into `pDescriptorSets`

- VUID-vkCmdBindDescriptorSets-dynamicOffsetCount-00359
  `dynamicOffsetCount` must be equal to the total number of dynamic descriptors in `pDescriptorSets`

- VUID-vkCmdBindDescriptorSets-firstSet-00360
  The sum of `firstSet` and `descriptorSetCount` must be less than or equal to `VkPipelineLayoutCreateInfo::setLayoutCount` provided when `layout` was created

- VUID-vkCmdBindDescriptorSets-pipelineBindPoint-00361
pipelineBindPoint must be supported by the commandBuffer's parent VkCommandPool's queue family

- VUID-vkCmdBindDescriptorSets-pDynamicOffsets-01971
  Each element of pDynamicOffsets which corresponds to a descriptor binding with type VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC must be a multiple of VkPhysicalDeviceLimits::minUniformBufferOffsetAlignment

- VUID-vkCmdBindDescriptorSets-pDynamicOffsets-01972
  Each element of pDynamicOffsets which corresponds to a descriptor binding with type VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC must be a multiple of VkPhysicalDeviceLimits::minStorageBufferOffsetAlignment

- VUID-vkCmdBindDescriptorSets-pDescriptorSets-01979
  For each dynamic uniform or storage buffer binding in pDescriptorSets, the sum of the effective offset and the range of the binding must be less than or equal to the size of the buffer

- VUID-vkCmdBindDescriptorSets-pDescriptorSets-06715
  For each dynamic uniform or storage buffer binding in pDescriptorSets, if the range was set with VK_WHOLE_SIZE then pDynamicOffsets which corresponds to the descriptor binding must be 0

- VUID-vkCmdBindDescriptorSets-pDescriptorSets-06563
  Each element of pDescriptorSets must be a valid VkDescriptorSet

**Valid Usage (Implicit)**

- VUID-vkCmdBindDescriptorSets-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdBindDescriptorSets-pipelineBindPoint-parameter
  pipelineBindPoint must be a valid VkPipelineBindPoint value

- VUID-vkCmdBindDescriptorSets-layout-parameter
  layout must be a valid VkPipelineLayout handle

- VUID-vkCmdBindDescriptorSets-pDescriptorSets-parameter
  pDescriptorSets must be a valid pointer to an array of descriptorSetCount valid or VK_NULL_HANDLE VkDescriptorSet handles

- VUID-vkCmdBindDescriptorSets-pDynamicOffsets-parameter
  If dynamicOffsetCount is not 0, pDynamicOffsets must be a valid pointer to an array of dynamicOffsetCount uint32_t values

- VUID-vkCmdBindDescriptorSets-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdBindDescriptorSets-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations

- VUID-vkCmdBindDescriptorSets-descriptorSetCount-arraylength
  descriptorSetCount must be greater than 0
Each of `commandBuffer`, `layout`, and the elements of `pDescriptorSets` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`.

### Host Synchronization

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

### 14.2.8. Push Constant Updates

As described above in section *Pipeline Layouts*, the pipeline layout defines shader push constants which are updated via Vulkan commands rather than via writes to memory or copy commands.

**Note**

Push constants represent a high speed path to modify constant data in pipelines that is expected to outperform memory-backed resource updates.

To update push constants, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdPushConstants(
    VkCommandBuffer commandBuffer,
    VkPipelineLayout layout,
    VkShaderStageFlags stageFlags,
    uint32_t offset,
    uint32_t size,
    const void* pValues);
```

- **commandBuffer** is the command buffer in which the push constant update will be recorded.
- **layout** is the pipeline layout used to program the push constant updates.
- **stageFlags** is a bitmask of `VkShaderStageFlagBits` specifying the shader stages that will use the push constants in the updated range.
• **offset** is the start offset of the push constant range to update, in units of bytes.
• **size** is the size of the push constant range to update, in units of bytes.
• **pValues** is a pointer to an array of **size** bytes containing the new push constant values.

When a command buffer begins recording, all push constant values are undefined. Reads of undefined push constant values by the executing shader return undefined values.

Push constant values can be updated incrementally, causing shader stages in **stageFlags** to read the new data from **pValues** for push constants modified by this command, while still reading the previous data for push constants not modified by this command. When a bound pipeline command is issued, the bound pipeline’s layout must be compatible with the layouts used to set the values of all push constants in the pipeline layout’s push constant ranges, as described in Pipeline Layout Compatibility. Binding a pipeline with a layout that is not compatible with the push constant layout does not disturb the push constant values.

**Note**

As **stageFlags** needs to include all flags the relevant push constant ranges were created with, any flags that are not supported by the queue family that the **VkCommandPool** used to allocate **commandBuffer** was created on are ignored.

---

**Valid Usage**

- VUID-vkCmdPushConstants-offset-01795
  For each byte in the range specified by **offset** and **size** and for each shader stage in **stageFlags**, there must be a push constant range in **layout** that includes that byte and that stage.

- VUID-vkCmdPushConstants-offset-01796
  For each byte in the range specified by **offset** and **size** and for each push constant range that overlaps that byte, **stageFlags** must include all stages in that push constant range’s **VkPushConstantRange::stageFlags**.

- VUID-vkCmdPushConstants-offset-00368
  **offset** must be a multiple of 4.

- VUID-vkCmdPushConstants-size-00369
  **size** must be a multiple of 4.

- VUID-vkCmdPushConstants-offset-00370
  **offset** must be less than **VkPhysicalDeviceLimits::maxPushConstantsSize**.

- VUID-vkCmdPushConstants-size-00371
  **size** must be less than or equal to **VkPhysicalDeviceLimits::maxPushConstantsSize** minus **offset**.

---

**Valid Usage (Implicit)**

- VUID-vkCmdPushConstants-commandBuffer-parameter
commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdPushConstants-layout-parameter
  layout must be a valid VkPipelineLayout handle

- VUID-vkCmdPushConstants-stageFlags-parameter
  stageFlags must be a valid combination of VkShaderStageFlagBits values

- VUID-vkCmdPushConstants-stageFlags-requiredbitmask
  stageFlags must not be 0

- VUID-vkCmdPushConstants-pValues-parameter
  pValues must be a valid pointer to an array of size bytes

- VUID-vkCmdPushConstants-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdPushConstants-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations

- VUID-vkCmdPushConstants-size-arraylength
  size must be greater than 0

- VUID-vkCmdPushConstants-commonparent
  Both of commandBuffer, and layout must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

14.3. Physical Storage Buffer Access

To query a 64-bit buffer device address value through which buffer memory can be accessed in a shader, call:

```c
// Provided by VK_VERSION_1_2
VkDeviceAddress vkGetBufferDeviceAddress(
```
• `device` is the logical device that the buffer was created on.
• `pInfo` is a pointer to a `VkBufferDeviceAddressInfo` structure specifying the buffer to retrieve an address for.

The 64-bit return value is an address of the start of `pInfo->buffer`. The address range starting at this value and whose size is the size of the buffer can be used in a shader to access the memory bound to that buffer, using the `SPV_KHR_physical_storage_buffer` extension and the `PhysicalStorageBuffer` storage class. For example, this value can be stored in a uniform buffer, and the shader can read the value from the uniform buffer and use it to do a dependent read/write to this buffer. A value of zero is reserved as a “null” pointer and must not be returned as a valid buffer device address. All loads, stores, and atomics in a shader through `PhysicalStorageBuffer` pointers must access addresses in the address range of some buffer.

If the buffer was created with a non-zero value of `VkBufferOpaqueCaptureAddressCreateInfo::opaqueCaptureAddress`, the return value will be the same address that was returned at capture time.

The returned address must satisfy the alignment requirement specified by `VkMemoryRequirements::alignment` for the buffer in `VkBufferDeviceAddressInfo::buffer`.

If multiple `VkBuffer` objects are bound to overlapping ranges of `VkDeviceMemory`, implementations may return address ranges which overlap. In this case, it is ambiguous which `VkBuffer` is associated with any given device address. For purposes of valid usage, if multiple `VkBuffer` objects can be attributed to a device address, a `VkBuffer` is selected such that valid usage passes, if it exists.

### Valid Usage

- VUID-vkGetBufferDeviceAddress-None-06542
  The `bufferDeviceAddress` feature must be enabled

- VUID-vkGetBufferDeviceAddress-device-parameter
  If `device` was created with multiple physical devices, then the `bufferDeviceAddressMultiDevice` feature must be enabled

### Valid Usage (Implicit)

- VUID-vkGetBufferDeviceAddress-device-parameter
  `device` must be a valid `VkDevice` handle

- VUID-vkGetBufferDeviceAddress-pInfo-parameter
  `pInfo` must be a valid pointer to a valid `VkBufferDeviceAddressInfo` structure

The `VkBufferDeviceAddressInfo` structure is defined as:
typedef struct VkBufferDeviceAddressInfo {
    VkStructureType sType;
    const void* pNext;
    VkBuffer buffer;
} VkBufferDeviceAddressInfo;

• `sType` is the type of this structure.
• `pNext` is `NULL` or a pointer to a structure extending this structure.
• `buffer` specifies the buffer whose address is being queried.

**Valid Usage**

• VUID-VkBufferDeviceAddressInfo-buffer-02600
  If `buffer` is non-sparse and was not created with the `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT` flag, then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object

• VUID-VkBufferDeviceAddressInfo-buffer-02601
  `buffer` **must** have been created with `VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT`

**Valid Usage (Implicit)**

• VUID-VkBufferDeviceAddressInfo-sType-sType
  `sType` **must** be `VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO`

• VUID-VkBufferDeviceAddressInfo-pNext-pNext
  `pNext` **must** be `NULL`

• VUID-VkBufferDeviceAddressInfo-buffer-parameter
  `buffer` **must** be a valid `VkBuffer` handle

To query a 64-bit buffer opaque capture address, call:

```c
// Provided by VK_VERSION_1_2
uint64_t vkGetBufferOpaqueCaptureAddress(
    VkDevice device, 
    const VkBufferDeviceAddressInfo* pInfo);
```

• `device` is the logical device that the buffer was created on.

• `pInfo` is a pointer to a `VkBufferDeviceAddressInfo` structure specifying the buffer to retrieve an address for.

The 64-bit return value is an opaque capture address of the start of `pInfo->buffer`. 
If the buffer was created with a non-zero value of `VkBufferOpaqueCaptureAddressCreateInfo::opaqueCaptureAddress` the return value **must** be the same address.

### Valid Usage

- VUID-vkGetBufferOpaqueCaptureAddress-None-03326
  The `bufferDeviceAddress` feature **must** be enabled

- VUID-vkGetBufferOpaqueCaptureAddress-device-03327
  If `device` was created with multiple physical devices, then the `bufferDeviceAddressMultiDevice` feature **must** be enabled

### Valid Usage (Implicit)

- VUID-vkGetBufferOpaqueCaptureAddress-device-parameter
  `device` **must** be a valid `VkDevice` handle

- VUID-vkGetBufferOpaqueCaptureAddress-pInfo-parameter
  `pInfo` **must** be a valid pointer to a valid `VkBufferDeviceAddressInfo` structure
Chapter 15. Shader Interfaces

When a pipeline is created, the set of shaders specified in the corresponding Vk*PipelineCreateInfo structure are implicitly linked at a number of different interfaces.

- Shader Input and Output Interface
- Vertex Input Interface
- Fragment Output Interface
- Fragment Input Attachment Interface
- Shader Resource Interface

This chapter describes valid uses for a set of SPIR-V decorations. Any other use of one of these decorations is invalid, with the exception that, when using SPIR-V versions 1.4 and earlier: Block, BufferBlock, Offset, ArrayStride, and MatrixStride can also decorate types and type members used by variables in the Private and Function storage classes.

Note

In this chapter, there are references to SPIR-V terms such as the MeshNV execution model. These terms will appear even in a build of the specification which does not support any extensions. This is as intended, since these terms appear in the unified SPIR-V specification without such qualifiers.

15.1. Shader Input and Output Interfaces

When multiple stages are present in a pipeline, the outputs of one stage form an interface with the inputs of the next stage. When such an interface involves a shader, shader outputs are matched against the inputs of the next stage, and shader inputs are matched against the outputs of the previous stage.

All the variables forming the shader input and output interfaces are listed as operands to the OpEntryPoint instruction and are declared with the Input or Output storage classes, respectively, in the SPIR-V module. These generally form the interfaces between consecutive shader stages, regardless of any non-shader stages between the consecutive shader stages.

There are two classes of variables that can be matched between shader stages, built-in variables and user-defined variables. Each class has a different set of matching criteria.

Output variables of a shader stage have undefined values until the shader writes to them or uses the Initializer operand when declaring the variable.

15.1.1. Built-in Interface Block

Shader built-in variables meeting the following requirements define the built-in interface block. They must

- be explicitly declared (there are no implicit built-ins),
• be identified with a `BuiltIn` decoration,
• form object types as described in the `Built-in Variables` section, and
• be declared in a block whose top-level members are the built-ins.

There must be no more than one built-in interface block per shader per interface.

Built-ins must not have any `Location` or `Component` decorations.

### 15.1.2. User-defined Variable Interface

The non-built-in variables listed by `OpEntryPoint` with the `Input` or `Output` storage class form the user-defined variable interface. These must have SPIR-V numerical types or, recursively, composite types of such types. By default, the components of such types have a width of 32 or 64 bits. If an implementation supports `storageInputOutput16`, components can also have a width of 16 bits. These variables must be identified with a `Location` decoration and can also be identified with a `Component` decoration.

### 15.1.3. Interface Matching

An output variable, block, or structure member in a given shader stage has an interface match with an input variable, block, or structure member in a subsequent shader stage if they both adhere to the following conditions:

• They have equivalent decorations, other than:
  ◦ one is not decorated with `Component` and the other is declared with a `Component` of 0
  ◦ `Interpolation` decorations
  ◦ `RelaxedPrecision` if one is an input variable and the other an output variable

• Their types match as follows:
  ◦ if the input is declared in a tessellation control or geometry shader as an `OpTypeArray` with an `Element Type` equivalent to the `OpType*` declaration of the output, and neither is a structure member; or
  ◦ if the `maintenance4` feature is enabled, they are declared as `OpTypeVector` variables, and the output has a `Component Count` value higher than that of the input but the same `Component Type`; or
  ◦ if in any other case they are declared with an equivalent `OpType*` declaration.

• If both are structures and every member has an interface match.

#### Note

The word “structure” above refers to both variables that have an `OpTypeStruct` type and interface blocks (which are also declared as `OpTypeStruct`).

All input variables and blocks must have an interface match in the preceding shader stage, except for built-in variables in fragment shaders. Shaders can declare and write to output variables that are not declared or read by the subsequent stage.
The value of an input variable is undefined if the preceding stage does not write to a matching output variable, as described above.

### 15.1.4. Location Assignment

This section describes location assignments for user-defined variables and how many locations are consumed by a given user-variable type. As mentioned above, some inputs and outputs have an additional level of arrayness relative to other shader inputs and outputs. This outer array level is removed from the type before considering how many locations the type consumes.

The **Location** value specifies an interface slot comprised of a 32-bit four-component vector conveyed between stages. The **Component** specifies components within these vector locations. Only types with widths of 16, 32 or 64 are supported in shader interfaces.

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

- 16-bit scalar and vector types, and
- 32-bit scalar and vector types, and
- 64-bit scalar and 2-component vector types.

64-bit three- and four-component vectors consume two consecutive locations.

If a declared input or output is an array of size $n$ and each element takes $m$ locations, it will be assigned $m \times n$ consecutive locations starting with the location specified.

If the declared input or output is an $n \times m$ 16-, 32- or 64-bit matrix, it will be assigned multiple locations starting with the location specified. The number of locations assigned for each matrix will be the same as for an $n$-element array of $m$-component vectors.

An **OpVariable** with a structure type that is not a block must be decorated with a **Location**.

When an **OpVariable** with a structure type (either block or non-block) is decorated with a **Location**, the members in the structure type must not be decorated with a **Location**. The **OpVariable**'s members are assigned consecutive locations in declaration order, starting from the first member, which is assigned the location decoration from the **OpVariable**.

When a block-type **OpVariable** is declared without a **Location** decoration, each member in its structure type must be decorated with a **Location**. Types nested deeper than the top-level members must not have **Location** decorations.

The locations consumed by block and structure members are determined by applying the rules above in a depth-first traversal of the instantiated members as though the structure or block member were declared as an input or output variable of the same type.

Any two inputs listed as operands on the same **OpEntryPoint** must not be assigned the same location, either explicitly or implicitly. Any two outputs listed as operands on the same **OpEntryPoint** must not be assigned the same location, either explicitly or implicitly.

The number of input and output locations available for a shader input or output interface are limited, and dependent on the shader stage as described in **Shader Input and Output Locations**. All
variables in both the built-in interface block and the user-defined variable interface count against these limits. Each effective Location must have a value less than the number of locations available for the given interface, as specified in the “Locations Available” column in Shader Input and Output Locations.

Table 10. Shader Input and Output Locations

<table>
<thead>
<tr>
<th>Shader Interface</th>
<th>Locations Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex input</td>
<td>maxVertexInputAttributes</td>
</tr>
<tr>
<td>vertex output</td>
<td>maxVertexOutputComponents / 4</td>
</tr>
<tr>
<td>tessellation control input</td>
<td>maxTessellationControlPerVertexInputComponents / 4</td>
</tr>
<tr>
<td>tessellation control output</td>
<td>maxTessellationControlPerVertexOutputComponents / 4</td>
</tr>
<tr>
<td>tessellation evaluation input</td>
<td>maxTessellationEvaluationInputComponents / 4</td>
</tr>
<tr>
<td>tessellation evaluation output</td>
<td>maxTessellationEvaluationOutputComponents / 4</td>
</tr>
<tr>
<td>geometry input</td>
<td>maxGeometryInputComponents / 4</td>
</tr>
<tr>
<td>geometry output</td>
<td>maxGeometryOutputComponents / 4</td>
</tr>
<tr>
<td>fragment input</td>
<td>maxFragmentInputComponents / 4</td>
</tr>
<tr>
<td>fragment output</td>
<td>maxFragmentOutputAttachments</td>
</tr>
</tbody>
</table>

15.1.5. Component Assignment

The Component decoration allows the Location to be more finely specified for scalars and vectors, down to the individual components within a location that are consumed. The components within a location are 0, 1, 2, and 3. A variable or block member starting at component N will consume components N, N+1, N+2, ... up through its size. For 16-, and 32-bit types, it is invalid if this sequence of components gets larger than 3. A scalar 64-bit type will consume two of these components in sequence, and a two-component 64-bit vector type will consume all four components available within a location. A three- or four-component 64-bit vector type must not specify a Component decoration. A three-component 64-bit vector type will consume all four components of the first location and components 0 and 1 of the second location. This leaves components 2 and 3 available for other component-qualified declarations.

A scalar or two-component 64-bit data type must not specify a Component decoration of 1 or 3. A Component decoration must not be specified for any type that is not a scalar or vector.

15.2. Vertex Input Interface

When the vertex stage is present in a pipeline, the vertex shader input variables form an interface with the vertex input attributes. The vertex shader input variables are matched by the Location and Component decorations to the vertex input attributes specified in the pVertexInputState member of the VkGraphicsPipelineCreateInfo structure.

The vertex shader input variables listed by OpEntryPoint with the Input storage class form the vertex
input interface. These variables must be identified with a Location decoration and can also be identified with a Component decoration.

For the purposes of interface matching: variables declared without a Component decoration are considered to have a Component decoration of zero. The number of available vertex input locations is given by the maxVertexInputAttributes member of the VkPhysicalDeviceLimits structure.

See Attribute Location and Component Assignment for details.

All vertex shader inputs declared as above must have a corresponding attribute and binding in the pipeline.

### 15.3. Fragment Output Interface

When the fragment stage is present in a pipeline, the fragment shader outputs form an interface with the output attachments defined by a render pass instance. The fragment shader output variables are matched by the Location and Component decorations to specified color attachments.

The fragment shader output variables listed by OpEntryPoint with the Output storage class form the fragment output interface. These variables must be identified with a Location decoration. They can also be identified with a Component decoration and/or an Index decoration. For the purposes of interface matching: variables declared without a Component decoration are considered to have a Component decoration of zero, and variables declared without an Index decoration are considered to have an Index decoration of zero.

A fragment shader output variable identified with a Location decoration of \( i \) is associated with the element of VkRenderingInfo::pColorAttachments with a location equal to \( i \). When using render pass objects, it is associated with the color attachment indicated by pColorAttachments[\( i \)]. Values are written to those attachments after passing through the blending unit as described in Blending, if enabled. Locations are consumed as described in Location Assignment. The number of available fragment output locations is given by the maxFragmentOutputAttachments member of the VkPhysicalDeviceLimits structure.

Components of the output variables are assigned as described in Component Assignment. Output components identified as 0, 1, 2, and 3 will be directed to the R, G, B, and A inputs to the blending unit, respectively, or to the output attachment if blending is disabled. If two variables are placed within the same location, they must have the same underlying type (floating-point or integer). The input values to blending or color attachment writes are undefined for components which do not correspond to a fragment shader output.

Fragment outputs identified with an Index of zero are directed to the first input of the blending unit associated with the corresponding Location. Outputs identified with an Index of one are directed to the second input of the corresponding blending unit.

No component aliasing of output variables is allowed, that is there must not be two output variables which have the same location, component, and index, either explicitly declared or implied.

Output values written by a fragment shader must be declared with either OpTypeFloat or OpTypeInt, and a Width of 32. If storageInputOutput16 is supported, output values written by a fragment shader
can be also declared with either `OpTypeFloat` or `OpTypeInt` and a `Width` of 16. Composites of these types are also permitted. If the color attachment has a signed or unsigned normalized fixed-point format, color values are assumed to be floating-point and are converted to fixed-point as described in Conversion from Floating-Point to Normalized Fixed-Point; If the color attachment has an integer format, color values are assumed to be integers and converted to the bit-depth of the target. Any value that cannot be represented in the attachment’s format is undefined. For any other attachment format no conversion is performed. If the type of the values written by the fragment shader do not match the format of the corresponding color attachment, the resulting values are undefined for those components.

### 15.4. Fragment Input Attachment Interface

When a fragment stage is present in a pipeline, the fragment shader subpass inputs form an interface with the input attachments of the current subpass. The fragment shader subpass input variables are matched by `InputAttachmentIndex` decorations to the input attachments specified in the `pInputAttachments` array of the `VkSubpassDescription` structure describing the subpass that the fragment shader is executed in.

The fragment shader subpass input variables with the `UniformConstant` storage class and a decoration of `InputAttachmentIndex` that are statically used by `OpEntryPoint` form the fragment input attachment interface. These variables must be declared with a type of `OpTypeImage`, a `Dim` operand of `SubpassData`, an `Arrayed` operand of 0, and a `Sampled` operand of 2. The `MS` operand of the `OpTypeImage` must be 0 if the samples field of the corresponding `VkAttachmentDescription` is `VK_SAMPLE_COUNT_1_BIT` and 1 otherwise.

A subpass input variable identified with an `InputAttachmentIndex` decoration of `i` reads from the input attachment indicated by `pInputAttachments[i]` member of `VkSubpassDescription`. If the subpass input variable is declared as an array of size `N`, it consumes `N` consecutive input attachments, starting with the index specified. There must not be more than one input variable with the same `InputAttachmentIndex` whether explicitly declared or implied by an array declaration. The number of available input attachment indices is given by the `maxPerStageDescriptorInputAttachments` member of the `VkPhysicalDeviceLimits` structure.

Variables identified with the `InputAttachmentIndex` must only be used by a fragment stage. The basic data type (floating-point, integer, unsigned integer) of the subpass input must match the basic format of the corresponding input attachment, or the values of subpass loads from these variables are undefined. If the framebuffer attachment contains both depth and stencil aspects, the basic data type of the subpass input determines if depth or stencil aspect is accessed by the shader.

See Input Attachment for more details.

### 15.4.1. Fragment Input Attachment Compatibility

An input attachment that is statically accessed by a fragment shader must be backed by a descriptor that is equivalent to the `VkImageView` in the `VkFramebuffer`, except for `subresourceRange.aspectMask`. The `aspectMask` must be equal to the aspect accessed by the shader.
15.5. Shader Resource Interface

When a shader stage accesses buffer or image resources, as described in the Resource Descriptors section, the shader resource variables must be matched with the pipeline layout that is provided at pipeline creation time.

The set of shader variables that form the shader resource interface for a stage are the variables statically used by that stage’s OpEntryPoint with a storage class of Uniform, UniformConstant, StorageBuffer, or PushConstant. For the fragment shader, this includes the fragment input attachment interface.

The shader resource interface consists of two sub-interfaces: the push constant interface and the descriptor set interface.

15.5.1. Push Constant Interface

The shader variables defined with a storage class of PushConstant that are statically used by the shader entry points for the pipeline define the push constant interface. They must be:

- typed as OpTypeStruct,
- identified with a Block decoration, and
- laid out explicitly using the Offset, ArrayStride, and MatrixStride decorations as specified in Offset and Stride Assignment.

There must be no more than one push constant block statically used per shader entry point.

Each statically used member of a push constant block must be placed at an Offset such that the entire member is entirely contained within the VkPushConstantRange for each OpEntryPoint that uses it, and the stageFlags for that range must specify the appropriate VkShaderStageFlagBits for that stage. The Offset decoration for any member of a push constant block must not cause the space required for that member to extend outside the range [0, maxPushConstantsSize).

Any member of a push constant block that is declared as an array must only be accessed with dynamically uniform indices.

15.5.2. Descriptor Set Interface

The descriptor set interface is comprised of the shader variables with the storage class of StorageBuffer, Uniform or UniformConstant (including the variables in the fragment input attachment interface) that are statically used by the shader entry points for the pipeline.

These variables must have DescriptorSet and Binding decorations specified, which are assigned and matched with the VkDescriptorSetLayout objects in the pipeline layout as described in DescriptorSet and Binding Assignment.

The Image Format of an OpTypeImage declaration must not be Unknown, for variables which are used for OpImageRead, OpImageSparseRead, or OpImageWrite operations, except under the following conditions:
• For `OpImageWrite`, if the image format is listed in the `storage without format` list and if the `shaderStorageImageWriteWithoutFormat` feature is enabled and the shader module declares the `StorageImageWriteWithoutFormat` capability.

• For `OpImageWrite`, if the image format supports `VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT` and the shader module declares the `StorageImageWriteWithoutFormat` capability.

• For `OpImageRead` or `OpImageSparseRead`, if the image format is listed in the `storage without format` list and if the `shaderStorageImageReadWithoutFormat` feature is enabled and the shader module declares the `StorageImageReadWithoutFormat` capability.

• For `OpImageRead` or `OpImageSparseRead`, if the image format supports `VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT` and the shader module declares the `StorageImageReadWithoutFormat` capability.

• For `OpImageRead`, if `Dim` is `SubpassData` (indicating a read from an input attachment).

The `Image Format` of an `OpTypeImage` declaration must not be `Unknown`, for variables which are used for `OpAtomic*` operations.

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

• typed as `OpTypeStruct`, or an array of this type,
• identified with a `Block` or `BufferBlock` decoration, and
• laid out explicitly using the `Offset`, `ArrayStride`, and `MatrixStride` decorations as specified in `Offset and Stride Assignment`.

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

• typed as `OpTypeStruct`, or an array of this type,
• identified with a `Block` decoration, and
• laid out explicitly using the `Offset`, `ArrayStride`, and `MatrixStride` decorations as specified in `Offset and Stride Assignment`.

The `Offset` decoration for any member of a `Block`-decorated variable in the `Uniform` storage class must not cause the space required for that variable to extend outside the range `[0, maxUniformBufferRange)`. The `Offset` decoration for any member of a `Block`-decorated variable in the `StorageBuffer` storage class must not cause the space required for that variable to extend outside the range `[0, maxStorageBufferRange)`. Variables identified with the `Uniform` storage class can also be used to access transparent descriptor set backed resources when the variable is assigned to a descriptor set layout binding with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK`. In this case the variable must be typed as `OpTypeStruct` and cannot be aggregated into arrays of that type. Further, the `Offset` decoration for any member of such a variable must not cause the space required for that variable to extend outside the range `[0, maxInlineUniformBlockSize)`. 
Variables identified with a storage class of UniformConstant and a decoration of InputAttachmentIndex must be declared as described in Fragment Input Attachment Interface.

SPIR-V variables decorated with a descriptor set and binding that identify a combined image sampler descriptor can have a type of OpTypeImage, OpTypeSampler (Sampled=1), or OpTypeSampledImage.

Arrays of any of these types can be indexed with constant integral expressions. The following features must be enabled and capabilities must be declared in order to index such arrays with dynamically uniform or non-uniform indices:

- Storage images (except storage texel buffers and input attachments):
  - Dynamically uniform: shaderStorageImageArrayDynamicIndexing and StorageImageArrayDynamicIndexing
  - Non-uniform: shaderStorageImageArrayNonUniformIndexing and StorageImageArrayNonUniformIndexing

- Storage texel buffers:
  - Dynamically uniform: shaderStorageTexelBufferArrayDynamicIndexing and StorageTexelBufferArrayDynamicIndexing
  - Non-uniform: shaderStorageTexelBufferArrayNonUniformIndexing and StorageTexelBufferArrayNonUniformIndexing

- Input attachments:
  - Dynamically uniform: shaderInputAttachmentArrayDynamicIndexing and InputAttachmentArrayDynamicIndexing
  - Non-uniform: shaderInputAttachmentArrayNonUniformIndexing and InputAttachmentArrayNonUniformIndexing

- Sampled images (except uniform texel buffers), samplers and combined image samplers:
  - Dynamically uniform: shaderSampledImageArrayDynamicIndexing and SampledImageArrayDynamicIndexing
  - Non-uniform: shaderSampledImageArrayNonUniformIndexing and SampledImageArrayNonUniformIndexing

- Uniform texel buffers:
  - Dynamically uniform: shaderUniformTexelBufferArrayDynamicIndexing and UniformTexelBufferArrayDynamicIndexing
  - Non-uniform: shaderUniformTexelBufferArrayNonUniformIndexing and UniformTexelBufferArrayNonUniformIndexing

- Uniform buffers:
  - Dynamically uniform: shaderUniformBufferArrayDynamicIndexing and UniformBufferArrayDynamicIndexing
  - Non-uniform: shaderUniformBufferArrayNonUniformIndexing and UniformBufferArrayNonUniformIndexing

- Storage buffers:
Dynamically uniform: `shaderStorageBufferArrayDynamicIndexing` and `StorageBufferArrayDynamicIndexing`

Non-uniform: `shaderStorageBufferArrayNonUniformIndexing` and `StorageBufferArrayNonUniformIndexing`

If an instruction loads from or stores to a resource (including atomics and image instructions) and the resource descriptor being accessed is not dynamically uniform, then the corresponding non-uniform indexing feature **must** be enabled and the capability **must** be declared. If an instruction loads from or stores to a resource (including atomics and image instructions) and the resource descriptor being accessed is loaded from an array element with a non-constant index, then the corresponding dynamic or non-uniform indexing feature **must** be enabled and the capability **must** be declared.

If the combined image sampler enables sampler \( \text{YC}_{a}C_{b} \) conversion, it **must** be indexed only by constant integral expressions when aggregated into arrays in shader code, irrespective of the `shaderSampledImageArrayDynamicIndexing` feature.

<table>
<thead>
<tr>
<th>Table 11. Shader Resource and Descriptor Type Correspondence</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Resource type</strong></td>
</tr>
<tr>
<td>sampler</td>
</tr>
<tr>
<td>sampled image</td>
</tr>
<tr>
<td>storage image</td>
</tr>
<tr>
<td>combined image sampler</td>
</tr>
<tr>
<td>uniform texel buffer</td>
</tr>
<tr>
<td>storage texel buffer</td>
</tr>
<tr>
<td>uniform buffer</td>
</tr>
<tr>
<td>storage buffer</td>
</tr>
<tr>
<td>input attachment</td>
</tr>
<tr>
<td>inline uniform block</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 12. Shader Resource and Storage Class Correspondence</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Resource type</strong></td>
</tr>
<tr>
<td>sampler</td>
</tr>
<tr>
<td>sampled image</td>
</tr>
<tr>
<td>storage image</td>
</tr>
<tr>
<td>combined image sampler</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Resource type</td>
</tr>
<tr>
<td>----------------------------</td>
</tr>
<tr>
<td>uniform texel buffer</td>
</tr>
<tr>
<td>storage texel buffer</td>
</tr>
<tr>
<td>uniform buffer</td>
</tr>
<tr>
<td>storage buffer</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>input attachment</td>
</tr>
<tr>
<td>inline uniform block</td>
</tr>
</tbody>
</table>

1
Where OpTypeImage is referenced, the Dim values Buffer and Subpassdata are only accepted where they are specifically referenced. They do not correspond to resource types where a generic OpTypeImage is specified.

2
In addition to DescriptorSet and Binding.

15.5.3. DescriptorSet and Binding Assignment

A variable decorated with a DescriptorSet decoration of s and a Binding decoration of b indicates that this variable is associated with the VkDescriptorSetLayoutBinding that has a binding equal to b in pSetLayouts[s] that was specified in VkPipelineLayoutCreateInfo.

DescriptorSet decoration values must be between zero and maxBoundDescriptorSets minus one, inclusive. Binding decoration values can be any 32-bit unsigned integer value, as described in Descriptor Set Layout. Each descriptor set has its own binding name space.

If the Binding decoration is used with an array, the entire array is assigned that binding value. The array must be a single-dimensional array and size of the array must be no larger than the number of descriptors in the binding. If the array is runtime-sized, then array elements greater than or equal to the size of that binding in the bound descriptor set must not be used. If the array is runtime-sized, the runtimeDescriptorArray feature must be enabled and the RuntimeDescriptorArray capability must be declared. The index of each element of the array is referred to as the arrayElement. For the purposes of interface matching and descriptor set operations, if a resource variable is not an array, it is treated as if it has an arrayElement of zero.

There is a limit on the number of resources of each type that can be accessed by a pipeline stage as shown in Shader Resource Limits. The “Resources Per Stage” column gives the limit on the number...
each type of resource that can be statically used for an entry point in any given stage in a pipeline. The “Resource Types” column lists which resource types are counted against the limit. Some resource types count against multiple limits.

The pipeline layout may include descriptor sets and bindings which are not referenced by any variables statically used by the entry points for the shader stages in the binding’s stageFlags.

However, if a variable assigned to a given DescriptorSet and Binding is statically used by the entry point for a shader stage, the pipeline layout must contain a descriptor set layout binding in that descriptor set layout and for that binding number, and that binding’s stageFlags must include the appropriate VkShaderStageFlagBits for that stage. The variable must be of a valid resource type determined by its SPIR-V type and storage class, as defined in Shader Resource and Storage Class Correspondence. The descriptor set layout binding must be of a corresponding descriptor type, as defined in Shader Resource and Descriptor Type Correspondence.

Note
There are no limits on the number of shader variables that can have overlapping set and binding values in a shader; but which resources are statically used has an impact. If any shader variable identifying a resource is statically used in a shader, then the underlying descriptor bound at the declared set and binding must support the declared type in the shader when the shader executes.

If multiple shader variables are declared with the same set and binding values, and with the same underlying descriptor type, they can all be statically used within the same shader. However, accesses are not automatically synchronized, and Aliased decorations should be used to avoid data hazards (see section 2.18.2 Aliasing in the SPIR-V specification).

If multiple shader variables with the same set and binding values are declared in a single shader, but with different declared types, where any of those are not supported by the relevant bound descriptor, that shader can only be executed if the variables with the unsupported type are not statically used.

A noteworthy example of using multiple statically-used shader variables sharing the same descriptor set and binding values is a descriptor of type VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER that has multiple corresponding shader variables in the UniformConstant storage class, where some could be OpTypeImage (Sampled=1), some could be OpTypeSampler, and some could be OpTypeSampledImage.

Table 13. Shader Resource Limits

<table>
<thead>
<tr>
<th>Resources per Stage</th>
<th>Resource Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxPerStageDescriptorSamplers or maxPerStageDescriptorUpdateAfterBindSamplers</td>
<td>sampler</td>
</tr>
<tr>
<td></td>
<td>combined image sampler</td>
</tr>
</tbody>
</table>
### Resources per Stage

<table>
<thead>
<tr>
<th>Resources per Stage</th>
<th>Resource Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxPerStageDescriptorSampledImages or maxPerStageDescriptorUpdateAfterBindSampledImages</td>
<td>sampled image</td>
</tr>
<tr>
<td></td>
<td>combined image sampler</td>
</tr>
<tr>
<td></td>
<td>uniform texel buffer</td>
</tr>
<tr>
<td>maxPerStageDescriptorStorageImages or maxPerStageDescriptorUpdateAfterBindStorageImages</td>
<td>storage image</td>
</tr>
<tr>
<td></td>
<td>storage texel buffer</td>
</tr>
<tr>
<td>maxPerStageDescriptorUniformBuffers or maxPerStageDescriptorUpdateAfterBindUniformBuffers</td>
<td>uniform buffer</td>
</tr>
<tr>
<td></td>
<td>uniform buffer dynamic</td>
</tr>
<tr>
<td>maxPerStageDescriptorStorageBuffers or maxPerStageDescriptorUpdateAfterBindStorageBuffers</td>
<td>storage buffer</td>
</tr>
<tr>
<td></td>
<td>storage buffer dynamic</td>
</tr>
<tr>
<td>maxPerStageDescriptorInputAttachments or maxPerStageDescriptorUpdateAfterBindInputAttachments</td>
<td>input attachment¹</td>
</tr>
<tr>
<td>maxPerStageDescriptorInlineUniformBlocks or maxPerStageDescriptorUpdateAfterBindInlineUniformBlocks</td>
<td>inline uniform block</td>
</tr>
</tbody>
</table>

1

Input attachments **can** only be used in the fragment shader stage.

### 15.5.4. Offset and Stride Assignment

Certain objects **must** be explicitly laid out using the `Offset`, `ArrayStride`, and `MatrixStride`, as described in [SPIR-V explicit layout validation rules](#). All such layouts also **must** conform to the following requirements.

#### Note

The numeric order of `Offset` decorations does not need to follow member declaration order.

#### Alignment Requirements

There are different alignment requirements depending on the specific resources and on the features enabled on the device.

Matrix types are defined in terms of arrays as follows:

- A column-major matrix with C columns and R rows is equivalent to a C element array of vectors with R components.
- A row-major matrix with C columns and R rows is equivalent to an R element array of vectors with C components.

The *scalar alignment* of the type of an `OpTypeStruct` member is defined recursively as follows:
A scalar of size N has a scalar alignment of N.
A vector type has a scalar alignment equal to that of its component type.
An array type has a scalar alignment equal to that of its element type.
A structure has a scalar alignment equal to the largest scalar alignment of any of its members.
A matrix type inherits scalar alignment from the equivalent array declaration.

The base alignment of the type of an OpTypeStruct member is defined recursively as follows:

- A scalar has a base alignment equal to its scalar alignment.
- A two-component vector has a base alignment equal to twice its scalar alignment.
- A three- or four-component vector has a base alignment equal to four times its scalar alignment.
- An array has a base alignment equal to the base alignment of its element type.
- A structure has a base alignment equal to the largest base alignment of any of its members. An empty structure has a base alignment equal to the size of the smallest scalar type permitted by the capabilities declared in the SPIR-V module. (e.g., for a 1 byte aligned empty struct in the StorageBuffer storage class, StorageBuffer8BitAccess or UniformAndStorageBuffer8BitAccess must be declared in the SPIR-V module.)
- A matrix type inherits base alignment from the equivalent array declaration.

The extended alignment of the type of an OpTypeStruct member is similarly defined as follows:

- A scalar or vector type has an extended alignment equal to its base alignment.
- An array or structure type has an extended alignment equal to the largest extended alignment of any of its members, rounded up to a multiple of 16.
- A matrix type inherits extended alignment from the equivalent array declaration.

A member is defined to improperly straddle if either of the following are true:

- It is a vector with total size less than or equal to 16 bytes, and has Offset decorations placing its first byte at F and its last byte at L, where floor(F / 16) != floor(L / 16).
- It is a vector with total size greater than 16 bytes and has its Offset decorations placing its first byte at a non-integer multiple of 16.

**Standard Buffer Layout**

Every member of an OpTypeStruct that is required to be explicitly laid out must be aligned according to the first matching rule as follows. If the struct is contained in pointer types of multiple storage classes, it must satisfy the requirements for every storage class used to reference it.

1. If the scalarBlockLayout feature is enabled on the device and the storage class is Uniform, StorageBuffer, PhysicalStorageBuffer, or PushConstant then every member must be aligned according to its scalar alignment.
2. All vectors must be aligned according to their scalar alignment.
3. If the uniformBufferStandardLayout feature is not enabled on the device, then any member of an
OpTypeStruct with a storage class of Uniform and a decoration of Block must be aligned according to its extended alignment.

4. Every other member must be aligned according to its base alignment.

```
Note
Even if scalar alignment is supported, it is generally more performant to use the base alignment.
```

The memory layout must obey the following rules:

- The Offset decoration of any member must be a multiple of its alignment.
- Any ArrayStride or MatrixStride decoration must be a multiple of the alignment of the array or matrix as defined above.

If one of the conditions below applies

- The storage class is Uniform, StorageBuffer, PhysicalStorageBuffer, or PushConstant, and the scalarBlockLayout feature is not enabled on the device.
- The storage class is any other storage class.

the memory layout must also obey the following rules:

- Vectors must not improperly straddle, as defined above.
- The Offset decoration of a member must not place it between the end of a structure, an array or a matrix and the next multiple of the alignment of that structure, array or matrix.

```
Note
The std430 layout in GLSL satisfies these rules for types using the base alignment. The std140 layout satisfies the rules for types using the extended alignment.
```

## 15.6. Built-In Variables

Built-in variables are accessed in shaders by declaring a variable decorated with a BuiltIn SPIR-V decoration. The meaning of each BuiltIn decoration is as follows. In the remainder of this section, the name of a built-in is used interchangeably with a term equivalent to a variable decorated with that particular built-in. Built-ins that represent integer values can be declared as either signed or unsigned 32-bit integers.

As mentioned above, some inputs and outputs have an additional level of arrayness relative to other shader inputs and outputs. This level of arrayness is not included in the type descriptions below, but must be included when declaring the built-in.

**BaseInstance**

Decorating a variable with the BaseInstance built-in will make that variable contain the integer value corresponding to the first instance that was passed to the command that invoked the current vertex shader invocation. BaseInstance is the firstInstance parameter to a direct drawing command or the firstInstance member of a structure consumed by an indirect drawing
Valid Usage

- VUID-BaseInstance-BaseInstance-04181
  The BaseInstance decoration must be used only within the Vertex Execution Model

- VUID-BaseInstance-BaseInstance-04182
  The variable decorated with BaseInstance must be declared using the Input Storage Class

- VUID-BaseInstance-BaseInstance-04183
  The variable decorated with BaseInstance must be declared as a scalar 32-bit integer value

BaseVertex

Decorating a variable with the BaseVertex built-in will make that variable contain the integer value corresponding to the first vertex or vertex offset that was passed to the command that invoked the current vertex shader invocation. For non-indexed drawing commands, this variable is the firstVertex parameter to a direct drawing command or the firstVertex member of the structure consumed by an indirect drawing command. For indexed drawing commands, this variable is the vertexOffset parameter to a direct drawing command or the vertexOffset member of the structure consumed by an indirect drawing command.

Valid Usage

- VUID-BaseVertex-BaseVertex-04184
  The BaseVertex decoration must be used only within the Vertex Execution Model

- VUID-BaseVertex-BaseVertex-04185
  The variable decorated with BaseVertex must be declared using the Input Storage Class

- VUID-BaseVertex-BaseVertex-04186
  The variable decorated with BaseVertex must be declared as a scalar 32-bit integer value

ClipDistance

Decorating a variable with the ClipDistance built-in decoration will make that variable contain the mechanism for controlling user clipping. ClipDistance is an array such that the i\textsuperscript{th} element of the array specifies the clip distance for plane i. A clip distance of 0 means the vertex is on the plane, a positive distance means the vertex is inside the clip half-space, and a negative distance means the vertex is outside the clip half-space.

Note

The array variable decorated with ClipDistance is explicitly sized by the shader.

Note

In the last pre-rasterization shader stage, these values will be linearly interpolated across the primitive and the portion of the primitive with interpolated distances...
less than 0 will be considered outside the clip volume. If `ClipDistance` is then used by a fragment shader, `ClipDistance` contains these linearly interpolated values.

### Valid Usage

- **VUID-ClipDistance-ClipDistance-04187**
  The `ClipDistance` decoration **must** be used only within the `MeshEXT`, `MeshNV`, `Vertex`, `Fragment`, `TessellationControl`, `TessellationEvaluation`, or `Geometry Execution Model`.

- **VUID-ClipDistance-ClipDistance-04188**
  The variable decorated with `ClipDistance` within the `MeshEXT`, `MeshNV`, or `Vertex Execution Model` **must** be declared using the `Output Storage Class`.

- **VUID-ClipDistance-ClipDistance-04189**
  The variable decorated with `ClipDistance` within the `Fragment Execution Model` **must** be declared using the `Input Storage Class`.

- **VUID-ClipDistance-ClipDistance-04190**
  The variable decorated with `ClipDistance` within the `TessellationControl`, `TessellationEvaluation`, or `Geometry Execution Model` **must** not be declared in a `Storage Class` other than `Input` or `Output`.

- **VUID-ClipDistance-ClipDistance-04191**
  The variable decorated with `ClipDistance` **must** be declared as an array of 32-bit floating-point values.

### CullDistance

Decorating a variable with the `CullDistance` built-in decoration will make that variable contain the mechanism for controlling user culling. If any member of this array is assigned a negative value for all vertices belonging to a primitive, then the primitive is discarded before rasterization.

**Note**

In fragment shaders, the values of the `CullDistance` array are linearly interpolated across each primitive.

**Note**

If `CullDistance` decorates an input variable, that variable will contain the corresponding value from the `CullDistance` decorated output variable from the previous shader stage.

### Valid Usage

- **VUID-CullDistance-CullDistance-04196**
  The `CullDistance` decoration **must** be used only within the `MeshEXT`, `MeshNV`, `Vertex`, `Fragment`, `TessellationControl`, `TessellationEvaluation`, or `Geometry Execution Model`.

- **VUID-CullDistance-CullDistance-04197**
The variable decorated with `CullDistance` within the MeshEXT, MeshNV or Vertex Execution Model must be declared using the Output Storage Class.

- **VUID-CullDistance-CullDistance-04198**
  The variable decorated with `CullDistance` within the Fragment Execution Model must be declared using the Input Storage Class.

- **VUID-CullDistance-CullDistance-04199**
  The variable decorated with `CullDistance` within the TessellationControl, TessellationEvaluation, or Geometry Execution Model must not be declared using a Storage Class other than Input or Output.

- **VUID-CullDistance-CullDistance-04200**
  The variable decorated with `CullDistance` must be declared as an array of 32-bit floating-point values.

**DeviceIndex**

The `DeviceIndex` decoration can be applied to a shader input which will be filled with the device index of the physical device that is executing the current shader invocation. This value will be in the range \(0, \max(1, \text{physicalDeviceCount})\), where `physicalDeviceCount` is the `physicalDeviceCount` member of `VkDeviceGroupDeviceCreateInfo`.

**Valid Usage**

- **VUID-DeviceIndex-DeviceIndex-04205**
  The variable decorated with `DeviceIndex` must be declared using the Input Storage Class.

- **VUID-DeviceIndex-DeviceIndex-04206**
  The variable decorated with `DeviceIndex` must be declared as a scalar 32-bit integer value.

**DrawIndex**

Decorating a variable with the `DrawIndex` built-in will make that variable contain the integer value corresponding to the zero-based index of the drawing command that invoked the current vertex shader invocation. For *indirect drawing commands*, `DrawIndex` begins at zero and increments by one for each drawing command executed. The number of drawing commands is given by the `drawCount` parameter. For *direct drawing commands*, `DrawIndex` is always zero. `DrawIndex` is dynamically uniform.

**Valid Usage**

- **VUID-DrawIndex-DrawIndex-04207**
  The `DrawIndex` decoration must be used only within the Vertex, MeshEXT, TaskEXT, MeshNV, or TaskNV Execution Model.

- **VUID-DrawIndex-DrawIndex-04208**
  The variable decorated with `DrawIndex` must be declared using the Input Storage Class.

- **VUID-DrawIndex-DrawIndex-04209**
  The variable decorated with `DrawIndex` must be declared as a scalar 32-bit integer value.
**FragCoord**

Decorating a variable with the **FragCoord** built-in decoration will make that variable contain the framebuffer coordinate \((x, y, z, \frac{1}{w})\) of the fragment being processed. The \((x,y)\) coordinate \((0,0)\) is the upper left corner of the upper left pixel in the framebuffer.

When **Sample Shading** is enabled, the \(x\) and \(y\) components of **FragCoord** reflect the location of one of the samples corresponding to the shader invocation.

Otherwise, the \(x\) and \(y\) components of **FragCoord** reflect the location of the center of the fragment.

The \(z\) component of **FragCoord** is the interpolated depth value of the primitive.

The \(w\) component is the interpolated \(\frac{1}{w}\).

The **Centroid** interpolation decoration is ignored, but allowed, on **FragCoord**.

---

### Valid Usage

- **VUID-FragCoord-FragCoord-04210**
  - The **FragCoord** decoration **must** be used only within the **Fragment Execution Model**

- **VUID-FragCoord-FragCoord-04211**
  - The variable decorated with **FragCoord** **must** be declared using the **Input Storage Class**

- **VUID-FragCoord-FragCoord-04212**
  - The variable decorated with **FragCoord** **must** be declared as a four-component vector of 32-bit floating-point values

---

**FragDepth**

To have a shader supply a fragment-depth value, the shader **must** declare the **DepthReplacing** execution mode. Such a shader's fragment-depth value will come from the variable decorated with the **FragDepth** built-in decoration.

This value will be used for any subsequent depth testing performed by the implementation or writes to the depth attachment. See **fragment shader depth replacement** for details.

---

### Valid Usage

- **VUID-FragDepth-FragDepth-04213**
  - The **FragDepth** decoration **must** be used only within the **Fragment Execution Model**

- **VUID-FragDepth-FragDepth-04214**
  - The variable decorated with **FragDepth** **must** be declared using the **Output Storage Class**

- **VUID-FragDepth-FragDepth-04215**
  - The variable decorated with **FragDepth** **must** be declared as a scalar 32-bit floating-point value

- **VUID-FragDepth-FragDepth-04216**
  - If the shader dynamically writes to the variable decorated with **FragDepth**, the
DepthReplacing Execution Mode must be declared

**FrontFacing**
Decorating a variable with the `FrontFacing` built-in decoration will make that variable contain whether the fragment is front or back facing. This variable is non-zero if the current fragment is considered to be part of a front-facing polygon primitive or of a non-polygon primitive and is zero if the fragment is considered to be part of a back-facing polygon primitive.

**Valid Usage**
- VUID-FrontFacing-FrontFacing-04229
  The `FrontFacing` decoration must be used only within the Fragment Execution Model
- VUID-FrontFacing-FrontFacing-04230
  The variable decorated with `FrontFacing` must be declared using the Input Storage Class
- VUID-FrontFacing-FrontFacing-04231
  The variable decorated with `FrontFacing` must be declared as a boolean value

**GlobalInvocationId**
Decorating a variable with the `GlobalInvocationId` built-in decoration will make that variable contain the location of the current invocation within the global workgroup. Each component is equal to the index of the local workgroup multiplied by the size of the local workgroup plus `LocalInvocationId`.

**Valid Usage**
- VUID-GlobalInvocationId-GlobalInvocationId-04236
  The `GlobalInvocationId` decoration must be used only within the GLCompute, MeshEXT, TaskEXT, MeshNV, or TaskNV Execution Model
- VUID-GlobalInvocationId-GlobalInvocationId-04237
  The variable decorated with `GlobalInvocationId` must be declared using the Input Storage Class
- VUID-GlobalInvocationId-GlobalInvocationId-04238
  The variable decorated with `GlobalInvocationId` must be declared as a three-component vector of 32-bit integer values

**HelperInvocation**
Decorating a variable with the `HelperInvocation` built-in decoration will make that variable contain whether the current invocation is a helper invocation. This variable is non-zero if the current fragment being shaded is a helper invocation and zero otherwise. A helper invocation is an invocation of the shader that is produced to satisfy internal requirements such as the generation of derivatives.

**Note**
It is very likely that a helper invocation will have a value of \texttt{SampleMask} fragment shader input value that is zero.

**Valid Usage**

- VUID-HelperInvocation-HelperInvocation-04239
  The \texttt{HelperInvocation} decoration \textbf{must} be used only within the \texttt{Fragment Execution Model}
- VUID-HelperInvocation-HelperInvocation-04240
  The variable decorated with \texttt{HelperInvocation} \textbf{must} be declared using the \texttt{Input Storage Class}
- VUID-HelperInvocation-HelperInvocation-04241
  The variable decorated with \texttt{HelperInvocation} \textbf{must} be declared as a boolean value

**InvocationId**

Decorating a variable with the \texttt{InvocationId} built-in decoration will make that variable contain the index of the current shader invocation in a geometry shader, or the index of the output patch vertex in a tessellation control shader.

In a geometry shader, the index of the current shader invocation ranges from zero to the number of \texttt{instances} declared in the shader minus one. If the instance count of the geometry shader is one or is not specified, then \texttt{InvocationId} will be zero.

**Valid Usage**

- VUID-InvocationId-InvocationId-04257
  The \texttt{InvocationId} decoration \textbf{must} be used only within the \texttt{TessellationControl} or \texttt{Geometry Execution Model}
- VUID-InvocationId-InvocationId-04258
  The variable decorated with \texttt{InvocationId} \textbf{must} be declared using the \texttt{Input Storage Class}
- VUID-InvocationId-InvocationId-04259
  The variable decorated with \texttt{InvocationId} \textbf{must} be declared as a scalar 32-bit integer value

**InstanceIndex**

Decorating a variable in a vertex shader with the \texttt{InstanceIndex} built-in decoration will make that variable contain the index of the instance that is being processed by the current vertex shader invocation. \texttt{InstanceIndex} begins at the \texttt{firstInstance} parameter to \texttt{vkCmdDraw} or \texttt{vkCmdDrawIndexed} or at the \texttt{firstInstance} member of a structure consumed by \texttt{vkCmdDrawIndirect} or \texttt{vkCmdDrawIndexedIndirect}.

**Valid Usage**

- VUID-InstanceIndex-InstanceIndex-04263
  The \texttt{InstanceIndex} decoration \textbf{must} be used only within the \texttt{Vertex Execution Model}
Layer

Decorating a variable with the `Layer` built-in decoration will make that variable contain the select layer of a multi-layer framebuffer attachment.

In a vertex, tessellation evaluation, or geometry shader, any variable decorated with `Layer` can be written with the framebuffer layer index to which the primitive produced by that shader will be directed.

The last active pre-rasterization shader stage (in pipeline order) controls the `Layer` that is used. Outputs in previous shader stages are not used, even if the last stage fails to write the `Layer`.

If the last active pre-rasterization shader stage shader entry point's interface does not include a variable decorated with `Layer`, then the first layer is used. If a pre-rasterization shader stage shader entry point's interface includes a variable decorated with `Layer`, it must write the same value to `Layer` for all output vertices of a given primitive. If the `Layer` value is less than 0 or greater than or equal to the number of layers in the framebuffer, then primitives may still be rasterized, fragment shaders may be executed, and the framebuffer values for all layers are undefined.

In a fragment shader, a variable decorated with `Layer` contains the layer index of the primitive that the fragment invocation belongs to.

Valid Usage

- **VUID-Layer-Layer-04272**
  The `Layer` decoration must be used only within the `MeshEXT`, `MeshNV`, `Vertex`, `TessellationEvaluation`, `Geometry`, or `Fragment Execution Model`

- **VUID-Layer-Layer-04273**
  If the `shaderOutputLayer` feature is not enabled then the `Layer` decoration must be used only within the `Geometry` or `Fragment Execution Model`

- **VUID-Layer-Layer-04274**
  The variable decorated with `Layer` within the `MeshEXT`, `MeshNV`, `Vertex`, `TessellationEvaluation`, or `Geometry Execution Model` must be declared using the `Output Storage Class`

- **VUID-Layer-Layer-04275**
  The variable decorated with `Layer` within the `Fragment Execution Model` must be declared using the `Input Storage Class`

- **VUID-Layer-Layer-04276**
  The variable decorated with `Layer` must be declared as a scalar 32-bit integer value
The variable decorated with `Layer` within the `MeshEXT Execution Model` must also be decorated with the `PerPrimitiveEXT` decoration.

### LocalInvocationId

Decorating a variable with the `LocalInvocationId` built-in decoration will make that variable contain the location of the current compute shader invocation within the local workgroup. Each component ranges from zero through to the size of the workgroup in that dimension minus one.

**Note**

If the size of the workgroup in a particular dimension is one, then the `LocalInvocationId` in that dimension will be zero. If the workgroup is effectively two-dimensional, then `LocalInvocationId.z` will be zero. If the workgroup is effectively one-dimensional, then both `LocalInvocationId.y` and `LocalInvocationId.z` will be zero.

### Valid Usage

- **VUID-LocalInvocationId-LocalInvocationId-04281**
  The `LocalInvocationId` decoration must be used only within the `GLCompute`, `MeshEXT`, `TaskEXT`, `MeshNV`, or `TaskNV Execution Model`.

- **VUID-LocalInvocationId-LocalInvocationId-04282**
  The variable decorated with `LocalInvocationId` must be declared using the `Input Storage Class`.

- **VUID-LocalInvocationId-LocalInvocationId-04283**
  The variable decorated with `LocalInvocationId` must be declared as a three-component vector of 32-bit integer values.

### LocalInvocationIndex

Decorating a variable with the `LocalInvocationIndex` built-in decoration will make that variable contain a one-dimensional representation of `LocalInvocationId`. This is computed as:

\[
\text{LocalInvocationIndex} = \text{LocalInvocationId.z} \times \text{WorkgroupSize.x} \times \text{WorkgroupSize.y} + \\
\text{LocalInvocationId.y} \times \text{WorkgroupSize.x} + \\
\text{LocalInvocationId.x};
\]

### Valid Usage

- **VUID-LocalInvocationIndex-LocalInvocationIndex-04284**
  The `LocalInvocationIndex` decoration must be used only within the `GLCompute`, `MeshEXT`, `TaskEXT`, `MeshNV`, or `TaskNV Execution Model`.

- **VUID-LocalInvocationIndex-LocalInvocationIndex-04285**
The variable decorated with `LocalInvocationIndex` **must** be declared using the **Input Storage Class**

- VUID-LocalInvocationIndex-LocalInvocationIndex-04286
  The variable decorated with `LocalInvocationIndex` **must** be declared as a scalar 32-bit integer value

**NumSubgroups**

Decorating a variable with the `NumSubgroups` built-in decoration will make that variable contain the number of subgroups in the local workgroup.

**Valid Usage**

- VUID-NumSubgroups-NumSubgroups-04293
  The `NumSubgroups` decoration **must** be used only within the `GLCompute`, `MeshEXT`, `TaskEXT`, `MeshNV`, or `TaskNV Execution Model`

- VUID-NumSubgroups-NumSubgroups-04294
  The variable decorated with `NumSubgroups` **must** be declared using the **Input Storage Class**

- VUID-NumSubgroups-NumSubgroups-04295
  The variable decorated with `NumSubgroups` **must** be declared as a scalar 32-bit integer value

**NumWorkgroups**

Decorating a variable with the `NumWorkgroups` built-in decoration will make that variable contain the number of local workgroups that are part of the dispatch that the invocation belongs to. Each component is equal to the values of the workgroup count parameters passed into the dispatching commands.

**Valid Usage**

- VUID-NumWorkgroups-NumWorkgroups-04296
  The `NumWorkgroups` decoration **must** be used only within the `GLCompute`, `MeshEXT`, or `TaskEXT Execution Model`

- VUID-NumWorkgroups-NumWorkgroups-04297
  The variable decorated with `NumWorkgroups` **must** be declared using the **Input Storage Class**

- VUID-NumWorkgroups-NumWorkgroups-04298
  The variable decorated with `NumWorkgroups` **must** be declared as a three-component vector of 32-bit integer values

**PatchVertices**

Decorating a variable with the `PatchVertices` built-in decoration will make that variable contain the number of vertices in the input patch being processed by the shader. In a Tessellation Control Shader, this is the same as the name:`patchControlPoints` member of `VkPipelineTessellationStateCreateInfo`. In a Tessellation Evaluation Shader, `PatchVertices` is
equal to the tessellation control output patch size. When the same shader is used in different pipelines where the patch sizes are configured differently, the value of the `PatchVertices` variable will also differ.

**Valid Usage**

- **VUID-PatchVertices-PatchVertices-04308**
  The `PatchVertices` decoration **must** be used only within the `TessellationControl` or `TessellationEvaluation` Execution Model

- **VUID-PatchVertices-PatchVertices-04309**
  The variable decorated with `PatchVertices` **must** be declared using the `Input Storage Class`

- **VUID-PatchVertices-PatchVertices-04310**
  The variable decorated with `PatchVertices` **must** be declared as a scalar 32-bit integer value

**PointCoord**

Decorating a variable with the `PointCoord` built-in decoration will make that variable contain the coordinate of the current fragment within the point being rasterized, normalized to the size of the point with origin in the upper left corner of the point, as described in Basic Point Rasterization. If the primitive the fragment shader invocation belongs to is not a point, then the variable decorated with `PointCoord` contains an undefined value.

*Note*

Depending on how the point is rasterized, `PointCoord` may never reach (0,0) or (1,1).

**Valid Usage**

- **VUID-PointCoord-PointCoord-04311**
  The `PointCoord` decoration **must** be used only within the `Fragment Execution Model`

- **VUID-PointCoord-PointCoord-04312**
  The variable decorated with `PointCoord` **must** be declared using the `Input Storage Class`

- **VUID-PointCoord-PointCoord-04313**
  The variable decorated with `PointCoord` **must** be declared as a two-component vector of 32-bit floating-point values

**PointSize**

Decorating a variable with the `PointSize` built-in decoration will make that variable contain the size of point primitives. The value written to the variable decorated with `PointSize` by the last pre-rasterization shader stage in the pipeline is used as the framebuffer-space size of points produced by rasterization.

*Note*
When `PointSize` decorates a variable in the **Input Storage Class**, it contains the data written to the output variable decorated with `PointSize` from the previous shader stage.

**Valid Usage**

- **VUID-PointSize-PointSize-04314**
  The `PointSize` decoration **must** be used only within the `MeshEXT`, `MeshNV`, `Vertex`, `TessellationControl`, `TessellationEvaluation`, or `Geometry Execution Model`

- **VUID-PointSize-PointSize-04315**
  The variable decorated with `PointSize` within the `MeshEXT`, `MeshNV`, or `Vertex Execution Model` **must** be declared using the **Output Storage Class**

- **VUID-PointSize-PointSize-04316**
  The variable decorated with `PointSize` within the `TessellationControl`, `TessellationEvaluation`, or `Geometry Execution Model` **must** not be declared using a **Storage Class** other than **Input** or **Output**

- **VUID-PointSize-PointSize-04317**
  The variable decorated with `PointSize` **must** be declared as a scalar 32-bit floating-point value

**Position**

Decorating a variable with the **Position** built-in decoration will make that variable contain the position of the current vertex. In the last **pre-rasterization shader stage**, the value of the variable decorated with **Position** is used in subsequent primitive assembly, clipping, and rasterization operations.

**Note**

When `Position` decorates a variable in the **Input Storage Class**, it contains the data written to the output variable decorated with `Position` from the previous shader stage.

**Valid Usage**

- **VUID-Position-Position-04318**
  The **Position** decoration **must** be used only within the `MeshEXT`, `MeshNV`, `Vertex`, `TessellationControl`, `TessellationEvaluation`, or `Geometry Execution Model`

- **VUID-Position-Position-04319**
  The variable decorated with **Position** within the `MeshEXT`, `MeshNV`, or `Vertex Execution Model` **must** be declared using the **Output Storage Class**

- **VUID-Position-Position-04320**
  The variable decorated with **Position** within the `TessellationControl`, `TessellationEvaluation`, or `Geometry Execution Model` **must** not be declared using a **Storage Class** other than **Input** or **Output**
The variable decorated with Position must be declared as a four-component vector of 32-bit floating-point values.

**PrimitiveId**

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

The index of the first primitive generated by a drawing command is zero, and the index is incremented after every individual point, line, or triangle primitive is processed.

For triangles drawn as points or line segments (see Polygon Mode), the primitive index is incremented only once, even if multiple points or lines are eventually drawn.

Variables decorated with PrimitiveId are reset to zero between each instance drawn.

Restarting a primitive topology using primitive restart has no effect on the value of variables decorated with PrimitiveId.

In tessellation control and tessellation evaluation shaders, it will contain the index of the patch within the current set of rendering primitives that corresponds to the shader invocation.

In a geometry shader, it will contain the number of primitives presented as input to the shader since the current set of rendering primitives was started.

In a fragment shader, it will contain the primitive index written by the geometry shader if a geometry shader is present, or with the value that would have been presented as input to the geometry shader had it been present.

*Note*

When the PrimitiveId decoration is applied to an output variable in the geometry shader, the resulting value is seen through the PrimitiveId decorated input variable in the fragment shader.

The fragment shader using PrimitiveId will need to declare either the Geometry or Tessellation capability to satisfy the requirement SPIR-V has to use PrimitiveId.

**Valid Usage**

- VUID-PrimitiveId-PrimitiveId-04330
  The PrimitiveId decoration must be used only within the MeshEXT, MeshNV, IntersectionKHR, AnyHitKHR, ClosestHitKHR, TessellationControl, TessellationEvaluation, Geometry, or Fragment Execution Model

- VUID-PrimitiveId-Fragment-04331
  If pipeline contains both the Fragment and Geometry Execution Model and a variable decorated with PrimitiveId is read from Fragment shader, then the Geometry shader must write to the output variables decorated with PrimitiveId in all execution paths
If pipeline contains both the Fragment and MeshEXT or MeshNV Execution Model and a variable decorated with PrimitiveId is read from Fragment shader, then the MeshEXT or MeshNV shader must write to the output variables decorated with PrimitiveId in all execution paths.

If Fragment Execution Model contains a variable decorated with PrimitiveId, then either the MeshShadingEXT, MeshShadingNV, Geometry or Tessellation capability must also be declared.

The variable decorated with PrimitiveId within the TessellationControl, TessellationEvaluation, Fragment, IntersectionKHR, AnyHitKHR, or ClosestHitKHR Execution Model must be declared using the Input Storage Class.

The variable decorated with PrimitiveId within the Geometry Execution Model must be declared using the Input or Output Storage Class.

The variable decorated with PrimitiveId within the MeshEXT or MeshNV Execution Model must be declared using the Output Storage Class.

The variable decorated with PrimitiveId must be declared as a scalar 32-bit integer value.

The variable decorated with PrimitiveId within the MeshEXT Execution Model must also be decorated with the PerPrimitiveEXT decoration.

**SampleId**

Decorating a variable with the SampleId built-in decoration will make that variable contain the coverage index for the current fragment shader invocation. SampleId ranges from zero to the number of samples in the framebuffer minus one. If a fragment shader entry point's interface includes an input variable decorated with SampleId, Sample Shading is considered enabled with a minSampleShading value of 1.0.

**Valid Usage**

- The SampleId decoration must be used only within the Fragment Execution Model.
- The variable decorated with SampleId must be declared using the Input Storage Class.
- The variable decorated with SampleId must be declared as a scalar 32-bit integer value.

**SampleMask**

Decorating a variable with the SampleMask built-in decoration will make any variable contain the sample mask for the current fragment shader invocation.
A variable in the Input storage class decorated with SampleMask will contain a bitmask of the set of samples covered by the primitive generating the fragment during rasterization. It has a sample bit set if and only if the sample is considered covered for this fragment shader invocation. SampleMask[] is an array of integers. Bits are mapped to samples in a manner where bit B of mask M (SampleMask[M]) corresponds to sample \(32 \times M + B\).

A variable in the Output storage class decorated with SampleMask is an array of integers forming a bit array in a manner similar to an input variable decorated with SampleMask, but where each bit represents coverage as computed by the shader. This computed SampleMask is combined with the generated coverage mask in the multisample coverage operation.

Variables decorated with SampleMask must be either an unsized array, or explicitly sized to be no larger than the implementation-dependent maximum sample-mask (as an array of 32-bit elements), determined by the maximum number of samples.

If a fragment shader entry point's interface includes an output variable decorated with SampleMask, the sample mask will be undefined for any array elements of any fragment shader invocations that fail to assign a value. If a fragment shader entry point's interface does not include an output variable decorated with SampleMask, the sample mask has no effect on the processing of a fragment.

---

### Valid Usage

- **VUID-SampleMask-SampleMask-04357**  
The SampleMask decoration must be used only within the Fragment Execution Model

- **VUID-SampleMask-SampleMask-04358**  
The variable decorated with SampleMask must be declared using the Input or Output Storage Class

- **VUID-SampleMask-SampleMask-04359**  
The variable decorated with SampleMask must be declared as an array of 32-bit integer values

---

### SamplePosition

Decorating a variable with the SamplePosition built-in decoration will make that variable contain the sub-pixel position of the sample being shaded. The top left of the pixel is considered to be at coordinate (0,0) and the bottom right of the pixel is considered to be at coordinate (1,1).

If a fragment shader entry point's interface includes an input variable decorated with SamplePosition, Sample Shading is considered enabled with a \(\text{minSampleShading}\) value of 1.0.

---

### Valid Usage

- **VUID-SamplePosition-SamplePosition-04360**  
The SamplePosition decoration must be used only within the Fragment Execution Model

- **VUID-SamplePosition-SamplePosition-04361**  
The variable decorated with SamplePosition must be declared using the Input Storage
Class

• VUID-SamplePosition-SamplePosition-04362
  The variable decorated with SamplePosition must be declared as a two-component vector of 32-bit floating-point values

SubgroupId

Decorating a variable with the SubgroupId built-in decoration will make that variable contain the index of the subgroup within the local workgroup. This variable is in range [0, NumSubgroups-1].

Valid Usage

• VUID-SubgroupId-SubgroupId-04367
  The SubgroupId decoration must be used only within the GLCompute, MeshEXT, TaskEXT, MeshNV, or TaskNV Execution Model
• VUID-SubgroupId-SubgroupId-04368
  The variable decorated with SubgroupId must be declared using the Input Storage Class
• VUID-SubgroupId-SubgroupId-04369
  The variable decorated with SubgroupId must be declared as a scalar 32-bit integer value

SubgroupEqMask

Decorating a variable with the SubgroupEqMask builtin decoration will make that variable contain the subgroup mask of the current subgroup invocation. The bit corresponding to the SubgroupLocalInvocationId is set in the variable decorated with SubgroupEqMask. All other bits are set to zero.

SubgroupEqMaskKHR is an alias of SubgroupEqMask.

Valid Usage

• VUID-SubgroupEqMask-SubgroupEqMask-04370
  The variable decorated with SubgroupEqMask must be declared using the Input Storage Class
• VUID-SubgroupEqMask-SubgroupEqMask-04371
  The variable decorated with SubgroupEqMask must be declared as a four-component vector of 32-bit integer values

SubgroupGeMask

Decorating a variable with the SubgroupGeMask builtin decoration will make that variable contain the subgroup mask of the current subgroup invocation. The bits corresponding to the invocations greater than or equal to SubgroupLocalInvocationId through SubgroupSize-1 are set in the variable decorated with SubgroupGeMask. All other bits are set to zero.

SubgroupGeMaskKHR is an alias of SubgroupGeMask.
Valid Usage

- VUID-SubgroupGeMask-SubgroupGeMask-04372
The variable decorated with `SubgroupGeMask` must be declared using the `Input Storage Class`

- VUID-SubgroupGeMask-SubgroupGeMask-04373
The variable decorated with `SubgroupGeMask` must be declared as a four-component vector of 32-bit integer values

**SubgroupGtMask**

Decorating a variable with the `SubgroupGtMask` builtin decoration will make that variable contain the `subgroup mask` of the current subgroup invocation. The bits corresponding to the invocations greater than `SubgroupLocalInvocationId` through `SubgroupSize-1` are set in the variable decorated with `SubgroupGtMask`. All other bits are set to zero.

`SubgroupGtMaskKHR` is an alias of `SubgroupGtMask`.

Valid Usage

- VUID-SubgroupGtMask-SubgroupGtMask-04374
The variable decorated with `SubgroupGtMask` must be declared using the `Input Storage Class`

- VUID-SubgroupGtMask-SubgroupGtMask-04375
The variable decorated with `SubgroupGtMask` must be declared as a four-component vector of 32-bit integer values

**SubgroupLeMask**

Decorating a variable with the `SubgroupLeMask` builtin decoration will make that variable contain the `subgroup mask` of the current subgroup invocation. The bits corresponding to the invocations less than or equal to `SubgroupLocalInvocationId` are set in the variable decorated with `SubgroupLeMask`. All other bits are set to zero.

`SubgroupLeMaskKHR` is an alias of `SubgroupLeMask`.

Valid Usage

- VUID-SubgroupLeMask-SubgroupLeMask-04376
The variable decorated with `SubgroupLeMask` must be declared using the `Input Storage Class`

- VUID-SubgroupLeMask-SubgroupLeMask-04377
The variable decorated with `SubgroupLeMask` must be declared as a four-component vector of 32-bit integer values
SubgroupLtMask

Decorating a variable with the SubgroupLtMask builtin decoration will make that variable contain the subgroup mask of the current subgroup invocation. The bits corresponding to the invocations less than SubgroupLocalInvocationId are set in the variable decorated with SubgroupLtMask. All other bits are set to zero.

SubgroupLtMaskKHR is an alias of SubgroupLtMask.

Valid Usage

• VUID-SubgroupLtMask-SubgroupLtMask-04378
  The variable decorated with SubgroupLtMask must be declared using the Input Storage Class

• VUID-SubgroupLtMask-SubgroupLtMask-04379
  The variable decorated with SubgroupLtMask must be declared as a four-component vector of 32-bit integer values

SubgroupLocalInvocationId

Decorating a variable with the SubgroupLocalInvocationId builtin decoration will make that variable contain the index of the invocation within the subgroup. This variable is in range [0,SubgroupSize-1).

If VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT is specified, or if module declares SPIR-V version 1.6 or higher, and the local workgroup size in the X dimension of the stage is a multiple of SubgroupSize, full subgroups are enabled for that pipeline stage. When full subgroups are enabled, subgroups must be launched with all invocations active, i.e., there is an active invocation with SubgroupLocalInvocationId for each value in range [0,SubgroupSize-1).

Note

There is no direct relationship between SubgroupLocalInvocationId and LocalInvocationId or LocalInvocationIndex. If the pipeline was created with full subgroups applications can compute their own local invocation index to serve the same purpose:

\[
\text{index} = \text{SubgroupLocalInvocationId} + \text{SubgroupId} \times \text{SubgroupSize}
\]

If full subgroups are not enabled, some subgroups may be dispatched with inactive invocations that do not correspond to a local workgroup invocation, making the value of index unreliable.

Note

VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT is effectively deprecated when compiling SPIR-V 1.6 shaders, as this behavior is the default for Vulkan with SPIR-V 1.6. This is more aligned with developer expectations, and avoids applications unexpectedly breaking in the future.
Valid Usage

- VUID-SubgroupLocalInvocationId-SubgroupLocalInvocationId-04380
  The variable decorated with `SubgroupLocalInvocationId` must be declared using the Input Storage Class
- VUID-SubgroupLocalInvocationId-SubgroupLocalInvocationId-04381
  The variable decorated with `SubgroupLocalInvocationId` must be declared as a scalar 32-bit integer value

SubgroupSize

Decorating a variable with the `SubgroupSize` builtin decoration will make that variable contain the implementation-dependent number of invocations in a subgroup. This value must be a power-of-two integer.

If the pipeline was created with the `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT` flag set, or the SPIR-V module is at least version 1.6, the `SubgroupSize` decorated variable will contain the subgroup size for each subgroup that gets dispatched. This value must be between `minSubgroupSize` and `maxSubgroupSize` and must be uniform with subgroup scope. The value may vary across a single draw call, and for fragment shaders may vary across a single primitive. In compute dispatches, `SubgroupSize` must be uniform with command scope.

If the pipeline was created with a chained `VkPipelineShaderStageRequiredSubgroupSizeCreateInfo` structure, the `SubgroupSize` decorated variable will match `requiredSubgroupSize`.

If the pipeline stage SPIR-V module is less than version 1.6 and was not created with the `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT` flag set and no `VkPipelineShaderStageRequiredSubgroupSizeCreateInfo` structure was chained, the variable decorated with `SubgroupSize` will match `subgroupSize`.

The maximum number of invocations that an implementation can support per subgroup is 128.

Note

The old behavior for `SubgroupSize` is considered deprecated as certain compute algorithms cannot be easily implemented without the guarantees of `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT` and `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT`.

Valid Usage

- VUID-SubgroupSize-SubgroupSize-04382
  The variable decorated with `SubgroupSize` must be declared using the Input Storage Class
- VUID-SubgroupSize-SubgroupSize-04383
  The variable decorated with `SubgroupSize` must be declared as a scalar 32-bit integer value
**TessCoord**

Decorating a variable with the `TessCoord` built-in decoration will make that variable contain the three-dimensional $\{u,v,w\}$ barycentric coordinate of the tessellated vertex within the patch. $u$, $v$, and $w$ are in the range $[0,1]$ and vary linearly across the primitive being subdivided. For the tessellation modes of `Quads` or `IsoLines`, the third component is always zero.

### Valid Usage

- **VUID-TessCoord-TessCoord-04387**
  The `TessCoord` decoration **must** be used only within the **TessellationEvaluation Execution Model**

- **VUID-TessCoord-TessCoord-04388**
  The variable decorated with `TessCoord` **must** be declared using the **Input Storage Class**

- **VUID-TessCoord-TessCoord-04389**
  The variable decorated with `TessCoord` **must** be declared as a three-component vector of 32-bit floating-point values

**TessLevelOuter**

Decorating a variable with the `TessLevelOuter` built-in decoration will make that variable contain the outer tessellation levels for the current patch.

In tessellation control shaders, the variable decorated with `TessLevelOuter` **can** be written to, controlling the tessellation factors for the resulting patch. These values are used by the tessellator to control primitive tessellation and **can** be read by tessellation evaluation shaders.

In tessellation evaluation shaders, the variable decorated with `TessLevelOuter` **can** read the values written by the tessellation control shader.

### Valid Usage

- **VUID-TessLevelOuter-TessLevelOuter-04390**
  The `TessLevelOuter` decoration **must** be used only within the **TessellationControl or TessellationEvaluation Execution Model**

- **VUID-TessLevelOuter-TessLevelOuter-04391**
  The variable decorated with `TessLevelOuter` within the **TessellationControl Execution Model** **must** be declared using the **Output Storage Class**

- **VUID-TessLevelOuter-TessLevelOuter-04392**
  The variable decorated with `TessLevelOuter` within the **TessellationEvaluation Execution Model** **must** be declared using the **Input Storage Class**

- **VUID-TessLevelOuter-TessLevelOuter-04393**
  The variable decorated with `TessLevelOuter` **must** be declared as an array of size four, containing 32-bit floating-point values
**TessLevelInner**

Decorating a variable with the `TessLevelInner` built-in decoration will make that variable contain the inner tessellation levels for the current patch.

In tessellation control shaders, the variable decorated with `TessLevelInner` can be written to, controlling the tessellation factors for the resulting patch. These values are used by the tessellator to control primitive tessellation and can be read by tessellation evaluation shaders.

In tessellation evaluation shaders, the variable decorated with `TessLevelInner` can read the values written by the tessellation control shader.

---

**Valid Usage**

- VUID-TessLevelInner-TessLevelInner-04394
  The `TessLevelInner` decoration must be used only within the `TessellationControl` or `TessellationEvaluation Execution Model`.

- VUID-TessLevelInner-TessLevelInner-04395
  The variable decorated with `TessLevelInner` within the `TessellationControl Execution Model` must be declared using the `Output Storage Class`.

- VUID-TessLevelInner-TessLevelInner-04396
  The variable decorated with `TessLevelInner` within the `TessellationEvaluation Execution Model` must be declared using the `Input Storage Class`.

- VUID-TessLevelInner-TessLevelInner-04397
  The variable decorated with `TessLevelInner` must be declared as an array of size two, containing 32-bit floating-point values.

---

**VertexIndex**

Decorating a variable with the `VertexIndex` built-in decoration will make that variable contain the index of the vertex that is being processed by the current vertex shader invocation. For non-indexed draws, this variable begins at the `firstVertex` parameter to `vkCmdDraw` or the `firstVertex` member of a structure consumed by `vkCmdDrawIndirect` and increments by one for each vertex in the draw. For indexed draws, its value is the content of the index buffer for the vertex plus the `vertexOffset` parameter to `vkCmdDrawIndexed` or the `vertexOffset` member of the structure consumed by `vkCmdDrawIndexedIndirect`.

**Note**

`VertexIndex` starts at the same starting value for each instance.

---

**Valid Usage**

- VUID-VertexIndex-VertexIndex-04398
  The `VertexIndex` decoration must be used only within the `Vertex Execution Model`.

- VUID-VertexIndex-VertexIndex-04399
  The variable decorated with `VertexIndex` must be declared using the `Input Storage Class`.
The variable decorated with `VertexIndex` must be declared as a scalar 32-bit integer value.

ViewIndex

The `ViewIndex` decoration can be applied to a shader input which will be filled with the index of the view that is being processed by the current shader invocation.

If multiview is enabled in the render pass, this value will be one of the bits set in the view mask of the subpass the pipeline is compiled against. If multiview is not enabled in the render pass, this value will be zero.

Valid Usage

- `VUID-ViewIndex-ViewIndex-04401`
  The `ViewIndex` decoration must be used only within the `MeshEXT`, `Vertex`, `Geometry`, `TessellationControl`, `TessellationEvaluation` or `Fragment Execution Model`.

- `VUID-ViewIndex-ViewIndex-04402`
  The variable decorated with `ViewIndex` must be declared using the `Input Storage Class`.

- `VUID-ViewIndex-ViewIndex-04403`
  The variable decorated with `ViewIndex` must be declared as a scalar 32-bit integer value.

ViewportIndex

Decorating a variable with the `ViewportIndex` built-in decoration will make that variable contain the index of the viewport.

In a vertex, tessellation evaluation, or geometry shader, the variable decorated with `ViewportIndex` can be written to with the viewport index to which the primitive produced by that shader will be directed.

The selected viewport index is used to select the viewport transform and scissor rectangle.

The last active pre-rasterization shader stage (in pipeline order) controls the `ViewportIndex` that is used. Outputs in previous shader stages are not used, even if the last stage fails to write the `ViewportIndex`.

If the last active pre-rasterization shader stage shader entry point's interface does not include a variable decorated with `ViewportIndex` then the first viewport is used. If a pre-rasterization shader stage shader entry point's interface includes a variable decorated with `ViewportIndex`, it must write the same value to `ViewportIndex` for all output vertices of a given primitive.

In a fragment shader, the variable decorated with `ViewportIndex` contains the viewport index of the primitive that the fragment invocation belongs to.

Valid Usage

- `VUID-ViewportIndex-ViewportIndex-04404`
The `ViewportIndex` decoration **must** be used only within the MeshEXT, MeshNV, Vertex, TessellationEvaluation, Geometry, or Fragment Execution Model.

- **VUID-ViewportIndex-ViewportIndex-04405**
  If the `shaderOutputViewportIndex` feature is not enabled then the `ViewportIndex` decoration **must** be used only within the Geometry or Fragment Execution Model.

- **VUID-ViewportIndex-ViewportIndex-04406**
  The variable decorated with `ViewportIndex` within the MeshEXT, MeshNV, Vertex, TessellationEvaluation, or Geometry Execution Model **must** be declared using the Output Storage Class.

- **VUID-ViewportIndex-ViewportIndex-04407**
  The variable decorated with `ViewportIndex` within the Fragment Execution Model **must** be declared using the Input Storage Class.

- **VUID-ViewportIndex-ViewportIndex-04408**
  The variable decorated with `ViewportIndex` **must** be declared as a scalar 32-bit integer value.

- **VUID-ViewportIndex-ViewportIndex-07060**
  The variable decorated with `ViewportIndex` within the MeshEXT Execution Model **must** also be decorated with the `PerPrimitiveEXT` decoration.

---

**WorkgroupId**

Decorating a variable with the `WorkgroupId` built-in decoration will make that variable contain the global workgroup that the current invocation is a member of. Each component ranges from a base value to a base + count value, based on the parameters passed into the dispatching commands.

---

**Valid Usage**

- **VUID-WorkgroupId-WorkgroupId-04422**
  The `WorkgroupId` decoration **must** be used only within the GLCompute, MeshEXT, TaskEXT, MeshNV, or TaskNV Execution Model.

- **VUID-WorkgroupId-WorkgroupId-04423**
  The variable decorated with `WorkgroupId` **must** be declared using the Input Storage Class.

- **VUID-WorkgroupId-WorkgroupId-04424**
  The variable decorated with `WorkgroupId` **must** be declared as a three-component vector of 32-bit integer values.

---

**WorkgroupSize**

*Note*

SPIR-V 1.6 deprecated `WorkgroupSize` in favor of using the `LocalSizeId` Execution Mode instead. Support for `LocalSizeId` was added with `VK_KHR_maintenance4` and promoted to core in Version 1.3.
Decorating an object with the `WorkgroupSize` built-in decoration will make that object contain the dimensions of a local workgroup. If an object is decorated with the `WorkgroupSize` decoration, this takes precedence over any `LocalSize` or `LocalSizeId` execution mode.

Valid Usage

- **VUID-WorkgroupSize-WorkgroupSize-04425**
  The `WorkgroupSize` decoration must be used only within the `GLCompute`, `MeshEXT`, `TaskEXT`, `MeshNV`, or `TaskNV Execution Model`

- **VUID-WorkgroupSize-WorkgroupSize-04426**
  The variable decorated with `WorkgroupSize` must be a specialization constant or a constant

- **VUID-WorkgroupSize-WorkgroupSize-04427**
  The variable decorated with `WorkgroupSize` must be declared as a three-component vector of 32-bit integer values
Chapter 16. Image Operations

16.1. Image Operations Overview

Vulkan Image Operations are operations performed by those SPIR-V Image Instructions which take an `OpTypeImage` (representing a `VkImageView`) or `OpTypeSampledImage` (representing a `(VkImageView, VkSampler)` pair). Read, write, and atomic operations also take texel coordinates as operands, and return a value based on a neighborhood of texture elements (texels) within the image. Query operations return properties of the bound image or of the lookup itself. The “Depth” operand of `OpTypeImage` is ignored.

Note
Texel is a term which is a combination of the words texture and element. Early interactive computer graphics supported texture operations on textures, a small subset of the image operations on images described here. The discrete samples remain essentially equivalent, however, so we retain the historical term texel to refer to them.

Image Operations include the functionality of the following SPIR-V Image Instructions:

- `OpImageSample*` and `OpImageSparseSample*` read one or more neighboring texels of the image, and filter the texel values based on the state of the sampler.
  - Instructions with `ImplicitLod` in the name determine the LOD used in the sampling operation based on the coordinates used in neighboring fragments.
  - Instructions with `ExplicitLod` in the name determine the LOD used in the sampling operation based on additional coordinates.
  - Instructions with `Proj` in the name apply homogeneous projection to the coordinates.
- `OpImageFetch` and `OpImageSparseFetch` return a single texel of the image. No sampler is used.
- `OpImage*Gather` and `OpImageSparse*Gather` read neighboring texels and return a single component of each.
- `OpImageRead` (and `OpImageSparseRead`) and `OpImageWrite` read and write, respectively, a texel in the image. No sampler is used.
- `OpImage*Dref*` instructions apply depth comparison on the texel values.
- `OpImageSparse*` instructions additionally return a sparse residency code.
- `OpImageQueryLod` returns the lod parameters that would be used in a sample operation. The actual operation is not performed.

16.1.1. Texel Coordinate Systems

Images are addressed by texel coordinates. There are three texel coordinate systems:
• normalized texel coordinates [0.0, 1.0]
• unnormalized texel coordinates [0.0, width / height / depth)
• integer texel coordinates [0, width / height / depth)

instructions use integer texel coordinates.

Other image instructions can use either normalized or unnormalized texel coordinates (selected by the `unnormalizedCoordinates` state of the sampler used in the instruction), but there are limitations on what operations, image state, and sampler state is supported. Normalized coordinates are logically converted to unnormalized as part of image operations, and certain steps are only performed on normalized coordinates. The array layer coordinate is always treated as unnormalized even when other coordinates are normalized.

Normalized texel coordinates are referred to as `(s,t,r,q,a)`, with the coordinates having the following meanings:

- s: Coordinate in the first dimension of an image.
- t: Coordinate in the second dimension of an image.
- r: Coordinate in the third dimension of an image.
  - (s,t,r) are interpreted as a direction vector for Cube images.
- q: Fourth coordinate, for homogeneous (projective) coordinates.
- a: Coordinate for array layer.

The coordinates are extracted from the SPIR-V operand based on the dimensionality of the image variable and type of instruction. For Proj instructions, the components are in order `(s, [t], [r] q)`, with `t` and `r` being conditionally present based on the `Dim` of the image. For non-Proj instructions, the coordinates are `(s, [t] [r] [a])`, with `t` and `r` being conditionally present based on the `Dim` of the image and `a` being conditionally present based on the Arrayed property of the image. Projective image instructions are not supported on Arrayed images.

Unnormalized texel coordinates are referred to as `(u,v,w,a)`, with the coordinates having the following meanings:

- u: Coordinate in the first dimension of an image.
- v: Coordinate in the second dimension of an image.
- w: Coordinate in the third dimension of an image.
- a: Coordinate for array layer.

Only the `u` and `v` coordinates are directly extracted from the SPIR-V operand, because only 1D and 2D (non-Arrayed) dimensionalities support unnormalized coordinates. The components are in order `(u [,v])`, with `v` being conditionally present when the dimensionality is 2D. When normalized coordinates are converted to unnormalized coordinates, all four coordinates are used.

Integer texel coordinates are referred to as `(i,j,k,l,n)`, with the coordinates having the following meanings:
• i: Coordinate in the first dimension of an image.
• j: Coordinate in the second dimension of an image.
• k: Coordinate in the third dimension of an image.
• l: Coordinate for array layer.
• n: Index of the sample within the texel.

They are extracted from the SPIR-V operand in order \((i, j, k, l, n)\), with \(j\) and \(k\) conditionally present based on the \texttt{Dim} of the image, and \(l\) conditionally present based on the \texttt{Arrayed} property of the image. \(n\) is conditionally present and is taken from the \texttt{Sample} image operand.

For all coordinate types, unused coordinates are assigned a value of zero.

![Figure 3. Texel Coordinate Systems, Linear Filtering](image)

The Texel Coordinate Systems - For the example shown of an 8×4 texel two dimensional image.

- Normalized texel coordinates:
  - The \(s\) coordinate goes from 0.0 to 1.0.
  - The \(t\) coordinate goes from 0.0 to 1.0.

- Unnormalized texel coordinates:
  - The \(u\) coordinate within the range 0.0 to 8.0 is within the image, otherwise it is outside the image.
  - The \(v\) coordinate within the range 0.0 to 4.0 is within the image, otherwise it is outside the image.

- Integer texel coordinates:
  - The \(i\) coordinate within the range 0 to 7 addresses texels within the image, otherwise it is outside the image.
• The j coordinate within the range 0 to 3 addresses texels within the image, otherwise it is outside the image.

• Also shown for linear filtering:
  ◦ Given the unnormalized coordinates (u,v), the four texels selected are \(i_0j_0, i_1j_0, i_0j_1,\) and \(i_1j_1.\)
  ◦ The fractions \(\alpha\) and \(\beta.\)
  ◦ Given the offset \(\Delta_i\) and \(\Delta_j,\) the four texels selected by the offset are \(i_0j'_0, i_1j'_0, i_0j'_1,\) and \(i_1j'_1.\)

Note

For formats with reduced-resolution components, \(\Delta_i\) and \(\Delta_j\) are relative to the resolution of the highest-resolution component, and therefore may be divided by two relative to the unnormalized coordinate space of the lower-resolution components.

The Texel Coordinate Systems - For the example shown of an 8×4 texel two dimensional image.

• Texel coordinates as above. Also shown for nearest filtering:
  ◦ Given the unnormalized coordinates (u,v), the texel selected is \(ij.\)
  ◦ Given the offset \(\Delta_i\) and \(\Delta_j,\) the texel selected by the offset is \(ij'.\)

16.2. Conversion Formulas

16.2.1. RGB to Shared Exponent Conversion

An RGB color (red, green, blue) is transformed to a shared exponent color \(\text{red}_{\text{shared}}, \text{green}_{\text{shared}}, \text{blue}_{\text{shared}}, \text{exp}_{\text{shared}}\) as follows:
First, the components (red, green, blue) are clamped to (\text{red}_{\text{clamped}}, \text{green}_{\text{clamped}}, \text{blue}_{\text{clamped}}) as:

\[
\text{red}_{\text{clamped}} = \max(0, \min(\text{sharedexp}_{\text{max}}, \text{red}))
\]

\[
\text{green}_{\text{clamped}} = \max(0, \min(\text{sharedexp}_{\text{max}}, \text{green}))
\]

\[
\text{blue}_{\text{clamped}} = \max(0, \min(\text{sharedexp}_{\text{max}}, \text{blue}))
\]

where:

\[
\begin{align*}
N &= 9 & \text{number of mantissa bits per component} \\
B &= 15 & \text{exponent bias} \\
E_{\text{max}} &= 31 & \text{maximum possible biased exponent value} \\
\text{sharedexp}_{\text{max}} &= \frac{(2^N - 1)}{2^N} \times 2^{(E_{\text{max}} - B)}
\end{align*}
\]

\[\text{Note}\] NaN, if supported, is handled as in IEEE 754-2008 \text{minNum() and maxNum().} This results in any NaN being mapped to zero.

The largest clamped component, \text{max}_{\text{clamped}} is determined:

\[
\text{max}_{\text{clamped}} = \max(\text{red}_{\text{clamped}}, \text{green}_{\text{clamped}}, \text{blue}_{\text{clamped}})
\]

A preliminary shared exponent \text{exp'} is computed:

\[
\text{exp'} = \begin{cases} 
\lfloor \log_2(\text{max}_{\text{clamped}}) \rfloor + (B + 1) & \text{for } \text{max}_{\text{clamped}} > 2^{-(B + 1)} \\
0 & \text{for } \text{max}_{\text{clamped}} \leq 2^{-(B + 1)}
\end{cases}
\]

The shared exponent \text{exp}_{\text{shared}} is computed:

\[
\text{max}_{\text{shared}} = \left[ \frac{\text{max}_{\text{clamped}}}{2^{(\text{exp'} - B - N)}} + \frac{1}{2} \right]
\]

\[
\text{exp}_{\text{shared}} = \begin{cases} 
\text{exp'} & \text{for } 0 \leq \text{max}_{\text{shared}} < 2^N \\
\text{exp'} + 1 & \text{for } \text{max}_{\text{shared}} = 2^N
\end{cases}
\]

Finally, three integer values in the range 0 to \(2^N\) are computed:
16.2.2. Shared Exponent to RGB

A shared exponent color \((\text{red}_{\text{shared}}, \text{green}_{\text{shared}}, \text{blue}_{\text{shared}}, \text{exp}_{\text{shared}})\) is transformed to an RGB color \((\text{red}, \text{green}, \text{blue})\) as follows:

\[
\begin{align*}
\text{red}_{\text{shared}} &= \left[ \frac{\text{red}_{\text{clamped}}}{2^{(\text{exp}_{\text{shared}} - B - N)}} + \frac{1}{2} \right] \\
\text{green}_{\text{shared}} &= \left[ \frac{\text{green}_{\text{clamped}}}{2^{(\text{exp}_{\text{shared}} - B - N)}} + \frac{1}{2} \right] \\
\text{blue}_{\text{shared}} &= \left[ \frac{\text{blue}_{\text{clamped}}}{2^{(\text{exp}_{\text{shared}} - B - N)}} + \frac{1}{2} \right]
\end{align*}
\]

where:

\(N = 9\) (number of mantissa bits per component)

\(B = 15\) (exponent bias)

16.3. Texel Input Operations

Texel input instructions are SPIR-V image instructions that read from an image. Texel input operations are a set of steps that are performed on state, coordinates, and texel values while processing a texel input instruction, and which are common to some or all texel input instructions. They include the following steps, which are performed in the listed order:

- Validation operations
  - Instruction/Sampler/Image validation
  - Coordinate validation
  - Sparse validation
  - Layout validation
- Format conversion
- Texel replacement
- Depth comparison
• Conversion to RGBA
• Component swizzle
• Chroma reconstruction
• Y’CₙCₔ conversion

For texel input instructions involving multiple texels (for sampling or gathering), these steps are applied for each texel that is used in the instruction. Depending on the type of image instruction, other steps are conditionally performed between these steps or involving multiple coordinate or texel values.

If Chroma Reconstruction is implicit, Texel Filtering instead takes place during chroma reconstruction, before sampler Y’CₙCₔ conversion occurs.

16.3.1. Texel Input Validation Operations

Texel input validation operations inspect instruction/image/sampler state or coordinates, and in certain circumstances cause the texel value to be replaced or become undefined. There are a series of validations that the texel undergoes.

Instruction/Sampler/Image View Validation

There are a number of cases where a SPIR-V instruction can mismatch with the sampler, the image view, or both, and a number of further cases where the sampler can mismatch with the image view. In such cases the value of the texel returned is undefined.

These cases include:

• The sampler borderColor is an integer type and the image view format is not one of the VkFormat integer types or a stencil component of a depth/stencil format.
• The sampler borderColor is a float type and the image view format is not one of the VkFormat float types or a depth component of a depth/stencil format.
• The sampler borderColor is one of the opaque black colors (VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK or VK_BORDER_COLOR_INT_OPAQUE_BLACK) and the image view VkComponentSwizzle for any of the VkComponentMapping components is not the identity swizzle.
• The VkImageLayout of any subresource in the image view does not match the VkDescriptorImageInfo::imageLayout used to write the image descriptor.
• The SPIR-V Image Format is not compatible with the image view's format.
• The sampler unnormalizedCoordinates is VK_TRUE and any of the limitations of unnormalized coordinates are violated.
• The SPIR-V instruction is one of the OpImage*Dref* instructions and the sampler compareEnable is VK_FALSE
• The SPIR-V instruction is not one of the OpImage*Dref* instructions and the sampler compareEnable is VK_TRUE
• The SPIR-V instruction is one of the OpImage*Dref* instructions, the image view format is one of the depth/stencil formats, and the image view aspect is not VK_IMAGE_ASPECT_DEPTH_BIT.
• The SPIR-V instruction’s image variable’s properties are not compatible with the image view:
  ◦ Rules for viewType:
    ▪ VK_IMAGE_VIEW_TYPE_1D must have Dim = 1D, Arrayed = 0, MS = 0.
    ▪ VK_IMAGE_VIEW_TYPE_2D must have Dim = 2D, Arrayed = 0.
    ▪ VK_IMAGE_VIEW_TYPE_3D must have Dim = 3D, Arrayed = 0, MS = 0.
    ▪ VK_IMAGE_VIEW_TYPE_CUBE must have Dim = Cube, Arrayed = 0, MS = 0.
    ▪ VK_IMAGE_VIEW_TYPE_1D_ARRAY must have Dim = 1D, Arrayed = 1, MS = 0.
    ▪ VK_IMAGE_VIEW_TYPE_2D_ARRAY must have Dim = 2D, Arrayed = 1.
    ▪ VK_IMAGE_VIEW_TYPE_CUBE_ARRAY must have Dim = Cube, Arrayed = 1, MS = 0.
  ◦ If the image was created with VkImageCreateInfo::samples equal to VK_SAMPLE_COUNT_1_BIT, the instruction must have MS = 0.
  ◦ If the image was created with VkImageCreateInfo::samples not equal to VK_SAMPLE_COUNT_1_BIT, the instruction must have MS = 1.
  ◦ If the Sampled Type of the OpTypeImage does not match the numeric format of the image, as shown in the SPIR-V Sampled Type column of the Interpretation of Numeric Format table.
  ◦ If the signedness of any read or sample operation does not match the signedness of the image's format.

Only OpImageSample* and OpImageSparseSample* can be used with a sampler or image view that enables sampler Y'CbCr conversion.

OpImageFetch, OpImageSparseFetch, OpImage*Gather, and OpImageSparse*Gather must not be used with a sampler or image view that enables sampler Y'CbCr conversion.

The ConstOffset and Offset operands must not be used with a sampler or image view that enables sampler Y'CbCr conversion.

**Integer Texel Coordinate Validation**

Integer texel coordinates are validated against the size of the image level, and the number of layers and number of samples in the image. For SPIR-V instructions that use integer texel coordinates, this is performed directly on the integer coordinates. For instructions that use normalized or unnormalized texel coordinates, this is performed on the coordinates that result after conversion to integer texel coordinates.

If the integer texel coordinates do not satisfy all of the conditions

\[
0 \leq i < w_s
\]

\[
0 \leq j < h_s
\]
0 ≤ k < \text{d}_s

0 ≤ l < \text{layers}

0 ≤ n < \text{samples}

where:

\text{w}_s = \text{width of the image level}

\text{h}_s = \text{height of the image level}

\text{d}_s = \text{depth of the image level}

\text{layers} = \text{number of layers in the image}

\text{samples} = \text{number of samples per texel in the image}

then the texel fails integer texel coordinate validation.

There are four cases to consider:

1. Valid Texel Coordinates
   - If the texel coordinates pass validation (that is, the coordinates lie within the image),
     then the texel value comes from the value in image memory.

2. Border Texel
   - If the texel coordinates fail validation, and
   - If the read is the result of an image sample instruction or image gather instruction, and
   - If the image is not a cube image,
     then the texel is a border texel and \textit{texel replacement} is performed.

3. Invalid Texel
   - If the texel coordinates fail validation, and
   - If the read is the result of an image fetch instruction, image read instruction, or atomic instruction,
     then the texel is an invalid texel and \textit{texel replacement} is performed.
4. Cube Map Edge or Corner

Otherwise the texel coordinates lie beyond the edges or corners of the selected cube map face, and Cube map edge handling is performed.

**Cube Map Edge Handling**

If the texel coordinates lie beyond the edges or corners of the selected cube map face (as described in the prior section), the following steps are performed. Note that this does not occur when using VK_FILTER_NEAREST filtering within a mip level, since VK_FILTER_NEAREST is treated as using VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE.

- Cube Map Edge Texel
  - If the texel lies beyond the selected cube map face in either only i or only j, then the coordinates (i,j) and the array layer l are transformed to select the adjacent texel from the appropriate neighboring face.

- Cube Map Corner Texel
  - If the texel lies beyond the selected cube map face in both i and j, then there is no unique neighboring face from which to read that texel. The texel should be replaced by the average of the three values of the adjacent texels in each incident face. However, implementations may replace the cube map corner texel by other methods. The methods are subject to the constraint that if the three available texels have the same value, the resulting filtered texel must have that value.

**Sparse Validation**

If the texel reads from an unbound region of a sparse image, the texel is a sparse unbound texel, and processing continues with texel replacement.

**Layout Validation**

If all planes of a disjoint multi-planar image are not in the same image layout, the image must not be sampled with sampler YCbCr conversion enabled.

**16.3.2. Format Conversion**

Texels undergo a format conversion from the VkFormat of the image view to a vector of either floating point or signed or unsigned integer components, with the number of components based on the number of components present in the format.

- Color formats have one, two, three, or four components, according to the format.
- Depth/stencil formats are one component. The depth or stencil component is selected by the aspectMask of the image view.

Each component is converted based on its type and size (as defined in the Format Definition section for each VkFormat), using the appropriate equations in 16-Bit Floating-Point Numbers, Unsigned 11-Bit Floating-Point Numbers, Unsigned 10-Bit Floating-Point Numbers, Fixed-Point Data Conversion, and Shared Exponent to RGB. Signed integer components smaller than 32 bits are sign-
extended.

If the image view format is sRGB, the color components are first converted as if they are UNORM, and then sRGB to linear conversion is applied to the R, G, and B components as described in the “sRGB EOTF” section of the Khronos Data Format Specification. The A component, if present, is unchanged.

If the image view format is block-compressed, then the texel value is first decoded, then converted based on the type and number of components defined by the compressed format.

### 16.3.3. Texel Replacement

A texel is replaced if it is one (and only one) of:

- a border texel,
- an invalid texel, or
- a sparse unbound texel.

Border texels are replaced with a value based on the image format and the `borderColor` of the sampler. The border color is:

#### Table 14. Border Color B

<table>
<thead>
<tr>
<th>Sampler</th>
<th>Corresponding Border Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK</td>
<td>([B_r, B_g, B_b, B_a] = [0.0, 0.0, 0.0, 0.0])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK</td>
<td>([B_r, B_g, B_b, B_a] = [0.0, 0.0, 0.0, 1.0])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE</td>
<td>([B_r, B_g, B_b, B_a] = [1.0, 1.0, 1.0, 1.0])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_INT_TRANSPARENT_BLACK</td>
<td>([B_r, B_g, B_b, B_a] = [0, 0, 0, 0])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_INT_OPAQUE_BLACK</td>
<td>([B_r, B_g, B_b, B_a] = [0, 0, 0, 1])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_INT_OPAQUE_WHITE</td>
<td>([B_r, B_g, B_b, B_a] = [1, 1, 1, 1])</td>
</tr>
</tbody>
</table>

**Note**

The names `VK_BORDER_COLOR_*_TRANSPARENT_BLACK`, `VK_BORDER_COLOR_*_OPAQUE_BLACK`, and `VK_BORDER_COLOR_*_OPAQUE_WHITE` are meant to describe which components are zeros and ones in the vocabulary of compositing, and are not meant to imply that the numerical value of `VK_BORDER_COLOR_INT_OPAQUE_WHITE` is a saturating value for integers.

This is substituted for the texel value by replacing the number of components in the image format.

#### Table 15. Border Texel Components After Replacement

<table>
<thead>
<tr>
<th>Texel Aspect or Format</th>
<th>Component Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth aspect</td>
<td>(D = B_r)</td>
</tr>
<tr>
<td>Stencil aspect</td>
<td>(S = B_r)</td>
</tr>
</tbody>
</table>
### Texel Aspect or Format

<table>
<thead>
<tr>
<th>Texel Aspect or Format</th>
<th>Component Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>One component color format</td>
<td>( Color_r = B_r )</td>
</tr>
<tr>
<td>Two component color format</td>
<td>( [Color_r, Color_g] = [B_r, B_g] )</td>
</tr>
<tr>
<td>Three component color format</td>
<td>( [Color_r, Color_g, Color_b] = [B_r, B_g, B_b] )</td>
</tr>
<tr>
<td>Four component color format</td>
<td>( [Color_r, Color_g, Color_b, Color_a] = [B_r, B_g, B_b, B_a] )</td>
</tr>
</tbody>
</table>

The value returned by a read of an invalid texel is undefined, unless that read operation is from a buffer resource and the `robustBufferAccess` feature is enabled. In that case, an invalid texel is replaced as described by the `robustBufferAccess` feature. If the access is to an image resource and the x, y, z, or layer coordinate validation fails and the `robustImageAccess` feature is enabled, then zero **must** be returned for the R, G, and B components, if present. Either zero or one **must** be returned for the A component, if present. If only the sample index was invalid, the values returned are undefined.

Additionally, if the `robustImageAccess` feature is enabled, any invalid texels **may** be expanded to four components prior to texel replacement. This means that components not present in the image format may be replaced with 0 or may undergo conversion to RGBA as normal.

If the `VkPhysicalDeviceSparseProperties::residencyNonResidentStrict` property is `VK_TRUE`, a sparse unbound texel is replaced with 0 or 0.0 values for integer and floating-point components of the image format, respectively.

If `residencyNonResidentStrict` is `VK_FALSE`, the value of the sparse unbound texel is undefined.

### 16.3.4. Depth Compare Operation

If the image view has a depth/stencil format, the depth component is selected by the `aspectMask`, and the operation is an `OpImage*Dref*` instruction, a depth comparison is performed. The result is 1.0 if the comparison evaluates to `true`, and 0.0 otherwise. This value replaces the depth component \( D \).

The compare operation is selected by the `VkCompareOp` value set by `VkSamplerCreateInfo::compareOp`. The reference value from the SPIR-V operand \( D_{ref} \) and the texel depth value \( D_{tex} \) are used as the `reference` and `test` values, respectively, in that operation.

If the image being sampled has an unsigned normalized fixed-point format, then \( D_{ref} \) is clamped to [0,1] before the compare operation.

### 16.3.5. Conversion to RGBA

The texel is expanded from one, two, or three components to four components based on the image base color:

*Table 16. Texel Color After Conversion To RGBA*

<table>
<thead>
<tr>
<th>Texel Aspect or Format</th>
<th>RGBA Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth aspect</td>
<td>([Color_r, Color_g, Color_b, Color_a] = [D,0,0,one])</td>
</tr>
<tr>
<td>Stencil aspect</td>
<td>([Color_r, Color_g, Color_b, Color_a] = [S,0,0,one])</td>
</tr>
</tbody>
</table>
Texel Aspect or Format | RGBA Color
--- | ---
One component color format | \([\text{Color}_r,\text{Color}_g,\text{Color}_b,\text{Color}_a] = [\text{Color}_r,0,0,\text{one}]\)
Two component color format | \([\text{Color}_r,\text{Color}_g,\text{Color}_b,\text{Color}_a] = [\text{Color}_r,\text{Color}_g,0,\text{one}]\)
Three component color format | \([\text{Color}_r,\text{Color}_g,\text{Color}_b,\text{Color}_a] = [\text{Color}_r,\text{Color}_g,\text{Color}_b,\text{one}]\)
Four component color format | \([\text{Color}_r,\text{Color}_g,\text{Color}_b,\text{Color}_a] = [\text{Color}_r,\text{Color}_g,\text{Color}_b,\text{Color}_a]\)

where one = 1.0f for floating-point formats and depth aspects, and one = 1 for integer formats and stencil aspects.

16.3.6. Component Swizzle

All texel input instructions apply a swizzle based on:

- the VkComponentSwizzle enums in the components member of the VkImageViewCreateInfo structure for the image being read if sampler Y’C₈C₆ conversion is not enabled, and
- the VkComponentSwizzle enums in the components member of the VkSamplerYcbcrConversionCreateInfo structure for the sampler Y’C₈C₆ conversion if sampler Y’C₈C₆ conversion is enabled.

The swizzle can rearrange the components of the texel, or substitute zero or one for any components. It is defined as follows for each color component:

\[
\text{Color}'_\text{component} = \begin{cases} 
\text{Color}_r & \text{for RED swizzle} \\
\text{Color}_g & \text{for GREEN swizzle} \\
\text{Color}_b & \text{for BLUE swizzle} \\
\text{Color}_a & \text{for ALPHA swizzle} \\
0 & \text{for ZERO swizzle} \\
\text{one} & \text{for ONE swizzle} \\
\text{identity} & \text{for IDENTITY swizzle} \\
\end{cases}
\]

where:

\[
\text{one} = \begin{cases} 
1.0f & \text{for floating point components} \\
1 & \text{for integer components} \\
\end{cases}
\]

\[
\text{identity} = \begin{cases} 
\text{Color}_r & \text{for component} = r \\
\text{Color}_g & \text{for component} = g \\
\text{Color}_b & \text{for component} = b \\
\text{Color}_a & \text{for component} = a \\
\end{cases}
\]

If the border color is one of the VK_BORDER_COLOR_*_OPAQUE_BLACK enums and the VkComponentSwizzle is not the identity swizzle for all components, the value of the texel after swizzle is undefined.
16.3.7. Sparse Residency

`OpImageSparse*` instructions return a structure which includes a *residency code* indicating whether any texels accessed by the instruction are sparse unbound texels. This code can be interpreted by the `OpImageSparseTexelsResident` instruction which converts the residency code to a boolean value.

16.3.8. Chroma Reconstruction

In some color models, the color representation is defined in terms of monochromatic light intensity (often called “luma”) and color differences relative to this intensity, often called “chroma”. It is common for color models other than RGB to represent the chroma components at lower spatial resolution than the luma component. This approach is used to take advantage of the eye’s lower spatial sensitivity to color compared with its sensitivity to brightness. Less commonly, the same approach is used with additive color, since the green component dominates the eye’s sensitivity to light intensity and the spatial sensitivity to color introduced by red and blue is lower.

Lower-resolution components are “downsampled” by resizing them to a lower spatial resolution than the component representing luminance. This process is also commonly known as “chroma subsampling”. There is one luminance sample in each texture texel, but each chrominance sample may be shared among several texels in one or both texture dimensions.

- “.444” formats do not spatially downsample chroma values compared with luma: there are unique chroma samples for each texel.
- “.422” formats have downsampling in the x dimension (corresponding to u or s coordinates): they are sampled at half the resolution of luma in that dimension.
- “.420” formats have downsampling in the x dimension (corresponding to u or s coordinates) and the y dimension (corresponding to v or t coordinates): they are sampled at half the resolution of luma in both dimensions.

The process of reconstructing a full color value for texture access involves accessing both chroma and luma values at the same location. To generate the color accurately, the values of the lower-resolution components at the location of the luma samples must be reconstructed from the lower-resolution sample locations, an operation known here as “chroma reconstruction” irrespective of the actual color model.

The location of the chroma samples relative to the luma coordinates is determined by the `xChromaOffset` and `yChromaOffset` members of the `VkSamplerYcbcrConversionCreateInfo` structure used to create the sampler Y’CbCr conversion.

The following diagrams show the relationship between unnormalized (u,v) coordinates and (i,j) integer texel positions in the luma component (shown in black, with circles showing integer sample positions) and the texel coordinates of reduced-resolution chroma components, shown as crosses in red.

---

**Note**

If the chroma values are reconstructed at the locations of the luma samples by means of interpolation, chroma samples from outside the image bounds are needed; these are determined according to *Wrapping Operation*. These diagrams
represent this by showing the bounds of the “chroma texel” extending beyond the image bounds, and including additional chroma sample positions where required for interpolation. The limits of a sample for NEAREST sampling is shown as a grid.

Figure 5. 422 downsampling, xChromaOffset=COSITED_EVEN

Figure 6. 422 downsampling, xChromaOffset=MIDPOINT
Figure 7. 4:2:0 downsampling, xChromaOffset=COSITED_EVEN, yChromaOffset=COSITED_EVEN

Figure 8. 4:2:0 downsampling, xChromaOffset=MIDPOINT, yChromaOffset=COSITED_EVEN
Reconstruction is implemented in one of two ways:

If the format of the image that is to be sampled sets

```
VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT,
```

or the

```
VkSamplerYcbcrConversionCreateInfo's forceExplicitReconstruction is set to VK_TRUE,
```

reconstruction is performed as an explicit step independent of filtering, described in the Explicit Reconstruction section.

If the format of the image that is to be sampled does not set

```
VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT
```

and if the
VkSamplerYcbcrConversionCreateInfo’s `forceExplicitReconstruction` is set to VK_FALSE, reconstruction is performed as an implicit part of filtering prior to color model conversion, with no separate post-conversion texel filtering step, as described in the Implicit Reconstruction section.

Explicit Reconstruction

- If the `chromaFilter` member of the VkSamplerYcbcrConversionCreateInfo structure is VK_FILTER_NEAREST:
  - If the format’s R and B components are reduced in resolution in just width by a factor of two relative to the G component (i.e. this is a “_422” format), the \( \tau_{ijk}[level] \) values accessed by texel filtering are reconstructed as follows:
    \[
    \begin{align*}
    \tau_R'(i, j) &= \tau_R([i \times 0.5], [j])[level] \\
    \tau_B'(i, j) &= \tau_B([i \times 0.5], [j])[level]
    \end{align*}
    \]
  - If the format’s R and B components are reduced in resolution in width and height by a factor of two relative to the G component (i.e. this is a “_420” format), the \( \tau_{ijk}[level] \) values accessed by texel filtering are reconstructed as follows:
    \[
    \begin{align*}
    \tau_R'(i, j) &= \tau_R([i \times 0.5], [j \times 0.5])[level] \\
    \tau_B'(i, j) &= \tau_B([i \times 0.5], [j \times 0.5])[level]
    \end{align*}
    \]

  Note
  xChromaOffset and yChromaOffset have no effect if chromaFilter is VK_FILTER_NEAREST for explicit reconstruction.

- If the `chromaFilter` member of the VkSamplerYcbcrConversionCreateInfo structure is VK_FILTER_LINEAR:
  - If the format’s R and B components are reduced in resolution in just width by a factor of two relative to the G component (i.e. this is a “_422” format):
    - If xChromaOffset is VK_CHROMA_LOCATION_COSITED_EVEN:
      \[
      \tau_{RB}'(i, j) = \begin{cases} 
      \tau_{RB}([i \times 0.5], [j])[level], & 0.5 \times i = [0.5 \times i] \\
      0.5 \times \tau_{RB}([i \times 0.5], [j])[level] + \\
      0.5 \times \tau_{RB}([i \times 0.5] + 1, [j])[level], & 0.5 \times i \neq [0.5 \times i]
      \end{cases}
      \]
    - If xChromaOffset is VK_CHROMA_LOCATION_MIDPOINT:
      \[
      \tau_{RB}'(i, j) = \begin{cases} 
      0.25 \times \tau_{RB}([i \times 0.5] - 1, [j])[level] + \\
      0.75 \times \tau_{RB}([i \times 0.5], [j])[level], & 0.5 \times i = [0.5 \times i] \\
      0.75 \times \tau_{RB}([i \times 0.5], [j])[level] + \\
      0.25 \times \tau_{RB}([i \times 0.5] + 1, [j])[level], & 0.5 \times i \neq [0.5 \times i]
      \end{cases}
      \]
    - If the format’s R and B components are reduced in resolution in width and height by a factor of two relative to the G component (i.e. this is a “_420” format), a similar relationship applies.
      Due to the number of options, these formulae are expressed more concisely as follows:
\[
i_{RB} = \begin{cases} 
0.5 \times (i) & \text{xChromaOffset=COSITED\_EVEN} \\
0.5 \times (i - 0.5) & \text{xChromaOffset=MIDPOINT}
\end{cases}
\]
\[
j_{RB} = \begin{cases} 
0.5 \times (j) & \text{yChromaOffset=COSITED\_EVEN} \\
0.5 \times (j - 0.5) & \text{yChromaOffset=MIDPOINT}
\end{cases}
\]
\[
i_{\text{floor}} = \lfloor i_{RB} \rfloor \\
j_{\text{floor}} = \lfloor j_{RB} \rfloor \\
i_{\text{frac}} = i_{RB} - i_{\text{floor}} \\
j_{\text{frac}} = j_{RB} - j_{\text{floor}}
\]
\[
\tau_{RB}^\prime(i, j) = \begin{cases} 
\tau_{RB}(i_{\text{floor}}, j_{\text{floor}})[\text{level}] & \times (1 - i_{\text{frac}}) & \times (1 - j_{\text{frac}}) +
\tau_{RB}(1 + i_{\text{floor}}, j_{\text{floor}})[\text{level}] & \times (i_{\text{frac}}) & \times (1 - j_{\text{frac}}) +
\tau_{RB}(i_{\text{floor}}, 1 + j_{\text{floor}})[\text{level}] & \times (1 - i_{\text{frac}}) & \times (j_{\text{frac}}) +
\tau_{RB}(1 + i_{\text{floor}}, 1 + j_{\text{floor}})[\text{level}] & \times (i_{\text{frac}}) & \times (j_{\text{frac}})
\end{cases}
\]

Note

In the case where the texture itself is bilinearly interpolated as described in Texel Filtering, thus requiring four full-color samples for the filtering operation, and where the reconstruction of these samples uses bilinear interpolation in the chroma components due to \text{chromaFilter}=VK\_FILTER\_LINEAR, up to nine chroma samples may be required, depending on the sample location.

Implicit Reconstruction

Implicit reconstruction takes place by the samples being interpolated, as required by the filter settings of the sampler, except that \text{chromaFilter} takes precedence for the chroma samples.

If \text{chromaFilter} is VK\_FILTER\_NEAREST, an implementation \text{may} behave as if \text{xChromaOffset} and \text{yChromaOffset} were both VK\_CHROMA\_LOCATION\_MIDPOINT, irrespective of the values set.

Note

This will not have any visible effect if the locations of the luma samples coincide with the location of the samples used for rasterization.

The sample coordinates are adjusted by the downsample factor of the component (such that, for example, the sample coordinates are divided by two if the component has a downsample factor of two relative to the luma component):

\[
u_{RB}^\prime(422 / 420) = \begin{cases} 
0.5 \times (u + 0.5), & \text{xChromaOffset=COSITED\_EVEN} \\
0.5 \times u, & \text{xChromaOffset=MIDPOINT}
\end{cases}
\]
\[
v_{RB}^\prime(420) = \begin{cases} 
0.5 \times (v + 0.5), & \text{yChromaOffset=COSITED\_EVEN} \\
0.5 \times v, & \text{yChromaOffset=MIDPOINT}
\end{cases}
\]

16.3.9. Sampler Y'CbCr Conversion

Sampler Y'CbCr conversion performs the following operations, which an implementation \text{may}
combine into a single mathematical operation:

- **Sampler Y’CₐCᵦ Range Expansion**
- **Sampler Y’CₐCᵦ Model Conversion**

### Sampler Y’CₐCᵦ Range Expansion

Sampler Y’CₐCᵦ range expansion is applied to color component values after all texel input operations which are not specific to sampler Y’CₐCᵦ conversion. For example, the input values to this stage have been converted using the normal format conversion rules.

Sampler Y’CₐCᵦ range expansion is not applied if `ycbcrModel` is `VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY`. That is, the shader receives the vector C’rgba as output by the Component Swizzle stage without further modification.

For other values of `ycbcrModel`, range expansion is applied to the texel component values output by the Component Swizzle defined by the `components` member of `VkSamplerYcbcrConversionCreateInfo`. Range expansion applies independently to each component of the image. For the purposes of range expansion and Y’CₐCᵦ model conversion, the R and B components contain color difference (chroma) values and the G component contains luma. The A component is not modified by sampler Y’CₐCᵦ range expansion.

The range expansion to be applied is defined by the `ycbcrRange` member of the `VkSamplerYcbcrConversionCreateInfo` structure:

- If `ycbcrRange` is `VK_SAMPLER_YCBCR_RANGE_ITU_FULL`, the following transformations are applied:

  \[
  Y' = C'_rgba \lfloor G \rfloor \\
  C_B = C'_rgba \lfloor B \rfloor - \frac{2^{(n-1)}}{(2^n)-1} \\
  C_R = C'_rgba \lfloor R \rfloor - \frac{2^{(n-1)}}{(2^n)-1}
  \]

  **Note**

  These formulae correspond to the “full range” encoding in the “Quantization schemes” chapter of the Khronos Data Format Specification.

  Should any future amendments be made to the ITU specifications from which these equations are derived, the formulae used by Vulkan may also be updated to maintain parity.

- If `ycbcrRange` is `VK_SAMPLER_YCBCR_RANGE_ITU_NARROW`, the following transformations are applied:
Note

These formulae correspond to the “narrow range” encoding in the “Quantization schemes” chapter of the Khronos Data Format Specification.

\[
Y' = \frac{C'_rgba[G] \times (2^n - 1) - 16 \times 2^n - 8}{219 \times 2^n - 8}
\]
\[
C_B = \frac{C'_rgba[B'] \times (2^n - 1) - 128 \times 2^n - 8}{224 \times 2^n - 8}
\]
\[
C_R = \frac{C'_rgba[R] \times (2^n - 1) - 128 \times 2^n - 8}{224 \times 2^n - 8}
\]

• \( n \) is the bit-depth of the components in the format.

The precision of the operations performed during range expansion must be at least that of the source format.

An implementation may clamp the results of these range expansion operations such that \( Y' \) falls in the range \([0,1]\), and/or such that \( C_n \) and \( C_n \) fall in the range \([-0.5,0.5]\).

**Sampler Y'CbCr Model Conversion**

The range-expanded values are converted between color models, according to the color model conversion specified in the \( \text{ycbcrModel} \) member:

**VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY**

The color components are not modified by the color model conversion since they are assumed already to represent the desired color model in which the shader is operating; \( Y'C_nC_R \) range expansion is also ignored.

**VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_IDENTITY**

The color components are not modified by the color model conversion and are assumed to be treated as though in \( Y'C_nC_R \) form both in memory and in the shader; \( Y'C_nC_R \) range expansion is applied to the components as for other \( Y'C_nC_R \) models, with the vector \((C_R,Y',C_B,A)\) provided to the shader.

**VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_709**

The color components are transformed from a \( Y'C_nC_R \) representation to an \( R'G'B' \) representation as described in the “BT.709 Y'C_nC_R conversion” section of the Khronos Data Format Specification.

**VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_601**

The color components are transformed from a \( Y'C_nC_R \) representation to an \( R'G'B' \) representation as described in the “BT.601 Y'C_nC_R conversion” section of the Khronos Data Format Specification.

**VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_2020**

The color components are transformed from a \( Y'C_nC_R \) representation to an \( R'G'B' \) representation as described in the “BT.2020 Y'C_nC_R conversion” section of the Khronos Data Format Specification.
In this operation, each output component is dependent on each input component.

An implementation may clamp the R'G'B' results of these conversions to the range [0,1].

The precision of the operations performed during model conversion must be at least that of the source format.

The alpha component is not modified by these model conversions.

Note

Sampling operations in a non-linear color space can introduce color and intensity shifts at sharp transition boundaries. To avoid this issue, the technically precise color correction sequence described in the “Introduction to Color Conversions” chapter of the Khronos Data Format Specification may be performed as follows:

- Calculate the unnormalized texel coordinates corresponding to the desired sample position.
- For a minFilter or magFilter of VK_FILTER_NEAREST:
  1. Calculate \((i,j)\) for the sample location as described under the “nearest filtering” formulae in \((u,v,w,a)\) to \((i,j,k,l,n)\) Transformation And Array Layer Selection
  2. Calculate the normalized texel coordinates corresponding to these integer coordinates.
  3. Sample using sampler Y'C_bC_r conversion at this location.
- For a minFilter or magFilter of VK_FILTER_LINEAR:
  1. Calculate \((i_a,j_b)\) for the sample location as described under the “linear filtering” formulae in \((u,v,w,a)\) to \((i,j,k,l,n)\) Transformation And Array Layer Selection
  2. Calculate the normalized texel coordinates corresponding to these integer coordinates.
  3. Sample using sampler Y'C_bC_r conversion at each of these locations.
  4. Convert the non-linear A'R'G'B' outputs of the Y'C_bC_r conversions to linear ARGB values as described in the “Transfer Functions” chapter of the Khronos Data Format Specification.
  5. Interpolate the linear ARGB values using the \(\alpha\) and \(\beta\) values described in the “linear filtering” section of \((u,v,w,a)\) to \((i,j,k,l,n)\) Transformation And Array Layer Selection and the equations in Texel Filtering.

The additional calculations and, especially, additional number of sampling operations in the VK_FILTER_LINEAR case can be expected to have a performance impact compared with using the outputs directly. Since the variations from “correct” results are subtle for most content, the application author should determine whether a more costly implementation is strictly necessary.

If chromaFilter, and minFilter or magFilter are both VK_FILTER_NEAREST, these
operations are redundant and sampling using sampler Y’C₆C₇ conversion at the desired sample coordinates will produce the “correct” results without further processing.

16.4. Texel Output Operations

Texel output instructions are SPIR-V image instructions that write to an image. Texel output operations are a set of steps that are performed on state, coordinates, and texel values while processing a texel output instruction, and which are common to some or all texel output instructions. They include the following steps, which are performed in the listed order:

- Validation operations
  - Format validation
  - Type validation
  - Coordinate validation
  - Sparse validation
- Texel output format conversion

16.4.1. Texel Output Validation Operations

Texel output validation operations inspect instruction/image state or coordinates, and in certain circumstances cause the write to have no effect. There are a series of validations that the texel undergoes.

Texel Format Validation

If the image format of the OpTypeImage is not compatible with the VkImageView’s format, the write causes the contents of the image’s memory to become undefined.

Texel Type Validation

If the Sampled Type of the OpTypeImage does not match the type defined for the format, as specified in the SPIR-V Sampled Type column of the Interpretation of Numeric Format table, the write causes the value of the texel to become undefined. For integer types, if the signedness of the access does not match the signedness of the accessed resource, the write causes the value of the texel to become undefined.

16.4.2. Integer Texel Coordinate Validation

The integer texel coordinates are validated according to the same rules as for texel input coordinate validation.

If the texel fails integer texel coordinate validation, then the write has no effect.
16.4.3. Sparse Texel Operation

If the texel attempts to write to an unbound region of a sparse image, the texel is a sparse unbound texel. In such a case, if the `VkPhysicalDeviceSparseProperties::residencyNonResidentStrict` property is `VK_TRUE`, the sparse unbound texel write has no effect. If `residencyNonResidentStrict` is `VK_FALSE`, the write may have a side effect that becomes visible to other accesses to unbound texels in any resource, but will not be visible to any device memory allocated by the application.

16.4.4. Texel Output Format Conversion

If the image format is sRGB, a linear to sRGB conversion is applied to the R, G, and B components as described in the “sRGB EOTF” section of the Khronos Data Format Specification. The A component, if present, is unchanged.

Texels then undergo a format conversion from the floating point, signed, or unsigned integer type of the texel data to the `VkFormat` of the image view. If the number of components in the texel data is larger than the number of components in the format, additional components are discarded.

Each component is converted based on its type and size (as defined in the Format Definition section for each `VkFormat`). Floating-point outputs are converted as described in Floating-Point Format Conversions and Fixed-Point Data Conversion. Integer outputs are converted such that their value is preserved. The converted value of any integer that cannot be represented in the target format is undefined.

16.5. Normalized Texel Coordinate Operations

If the image sampler instruction provides normalized texel coordinates, some of the following operations are performed.

16.5.1. Projection Operation

For Proj image operations, the normalized texel coordinates \((s, t, r, q, a)\) and (if present) the \(D_{\text{ref}}\) coordinate are transformed as follows:

\[
\begin{align*}
    s &= \frac{s}{q}, \quad \text{for 1D, 2D, or 3D image} \\
    t &= \frac{t}{q}, \quad \text{for 2D or 3D image} \\
    r &= \frac{r}{q}, \quad \text{for 3D image} \\
    D_{\text{ref}} &= \frac{D_{\text{ref}}}{q}, \quad \text{if provided}
\end{align*}
\]

16.5.2. Derivative Image Operations

Derivatives are used for LOD selection. These derivatives are either implicit (in an ImplicitLod image instruction in a fragment shader) or explicit (provided explicitly by shader to the image
instruction in any shader).

For implicit derivatives image instructions, the derivatives of texel coordinates are calculated in the same manner as derivative operations. That is:

\[
\frac{\partial s}{\partial x} = P_d x(s), \quad \frac{\partial s}{\partial y} = P_d y(s), \quad \text{for 1D, 2D, Cube, or 3D image}
\]

\[
\frac{\partial t}{\partial x} = P_d x(t), \quad \frac{\partial t}{\partial y} = P_d y(t), \quad \text{for 2D, Cube, or 3D image}
\]

\[
\frac{\partial r}{\partial x} = P_d x(r), \quad \frac{\partial r}{\partial y} = P_d y(r), \quad \text{for Cube or 3D image}
\]

Partial derivatives not defined above for certain image dimensionalities are set to zero.

For explicit LOD image instructions, if the optional SPIR-V operand Grad is provided, then the operand values are used for the derivatives. The number of components present in each derivative for a given image dimensionality matches the number of partial derivatives computed above.

If the optional SPIR-V operand Lod is provided, then derivatives are set to zero, the cube map derivative transformation is skipped, and the scale factor operation is skipped. Instead, the floating point scalar coordinate is directly assigned to \(\lambda_{\text{base}}\) as described in Level-of-Detail Operation.

If the image or sampler object used by an implicit derivative image instruction is not uniform across the quad and quadDivergentImplicitLod is not supported, then the derivative and LOD values are undefined. Implicit derivatives are well-defined when the image and sampler and control flow are uniform across the quad, even if they diverge between different quads.

If quadDivergentImplicitLod is supported, then derivatives and implicit LOD values are well-defined even if the image or sampler object are not uniform within a quad. The derivatives are computed as specified above, and the implicit LOD calculation proceeds for each shader invocation using its respective image and sampler object.

### 16.5.3. Cube Map Face Selection and Transformations

For cube map image instructions, the \((s,t,r)\) coordinates are treated as a direction vector \((r_x, r_y, r_z)\). The direction vector is used to select a cube map face. The direction vector is transformed to a per-face texel coordinate system \((s_{\text{face}}, t_{\text{face}})\). The direction vector is also used to transform the derivatives to per-face derivatives.

### 16.5.4. Cube Map Face Selection

The direction vector selects one of the cube map’s faces based on the largest magnitude coordinate direction (the major axis direction). Since two or more coordinates can have identical magnitude, the implementation must have rules to disambiguate this situation.

The rules should have as the first rule that \(r_z\) wins over \(r_y\) and \(r_x\), and the second rule that \(r_y\) wins over \(r_x\). An implementation may choose other rules, but the rules must be deterministic and depend only on \((r_x, r_y, r_z)\).

The layer number (corresponding to a cube map face), the coordinate selections for \(s_c\), \(t_c\), \(r_c\), and the selection of derivatives, are determined by the major axis direction as specified in the following two tables.
Table 17. Cube map face and coordinate selection

<table>
<thead>
<tr>
<th>Major Axis Direction</th>
<th>Layer Number</th>
<th>Cube Map Face</th>
<th>s_c</th>
<th>t_c</th>
<th>r_c</th>
</tr>
</thead>
<tbody>
<tr>
<td>+r_x</td>
<td>0</td>
<td>Positive X</td>
<td>-r_z</td>
<td>-r_y</td>
<td>r_x</td>
</tr>
<tr>
<td>-r_x</td>
<td>1</td>
<td>Negative X</td>
<td>+r_z</td>
<td>-r_y</td>
<td>r_x</td>
</tr>
<tr>
<td>+r_y</td>
<td>2</td>
<td>Positive Y</td>
<td>+r_x</td>
<td>+r_z</td>
<td>r_y</td>
</tr>
<tr>
<td>-r_y</td>
<td>3</td>
<td>Negative Y</td>
<td>+r_x</td>
<td>-r_z</td>
<td>r_y</td>
</tr>
<tr>
<td>+r_z</td>
<td>4</td>
<td>Positive Z</td>
<td>+r_x</td>
<td>-r_y</td>
<td>r_z</td>
</tr>
<tr>
<td>-r_z</td>
<td>5</td>
<td>Negative Z</td>
<td>-r_x</td>
<td>-r_y</td>
<td>r_z</td>
</tr>
</tbody>
</table>

Table 18. Cube map derivative selection

<table>
<thead>
<tr>
<th>Major Axis Direction</th>
<th>∂s_c / ∂x</th>
<th>∂s_c / ∂y</th>
<th>∂t_c / ∂x</th>
<th>∂t_c / ∂y</th>
<th>∂r_c / ∂x</th>
<th>∂r_c / ∂y</th>
</tr>
</thead>
<tbody>
<tr>
<td>+r_x</td>
<td>-∂r_z / ∂x</td>
<td>-∂r_z / ∂y</td>
<td>-∂r_y / ∂x</td>
<td>-∂r_y / ∂y</td>
<td>+∂r_z / ∂x</td>
<td>+∂r_z / ∂y</td>
</tr>
<tr>
<td>-r_x</td>
<td>+∂r_z / ∂x</td>
<td>+∂r_z / ∂y</td>
<td>-∂r_y / ∂x</td>
<td>-∂r_y / ∂y</td>
<td>-∂r_z / ∂x</td>
<td>-∂r_z / ∂y</td>
</tr>
<tr>
<td>+r_y</td>
<td>+∂r_x / ∂x</td>
<td>+∂r_x / ∂y</td>
<td>+∂r_z / ∂x</td>
<td>+∂r_z / ∂y</td>
<td>+∂r_y / ∂x</td>
<td>+∂r_y / ∂y</td>
</tr>
<tr>
<td>-r_y</td>
<td>+∂r_x / ∂x</td>
<td>+∂r_x / ∂y</td>
<td>-∂r_z / ∂x</td>
<td>-∂r_z / ∂y</td>
<td>-∂r_y / ∂x</td>
<td>-∂r_y / ∂y</td>
</tr>
<tr>
<td>+r_z</td>
<td>+∂r_x / ∂x</td>
<td>+∂r_x / ∂y</td>
<td>-∂r_y / ∂x</td>
<td>-∂r_y / ∂y</td>
<td>+∂r_z / ∂x</td>
<td>+∂r_z / ∂y</td>
</tr>
<tr>
<td>-r_z</td>
<td>-∂r_x / ∂x</td>
<td>-∂r_x / ∂y</td>
<td>-∂r_y / ∂x</td>
<td>-∂r_y / ∂y</td>
<td>-∂r_z / ∂x</td>
<td>-∂r_z / ∂y</td>
</tr>
</tbody>
</table>

16.5.5. Cube Map Coordinate Transformation

\[
s_{\text{face}} = \frac{1}{2} \times \frac{s_c}{|r_c|} + \frac{1}{2}
\]
\[
t_{\text{face}} = \frac{1}{2} \times \frac{t_c}{|r_c|} + \frac{1}{2}
\]

16.5.6. Cube Map Derivative Transformation

\[
\frac{\partial s_{\text{face}}}{\partial x} = \frac{\partial}{\partial x} \left( \frac{1}{2} \times \frac{s_c}{|r_c|} + \frac{1}{2} \right)
\]
\[
\frac{\partial s_{\text{face}}}{\partial x} = \frac{1}{2} \times \frac{\partial}{\partial x} \left( \frac{s_c}{|r_c|} \right)
\]
\[
\frac{\partial s_{\text{face}}}{\partial x} = \frac{1}{2} \times \left( \frac{|r_c| \times \partial s_c / \partial x - s_c \times \partial r_c / \partial x}{(r_c)^2} \right)
\]
16.5.7. Scale Factor Operation, Level-of-Detail Operation and Image Level(s) Selection

LOD selection can be either explicit (provided explicitly by the image instruction) or implicit (determined from a scale factor calculated from the derivatives). The LOD must be computed with `mipmapPrecisionBits` of accuracy.

Scale Factor Operation

The magnitude of the derivatives are calculated by:

\[
\frac{\partial s_{\text{face}}}{\partial y} = \frac{1}{2} \times \left( \frac{|r_c| \times \partial s_c / \partial y - s_c \times \partial r_c / \partial y}{(r_c)^2} \right)
\]

\[
\frac{\partial t_{\text{face}}}{\partial x} = \frac{1}{2} \times \left( \frac{|r_c| \times \partial t_c / \partial x - t_c \times \partial r_c / \partial x}{(r_c)^2} \right)
\]

\[
\frac{\partial t_{\text{face}}}{\partial y} = \frac{1}{2} \times \left( \frac{|r_c| \times \partial t_c / \partial y - t_c \times \partial r_c / \partial y}{(r_c)^2} \right)
\]

where:

\[\frac{\partial t}{\partial x} = \frac{\partial t}{\partial y} = 0 \text{ (for 1D images)}\]

\[\frac{\partial r}{\partial x} = \frac{\partial r}{\partial y} = 0 \text{ (for 1D, 2D or Cube images)}\]

and:

\[w_{\text{base}} = \text{image.w}\]
A point sampled in screen space has an elliptical footprint in texture space. The minimum and maximum scale factors \((\rho_{\text{min}}, \rho_{\text{max}})\) should be the minor and major axes of this ellipse.

The scale factors \(\rho_x\) and \(\rho_y\), calculated from the magnitude of the derivatives in x and y, are used to compute the minimum and maximum scale factors.

\(\rho_x\) and \(\rho_y\) may be approximated with functions \(f_x\) and \(f_y\), subject to the following constraints:

\[
\begin{align*}
\text{max}(|m_{ux}|, |m_{vx}|, |m_{wx}|) & \leq f_x \leq \sqrt{2(|m_{ux}| + |m_{vx}| + |m_{wx}|)} \\
\text{max}(|m_{uy}|, |m_{vy}|, |m_{wy}|) & \leq f_y \leq \sqrt{2(|m_{uy}| + |m_{vy}| + |m_{wy}|)}
\end{align*}
\]

The minimum and maximum scale factors \((\rho_{\text{min}}, \rho_{\text{max}})\) are determined by:

\[
\begin{align*}
\rho_{\text{max}} & = \max(\rho_x, \rho_y) \\
\rho_{\text{min}} & = \min(\rho_x, \rho_y)
\end{align*}
\]

The ratio of anisotropy is determined by:

\[
\eta = \min(\rho_{\text{max}}/\rho_{\text{min}}, \text{maxAniso})
\]

where:

\[
sampler.\text{maxAniso} = \text{maxAnisotropy} \quad (\text{from sampler descriptor})
\]

\[
\text{limits.\text{maxAniso}} = \text{maxSamplerAnisotropy} \quad (\text{from physical device limits})
\]

\[
\text{maxAniso} = \min(sampler.\text{maxAniso}, \text{limits.\text{maxAniso}})
\]

If \(\rho_{\text{max}} = \rho_{\text{min}} = 0\), then all the partial derivatives are zero, the fragment’s footprint in texel space is a point, and \(\eta\) should be treated as 1. If \(\rho_{\text{max}} \neq 0\) and \(\rho_{\text{min}} = 0\) then all partial derivatives along one axis are zero, the fragment’s footprint in texel space is a line segment, and \(\eta\) should be treated as
maxAniso. However, anytime the footprint is small in texel space the implementation may use a smaller value of η, even when \( \rho_{\text{min}} \) is zero or close to zero. If either VkPhysicalDeviceFeatures::samplerAnisotropy or VkSamplerCreateInfo::anisotropyEnable are VK_FALSE, maxAniso is set to 1.

If \( \eta = 1 \), sampling is isotropic. If \( \eta > 1 \), sampling is anisotropic.

The sampling rate (N) is derived as:

\[
N = \lceil \eta \rceil
\]

An implementation may round N up to the nearest supported sampling rate. An implementation may use the value of N as an approximation of \( \eta \).

**Level-of-Detail Operation**

The LOD parameter \( \lambda \) is computed as follows:

\[
\lambda_{\text{base}}(x, y) = \begin{cases} 
    \text{shaderOp.Lod} & \text{(from optional SPIR-V operand)} \\
    \log_2\left(\frac{\rho_{\text{max}}}{\eta}\right) & \text{otherwise}
\end{cases}
\]

\[
\lambda'(x, y) = \lambda_{\text{base}} + \text{clamp}(\text{sampler.bias} + \text{shaderOp.bias}, -\max\text{SamplerLodBias}, \max\text{SamplerLodBias})
\]

\[
\lambda = \begin{cases} 
    \text{lod}_{\max}, & \lambda' > \text{lod}_{\max} \\
    \lambda', & \text{lod}_{\min} \leq \lambda' \leq \text{lod}_{\max} \\
    \text{lod}_{\min}, & \lambda' < \text{lod}_{\min} \\
    \text{undefined}, & \text{lod}_{\min} > \text{lod}_{\max}
\end{cases}
\]

where:

\[
\text{sampler.bias} = \text{mipLodBias}
\]

(from sampler descriptor)

\[
\text{shaderOp.bias} = \begin{cases} 
    \text{Bias} & \text{(from optional SPIR-V operand)} \\
    0 & \text{otherwise}
\end{cases}
\]

\[
\text{sampler.lod}_{\min} = \text{minLod}
\]

(from sampler descriptor)

\[
\text{shaderOp.lod}_{\min} = \begin{cases} 
    \text{MinLod} & \text{(from optional SPIR-V operand)} \\
    0 & \text{otherwise}
\end{cases}
\]

\[
\text{lod}_{\min} = \max(\text{sampler.lod}_{\min}, \text{shaderOp.lod}_{\min})
\]

\[
\text{lod}_{\max} = \max\text{Lod}
\]

(from sampler descriptor)

and maxSamplerLodBias is the value of the VkPhysicalDeviceLimits feature maxSamplerLodBias.

**Image Level(s) Selection**

The image level(s) \( d, d_{hi}, \) and \( d_{lo} \) which texels are read from are determined by an image-level parameter \( d_{l} \) which is computed based on the LOD parameter, as follows:

\[
d_j = \begin{cases} 
    \text{nearest}(d'), & \text{mipmapMode is VK_SAMPLER_MIPMAP_MODE_NEAREST} \\
    d', & \text{otherwise}
\end{cases}
\]

where:

\[
d' = \text{level}_{\text{base}} + \text{clamp}(\lambda, 0, q)
\]
\[ nearest(d') = \begin{cases} 
[d' + 0.5] - 1, & \text{preferred} \\
[d' + 0.5], & \text{alternative} 
\end{cases} \]

and:

\[ \text{level}_\text{base} = \text{baseMipLevel} \]
\[ q = \text{levelCount} - 1 \]

\text{baseMipLevel} and \text{levelCount} are taken from the \text{subresourceRange} of the image view.

If the sampler's \text{mipmapMode} is \text{VK_SAMPLER_MIPMAP_MODE_NEAREST}, then the level selected is \( d = d' \).

If the sampler's \text{mipmapMode} is \text{VK_SAMPLER_MIPMAP_MODE_LINEAR}, two neighboring levels are selected:

\[ d_{hi} = \lfloor d' \rfloor \]
\[ d_{lo} = \min(d_{hi} + 1, \text{level}_\text{base} + q) \]
\[ \delta = d' - d_{hi} \]

\( \delta \) is the fractional value, quantized to the number of \text{mipmap precision bits}, used for linear filtering between levels.

16.5.8. (s,t,r,q,a) to (u,v,w,a) Transformation

The normalized texel coordinates are scaled by the image level dimensions and the array layer is selected.

This transformation is performed once for each level used in \text{filtering} (either \( d \), or \( d_{hi} \) and \( d_{lo} \)).

\[ u(x, y) = s(x, y) \times \text{width}_{\text{scale}} + \Delta_i \]
\[ v(x, y) = \begin{cases} 
0 & \text{for 1D images} \\
t(x, y) \times \text{height}_{\text{scale}} + \Delta_j & \text{otherwise} 
\end{cases} \]
\[ w(x, y) = \begin{cases} 
0 & \text{for 2D or Cube images} \\
r(x, y) \times \text{depth}_{\text{scale}} + \Delta_k & \text{otherwise} 
\end{cases} \]
\[ a(x, y) = \begin{cases} 
a(x, y) & \text{for array images} \\
0 & \text{otherwise} 
\end{cases} \]

where:

\[ \text{width}_{\text{scale}} = \text{width}_{\text{level}} \]
\[ \text{height}_{\text{scale}} = \text{height}_{\text{level}} \]
\[ \text{depth}_{\text{scale}} = \text{depth}_{\text{level}} \]
and where \((\Delta_i, \Delta_j, \Delta_k)\) are taken from the image instruction if it includes a ConstOffset or Offset operand, otherwise they are taken to be zero.

Operations then proceed to Unnormalized Texel Coordinate Operations.

### 16.6. Unnormalized Texel Coordinate Operations

#### 16.6.1. \((u,v,w,a)\) to \((i,j,k,l,n)\) Transformation And Array Layer Selection

The unnormalized texel coordinates are transformed to integer texel coordinates relative to the selected mipmap level.

The layer index \(l\) is computed as:

\[
l = \text{clamp}(\text{RNE}(a), 0, \text{layerCount} - 1) + \text{baseArrayLayer}
\]

where \(\text{layerCount}\) is the number of layers in the image subresource range of the image view, \(\text{baseArrayLayer}\) is the first layer from the subresource range, and where:

\[
\text{RNE}(a) = \begin{cases} 
\text{roundTiesToEven}(a) & \text{preferred, from IEEE Std 754-2008 Floating-Point Arithmetic} \\
[a + 0.5] & \text{alternative}
\end{cases}
\]

The sample index \(n\) is assigned the value 0.

Nearest filtering (\(\text{VK_FILTER_NEAREST}\)) computes the integer texel coordinates that the unnormalized coordinates lie within:

\[
i = \lfloor u + \text{shift} \rfloor \\
j = \lfloor v + \text{shift} \rfloor \\
k = \lfloor w + \text{shift} \rfloor
\]

where:

\[
\text{shift} = 0.0
\]

Linear filtering (\(\text{VK_FILTER_LINEAR}\)) computes a set of neighboring coordinates which bound the unnormalized coordinates. The integer texel coordinates are combinations of \(i_0\) or \(i_1\), \(j_0\) or \(j_1\), \(k_0\) or \(k_1\), as well as weights \(\alpha\), \(\beta\), and \(\gamma\).

\[
i_0 = \lfloor u - \text{shift} \rfloor \\
i_1 = i_0 + 1 \\
j_0 = \lfloor v - \text{shift} \rfloor \\
j_1 = j_0 + 1 \\
k_0 = \lfloor w - \text{shift} \rfloor \\
k_1 = k_0 + 1
\]
\[
\begin{align*}
\alpha &= \text{frac}(u - shift) \\
\beta &= \text{frac}(v - shift) \\
\gamma &= \text{frac}(w - shift)
\end{align*}
\]

where:

\[
\text{shift} = 0.5
\]

and where:

\[
\text{frac}(x) = x - \lfloor x \rfloor
\]

where the number of fraction bits retained is specified by \text{VkPhysicalDeviceLimits::subTexelPrecisionBits}.

16.7. Integer Texel Coordinate Operations

The \text{OpImageFetch} and \text{OpImageFetchSparse} SPIR-V instructions may supply a LOD from which texels are to be fetched using the optional SPIR-V operand \text{Lod}. Other integer-coordinate operations must not. If the \text{Lod} is provided then it must be an integer.

The image level selected is:

\[
d = level_{\text{base}} + \begin{cases} 
Lod & \text{(from optional SPIR-V operand)} \\
0 & \text{otherwise}
\end{cases}
\]

If \(d\) does not lie in the range \([\text{baseMipLevel}, \text{baseMipLevel} + \text{levelCount})\) then any values fetched are undefined, and any writes (if supported) are discarded.

16.8. Image Sample Operations

16.8.1. Wrapping Operation

\text{Cube} images ignore the wrap modes specified in the sampler. Instead, if \text{VK_FILTER_NEAREST} is used within a mip level then \text{VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE} is used, and if \text{VK_FILTER_LINEAR} is used within a mip level then sampling at the edges is performed as described earlier in the \text{Cube map edge handling} section.

The first integer texel coordinate \(i\) is transformed based on the \text{addressModeU} parameter of the sampler.
\[
    i = \begin{cases} 
        i \mod \text{size} & \text{for repeat} \\
        (\text{size} - 1) - \text{mirror} \left( (i \mod (2 \times \text{size})) - \text{size} \right) & \text{for mirrored repeat} \\
        \text{clamp} \left( i, 0, \text{size} - 1 \right) & \text{for clamp to edge} \\
        \text{clamp} \left( i, -1, \text{size} \right) & \text{for clamp to border} \\
        \text{clamp} \left( \text{mirror} \left( i \right), 0, \text{size} - 1 \right) & \text{for mirror clamp to edge} 
    \end{cases}
\]

where:

\[
    \text{mirror} \left( n \right) = \begin{cases} 
        n & \text{for } n \geq 0 \\
        -(1 + n) & \text{otherwise}
    \end{cases}
\]

\( j \) (for 2D and Cube image) and \( k \) (for 3D image) are similarly transformed based on the \textit{addressModeV} and \textit{addressModeW} parameters of the sampler, respectively.

### 16.8.2. Texel Gathering

SPIR-V instructions with \texttt{Gather} in the name return a vector derived from 4 texels in the base level of the image view. The rules for the \texttt{VK_FILTER_LINEAR} minification filter are applied to identify the four selected texels. Each texel is then converted to an RGBA value according to \textit{conversion to RGBA} and then \textit{swizzled}. A four-component vector is then assembled by taking the component indicated by the \textit{Component} value in the instruction from the swizzled color value of the four texels. If the operation does not use the \texttt{ConstOffsets} image operand then the four texels form the \( 2 \times 2 \) rectangle used for texture filtering:

\[
    \tau[R] = \tau_{i0j0}[\text{level base}][\text{comp}]
\]

\[
    \tau[G] = \tau_{i1j0}[\text{level base}][\text{comp}]
\]

\[
    \tau[B] = \tau_{i0j1}[\text{level base}][\text{comp}]
\]

\[
    \tau[A] = \tau_{i1j1}[\text{level base}][\text{comp}]
\]

If the operation does use the \texttt{ConstOffsets} image operand then the offsets allow a custom filter to be defined:

\[
    \tau[R] = \tau_{i0j0} + \Delta_0[\text{level base}][\text{comp}]
\]

\[
    \tau[G] = \tau_{i1j0} + \Delta_1[\text{level base}][\text{comp}]
\]

\[
    \tau[B] = \tau_{i0j1} + \Delta_2[\text{level base}][\text{comp}]
\]

\[
    \tau[A] = \tau_{i1j1} + \Delta_3[\text{level base}][\text{comp}]
\]

where:

\[
    \tau[\text{level base}][\text{comp}] = \begin{cases} 
        \tau[\text{level base}][R], & \text{for } \text{comp} = 0 \\
        \tau[\text{level base}][G], & \text{for } \text{comp} = 1 \\
        \tau[\text{level base}][B], & \text{for } \text{comp} = 2 \\
        \tau[\text{level base}][A], & \text{for } \text{comp} = 3 
    \end{cases}
\]

\( \text{comp} \) from SPIR-V operand Component
OpImage*Gather must not be used on a sampled image with sampler Y’CbCr conversion enabled.

16.8.3. Texel Filtering

Texel filtering is first performed for each level (either d or d_hi and d_lo).

If \( \lambda \) is less than or equal to zero, the texture is said to be magnified, and the filter mode within a mip level is selected by the `magFilter` in the sampler. If \( \lambda \) is greater than zero, the texture is said to be minified, and the filter mode within a mip level is selected by the `minFilter` in the sampler.

**Texel Nearest Filtering**

Within a mip level, `VK_FILTER_NEAREST` filtering selects a single value using the \((i, j, k)\) texel coordinates, with all texels taken from layer \( l \).

\[
\tau[\text{level}] = \begin{cases} 
\tau_{ijk}[\text{level}], & \text{for 3D image} \\
\tau_{ij}[\text{level}], & \text{for 2D or Cube image} \\
\tau_{i}[\text{level}], & \text{for 1D image}
\end{cases}
\]

**Texel Linear Filtering**

Within a mip level, `VK_FILTER_LINEAR` filtering combines 8 (for 3D), 4 (for 2D or Cube), or 2 (for 1D) texel values, together with their linear weights. The linear weights are derived from the fractions computed earlier:

\[
\begin{align*}
    w_{i_0} &= (1 - \alpha) \\
    w_{i_1} &= \alpha \\
    w_{j_0} &= (1 - \beta) \\
    w_{j_1} &= \beta \\
    w_{k_0} &= (1 - \gamma) \\
    w_{k_1} &= \gamma
\end{align*}
\]

The values of multiple texels, together with their weights, are combined to produce a filtered value.

The `VkSamplerReductionModeCreateInfo::reductionMode` can control the process by which multiple texels, together with their weights, are combined to produce a filtered texture value.

When the `reductionMode` is set (explicitly or implicitly) to `VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE`, a weighted average is computed:
However, if the reduction mode is `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX`, the process operates on the above set of multiple texels, together with their weights, computing a component-wise minimum or maximum, respectively, of the components of the set of texels with non-zero weights.

**Texel Mipmap Filtering**

`VK_SAMPLER_MIPMAP_MODE_NEAREST` filtering returns the value of a single mipmap level, \( \tau = \tau[d] \).

`VK_SAMPLER_MIPMAP_MODE_LINEAR` filtering combines the values of multiple mipmap levels (\( \tau[hi] \) and \( \tau[lo] \)), together with their linear weights.

The linear weights are derived from the fraction computed earlier:

\[
\begin{align*}
  w_{hi} &= (1 - \delta) \\
  w_{lo} &= (\delta)
\end{align*}
\]

The values of multiple mipmap levels, together with their weights, are combined to produce a final filtered value.

The \texttt{VkSamplerReductionModeCreateInfo::reductionMode} can control the process by which multiple texels, together with their weights, are combined to produce a filtered texture value.

When the \texttt{reductionMode} is set (explicitly or implicitly) to `VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE`, a weighted average is computed:

\[
\tau = (w_{hi})\tau[hi] + (w_{lo})\tau[lo]
\]

**Texel Anisotropic Filtering**

Anisotropic filtering is enabled by the \texttt{anisotropyEnable} in the sampler. When enabled, the image filtering scheme accounts for a degree of anisotropy.

The particular scheme for anisotropic texture filtering is implementation-dependent. Implementations should consider the \texttt{magFilter}, \texttt{minFilter} and \texttt{mipmapMode} of the sampler to control the specifics of the anisotropic filtering scheme used. In addition, implementations should consider \texttt{minLod} and \texttt{maxLod} of the sampler.
The following describes one particular approach to implementing anisotropic filtering for the 2D Image case, implementations may choose other methods:

Given a `magFilter`, `minFilter` of `VK_FILTER_LINEAR` and a `mipmapMode` of `VK_SAMPLER_MIPMAP_MODE_NEAREST`:

Instead of a single isotropic sample, N isotropic samples are sampled within the image footprint of the image level d to approximate an anisotropic filter. The sum $τ_{2Daniso}$ is defined using the single isotropic $τ_{2D}(u,v)$ at level d.

$$τ_{2Daniso} = \frac{1}{N} \sum_{i=1}^{N} τ_{2D}\left(u - \frac{1}{2} + \frac{i}{N+1}, v - \frac{1}{2} + \frac{i}{N+1}\right), \quad \text{when } ρ_x > ρ_y$$

$$τ_{2Daniso} = \frac{1}{N} \sum_{i=1}^{N} τ_{2D}\left(u - \frac{1}{2} + \frac{i}{N+1}, v - \frac{1}{2} + \frac{i}{N+1}\right), \quad \text{when } ρ_y ≥ ρ_x$$

When `VkSamplerReductionModeCreateInfo::reductionMode` is set to `VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE`, the above summation is used. However, if the reduction mode is `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX`, the process operates on the above values, together with their weights, computing a component-wise minimum or maximum, respectively, of the components of the values with non-zero weights.

### 16.9. Image Operation Steps

Each step described in this chapter is performed by a subset of the image instructions:

- Texel Input Validation Operations, Format Conversion, Texel Replacement, Conversion to RGBA, and Component Swizzle: Performed by all instructions except `OpImageWrite`.
- Depth Comparison: Performed by `OpImage*Dref` instructions.
- All Texel output operations: Performed by `OpImageWrite`.
- Projection: Performed by all `OpImage*Proj` instructions.
- Derivative Image Operations, Cube Map Operations, Scale Factor Operation, Level-of-Detail Operation and Image Level(s) Selection, and Texel Anisotropic Filtering: Performed by all `OpImageSample*` and `OpImageSparseSample*` instructions.
- (s,t,r,q,a) to (u,v,w,a) Transformation, Wrapping, and (u,v,w,a) to (i,j,k,l,n) Transformation And Array Layer Selection: Performed by all `OpImageSample`, `OpImageSparseSample`, and `OpImage*Gather` instructions.
- Texel Filtering: Performed by all `OpImageSample*` and `OpImageSparseSample*` instructions.
- Sparse Residency: Performed by all `OpImageSparse*` instructions.

### 16.10. Image Query Instructions
16.10.1. Image Property Queries

OpImageQuerySize, OpImageQuerySizeLod, OpImageQueryLevels, and OpImageQuerySamples query properties of the image descriptor that would be accessed by a shader image operation.

OpImageQuerySizeLod returns the size of the image level identified by the Level of Detail operand. If that level does not exist in the image, then the value returned is undefined.

16.10.2. Lod Query

OpImageQueryLod returns the Lod parameters that would be used in an image operation with the given image and coordinates. The steps described in this chapter are performed as if for OpImageSampleImplicitLod, up to Scale Factor Operation, Level-of-Detail Operation and Image Level(s) Selection. The return value is the vector ($\lambda'$, $d_l$). These values may be subject to implementation-specific maxima and minima for very large, out-of-range values.
Chapter 17. Queries

Queries provide a mechanism to return information about the processing of a sequence of Vulkan commands. Query operations are asynchronous, and as such, their results are not returned immediately. Instead, their results, and their availability status are stored in a Query Pool. The state of these queries can be read back on the host, or copied to a buffer object on the device.

The supported query types are Occlusion Queries, Pipeline Statistics Queries, and Timestamp Queries.

17.1. Query Pools

Queries are managed using query pool objects. Each query pool is a collection of a specific number of queries of a particular type.

Query pools are represented by VkQueryPool handles:

```cpp
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkQueryPool)
```

To create a query pool, call:

```cpp
// Provided by VK_VERSION_1_0
VkResult vkCreateQueryPool(
    VkDevice device,                        
    const VkQueryPoolCreateInfo* pCreateInfo, 
    const VkAllocationCallbacks* pAllocator, 
    VkQueryPool* pQueryPool);
```

- `device` is the logical device that creates the query pool.
- `pCreateInfo` is a pointer to a VkQueryPoolCreateInfo structure containing the number and type of queries to be managed by the pool.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pQueryPool` is a pointer to a VkQueryPool handle in which the resulting query pool object is returned.

Valid Usage (Implicit)

- VUID-vkCreateQueryPool-device-parameter `device` must be a valid VkDevice handle
- VUID-vkCreateQueryPool-pCreateInfo-parameter `pCreateInfo` must be a valid pointer to a valid VkQueryPoolCreateInfo structure
- VUID-vkCreateQueryPool-pAllocator-parameter If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid
VkAllocationCallbacks structure

- VUID-vkCreateQueryPool-pQueryPool-parameter
  pQueryPool must be a valid pointer to a VkQueryPool handle

## Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkQueryPoolCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkQueryPoolCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkQueryPoolCreateFlags flags;
    VkQueryType queryType;
    uint32_t queryCount;
    VkQueryPipelineStatisticFlags pipelineStatistics;
} VkQueryPoolCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `queryType` is a VkQueryType value specifying the type of queries managed by the pool.
- `queryCount` is the number of queries managed by the pool.
- `pipelineStatistics` is a bitmask of VkQueryPipelineStatisticFlagBits specifying which counters will be returned in queries on the new pool, as described below in Pipeline Statistics Queries.

`pipelineStatistics` is ignored if `queryType` is not VK_QUERY_TYPE_PIPELINE_STATISTICS.

## Valid Usage

- VUID-VkQueryPoolCreateInfo-queryType-00791
  If the pipelineStatisticsQuery feature is not enabled, `queryType` must not be VK_QUERY_TYPE_PIPELINE_STATISTICS

- VUID-VkQueryPoolCreateInfo-queryType-00792
  If `queryType` is VK_QUERY_TYPE_PIPELINE_STATISTICS, `pipelineStatistics` must be a valid combination of VkQueryPipelineStatisticFlagBits values
• VUID-VkQueryPoolCreateInfo-queryCount-02763
  queryCount must be greater than 0

Valid Usage (Implicit)

• VUID-VkQueryPoolCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO

• VUID-VkQueryPoolCreateInfo-pNext-pNext
  pNext must be NULL

• VUID-VkQueryPoolCreateInfo-flags-zerobitmask
  flags must be 0

• VUID-VkQueryPoolCreateInfo-queryType-parameter
  queryType must be a valid VkQueryType value

// Provided by VK_VERSION_1_0
typedef VkFlags VkQueryPoolCreateFlags;

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

To destroy a query pool, call:

// Provided by VK_VERSION_1_0
void vkDestroyQueryPool(
    VkDevice               device,
    VkQueryPool           queryPool,
    const VkAllocationCallbacks* pAllocator);

• device is the logical device that destroys the query pool.
• queryPool is the query pool to destroy.
• pAllocator controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

• VUID-vkDestroyQueryPool-queryPool-00793
  All submitted commands that refer to queryPool must have completed execution

• VUID-vkDestroyQueryPool-queryPool-00794
  If VkAllocationCallbacks were provided when queryPool was created, a compatible set of callbacks must be provided here

• VUID-vkDestroyQueryPool-queryPool-00795
  If no VkAllocationCallbacks were provided when queryPool was created, pAllocator must be NULL
Applications can verify that `queryPool` can be destroyed by checking that `vkGetQueryPoolResults()` without the `VK_QUERY_RESULT_PARTIAL_BIT` flag returns `VK_SUCCESS` for all queries that are used in command buffers submitted for execution.

### Valid Usage (Implicit)

- **VUID-vkDestroyQueryPool-device-parameter**
  - `device` must be a valid `VkDevice` handle
- **VUID-vkDestroyQueryPool-queryPool-parameter**
  - If `queryPool` is not `VK_NULL_HANDLE`, `queryPool` must be a valid `VkQueryPool` handle
- **VUID-vkDestroyQueryPool-pAllocator-parameter**
  - If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- **VUID-vkDestroyQueryPool-queryPool-parent**
  - If `queryPool` is a valid handle, it must have been created, allocated, or retrieved from `device`

### Host Synchronization

- Host access to `queryPool` must be externally synchronized

Possible values of `VkQueryPoolCreateInfo::queryType`, specifying the type of queries managed by the pool, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkQueryType {
    VK_QUERY_TYPE_OCCLUSION = 0,
    VK_QUERY_TYPE_PIPELINE_STATISTICS = 1,
    VK_QUERY_TYPE_TIMESTAMP = 2,
} VkQueryType;
```

- **`VK_QUERY_TYPE_OCCLUSION`** specifies an occlusion query.
- **`VK_QUERY_TYPE_PIPELINE_STATISTICS`** specifies a pipeline statistics query.
- **`VK_QUERY_TYPE_TIMESTAMP`** specifies a timestamp query.

### 17.2. Query Operation

The operation of queries is controlled by the commands `vkCmdBeginQuery`, `vkCmdEndQuery`, `vkCmdResetQueryPool`, `vkCmdCopyQueryPoolResults`, `vkCmdWriteTimestamp2`, and `vkCmdWriteTimestamp`.
In order for a `VkCommandBuffer` to record query management commands, the queue family for which its `VkCommandPool` was created **must** support the appropriate type of operations (graphics, compute) suitable for the query type of a given query pool.

Each query in a query pool has a status that is either **unavailable** or **available**, and also has state to store the numerical results of a query operation of the type requested when the query pool was created. Resetting a query via `vkCmdResetQueryPool` or `vkResetQueryPool` sets the status to unavailable and makes the numerical results undefined. A query is made available by the operation of `vkCmdEndQuery`, `vkCmdWriteTimestamp2`, or `vkCmdWriteTimestamp`. Both the availability status and numerical results can be retrieved by calling either `vkGetQueryPoolResults` or `vkCmdCopyQueryPoolResults`.

After query pool creation, each query **must** be reset before it is used. Queries **must** also be reset between uses.

If a logical device includes multiple physical devices, then each command that writes a query **must** execute on a single physical device, and any call to `vkCmdBeginQuery` **must** execute the corresponding `vkCmdEndQuery` command on the same physical device.

To reset a range of queries in a query pool on a queue, call:

```
// Provided by VK_VERSION_1_0
void vkCmdResetQueryPool(
    VkCommandBuffer commandBuffer,            // Provided by VK_VERSION_1_0
    VkQueryPool queryPool,                  // Provided by VK_VERSION_1_0
    uint32_t firstQuery,                    // Provided by VK_VERSION_1_0
    uint32_t queryCount);                   // Provided by VK_VERSION_1_0
```

- `commandBuffer` is the command buffer into which this command will be recorded.
- `queryPool` is the handle of the query pool managing the queries being reset.
- `firstQuery` is the initial query index to reset.
- `queryCount` is the number of queries to reset.

When executed on a queue, this command sets the status of query indices `[firstQuery, firstQuery + queryCount - 1]` to unavailable.

This command defines an execution dependency between other query commands that reference the same query.

The first synchronization scope includes all commands which reference the queries in `queryPool` indicated by `firstQuery` and `queryCount` that occur earlier in submission order.

The second synchronization scope includes all commands which reference the queries in `queryPool` indicated by `firstQuery` and `queryCount` that occur later in submission order.

The operation of this command happens after the first scope and happens before the second scope.
Valid Usage

- VUID-vkCmdResetQueryPool-firstQuery-00796
  `firstQuery` must be less than the number of queries in `queryPool`

- VUID-vkCmdResetQueryPool-firstQuery-00797
  The sum of `firstQuery` and `queryCount` must be less than or equal to the number of queries in `queryPool`

- VUID-vkCmdResetQueryPool-None-02841
  All queries used by the command must not be active

Valid Usage (Implicit)

- VUID-vkCmdResetQueryPool-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdResetQueryPool-queryPool-parameter
  `queryPool` must be a valid `VkQueryPool` handle

- VUID-vkCmdResetQueryPool-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdResetQueryPool-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, compute, decode, encode, or opticalflow operations

- VUID-vkCmdResetQueryPool-renderpass
  This command must only be called outside of a render pass instance

- VUID-vkCmdResetQueryPool-commonparent
  Both of `commandBuffer`, and `queryPool` must have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

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

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Decode</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Encode</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Opticalflow</td>
<td></td>
</tr>
</tbody>
</table>

To reset a range of queries in a query pool on the host, call:

```c
// Provided by VK_VERSION_1_2
void vkResetQueryPool(
    VkDevice device,
    VkQueryPool queryPool,
    uint32_t firstQuery,
    uint32_t queryCount);
```

- **device** is the logical device that owns the query pool.
- **queryPool** is the handle of the query pool managing the queries being reset.
- **firstQuery** is the initial query index to reset.
- **queryCount** is the number of queries to reset.

This command sets the status of query indices `[firstQuery, firstQuery + queryCount - 1]` to unavailable.

Valid Usage

- **VUID-vkResetQueryPool-None-02665**
  The `hostQueryReset` feature must be enabled

- **VUID-vkResetQueryPool-firstQuery-02666**
  `firstQuery` must be less than the number of queries in `queryPool`

- **VUID-vkResetQueryPool-firstQuery-02667**
  The sum of `firstQuery` and `queryCount` must be less than or equal to the number of queries in `queryPool`

- **VUID-vkResetQueryPool-firstQuery-02741**
  Submitted commands that refer to the range specified by `firstQuery` and `queryCount` in `queryPool` must have completed execution

- **VUID-vkResetQueryPool-firstQuery-02742**
  The range of queries specified by `firstQuery` and `queryCount` in `queryPool` must not be in use by calls to `vkGetQueryPoolResults` or `vkResetQueryPool` in other threads
Valid Usage (Implicit)

- VUID-vkResetQueryPool-device-parameter
  device must be a valid VkDevice handle
- VUID-vkResetQueryPool-queryPool-parameter
  queryPool must be a valid VkQueryPool handle
- VUID-vkResetQueryPool-queryPool-parent
  queryPool must have been created, allocated, or retrieved from device

Once queries are reset and ready for use, query commands can be issued to a command buffer. Occlusion queries and pipeline statistics queries count events - drawn samples and pipeline stage invocations, respectively - resulting from commands that are recorded between a vkCmdBeginQuery command and a vkCmdEndQuery command within a specified command buffer, effectively scoping a set of drawing and/or dispatching commands. Timestamp queries write timestamps to a query pool.

A query must begin and end in the same command buffer, although if it is a primary command buffer, and the inheritedQueries feature is enabled, it can execute secondary command buffers during the query operation. For a secondary command buffer to be executed while a query is active, it must set the occlusionQueryEnable, queryFlags, and/or pipelineStatistics members of VkCommandBufferInheritanceInfo to conservative values, as described in the Command Buffer Recording section. A query must either begin and end inside the same subpass of a render pass instance, or must both begin and end outside of a render pass instance (i.e. contain entire render pass instances).

If queries are used while executing a render pass instance that has multiview enabled, the query uses N consecutive query indices in the query pool (starting at query) where N is the number of bits set in the view mask in the subpass the query is used in. How the numerical results of the query are distributed among the queries is implementation-dependent. For example, some implementations may write each view's results to a distinct query, while other implementations may write the total result to the first query and write zero to the other queries. However, the sum of the results in all the queries must accurately reflect the total result of the query summed over all views. Applications can sum the results from all the queries to compute the total result.

Queries used with multiview rendering must not span subpasses, i.e. they must begin and end in the same subpass.

To begin a query, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdBeginQuery(
    VkCommandBuffer commandBuffer,
    VkQueryPool queryPool,
    uint32_t query,
    VkQueryControlFlags flags);
```
• **commandBuffer** is the command buffer into which this command will be recorded.
• **queryPool** is the query pool that will manage the results of the query.
• **query** is the query index within the query pool that will contain the results.
• **flags** is a bitmask of **VkQueryControlFlagBits** specifying constraints on the types of queries that can be performed.

If the **queryType** of the pool is **VK_QUERY_TYPE_OCCLUSION** and **flags** contains **VK_QUERY_CONTROL_PRECISE_BIT**, an implementation must return a result that matches the actual number of samples passed. This is described in more detail in **Occlusion Queries**.

After beginning a query, that query is considered active within the command buffer it was called in until that same query is ended. Queries active in a primary command buffer when secondary command buffers are executed are considered active for those secondary command buffers.

This command defines an execution dependency between other query commands that reference the same query.

The first **synchronization scope** includes all commands which reference the queries in **queryPool** indicated by **query** that occur earlier in submission order.

The second **synchronization scope** includes all commands which reference the queries in **queryPool** indicated by **query** that occur later in submission order.

The operation of this command happens after the first scope and happens before the second scope.

### Valid Usage

- **VUID-vkCmdBeginQuery-None-00807**
  All queries used by the command must be unavailable

- **VUID-vkCmdBeginQuery-queryType-02804**
  The **queryType** used to create **queryPool** must not be **VK_QUERY_TYPE_TIMESTAMP**

- **VUID-vkCmdBeginQuery-queryType-00800**
  If the **occlusionQueryPrecise** 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**

- **VUID-vkCmdBeginQuery-query-00802**
  **query** must be less than the number of queries in **queryPool**

- **VUID-vkCmdBeginQuery-queryType-00803**
  If the **queryType** used to create **queryPool** was **VK_QUERY_TYPE_OCCLUSION**, the **VkCommandPool** that **commandBuffer** was allocated from must support graphics operations

- **VUID-vkCmdBeginQuery-queryType-00804**
  If the **queryType** used to create **queryPool** was **VK_QUERY_TYPE_PIPELINE_STATISTICS** and any of the **pipelineStatistics** indicate graphics operations, the **VkCommandPool** that **commandBuffer** was allocated from must support graphics operations

- **VUID-vkCmdBeginQuery-queryType-00805**
If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_PIPELINE_STATISTICS` and any of the `pipelineStatistics` indicate compute operations, the `VkCommandPool` that `commandBuffer` was allocated from must support compute operations.

- VUID-vkCmdBeginQuery-commandBuffer-01885
  `commandBuffer` must not be a protected command buffer

- VUID-vkCmdBeginQuery-query-00808
  If called within a render pass instance, the sum of `query` and the number of bits set in the current subpass's view mask must be less than or equal to the number of queries in `queryPool`

- VUID-vkCmdBeginQuery-queryPool-01922
  `queryPool` must have been created with a `queryType` that differs from that of any queries that are active within `commandBuffer`

**Valid Usage (Implicit)**

- VUID-vkCmdBeginQuery-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdBeginQuery-queryPool-parameter
  `queryPool` must be a valid `VkQueryPool` handle

- VUID-vkCmdBeginQuery-flags-parameter
  `flags` must be a valid combination of `VkQueryControlFlagBits` values

- VUID-vkCmdBeginQuery-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdBeginQuery-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, compute, decode, or encode operations

- VUID-vkCmdBeginQuery-commonparent
  Both of `commandBuffer`, and `queryPool` must have been created, allocated, or retrieved from the same `VkDevice`

**Host Synchronization**

- Host access to `commandBuffer` must be externally synchronized

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

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics, Compute, Decode, Encode</td>
<td>Action, State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits which **can** be set in `vkCmdBeginQuery::flags`, specifying constraints on the types of queries that **can** be performed, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkQueryControlFlagBits {
    VK_QUERY_CONTROL_PRECISE_BIT = 0x00000001,
} VkQueryControlFlagBits;
```

- **VK_QUERY_CONTROL_PRECISE_BIT** specifies the precision of occlusion queries.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkQueryControlFlags;
```

`VkQueryControlFlags` is a bitmask type for setting a mask of zero or more `VkQueryControlFlagBits`.

To end a query after the set of desired drawing or dispatching commands is executed, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdEndQuery(
    VkCommandBuffer commandBuffer,
    VkQueryPool queryPool,
    uint32_t query);
```

- **commandBuffer** is the command buffer into which this command will be recorded.
- **queryPool** is the query pool that is managing the results of the query.
- **query** is the query index within the query pool where the result is stored.

The command completes the query in `queryPool` identified by `query`, and marks it as available.

This command defines an execution dependency between other query commands that reference the same query.

The first **synchronization scope** includes all commands which reference the queries in `queryPool` indicated by `query` that occur earlier in submission order.
The second synchronization scope includes only the operation of this command.

### Valid Usage

- **VUID-vkCmdEndQuery-None-01923**
  All queries used by the command **must** be **active**

- **VUID-vkCmdEndQuery-query-00810**
  Query **must** be less than the number of queries in **queryPool**

- **VUID-vkCmdEndQuery-commandBuffer-01886**
  CommandBuffer **must** not be a protected command buffer

- **VUID-vkCmdEndQuery-query-00812**
  If `vkCmdEndQuery` is called within a render pass instance, the sum of query and the number of bits set in the current subpass’s view mask **must** be less than or equal to the number of queries in **queryPool**

- **VUID-vkCmdBeginQuery-None-07007**
  If called within a subpass of a render pass instance, the corresponding `vkCmdBeginQuery`* command **must** have been called previously within the same subpass

- **VUID-vkCmdBeginQuery-None-07008**
  If called outside of a render pass instance, the corresponding `vkCmdBeginQuery`* command **must** have been called outside of a render pass instance

### Valid Usage (Implicit)

- **VUID-vkCmdEndQuery-commandBuffer-parameter**
  CommandBuffer **must** be a valid `VkCommandBuffer` handle

- **VUID-vkCmdEndQuery-queryPool-parameter**
  QueryPool **must** be a valid `VkQueryPool` handle

- **VUID-vkCmdEndQuery-commandBuffer-recording**
  CommandBuffer **must** be in the recording state

- **VUID-vkCmdEndQuery-commandBuffer-cmdpool**
  The `VkCommandPool` that CommandBuffer was allocated from **must** support graphics, compute, decode, or encode operations

- **VUID-vkCmdEndQuery-commonparent**
  Both of CommandBuffer, and QueryPool **must** have been created, allocated, or retrieved from the same `VkDevice`

### Host Synchronization

- Host access to CommandBuffer **must** be externally synchronized

- Host access to the `VkCommandPool` that CommandBuffer was allocated from **must** be externally synchronized
An application can retrieve results either by requesting they be written into application-provided memory, or by requesting they be copied into a VkBuffer. In either case, the layout in memory is defined as follows:

- The first query's result is written starting at the first byte requested by the command, and each subsequent query's result begins stride bytes later.
- Occlusion queries, pipeline statistics queries, and timestamp queries store results in a tightly packed array of unsigned integers, either 32- or 64-bits as requested by the command, storing the numerical results and, if requested, the availability status.
- If VK_QUERY_RESULT_WITH_AVAILABILITY_BIT is used, the final element of each query's result is an integer indicating whether the query's result is available, with any non-zero value indicating that it is available.
- Occlusion queries write one integer value - the number of samples passed. Pipeline statistics queries write one integer value for each bit that is enabled in the pipelineStatistics when the pool is created, and the statistics values are written in bit order starting from the least significant bit. Timestamp queries write one integer value.
- If more than one query is retrieved and stride is not at least as large as the size of the array of values corresponding to a single query, the values written to memory are undefined.

To retrieve status and results for a set of queries, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkGetQueryPoolResults(
    VkDevice device,  // Logical device that owns the query pool.
    VkQueryPool queryPool,  // Query pool managing the queries containing the desired results.
    uint32_t firstQuery,  // Initial query index.
    uint32_t queryCount,  // Number of queries to retrieve.
    size_t dataSize,  // Size of the data buffer.
    void* pData,  // Pointer to the data buffer.
    VkDeviceSize stride,  // Stride between query results.
    VkQueryResultFlags flags);  // Flags indicating availability status.
```
• **queryCount** is the number of queries to read.

• **dataSize** is the size in bytes of the buffer pointed to by **pData**.

• **pData** is a pointer to a user-allocated buffer where the results will be written

• **stride** is the stride in bytes between results for individual queries within **pData**.

• **flags** is a bitmask of **VkQueryResultFlagBits** specifying how and when results are returned.

Any results written for a query are written according to a layout dependent on the query type.

If no bits are set in **flags**, and all requested queries are in the available state, results are written as an array of 32-bit unsigned integer values. Behavior when not all queries are available is described below.

If **VK_QUERY_RESULT_WITH_AVAILABILITY_BIT** is set, results for all queries in **queryPool** identified by **firstQuery** and **queryCount** are copied to **pData**, along with an extra availability value written directly after the results of each query and interpreted as an unsigned integer. A value of zero indicates that the results are not yet available, otherwise the query is complete and results are available.

Results for any available query written by this command are final and represent the final result of the query. If **VK_QUERY_RESULT_PARTIAL_BIT** is set, then for any query that is unavailable, an intermediate result between zero and the final result value is written for that query. Otherwise, any result written by this command is undefined.

If **VK_QUERY_RESULT_64_BIT** is set, results and availability values for all queries are written as an array of 64-bit values. Otherwise, results and availability values are written as an array of 32-bit values. If an unsigned integer query’s value overflows the result type, the value may either wrap or saturate.

If **VK_QUERY_RESULT_WAIT_BIT** is set, this command defines an execution dependency with any earlier commands that writes one of the identified queries. The first synchronization scope includes all instances of **vkCmdEndQuery**, **vkCmdWriteTimestamp2**, and **vkCmdWriteTimestamp** that reference any query in **queryPool** indicated by **firstQuery** and **queryCount**. The second synchronization scope includes the host operations of this command.

If **VK_QUERY_RESULT_WAIT_BIT** is not set, **vkGetQueryPoolResults** may return **VK_NOT_READY** if there are queries in the unavailable state.

---

**Note**

Applications must take care to ensure that use of the **VK_QUERY_RESULT_WAIT_BIT** bit has the desired effect.

For example, if a query has been used previously and a command buffer records the commands **vkCmdResetQueryPool**, **vkCmdBeginQuery**, and **vkCmdEndQuery** for that query, then the query will remain in the available state until **vkResetQueryPool** is called or the **vkCmdResetQueryPool** command executes on a queue. Applications can use fences or events to ensure that a query has already been reset before checking for its results or availability status. Otherwise, a stale value could be returned from a previous use of the query.
The above also applies when `VKQUERYRESULT_WAIT_BIT` is used in combination with `VKQUERYRESULT_WITH_AVAILABILITY_BIT`. In this case, the returned availability status may reflect the result of a previous use of the query unless `vkResetQueryPool` is called or the `vkCmdResetQueryPool` command has been executed since the last use of the query.

**Note**
Applications can double-buffer query pool usage, with a pool per frame, and reset queries at the end of the frame in which they are read.

**Valid Usage**

- VUID-vkGetQueryPoolResults-firstQuery-00813
  *firstQuery* must be less than the number of queries in `queryPool`

- VUID-vkGetQueryPoolResults-flags-02827
  If `VK_QUERY_RESULT_64_BIT` is not set in `flags`, then `pData` and `stride` must be multiples of 4

- VUID-vkGetQueryPoolResults-flags-00815
  If `VK_QUERY_RESULT_64_BIT` is set in `flags` then `pData` and `stride` must be multiples of 8

- VUID-vkGetQueryPoolResults-firstQuery-00816
  The sum of `firstQuery` and `queryCount` must be less than or equal to the number of queries in `queryPool`

- VUID-vkGetQueryPoolResults-dataSize-00817
  `dataSize` must be large enough to contain the result of each query, as described here

- VUID-vkGetQueryPoolResults-queryType-00818
  If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TIMESTAMP`, `flags` must not contain `VK_QUERY_RESULT_PARTIAL_BIT`

**Valid Usage (Implicit)**

- VUID-vkGetQueryPoolResults-device-parameter
  `device` must be a valid `VkDevice` handle

- VUID-vkGetQueryPoolResults-queryPool-parameter
  `queryPool` must be a valid `VkQueryPool` handle

- VUID-vkGetQueryPoolResults-pData-parameter
  `pData` must be a valid pointer to an array of `dataSize` bytes

- VUID-vkGetQueryPoolResults-flags-parameter
  `flags` must be a valid combination of `VkQueryResultFlagBits` values

- VUID-vkGetQueryPoolResults-dataSize-arraylength
  `dataSize` must be greater than 0

- VUID-vkGetQueryPoolResults-queryPool-parent
  `queryPool` must have been created, allocated, or retrieved from `device`
Return Codes

Success
  • VK_SUCCESS
  • VK_NOT_READY

Failure
  • VK_ERROR_OUT_OF_HOST_MEMORY
  • VK_ERROR_OUT_OF_DEVICE_MEMORY
  • VK_ERROR_DEVICE_LOST

Bits which can be set in VkGetQueryPoolResults::flags and VkCmdCopyQueryPoolResults::flags, specifying how and when results are returned, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkQueryResultFlagBits {
    VK_QUERY_RESULT_64_BIT = 0x00000001,
    VK_QUERY_RESULT_WAIT_BIT = 0x00000002,
    VK_QUERY_RESULT_WITH_AVAILABILITY_BIT = 0x00000004,
    VK_QUERY_RESULT_PARTIAL_BIT = 0x00000008,
} VkQueryResultFlagBits;
```

- **VK_QUERY_RESULT_64_BIT** specifies the results will be written as an array of 64-bit unsigned integer values. If this bit is not set, the results will be written as an array of 32-bit unsigned integer values.
- **VK_QUERY_RESULT_WAIT_BIT** specifies that Vulkan will wait for each query’s status to become available before retrieving its results.
- **VK_QUERY_RESULT_WITH_AVAILABILITY_BIT** specifies that the availability status accompanies the results.
- **VK_QUERY_RESULT_PARTIAL_BIT** specifies that returning partial results is acceptable.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkQueryResultFlags;
```

VkQueryResultFlags is a bitmask type for setting a mask of zero or more VkQueryResultFlagBits.

To copy query statuses and numerical results directly to buffer memory, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdCopyQueryPoolResults(
    VkCommandBuffer commandBuffer,
    VkQueryPool queryPool,
    uint32_t firstQuery,
    ...) {
```
uint32_t
VkBuffer
VkDeviceSize
VkDeviceSize
VkQueryResultFlags

• **commandBuffer** is the command buffer into which this command will be recorded.

• **queryPool** is the query pool managing the queries containing the desired results.

• **firstQuery** is the initial query index.

• **queryCount** is the number of queries. **firstQuery** and **queryCount** together define a range of queries.

• **dstBuffer** is a **VkBuffer** object that will receive the results of the copy command.

• **dstOffset** is an offset into **dstBuffer**.

• **stride** is the stride in bytes between results for individual queries within **dstBuffer**. The required size of the backing memory for **dstBuffer** is determined as described above for **vkGetQueryPoolResults**.

• **flags** is a bitmask of **VkQueryResultFlagBits** specifying how and when results are returned.

Any results written for a query are written according to a **layout dependent on the query type**.

Results for any query in **queryPool** identified by **firstQuery** and **queryCount** that is available are copied to **dstBuffer**.

If **VK_QUERY_RESULT_WITH_AVAILABILITY_BIT** is set, results for all queries in **queryPool** identified by **firstQuery** and **queryCount** are copied to **dstBuffer**, along with an extra availability value written directly after the results of each query and interpreted as an unsigned integer. A value of zero indicates that the results are not yet available, otherwise the query is complete and results are available.

Results for any available query written by this command are final and represent the final result of the query. If **VK_QUERY_RESULT_PARTIAL_BIT** is set, then for any query that is unavailable, an intermediate result between zero and the final result value is written for that query. Otherwise, any result written by this command is undefined.

If **VK_QUERY_RESULT_64_BIT** is set, results and availability values for all queries are written as an array of 64-bit values. Otherwise, results and availability values are written as an array of 32-bit values. If an unsigned integer query's value overflows the result type, the value **may** either wrap or saturate.

This command defines an execution dependency between other query commands that reference the same query.

The first **synchronization scope** includes all commands which reference the queries in **queryPool** indicated by **query** that occur earlier in **submission order**. If **flags** does not include **VK_QUERY_RESULT_WAIT_BIT**, **vkCmdWriteTimestamp2**, **vkCmdEndQuery**, and **vkCmdWriteTimestamp** are excluded from this scope.
The second synchronization scope includes all commands which reference the queries in queryPool indicated by query that occur later in submission order.

The operation of this command happens after the first scope and happens before the second scope.

vkCmdCopyQueryPoolResults is considered to be a transfer operation, and its writes to buffer memory must be synchronized using VK_PIPELINE_STAGE_TRANSFER_BIT and VK_ACCESS_TRANSFER_WRITE_BIT before using the results.

 Valid Usage

- VUID-vkCmdCopyQueryPoolResults-dstOffset-00819
dstOffset must be less than the size of dstBuffer
- VUID-vkCmdCopyQueryPoolResults-firstQuery-00820
firstQuery must be less than the number of queries in queryPool
- VUID-vkCmdCopyQueryPoolResults-firstQuery-00821
The sum of firstQuery and queryCount must be less than or equal to the number of queries in queryPool
- VUID-vkCmdCopyQueryPoolResults-flags-00822
If VK_QUERY_RESULT_64_BIT is not set in flags then dstOffset and stride must be multiples of 4
- VUID-vkCmdCopyQueryPoolResults-flags-00823
If VK_QUERY_RESULT_64_BIT is set in flags then dstOffset and stride must be multiples of 8
- VUID-vkCmdCopyQueryPoolResults-dstBuffer-00824
dstBuffer must have enough storage, from dstOffset, to contain the result of each query, as described here
- VUID-vkCmdCopyQueryPoolResults-dstBuffer-00825
dstBuffer must have been created with VK_BUFFER_USAGE_TRANSFER_DST_BIT usage flag
- VUID-vkCmdCopyQueryPoolResults-dstBuffer-00826
If dstBuffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object
- VUID-vkCmdCopyQueryPoolResults-queryType-00827
If the queryType used to create queryPool was VK_QUERY_TYPE_TIMESTAMP, flags must not contain VK_QUERY_RESULT_PARTIAL_BIT
- VUID-vkCmdCopyQueryPoolResults-None-07429
All queries used by the command must not be active

 Valid Usage (Implicit)

- VUID-vkCmdCopyQueryPoolResults-commandBuffer-parameter
commandBuffer must be a valid VkCommandBuffer handle
- VUID-vkCmdCopyQueryPoolResults-queryPool-parameter
queryPool must be a valid VkQueryPool handle
• VUID-vkCmdCopyQueryPoolResults-dstBuffer-parameter
dstBuffer must be a valid VkBuffer handle

• VUID-vkCmdCopyQueryPoolResults-flags-parameter
flags must be a valid combination of VkQueryResultFlagBits values

• VUID-vkCmdCopyQueryPoolResults-commandBuffer-recording
commandBuffer must be in the recording state

• VUID-vkCmdCopyQueryPoolResults-commandBuffer-cmdpool
The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations

• VUID-vkCmdCopyQueryPoolResults-renderpass
This command must only be called outside of a render pass instance

• VUID-vkCmdCopyQueryPoolResults-commonparent
Each of commandBuffer, dstBuffer, and queryPool must have been created, allocated, or retrieved from the same VkDevice

### Host Synchronization

• Host access to commandBuffer must be externally synchronized

• Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

Rendering operations such as clears, MSAA resolves, attachment load/store operations, and blits may count towards the results of queries. This behavior is implementation-dependent and may vary depending on the path used within an implementation. For example, some implementations have several types of clears, some of which may include vertices and some not.

### 17.3. Occlusion Queries

Occlusion queries track the number of samples that pass the per-fragment tests for a set of drawing commands. As such, occlusion queries are only available on queue families supporting graphics operations. The application can then use these results to inform future rendering decisions. An occlusion query is begun and ended by calling vkCmdBeginQuery and vkCmdEndQuery, respectively. When an occlusion query begins, the count of passing samples always starts at zero. For each drawing command, the count is incremented as described in Sample Counting. If flags does not contain VK_QUERY_CONTROL_PRECISE_BIT an implementation may generate any non-zero result value.
for the query if the count of passing samples is non-zero.

**Note**

Not setting `VK_QUERY_CONTROL_PRECISE_BIT` mode **may** be more efficient on some implementations, and **should** be used where it is sufficient to know a boolean result on whether any samples passed the per-fragment tests. In this case, some implementations **may** only return zero or one, indifferent to the actual number of samples passing the per-fragment tests.

When an occlusion query finishes, the result for that query is marked as available. The application **can** then either copy the result to a buffer (via `vkCmdCopyQueryPoolResults`) or request it be put into host memory (via `vkGetQueryPoolResults`).

**Note**

If occluding geometry is not drawn first, samples **can** pass the depth test, but still not be visible in a final image.

### 17.4. Pipeline Statistics Queries

Pipeline statistics queries allow the application to sample a specified set of `VkPipeline` counters. These counters are accumulated by Vulkan for a set of either drawing or dispatching commands while a pipeline statistics query is active. As such, pipeline statistics queries are available on queue families supporting either graphics or compute operations. The availability of pipeline statistics queries is indicated by the `pipelineStatisticsQuery` member of the `VkPhysicalDeviceFeatures` object (see `vkGetPhysicalDeviceFeatures` and `vkCreateDevice` for detecting and requesting this query type on a `VkDevice`).

A pipeline statistics query is begun and ended by calling `vkCmdBeginQuery` and `vkCmdEndQuery`, respectively. When a pipeline statistics query begins, all statistics counters are set to zero. While the query is active, the pipeline type determines which set of statistics are available, but these **must** be configured on the query pool when it is created. If a statistic counter is issued on a command buffer that does not support the corresponding operation, the value of that counter is undefined after the query has been made available. At least one statistic counter relevant to the operations supported on the recording command buffer **must** be enabled.

Bits which **can** be set in `VkQueryPoolCreateInfo::pipelineStatistics` for query pools and in `VkCommandBufferInheritanceInfo::pipelineStatistics` for secondary command buffers, individually enabling pipeline statistics counters, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkQueryPipelineStatisticFlagBits {
    VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_VERTICES_BIT = 0x00000001,
    VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_PRIMITIVES_BIT = 0x00000002,
    VK_QUERY_PIPELINE_STATISTIC_VERTEX_SHADER_INVOCATIONS_BIT = 0x00000004,
    VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_INVOCATIONS_BIT = 0x00000008,
    VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_PRIMITIVES_BIT = 0x00000010,
    VK_QUERY_PIPELINE_STATISTIC_CLIPPING_INVOCATIONS_BIT = 0x00000020,
}```
\[
\begin{align*}
\text{VK_QUERY_PIPELINE_STATISTIC_CLIPPING_PRIMITIVES_BIT} & = 0x00000040, \\
\text{VK_QUERY_PIPELINE_STATISTIC_FRAGMENT_SHADER_INVOCATIONS_BIT} & = 0x00000080, \\
\text{VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_CONTROL_SHADER_PATCHES_BIT} & = 0x00000100, \\
\text{VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_EVALUATION_SHADER_INVOCATIONS_BIT} & = 0x00000200, \\
\text{VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT} & = 0x00000400,
\end{align*}
\]

\begin{itemize}
\item \textbf{VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_VERTICES_BIT} specifies that queries managed by the pool will count the number of vertices processed by the \textit{input assembly} stage. Vertices corresponding to incomplete primitives \textbf{may} contribute to the count.
\item \textbf{VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_PRIMITIVES_BIT} specifies that queries managed by the pool will count the number of primitives processed by the \textit{input assembly} stage. If primitive restart is enabled, restarting the primitive topology has no effect on the count. Incomplete primitives \textbf{may} be counted.
\item \textbf{VK_QUERY_PIPELINE_STATISTIC_VERTEX_SHADER_INVOCATIONS_BIT} specifies that queries managed by the pool will count the number of vertex shader invocations. This counter's value is incremented each time a vertex shader is \textit{invoked}.
\item \textbf{VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_INVOCATIONS_BIT} specifies that queries managed by the pool will count the number of geometry shader invocations. This counter's value is incremented each time a geometry shader is \textit{invoked}. In the case of \textit{instanced geometry shaders}, the geometry shader invocations count is incremented for each separate instanced invocation.
\item \textbf{VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_PRIMITIVES_BIT} specifies that queries managed by the pool will count the number of primitives generated by geometry shader invocations. The counter's value is incremented each time the geometry shader emits a primitive. Restarting primitive topology using the SPIR-V instructions \texttt{OpEndPrimitive} or \texttt{OpEndStreamPrimitive} has no effect on the geometry shader output primitives count.
\item \textbf{VK_QUERY_PIPELINE_STATISTIC_CLIPPING_INVOCATIONS_BIT} specifies that queries managed by the pool will count the number of primitives processed by the \textit{Primitive Clipping} stage of the pipeline. The counter's value is incremented each time a primitive reaches the primitive clipping stage.
\item \textbf{VK_QUERY_PIPELINE_STATISTIC_CLIPPING_PRIMITIVES_BIT} specifies that queries managed by the pool will count the number of primitives output by the \textit{Primitive Clipping} stage of the pipeline. The actual number of primitives output by the primitive clipping stage for a particular input primitive is implementation-dependent but \textbf{must} satisfy the following conditions:
\begin{itemize}
\item If at least one vertex of the input primitive lies inside the clipping volume, the counter is incremented by one or more.
\item Otherwise, the counter is incremented by zero or more.
\end{itemize}
\item \textbf{VK_QUERY_PIPELINE_STATISTIC_FRAGMENT_SHADER_INVOCATIONS_BIT} specifies that queries managed by the pool will count the number of fragment shader invocations. The counter's value is incremented each time the fragment shader is \textit{invoked}.
\item \textbf{VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_CONTROL_SHADER_PATCHES_BIT} specifies that queries
managed by the pool will count the number of patches processed by the tessellation control shader. The counter's value is incremented once for each patch for which a tessellation control shader is invoked.

- **VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_EVALUATION_SHADER_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of invocations of the tessellation evaluation shader. The counter's value is incremented each time the tessellation evaluation shader is invoked.

- **VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of compute shader invocations. The counter's value is incremented every time the compute shader is invoked. Implementations may skip the execution of certain compute shader invocations or execute additional compute shader invocations for implementation-dependent reasons as long as the results of rendering otherwise remain unchanged.

These values are intended to measure relative statistics on one implementation. Various device architectures will count these values differently. Any or all counters may be affected by the issues described in Query Operation.

**Note**

For example, tile-based rendering devices may need to replay the scene multiple times, affecting some of the counts.

If a pipeline has `rasterizerDiscardEnable` enabled, implementations may discard primitives after the final pre-rasterization shader stage. As a result, if `rasterizerDiscardEnable` is enabled, the clipping input and output primitives counters may not be incremented.

When a pipeline statistics query finishes, the result for that query is marked as available. The application can copy the result to a buffer (via `vkCmdCopyQueryPoolResults`), or request it be put into host memory (via `vkGetQueryPoolResults`).

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkQueryPipelineStatisticFlags;
```

`VkQueryPipelineStatisticFlags` is a bitmask type for setting a mask of zero or more `VkQueryPipelineStatisticFlagBits`.

### 17.5. Timestamp Queries

Timestamps provide applications with a mechanism for timing the execution of commands. A timestamp is an integer value generated by the `VkPhysicalDevice`. Unlike other queries, timestamps do not operate over a range, and so do not use `vkCmdBeginQuery` or `vkCmdEndQuery`. The mechanism is built around a set of commands that allow the application to tell the `VkPhysicalDevice` to write timestamp values to a query pool and then either read timestamp values on the host (using `vkGetQueryPoolResults`) or copy timestamp values to a `VkBuffer` (using `vkCmdCopyQueryPoolResults`). The application can then compute differences between timestamps to determine execution time.
The number of valid bits in a timestamp value is determined by the `VkQueueFamilyProperties::timestampValidBits` property of the queue on which the timestamp is written. Timestamps are supported on any queue which reports a non-zero value for `timestampValidBits` via `vkGetPhysicalDeviceQueueFamilyProperties`. If the `timestampComputeAndGraphics` limit is `VK_TRUE`, timestamps are supported by every queue family that supports either graphics or compute operations (see `VkQueueFamilyProperties`).

The number of nanoseconds it takes for a timestamp value to be incremented by 1 can be obtained from `VkPhysicalDeviceLimits::timestampPeriod` after a call to `vkGetPhysicalDeviceProperties`.

To request a timestamp and write the value to memory, call:

```c
// Provided by VK_VERSION_1_3
void vkCmdWriteTimestamp2(
    VkCommandBuffer commandBuffer,  // Provided by VK_VERSION_1_3
    VkPipelineStageFlags2 stage,    // Provided by VK_VERSION_1_3
    VkQueryPool queryPool,          // Provided by VK_VERSION_1_3
    uint32_t query);                // Provided by VK_VERSION_1_3
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `stage` specifies a stage of the pipeline.
- `queryPool` is the query pool that will manage the timestamp.
- `query` is the query within the query pool that will contain the timestamp.

When `vkCmdWriteTimestamp2` is submitted to a queue, it defines an execution dependency on commands that were submitted before it, and writes a timestamp to a query pool.

The first synchronization scope includes all commands that occur earlier in submission order. The synchronization scope is limited to operations on the pipeline stage specified by `stage`.

The second synchronization scope includes only the timestamp write operation.

When the timestamp value is written, the availability status of the query is set to available.

**Note**

If an implementation is unable to detect completion and latch the timer immediately after `stage` has completed, it **may** instead do so at any logically later stage.

Comparisons between timestamps are not meaningful if the timestamps are written by commands submitted to different queues.

**Note**

An example of such a comparison is subtracting an older timestamp from a newer one to determine the execution time of a sequence of commands.

If `vkCmdWriteTimestamp2` is called while executing a render pass instance that has multiview enabled,
the timestamp uses \( N \) consecutive query indices in the query pool (starting at \texttt{query}) where \( N \) is the number of bits set in the view mask of the subpass the command is executed in. The resulting query values are determined by an implementation-dependent choice of one of the following behaviors:

- The first query is a timestamp value and (if more than one bit is set in the view mask) zero is written to the remaining queries. If two timestamps are written in the same subpass, the sum of the execution time of all views between those commands is the difference between the first query written by each command.

- All \( N \) queries are timestamp values. If two timestamps are written in the same subpass, the sum of the execution time of all views between those commands is the sum of the difference between corresponding queries written by each command. The difference between corresponding queries \texttt{may} be the execution time of a single view.

In either case, the application \texttt{can} sum the differences between all \( N \) queries to determine the total execution time.

### Valid Usage

- \texttt{VUID-vkCmdWriteTimestamp2-stage-03929}
  If the \texttt{geometryShader} feature is not enabled, \texttt{stage} \texttt{must} not contain \texttt{VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT}

- \texttt{VUID-vkCmdWriteTimestamp2-stage-03930}
  If the \texttt{tessellationShader} feature is not enabled, \texttt{stage} \texttt{must} not contain \texttt{VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT} or \texttt{VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT}

- \texttt{VUID-vkCmdWriteTimestamp2-synchronization2-03858}
  The \texttt{synchronization2} feature \texttt{must} be enabled

- \texttt{VUID-vkCmdWriteTimestamp2-stage-03859}
  \texttt{stage} \texttt{must} only include a single pipeline stage

- \texttt{VUID-vkCmdWriteTimestamp2-stage-03860}
  \texttt{stage} \texttt{must} only include stages valid for the queue family that was used to create the command pool that \texttt{commandBuffer} was allocated from

- \texttt{VUID-vkCmdWriteTimestamp2-queryPool-03861}
  \texttt{queryPool} \texttt{must} have been created with a \texttt{queryType} of \texttt{VK_QUERY_TYPE_TIMESTAMP}

- \texttt{VUID-vkCmdWriteTimestamp2-queryPool-03862}
  The query identified by \texttt{queryPool} and \texttt{query} \texttt{must} be \texttt{unavailable}

- \texttt{VUID-vkCmdWriteTimestamp2-timestampValidBits-03863}
  The command pool's queue family \texttt{must} support a non-zero \texttt{timestampValidBits}

- \texttt{VUID-vkCmdWriteTimestamp2-query-04903}
  \texttt{query} \texttt{must} be less than the number of queries in \texttt{queryPool}

- \texttt{VUID-vkCmdWriteTimestamp2-None-03864}
  All queries used by the command \texttt{must} be \texttt{unavailable}

- \texttt{VUID-vkCmdWriteTimestamp2-query-03865}
  If \texttt{vkCmdWriteTimestamp2} is called within a render pass instance, the sum of \texttt{query} and the
number of bits set in the current subpass's view mask must be less than or equal to the number of queries in queryPool

**Valid Usage (Implicit)**

- VUID-vkCmdWriteTimestamp2-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle
- VUID-vkCmdWriteTimestamp2-stage-parameter
  stage must be a valid combination of VkPipelineStageFlagBits2 values
- VUID-vkCmdWriteTimestamp2-queryPool-parameter
  queryPool must be a valid VkQueryPool handle
- VUID-vkCmdWriteTimestamp2-commandBuffer-recording
  commandBuffer must be in the recording state
- VUID-vkCmdWriteTimestamp2-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support transfer, graphics, compute, decode, or encode operations
- VUID-vkCmdWriteTimestamp2-commonparent
  Both of commandBuffer, and queryPool must have been created, allocated, or retrieved from the same VkDevice

**Host Synchronization**

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

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Secondary</td>
<td>Both</td>
<td>Transfer Graphics</td>
<td>Action</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Decode</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Encode</td>
<td></td>
</tr>
</tbody>
</table>

To request a timestamp and write the value to memory, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdWriteTimestamp(
  VkCommandBuffer commandBuffer,
```
commandBuffer is the command buffer into which the command will be recorded.

pipelineStage is a VkPipelineStageFlagBits value, specifying a stage of the pipeline.

queryPool is the query pool that will manage the timestamp.

query is the query within the query pool that will contain the timestamp.

When vkCmdWriteTimestamp is submitted to a queue, it defines an execution dependency on commands that were submitted before it, and writes a timestamp to a query pool.

The first synchronization scope includes all commands that occur earlier in submission order. The synchronization scope is limited to operations on the pipeline stage specified by pipelineStage.

The second synchronization scope includes only the timestamp write operation.

When the timestamp value is written, the availability status of the query is set to available.

Note
If an implementation is unable to detect completion and latch the timer immediately after stage has completed, it may instead do so at any logically later stage.

Comparisons between timestamps are not meaningful if the timestamps are written by commands submitted to different queues.

Note
An example of such a comparison is subtracting an older timestamp from a newer one to determine the execution time of a sequence of commands.

If vkCmdWriteTimestamp is called while executing a render pass instance that has multiview enabled, the timestamp uses N consecutive query indices in the query pool (starting at query) where N is the number of bits set in the view mask of the subpass the command is executed in. The resulting query values are determined by an implementation-dependent choice of one of the following behaviors:

• The first query is a timestamp value and (if more than one bit is set in the view mask) zero is written to the remaining queries. If two timestamps are written in the same subpass, the sum of the execution time of all views between those commands is the difference between the first query written by each command.

• All N queries are timestamp values. If two timestamps are written in the same subpass, the sum of the execution time of all views between those commands is the sum of the difference between corresponding queries written by each command. The difference between corresponding queries may be the execution time of a single view.

In either case, the application can sum the differences between all N queries to determine the total execution time.
Valid Usage

- VUID-vkCmdWriteTimestamp-pipelineStage-04074
  
  ```
  pipelineStage must be a valid stage for the queue family that was used to create the command pool that commandBuffer was allocated from
  ```

- VUID-vkCmdWriteTimestamp-pipelineStage-04075
  
  If the geometryShader feature is not enabled, pipelineStage must not be `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`

- VUID-vkCmdWriteTimestamp-pipelineStage-04076
  
  If the tessellationShader feature is not enabled, pipelineStage must not be `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`

- VUID-vkCmdWriteTimestamp-synchronization2-06489
  
  If the synchronization2 feature is not enabled, pipelineStage must not be `VK_PIPELINE_STAGE_NONE`

- VUID-vkCmdWriteTimestamp-queryPool-01416
  
  queryPool must have been created with a queryType of `VK_QUERY_TYPE_TIMESTAMP`

- VUID-vkCmdWriteTimestamp-queryPool-00828
  
  The query identified by queryPool and query must be unavailable

- VUID-vkCmdWriteTimestamp-timestampValidBits-00829
  
  The command pool's queue family must support a non-zero timestampValidBits

- VUID-vkCmdWriteTimestamp-query-04904
  
  query must be less than the number of queries in queryPool

- VUID-vkCmdWriteTimestamp-None-00830
  
  All queries used by the command must be unavailable

- VUID-vkCmdWriteTimestamp-query-00831
  
  If vkCmdWriteTimestamp is called within a render pass instance, the sum of query and the number of bits set in the current subpass's view mask must be less than or equal to the number of queries in queryPool

Valid Usage (Implicit)

- VUID-vkCmdWriteTimestamp-commandBuffer-parameter
  
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdWriteTimestamp-pipelineStage-parameter
  
  pipelineStage must be a valid VkPipelineStageFlagBits value

- VUID-vkCmdWriteTimestamp-queryPool-parameter
  
  queryPool must be a valid VkQueryPool handle

- VUID-vkCmdWriteTimestamp-commandBuffer-recording
  
  commandBuffer must be in the recording state

- VUID-vkCmdWriteTimestamp-commandBuffer-cmdpool
The VkCommandPool that commandBuffer was allocated from must support transfer, graphics, compute, decode, encode, or opticalflow operations.

- VUID-vkCmdWriteTimestamp-commonparent
  Both of commandBuffer, and queryPool must have been created, allocated, or retrieved from the same VkDevice.

Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Transfer Graphics, Compute, Decode, Encode, Opticalflow</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 18. Clear Commands

18.1. Clearing Images Outside A Render Pass Instance

Color and depth/stencil images can be cleared outside a render pass instance using `vkCmdClearColorImage` or `vkCmdClearDepthStencilImage`, respectively. These commands are only allowed outside of a render pass instance.

To clear one or more subranges of a color image, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdClearColorImage(  
    VkCommandBuffer commandBuffer,  
    VkImage image,  
    VkImageLayout imageLayout,  
    const VkClearColorValue* pColor,  
    uint32_t rangeCount,  
    const VkImageSubresourceRange* pRanges);
```

- **commandBuffer** is the command buffer into which the command will be recorded.
- **image** is the image to be cleared.
- **imageLayout** specifies the current layout of the image subresource ranges to be cleared, and must be `VK_IMAGE_LAYOUT_GENERAL` or `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`.
- **pColor** is a pointer to a `VkClearColorValue` structure containing the values that the image subresource ranges will be cleared to (see Clear Values below).
- **rangeCount** is the number of image subresource range structures in **pRanges**.
- **pRanges** is a pointer to an array of `VkImageSubresourceRange` structures describing a range of mipmap levels, array layers, and aspects to be cleared, as described in Image Views.

Each specified range in **pRanges** is cleared to the value specified by **pColor**.

Valid Usage

- VUID-vkCmdClearColorImage-image-01993
  The format features of **image** must contain `VK_FORMAT_FEATURE_TRANSFER_DST_BIT`

- VUID-vkCmdClearColorImage-image-00002
  **image** must have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag

- VUID-vkCmdClearColorImage-image-01545
  **image** must not use any of the formats that require a sampler Y’CbCr conversion

- VUID-vkCmdClearColorImage-image-00003
  If **image** is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object

- VUID-vkCmdClearColorImage-imageLayout-00004
**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**

- **VID-vkCmdClearColorImage-imageLayout-00005**
  
  **imageLayout must** be **VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL** or **VK_IMAGE_LAYOUT_GENERAL**

- **VID-vkCmdClearColorImage-aspectMask-02498**
  
  The **VkImageSubresourceRange::aspectMask** members of the elements of the **pRanges** array **must** each only include **VK_IMAGE_ASPECT_COLOR_BIT**

- **VID-vkCmdClearColorImage-baseMipLevel-01470**
  
  The **VkImageSubresourceRange::baseMipLevel** members of the elements of the **pRanges** array **must** each be less than the **mipLevels** specified in **VkImageCreateInfo** when **image** was created

- **VID-vkCmdClearColorImage-pRanges-01692**
  
  For each **VkImageSubresourceRange** element of **pRanges**, if the **levelCount** member is not **VK_REMAINING_MIP_LEVELS**, then **baseMipLevel + levelCount must** be less than or equal to the **mipLevels** specified in **VkImageCreateInfo** when **image** was created

- **VID-vkCmdClearColorImage-baseArrayLayer-01472**
  
  The **VkImageSubresourceRange::baseArrayLayer** members of the elements of the **pRanges** array **must** each be less than the **arrayLayers** specified in **VkImageCreateInfo** when **image** was created

- **VID-vkCmdClearColorImage-pRanges-01693**
  
  For each **VkImageSubresourceRange** element of **pRanges**, if the **layerCount** member is not **VK_REMAINING_ARRAY_LAYERS**, then **baseArrayLayer + layerCount must** be less than or equal to the **arrayLayers** specified in **VkImageCreateInfo** when **image** was created

- **VID-vkCmdClearColorImage-image-00007**
  
  **image** **must** not have a compressed or depth/stencil format

- **VID-vkCmdClearColorImage-pColor-04961**
  
  **pColor must** be a valid pointer to a **VkClearColorValue** union

- **VID-vkCmdClearColorImage-commandBuffer-01805**
  
  If **commandBuffer** is an unprotected command buffer and **protectedNoFault** is not supported, **image must** not be a protected image

- **VID-vkCmdClearColorImage-commandBuffer-01806**
  
  If **commandBuffer** is a protected command buffer and **protectedNoFault** is not supported, **must** not be an unprotected image

---

**Valid Usage (Implicit)**

- **VID-vkCmdClearColorImage-commandBuffer-parameter**
  
  **commandBuffer must** be a valid **VkCommandBuffer** handle

- **VID-vkCmdClearColorImage-image-parameter**
  
  **image must** be a valid **VkImage** handle

- **VID-vkCmdClearColorImage-imageLayout-parameter**
  
  **imageLayout must** be a valid **VkImageLayout** value
To clear one or more subranges of a depth/stencil image, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdClearColorImage(
    VkCommandBuffer commandBuffer,
    VkImage image,
    VkImageLayout imageLayout,
    const VkClearColorValue* pColor,
    uint32_t rangeCount,
    const VkImageSubresourceRange* pRanges);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
• `image` is the image to be cleared.

• `imageLayout` specifies the current layout of the image subresource ranges to be cleared, and **must** be `VK_IMAGE_LAYOUT_GENERAL` or `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`.

• `pDepthStencil` is a pointer to a `VkClearDepthStencilValue` structure containing the values that the depth and stencil image subresource ranges will be cleared to (see **Clear Values** below).

• `rangeCount` is the number of image subresource range structures in `pRanges`.

• `pRanges` is a pointer to an array of `VkImageSubresourceRange` structures describing a range of mipmap levels, array layers, and aspects to be cleared, as described in **Image Views**.

---

**Valid Usage**

• VUID-vkCmdClearDepthStencilImage-image-01994
  The **format features** of `image` **must** contain `VK_FORMAT_FEATURE_TRANSFER_DST_BIT`

• VUID-vkCmdClearDepthStencilImage-pRanges-02658
  If the `aspect` member of any element of `pRanges` includes `VK_IMAGE_ASPECT_STENCIL_BIT`, and `image` was created with **separate stencil usage**, `VK_IMAGE_USAGE_TRANSFER_DST_BIT` **must** have been included in the `VkImageStencelUsageCreateInfo::stencilUsage` used to create `image`

• VUID-vkCmdClearDepthStencilImage-pRanges-02659
  If the `aspect` member of any element of `pRanges` includes `VK_IMAGE_ASPECT_STENCIL_BIT`, and `image` was not created with **separate stencil usage**, `VK_IMAGE_USAGE_TRANSFER_DST_BIT` **must** have been included in the `VkImageCreateInfo::usage` used to create `image`

• VUID-vkCmdClearDepthStencilImage-pRanges-02660
  If the `aspect` member of any element of `pRanges` includes `VK_IMAGE_ASPECT_DEPTH_BIT`, `VK_IMAGE_USAGE_TRANSFER_DST_BIT` **must** have been included in the `VkImageCreateInfo::usage` used to create `image`

• VUID-vkCmdClearDepthStencilImage-image-00010
  If `image` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object

• VUID-vkCmdClearDepthStencilImage-imageLayout-00011
  `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`

• VUID-vkCmdClearDepthStencilImage-imageLayout-00012
  `imageLayout` **must** be either of `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`

• VUID-vkCmdClearDepthStencilImage-aspectMask-02824
  The `VkImageSubresourceRange::aspectMask` member of each element of the `pRanges` array **must** not include bits other than `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT`

• VUID-vkCmdClearDepthStencilImage-aspectMask-02825
  If the `image`'s format does not have a stencil component, then the `VkImageSubresourceRange::aspectMask` member of each element of the `pRanges` array **must** not include the `VK_IMAGE_ASPECT_STENCIL_BIT` bit
If the image’s format does not have a depth component, then the `VkImageSubresourceRange::aspectMask` member of each element of the `pRanges` array **must** not include the `VK_IMAGE_ASPECT_DEPTH_BIT` bit.

The `VkImageSubresourceRange::baseMipLevel` members of the elements of the `pRanges` array **must** each be less than the `mipLevels` specified in `VkImageCreateInfo` when `image` was created.

For each `VkImageSubresourceRange` element of `pRanges`, if the `levelCount` member is not `VK_REMAINING_MIP_LEVELS`, then `baseMipLevel + levelCount` **must** be less than or equal to the `mipLevels` specified in `VkImageCreateInfo` when `image` was created.

The `VkImageSubresourceRange::baseArrayLayer` members of the elements of the `pRanges` array **must** each be less than the `arrayLayers` specified in `VkImageCreateInfo` when `image` was created.

For each `VkImageSubresourceRange` element of `pRanges`, if the `layerCount` member is not `VK_REMAINING_ARRAY_LAYERS`, then `baseArrayLayer + layerCount` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `image` was created.

**Valid Usage (Implicit)**

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `image` **must** be a valid `VkImage` handle
- `imageLayout` **must** be a valid `VkImageLayout` value
- `pDepthStencil` **must** be a valid pointer to a valid `VkClearColorValue` structure
- `pRanges` **must** be a valid pointer to an array of `rangeCount` valid `VkImageSubresourceRange` structures
Clears outside render pass instances are treated as transfer operations for the purposes of memory barriers.

18.2. Clearing Images Inside A Render Pass Instance

To clear one or more regions of color and depth/stencil attachments inside a render pass instance, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdClearAttachments(
    VkCommandBuffer commandBuffer,                   
    uint32_t attachmentCount,                        
    const VkClearAttachment* pAttachments,         
    uint32_t rectCount,                              
    const VkClearRect* pRects);
```
• **commandBuffer** is the command buffer into which the command will be recorded.

• **attachmentCount** is the number of entries in the **pAttachments** array.

• **pAttachments** is a pointer to an array of **VkClearAttachment** structures defining the attachments to clear and the clear values to use.

• **rectCount** is the number of entries in the **pRects** array.

• **pRects** is a pointer to an array of **VkClearRect** structures defining regions within each selected attachment to clear.

Unlike other clear commands, **vkCmdClearAttachments** is not a transfer command. It performs its operations in rasterization order. For color attachments, the operations are executed as color attachment writes, by the **VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT** stage. For depth/stencil attachments, the operations are executed as depth writes and stencil writes by the **VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT** and **VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT** stages.

**vkCmdClearAttachments** is not affected by the bound pipeline state.

---

**Note**

It is generally preferable to clear attachments by using the **VK_ATTACHMENT_LOAD_OP_CLEAR** load operation at the start of rendering, as it is more efficient on some implementations.

If any attachment's **aspectMask** to be cleared is not backed by an image view, the clear has no effect on that aspect.

---

**Valid Usage**

• **VUID-vkCmdClearAttachments-pAttachments-07270**
  For each element of **pAttachments**, the corresponding attachment in the current render pass instance must either not be backed by an image view, or contain each of the aspects specified in **aspectMask**

• **VUID-vkCmdClearAttachments-aspectMask-07271**
  If the **aspectMask** member of any element of **pAttachments** contains **VK_IMAGE_ASPECT_COLOR_BIT**, the colorAttachment must be a valid color attachment index in the current render pass instance

• **VUID-vkCmdClearAttachments-rect-02682**
  The rect member of each element of **pRects** must have an **extent.width** greater than 0

• **VUID-vkCmdClearAttachments-rect-02683**
  The rect member of each element of **pRects** must have an **extent.height** greater than 0

• **VUID-vkCmdClearAttachments-pRects-00016**
  The rectangular region specified by each element of **pRects** must be contained within the render area of the current render pass instance

• **VUID-vkCmdClearAttachments-pRects-06937**
  The layers specified by each element of **pRects** must be contained within every attachment that **pAttachments** refers to, i.e. for each element of **pRects**, **VkClearRect**
::baseArrayLayer + VkClearRect::layerCount must be less than or equal to the number of layers rendered to in the current render pass instance

- VUID-vkCmdClearAttachments-layerCount-01934
  The layerCount member of each element of pRects must not be 0

- VUID-vkCmdClearAttachments-commandBuffer-02504
  If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, each attachment to be cleared must not be a protected image

- VUID-vkCmdClearAttachments-commandBuffer-02505
  If commandBuffer is a protected command buffer and protectedNoFault is not supported, each attachment to be cleared must not be an unprotected image

- VUID-vkCmdClearAttachments-baseArrayLayer-00018
  If the render pass instance this is recorded in uses multiview, then baseArrayLayer must be zero and layerCount must be one

Valid Usage (Implicit)

- VUID-vkCmdClearAttachments-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdClearAttachments-pAttachments-parameter
  pAttachments must be a valid pointer to an array of attachmentCount valid VkClearAttachment structures

- VUID-vkCmdClearAttachments-pRects-parameter
  pRects must be a valid pointer to an array of rectCount VkClearRect structures

- VUID-vkCmdClearAttachments-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdClearAttachments-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

- VUID-vkCmdClearAttachments-renderpass
  This command must only be called inside of a render pass instance

- VUID-vkCmdClearAttachments-attachmentCount-arraylength
  attachmentCount must be greater than 0

- VUID-vkCmdClearAttachments-rectCount-arraylength
  rectCount must be greater than 0

Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized
Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The **VkClearRect** structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkClearRect {
    VkRect2D rect;
    uint32_t baseArrayLayer;
    uint32_t layerCount;
} VkClearRect;
```

- **rect** is the two-dimensional region to be cleared.
- **baseArrayLayer** is the first layer to be cleared.
- **layerCount** is the number of layers to clear.

The layers `[baseArrayLayer, baseArrayLayer + layerCount)` counting from the base layer of the attachment image view are cleared.

The **VkClearAttachment** structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkClearAttachment {
    VkImageAspectFlags aspectMask;
    uint32_t colorAttachment;
    VkClearColorValue clearValue;
} VkClearAttachment;
```

- **aspectMask** is a mask selecting the color, depth and/or stencil aspects of the attachment to be cleared.
- **colorAttachment** is only meaningful if VK_IMAGE_ASPECT_COLOR_BIT is set in **aspectMask**, in which case it is an index into the currently bound color attachments.
- **clearValue** is the color or depth/stencil value to clear the attachment to, as described in Clear Values below.

Valid Usage

- VUID-VkClearAttachment-aspectMask-00019
  If **aspectMask** includes **VK_IMAGE_ASPECT_COLOR_BIT**, it **must** not include...
18.3. Clear Values

The `VkClearColorValue` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef union VkClearColorValue {
    float float32[4];
    int32_t int32[4];
    uint32_t uint32[4];
} VkClearColorValue;
```

- `float32` are the color clear values when the format of the image or attachment is one of the formats in the Interpretation of Numeric Format table other than signed integer (SINT) or unsigned integer (UINT). Floating point values are automatically converted to the format of the image, with the clear value being treated as linear if the image is sRGB.

- `int32` are the color clear values when the format of the image or attachment is signed integer (SINT). Signed integer values are converted to the format of the image by casting to the smaller type (with negative 32-bit values mapping to negative values in the smaller type). If the integer clear value is not representable in the target type (e.g. would overflow in conversion to that type), the clear value is undefined.

- `uint32` are the color clear values when the format of the image or attachment is unsigned integer (UINT). Unsigned integer values are converted to the format of the image by casting to the integer type with fewer bits.

The four array elements of the clear color map to R, G, B, and A components of image formats, in order.

If the image has more than one sample, the same value is written to all samples for any pixels being cleared.

The `VkClearDepth_STENCILValue` structure is defined as:
typedef struct VkClearDepthStencilValue {
    float depth;
    uint32_t stencil;
} VkClearDepthStencilValue;

• depth is the clear value for the depth aspect of the depth/stencil attachment. It is a floating-point value which is automatically converted to the attachment’s format.

• stencil is the clear value for the stencil aspect of the depth/stencil attachment. It is a 32-bit integer value which is converted to the attachment’s format by taking the appropriate number of LSBs.

Valid Usage

• VUID-VkClearDepthStencilValue-depth-02506
depth must be between 0.0 and 1.0, inclusive

The VkClearValue union is defined as:

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

• color specifies the color image clear values to use when clearing a color image or attachment.

• depthStencil specifies the depth and stencil clear values to use when clearing a depth/stencil image or attachment.

This union is used where part of the API requires either color or depth/stencil clear values, depending on the attachment, and defines the initial clear values in the VkRenderPassBeginInfo structure.

18.4. Filling Buffers

To clear buffer data, call:

void vkCmdFillBuffer(
    VkCommandBuffer commandBuffer,  // Provided by VK_VERSION_1_0
    VkBuffer dstBuffer,              // Provided by VK_VERSION_1_0
    VkDeviceSize dstOffset,          // Provided by VK_VERSION_1_0
    VkDeviceSize size,               // Provided by VK_VERSION_1_0
    uint32_t data);                  // Provided by VK_VERSION_1_0
• **commandBuffer** is the command buffer into which the command will be recorded.
• **dstBuffer** is the buffer to be filled.
• **dstOffset** is the byte offset into the buffer at which to start filling, and must be a multiple of 4.
• **size** is the number of bytes to fill, and must be either a multiple of 4, or **VK_WHOLE_SIZE** to fill the range from offset to the end of the buffer. If **VK_WHOLE_SIZE** is used and the remaining size of the buffer is not a multiple of 4, then the nearest smaller multiple is used.
• **data** is the 4-byte word written repeatedly to the buffer to fill size bytes of data. The data word is written to memory according to the host endianness.

**vkCmdFillBuffer** is treated as a “transfer” operation for the purposes of synchronization barriers. The **VK_BUFFER_USAGE_TRANSFER_DST_BIT** must be specified in usage of VkBufferCreateInfo in order for the buffer to be compatible with **vkCmdFillBuffer**.

### Valid Usage

- VUID-vkCmdFillBuffer-dstOffset-00024
dstOffset must be less than the size of dstBuffer
- VUID-vkCmdFillBuffer-dstOffset-00025
dstOffset must be a multiple of 4
- VUID-vkCmdFillBuffer-size-00026
If size is not equal to **VK_WHOLE_SIZE**, size must be greater than 0
- VUID-vkCmdFillBuffer-size-00027
If size is not equal to **VK_WHOLE_SIZE**, size must be less than or equal to the size of dstBuffer minus dstOffset
- VUID-vkCmdFillBuffer-size-00028
If size is not equal to **VK_WHOLE_SIZE**, size must be a multiple of 4
- VUID-vkCmdFillBuffer-dstBuffer-00029
dstBuffer must have been created with **VK_BUFFER_USAGE_TRANSFER_DST_BIT** usage flag
- VUID-vkCmdFillBuffer-dstBuffer-00031
If dstBuffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object
- VUID-vkCmdFillBuffer-commandBuffer-01811
If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, dstBuffer must not be a protected buffer
- VUID-vkCmdFillBuffer-commandBuffer-01812
If commandBuffer is a protected command buffer and protectedNoFault is not supported, dstBuffer must not be an unprotected buffer

### Valid Usage (Implicit)

- VUID-vkCmdFillBuffer-commandBuffer-parameter
commandBuffer must be a valid VkCommandBuffer handle
• VUID-vkCmdFillBuffer-dstBuffer-parameter
dstBuffer must be a valid VkBuffer handle

• VUID-vkCmdFillBuffer-commandBuffer-recording
commandBuffer must be in the recording state

• VUID-vkCmdFillBuffer-commandBuffer-cmdpool
The VkCommandPool that commandBuffer was allocated from must support transfer, graphics or compute operations

• VUID-vkCmdFillBuffer-renderpass
This command must only be called outside of a render pass instance

• VUID-vkCmdFillBuffer-commonparent
Both of commandBuffer, and dstBuffer must have been created, allocated, or retrieved from the same VkDevice

### Host Synchronization

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

### 18.5. Updating Buffers

To update buffer data inline in a command buffer, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdUpdateBuffer(
    VkCommandBuffer commandBuffer,
    VkBuffer dstBuffer,
    VkDeviceSize dstOffset,
    VkDeviceSize dataSize,
    const void* pData);
```

- commandBuffer is the command buffer into which the command will be recorded.
- dstBuffer is a handle to the buffer to be updated.
• **dstOffset** is the byte offset into the buffer to start updating, and **must** be a multiple of 4.
• **dataSize** is the number of bytes to update, and **must** be a multiple of 4.
• **pData** is a pointer to the source data for the buffer update, and **must** be at least `dataSize` bytes in size.

`dataSize` **must** be less than or equal to 65536 bytes. For larger updates, applications **can** use buffer to buffer copies.

**Note**
Buffer updates performed with `vkCmdUpdateBuffer` first copy the data into command buffer memory when the command is recorded (which requires additional storage and may incur an additional allocation), and then copy the data from the command buffer into `dstBuffer` when the command is executed on a device.

The additional cost of this functionality compared to buffer to buffer copies means it is only recommended for very small amounts of data, and is why it is limited to only 65536 bytes.

Applications **can** work around this by issuing multiple `vkCmdUpdateBuffer` commands to different ranges of the same buffer, but it is strongly recommended that they **should** not.

The source data is copied from the user pointer to the command buffer when the command is called.

`vkCmdUpdateBuffer` is only allowed outside of a render pass. This command is treated as a “transfer” operation for the purposes of synchronization barriers. The **`VK_BUFFER_USAGE_TRANSFER_DST_BIT`** must be specified in `usage` of `VkBufferCreateInfo` in order for the buffer to be compatible with `vkCmdUpdateBuffer`.

## Valid Usage

- **VUID-vkCmdUpdateBuffer-dstOffset-00032**
  
  `dstOffset` **must** be less than the size of `dstBuffer`

- **VUID-vkCmdUpdateBuffer-dataSize-00033**
  
  `dataSize` **must** be less than or equal to the size of `dstBuffer` minus `dstOffset`

- **VUID-vkCmdUpdateBuffer-dstBuffer-00034**
  
  `dstBuffer` **must** have been created with **`VK_BUFFER_USAGE_TRANSFER_DST_BIT`** usage flag

- **VUID-vkCmdUpdateBuffer-dstBuffer-00035**
  
  If `dstBuffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-vkCmdUpdateBuffer-dstOffset-00036**
  
  `dstOffset` **must** be a multiple of 4

- **VUID-vkCmdUpdateBuffer-dataSize-00037**
\textbf{dataSize must} be less than or equal to 65536

- VUID-vkCmdUpdateBuffer-dataSize-00038
  \textit{dataSize must} be a multiple of 4

- VUID-vkCmdUpdateBuffer-commandBuffer-01813
  If \textit{commandBuffer} is an unprotected command buffer and \textit{protectedNoFault} is not supported, \textit{dstBuffer must} not be a protected buffer

- VUID-vkCmdUpdateBuffer-commandBuffer-01814
  If \textit{commandBuffer} is a protected command buffer and \textit{protectedNoFault} is not supported, \textit{dstBuffer must} not be an unprotected buffer

\begin{itemize}
\item \textbf{Valid Usage (Implicit)}
\begin{itemize}
\item VUID-vkCmdUpdateBuffer-commandBuffer-parameter
  \textit{commandBuffer must} be a valid \texttt{VkCommandBuffer} handle

\item VUID-vkCmdUpdateBuffer-dstBuffer-parameter
  \textit{dstBuffer must} be a valid \texttt{VkBuffer} handle

\item VUID-vkCmdUpdateBuffer-pData-parameter
  \textit{pData must} be a valid pointer to an array of \textit{dataSize} bytes

\item VUID-vkCmdUpdateBuffer-commandBuffer-recording
  \textit{commandBuffer must} be in the \textit{recording state}

\item VUID-vkCmdUpdateBuffer-commandBuffer-cmpool
  The \texttt{VkCommandPool} that \textit{commandBuffer} was allocated from \textit{must} support transfer, graphics, or compute operations

\item VUID-vkCmdUpdateBuffer-renderpass
  This command \textit{must} only be called outside of a render pass instance

\item VUID-vkCmdUpdateBuffer-dataSize-arraylength
  \textit{dataSize must} be greater than 0

\item VUID-vkCmdUpdateBuffer-commonparent
  Both of \textit{commandBuffer}, and \textit{dstBuffer must} have been created, allocated, or retrieved from the same \texttt{VkDevice}
\end{itemize}
\end{itemize}

\begin{itemize}
\item \textbf{Host Synchronization}
\begin{itemize}

\item Host access to \textit{commandBuffer must} be externally synchronized

\item Host access to the \texttt{VkCommandPool} that \textit{commandBuffer} was allocated from \textit{must} be externally synchronized
\end{itemize}
\end{itemize}
## Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

**Note**

The `pData` parameter was of type `uint32_t*` instead of `void*` prior to version 1.0.19 of the Specification and `VK_HEADER_VERSION` 19 of the Vulkan Header Files. This was a historical anomaly, as the source data may be of other types.
Chapter 19. Copy Commands

An application can copy buffer and image data using several methods described in this chapter, depending on the type of data transfer.

All copy commands are treated as “transfer” operations for the purposes of synchronization barriers.

All copy commands that have a source format with an X component in its format description read undefined values from those bits.

All copy commands that have a destination format with an X component in its format description write undefined values to those bits.

19.1. Copying Data Between Buffers

To copy data between buffer objects, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdCopyBuffer(
    VkCommandBuffer commandBuffer,
    VkBuffer srcBuffer,
    VkBuffer dstBuffer,
    uint32_t regionCount,
    const VkBufferCopy* pRegions);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `srcBuffer` is the source buffer.
- `dstBuffer` is the destination buffer.
- `regionCount` is the number of regions to copy.
- `pRegions` is a pointer to an array of `VkBufferCopy` structures specifying the regions to copy.

Each source region specified by `pRegions` is copied from the source buffer to the destination region of the destination buffer. If any of the specified regions in `srcBuffer` overlaps in memory with any of the specified regions in `dstBuffer`, values read from those overlapping regions are undefined.

Valid Usage

- VUID-vkCmdCopyBuffer-commandBuffer-01822
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `srcBuffer` must not be a protected buffer

- VUID-vkCmdCopyBuffer-commandBuffer-01823
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `dstBuffer` must not be a protected buffer

- VUID-vkCmdCopyBuffer-commandBuffer-01824
If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `dstBuffer` must not be an unprotected buffer.

- **VUID-vkCmdCopyBuffer-srcOffset-00113**
  The `srcOffset` member of each element of `pRegions` must be less than the size of `srcBuffer`.

- **VUID-vkCmdCopyBuffer-dstOffset-00114**
  The `dstOffset` member of each element of `pRegions` must be less than the size of `dstBuffer`.

- **VUID-vkCmdCopyBuffer-size-00115**
  The `size` member of each element of `pRegions` must be less than or equal to the size of `srcBuffer` minus `srcOffset`.

- **VUID-vkCmdCopyBuffer-size-00116**
  The `size` member of each element of `pRegions` must be less than or equal to the size of `dstBuffer` minus `dstOffset`.

- **VUID-vkCmdCopyBuffer-pRegions-00117**
  The union of the source regions, and the union of the destination regions, specified by the elements of `pRegions`, must not overlap in memory.

- **VUID-vkCmdCopyBuffer-srcBuffer-00118**
  `srcBuffer` must have been created with `VK_BUFFER_USAGE_TRANSFER_SRC_BIT` usage flag.

- **VUID-vkCmdCopyBuffer-srcBuffer-00119**
  If `srcBuffer` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.

- **VUID-vkCmdCopyBuffer-dstBuffer-00120**
  `dstBuffer` must have been created with `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage flag.

- **VUID-vkCmdCopyBuffer-dstBuffer-00121**
  If `dstBuffer` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.

### Valid Usage (Implicit)

- **VUID-vkCmdCopyBuffer-commandBuffer-parameter**
  `commandBuffer` must be a valid `VkCommandBuffer` handle.

- **VUID-vkCmdCopyBuffer-srcBuffer-parameter**
  `srcBuffer` must be a valid `VkBuffer` handle.

- **VUID-vkCmdCopyBuffer-dstBuffer-parameter**
  `dstBuffer` must be a valid `VkBuffer` handle.

- **VUID-vkCmdCopyBuffer-pRegions-parameter**
  `pRegions` must be a valid pointer to an array of `regionCount` valid `VkBufferCopy` structures.

- **VUID-vkCmdCopyBuffer-commandBuffer-recording**
  `commandBuffer` must be in the recording state.

- **VUID-vkCmdCopyBuffer-commandBuffer-cmdpool**
  The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations.
- VUID-vkCmdCopyBuffer-renderpass
  This command **must** only be called outside of a render pass instance

- VUID-vkCmdCopyBuffer-regionCount-arraylength
  `regionCount` **must** be greater than 0

- VUID-vkCmdCopyBuffer-commonparent
  Each of `commandBuffer`, `dstBuffer`, and `srcBuffer` **must** have been created, allocated, or retrieved from the same `VkDevice`

### Host Synchronization

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

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

The `VkBufferCopy` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkBufferCopy {
    VkDeviceSize srcOffset;
    VkDeviceSize dstOffset;
    VkDeviceSize size;
} VkBufferCopy;
```

- `srcOffset` is the starting offset in bytes from the start of `srcBuffer`.
- `dstOffset` is the starting offset in bytes from the start of `dstBuffer`.
- `size` is the number of bytes to copy.

### Valid Usage

- VUID-VkBufferCopy-size-01988
  The `size` **must** be greater than 0

A more extensible version of the copy buffer command is defined below.
To copy data between buffer objects, call:

```c
// Provided by VK_VERSION_1_3
void vkCmdCopyBuffer2(
    VkCommandBuffer commandBuffer,
    const VkCopyBufferInfo2* pCopyBufferInfo);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pCopyBufferInfo` is a pointer to a `VkCopyBufferInfo2` structure describing the copy parameters.

Each source region specified by `pCopyBufferInfo->pname:pRegions` is copied from the source buffer to the destination region of the destination buffer. If any of the specified regions in `pCopyBufferInfo->pname:srcBuffer` overlaps in memory with any of the specified regions in `pCopyBufferInfo->pname:dstBuffer`, values read from those overlapping regions are undefined.

### Valid Usage

- VUID-vkCmdCopyBuffer2-commandBuffer-01822
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `srcBuffer` must not be a protected buffer

- VUID-vkCmdCopyBuffer2-commandBuffer-01823
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `dstBuffer` must not be a protected buffer

- VUID-vkCmdCopyBuffer2-commandBuffer-01824
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `dstBuffer` must not be an unprotected buffer

### Valid Usage (Implicit)

- VUID-vkCmdCopyBuffer2-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdCopyBuffer2-pCopyBufferInfo-parameter
  `pCopyBufferInfo` must be a valid pointer to a valid `VkCopyBufferInfo2` structure

- VUID-vkCmdCopyBuffer2-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdCopyBuffer2-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations

- VUID-vkCmdCopyBuffer2-renderpass
  This command must only be called outside of a render pass instance
Host Synchronization

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

## Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer, Graphics, Compute</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkCopyBufferInfo2` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkCopyBufferInfo2 {
    VkStructureType sType;
    const void* pNext;
    VkBuffer srcBuffer;
    VkBuffer dstBuffer;
    uint32_t regionCount;
    const VkBufferCopy2* pRegions;
} VkCopyBufferInfo2;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcBuffer` is the source buffer.
- `dstBuffer` is the destination buffer.
- `regionCount` is the number of regions to copy.
- `pRegions` is a pointer to an array of `VkBufferCopy2` structures specifying the regions to copy.

## Valid Usage

- VUID-VkCopyBufferInfo2-srcOffset-00113
  The `srcOffset` member of each element of `pRegions` must be less than the size of `srcBuffer`.
- VUID-VkCopyBufferInfo2-dstOffset-00114
  The `dstOffset` member of each element of `pRegions` must be less than the size of `dstBuffer`.
- VUID-VkCopyBufferInfo2-size-00115
  The `size` member of each element of `pRegions` must be less than or equal to the size of
The **VkBufferCopy2** structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkBufferCopy2 {
```

- **srcBuffer** minus **srcOffset**
  - VUID-VkCopyBufferInfo2-size-00116
    The **size** member of each element of **pRegions** must be less than or equal to the size of **dstBuffer** minus **dstOffset**

- **pRegions**
  - VUID-VkCopyBufferInfo2-pRegions-00117
    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**
  - VUID-VkCopyBufferInfo2-srcBuffer-00118
    **srcBuffer** must have been created with **VK_BUFFER_USAGE_TRANSFER_SRC_BIT** usage flag

  - VUID-VkCopyBufferInfo2-srcBuffer-00119
    If **srcBuffer** is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object

- **dstBuffer**
  - VUID-VkCopyBufferInfo2-dstBuffer-00120
    **dstBuffer** must have been created with **VK_BUFFER_USAGE_TRANSFER_DST_BIT** usage flag

  - VUID-VkCopyBufferInfo2-dstBuffer-00121
    If **dstBuffer** is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object

**Valid Usage (Implicit)**

- **sType**
  - VUID-VkCopyBufferInfo2-sType-sType
    **sType** must be **VK_STRUCTURE_TYPE_COPY_BUFFER_INFO_2**

- **pNext**
  - VUID-VkCopyBufferInfo2-pNext-pNext
    **pNext** must be **NULL**

- **srcBuffer**
  - VUID-VkCopyBufferInfo2-srcBuffer-parameter
    **srcBuffer** must be a valid **VkBuffer** handle

- **dstBuffer**
  - VUID-VkCopyBufferInfo2-dstBuffer-parameter
    **dstBuffer** must be a valid **VkBuffer** handle

- **pRegions**
  - VUID-VkCopyBufferInfo2-pRegions-parameter
    **pRegions** must be a valid pointer to an array of **regionCount** valid **VkBufferCopy2** structures

- **regionCount**
  - VUID-VkCopyBufferInfo2-regionCount-arraylength
    **regionCount** must be greater than 0

- **VkDevice**
  - VUID-VkCopyBufferInfo2-commonparent
    Both of **dstBuffer**, and **srcBuffer** must have been created, allocated, or retrieved from the same **VkDevice**
• `sType` is the type of this structure.
• `pNext` is `NULL` or a pointer to a structure extending this structure.
• `srcOffset` is the starting offset in bytes from the start of `srcBuffer`.
• `dstOffset` is the starting offset in bytes from the start of `dstBuffer`.
• `size` is the number of bytes to copy.

**Valid Usage**

- VUID-VkBufferCopy2-size-01988
  The `size` must be greater than 0

**Valid Usage (Implicit)**

- VUID-VkBufferCopy2-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_BUFFER_COPY_2`
- VUID-VkBufferCopy2-pNext-pNext
  `pNext` must be `NULL`

### 19.2. Copying Data Between Images

To copy data between image objects, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdCopyImage(
    VkCommandBuffer commandBuffer,  // Provided by VK_VERSION_1_0
    VkImage srcImage,               // Provided by VK_VERSION_1_0
    VkImageLayout srcImageLayout,   // Provided by VK_VERSION_1_0
    VkImage dstImage,               // Provided by VK_VERSION_1_0
    VkImageLayout dstImageLayout,   // Provided by VK_VERSION_1_0
    uint32_t regionCount,           // Provided by VK_VERSION_1_0
    const VkImageCopy* pRegions);  // Provided by VK_VERSION_1_0
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `srcImage` is the source image.
- `srcImageLayout` is the current layout of the source image subresource.
• **dstImage** is the destination image.
• **dstImageLayout** is the current layout of the destination image subresource.
• **regionCount** is the number of regions to copy.
• **pRegions** is a pointer to an array of **VkImageCopy** structures specifying the regions to copy.

Each source region specified by **pRegions** is copied from the source image to the destination region of the destination image. If any of the specified regions in **srcImage** overlaps in memory with any of the specified regions in **dstImage**, values read from those overlapping regions are undefined.

**Multi-planar images** can only be copied on a per-plane basis, and the subresources used in each region when copying to or from such images must specify only one plane, though different regions can specify different planes. When copying planes of multi-planar images, the format considered is the **compatible format for that plane**, rather than the format of the multi-planar image.

If the format of the destination image has a different **block extent** than the source image (e.g. one is a compressed format), the offset and extent for each of the regions specified is scaled according to the block extents of each format to match in size. Copy regions for each image must be aligned to a multiple of the texel block extent in each dimension, except at the edges of the image, where region extents must match the edge of the image.

Image data can be copied between images with different image types. If one image is **VK_IMAGE_TYPE_3D** and the other image is **VK_IMAGE_TYPE_2D** with multiple layers, then each slice is copied to or from a different layer; depth slices in the 3D image correspond to **layerCount** layers in the 2D image, with an effective depth of 1 used for the 2D image.

### Valid Usage

- VUID-vkCmdCopyImage-commandBuffer-01825
  If **commandBuffer** is an unprotected command buffer and **protectedNoFault** is not supported, **srcImage** must not be a protected image

- VUID-vkCmdCopyImage-commandBuffer-01826
  If **commandBuffer** is an unprotected command buffer and **protectedNoFault** is not supported, **dstImage** must not be a protected image

- VUID-vkCmdCopyImage-commandBuffer-01827
  If **commandBuffer** is a protected command buffer and **protectedNoFault** is not supported, **dstImage** must not be an unprotected image

- VUID-vkCmdCopyImage-pRegions-00124
  The union of all source regions, and the union of all destination regions, specified by the elements of **pRegions**, must not overlap in memory

- VUID-vkCmdCopyImage-srcImage-01995
  The **format features** of **srcImage** must contain **VK_FORMAT_FEATURE_TRANSFER_SRC_BIT**

- VUID-vkCmdCopyImage-srcImage-01546
  If **srcImage** is non-sparse then the image or disjoint plane to be copied must be bound completely and contiguously to a single **VkDeviceMemory** object
srcImageLayout must specify the layout of the image subresources of srcImage specified in pRegions at the time this command is executed on a VkDevice.

srcImageLayout must be VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL.

The format features of dstImage must contain VK_FORMAT_FEATURE_TRANSFER_DST_BIT.

If dstImage is non-sparse then the image or disjoint plane that is the destination of the copy must be bound completely and contiguously to a single VkDeviceMemory object.

dstImageLayout must specify the layout of the image subresources of dstImage specified in pRegions at the time this command is executed on a VkDevice.

dstImageLayout must be VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL.

If the VkFormat of each of srcImage and dstImage is not a multi-planar format, the VkFormat of each of srcImage and dstImage must be size-compatible.

In a copy to or from a plane of a multi-planar image, the VkFormat of the image and plane must be compatible according to the description of compatible planes for the plane being copied.

The sample count of srcImage and dstImage must match.

The srcSubresource.mipLevel member of each element of pRegions must be less than the mipLevels specified in VkImageCreateInfo when srcImage was created.

The dstSubresource.mipLevel member of each element of pRegions must be less than the mipLevels specified in VkImageCreateInfo when dstImage was created.

The srcSubresource.baseArrayLayer + srcSubresource.layerCount of each element of pRegions must be less than or equal to the arrayLayers specified in VkImageCreateInfo when srcImage was created.

The dstSubresource.baseArrayLayer + dstSubresource.layerCount of each element of pRegions must be less than or equal to the arrayLayers specified in VkImageCreateInfo when dstImage was created.

The srcOffset and extent members of each element of pRegions must respect the image transfer granularity requirements of commandBuffer’s command pool’s queue family, as described in VkQueueFamilyProperties.
The dstOffset and extent members of each element of pRegions must respect the image transfer granularity requirements of commandBuffer’s command pool’s queue family, as described in VkQueueFamilyProperties.

If neither srcImage nor dstImage has a multi-planar image format then for each element of pRegions, srcSubresource.aspectMask and dstSubresource.aspectMask must match.

If srcImage has a VkFormat with two planes then for each element of pRegions, srcSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT or VK_IMAGE_ASPECT_PLANE_1_BIT.

If srcImage has a VkFormat with three planes then for each element of pRegions, srcSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or VK_IMAGE_ASPECT_PLANE_2_BIT.

If dstImage has a VkFormat with two planes then for each element of pRegions, dstSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT or VK_IMAGE_ASPECT_PLANE_1_BIT.

If dstImage has a VkFormat with three planes then for each element of pRegions, dstSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or VK_IMAGE_ASPECT_PLANE_2_BIT.

For each element of pRegions, srcSubresource.aspectMask must specify aspects present in srcImage.

For each element of pRegions, dstSubresource.aspectMask must specify aspects present in dstImage.
For each element of \( pRegions \), \( srcOffset.x \) and \((extent.width + srcOffset.x)\) must both be greater than or equal to 0 and less than or equal to the width of the specified \( srcSubresource \) of \( srcImage \).

For each element of \( pRegions \), \( srcOffset.y \) and \((extent.height + srcOffset.y)\) must both be greater than or equal to 0 and less than or equal to the height of the specified \( srcSubresource \) of \( srcImage \).

If \( srcImage \) is of type \( VK_IMAGE_TYPE_1D \), then for each element of \( pRegions \), \( srcOffset.y \) must be 0 and \( extent.height \) must be 1.

If \( srcImage \) is of type \( VK_IMAGE_TYPE_3D \), then for each element of \( pRegions \), \( srcOffset.z \) and \((extent.depth + srcOffset.z)\) must both be greater than or equal to 0 and less than or equal to the depth of the specified \( srcSubresource \) of \( srcImage \).

If \( srcImage \) is of type \( VK_IMAGE_TYPE_1D \), then for each element of \( pRegions \), \( srcOffset.z \) must be 0 and \( extent.depth \) must be 1.

If \( dstImage \) is of type \( VK_IMAGE_TYPE_1D \), then for each element of \( pRegions \), \( dstOffset.z \) must be 0 and \( extent.depth \) must be 1.

If \( srcImage \) is of type \( VK_IMAGE_TYPE_2D \), then for each element of \( pRegions \), \( dstOffset.z \) must be 0.

If \( srcImage \) and \( dstImage \) have a different \( VkImageType \), one must be \( VK_IMAGE_TYPE_3D \) and the other must be \( VK_IMAGE_TYPE_2D \).

If \( srcImage \) and \( dstImage \) have the same \( VkImageType \), the \( layerCount \) member of \( srcSubresource \) and \( dstSubresource \) in each element of \( pRegions \) must match.

If \( srcImage \) and \( dstImage \) are both of type \( VK_IMAGE_TYPE_2D \), then for each element of \( pRegions \), \( extent.depth \) must be 1.

If \( srcImage \) is of type \( VK_IMAGE_TYPE_2D \), and \( dstImage \) is of type \( VK_IMAGE_TYPE_3D \), then for each element of \( pRegions \), \( extent.depth \) must equal \( srcSubresource.layerCount \).

If \( dstImage \) is of type \( VK_IMAGE_TYPE_2D \), and \( srcImage \) is of type \( VK_IMAGE_TYPE_3D \), then for each element of \( pRegions \), \( extent.depth \) must equal \( dstSubresource.layerCount \).
For each element of \texttt{pRegions}, \texttt{dstOffset.x} and (\texttt{extent.width} + \texttt{dstOffset.x}) \textbf{must} both be greater than or equal to \(0\) and less than or equal to the width of the specified \texttt{dstSubresource} of \texttt{dstImage}

- VUID-vkCmdCopyImage-dstOffset-00151
  For each element of \texttt{pRegions}, \texttt{dstOffset.y} and (\texttt{extent.height} + \texttt{dstOffset.y}) \textbf{must} both be greater than or equal to \(0\) and less than or equal to the height of the specified \texttt{dstSubresource} of \texttt{dstImage}

- VUID-vkCmdCopyImage-dstOffset-00152
  If \texttt{dstImage} is of type \texttt{VK_IMAGE_TYPE_1D}, then for each element of \texttt{pRegions}, \texttt{dstOffset.y} \textbf{must} be \(0\) and \texttt{extent.height} \textbf{must} be \(1\)

- VUID-vkCmdCopyImage-pRegions-07278
  For each element of \texttt{pRegions}, \texttt{srcOffset.x} \textbf{must} be a multiple of the \texttt{texel block extent width} of the \texttt{VkFormat} of \texttt{srcImage}

- VUID-vkCmdCopyImage-pRegions-07279
  For each element of \texttt{pRegions}, \texttt{srcOffset.y} \textbf{must} be a multiple of the \texttt{texel block extent height} of the \texttt{VkFormat} of \texttt{srcImage}

- VUID-vkCmdCopyImage-pRegions-07280
  For each element of \texttt{pRegions}, \texttt{srcOffset.z} \textbf{must} be a multiple of the \texttt{texel block extent depth} of the \texttt{VkFormat} of \texttt{srcImage}

- VUID-vkCmdCopyImage-pRegions-07281
  For each element of \texttt{pRegions}, \texttt{dstOffset.x} \textbf{must} be a multiple of the \texttt{texel block extent width} of the \texttt{VkFormat} of \texttt{dstImage}

- VUID-vkCmdCopyImage-pRegions-07282
  For each element of \texttt{pRegions}, \texttt{dstOffset.y} \textbf{must} be a multiple of the \texttt{texel block extent height} of the \texttt{VkFormat} of \texttt{dstImage}

- VUID-vkCmdCopyImage-pRegions-07283
  For each element of \texttt{pRegions}, \texttt{dstOffset.z} \textbf{must} be a multiple of the \texttt{texel block extent depth} of the \texttt{VkFormat} of \texttt{dstImage}

- VUID-vkCmdCopyImage-srcImage-01728
  For each element of \texttt{pRegions}, if the sum of \texttt{srcOffset.x} and \texttt{extent.width} does not equal the width of the the subresource specified by \texttt{srcSubresource}, \texttt{extent.width} \textbf{must} be a multiple of the \texttt{texel block extent width} of the \texttt{VkFormat} of \texttt{srcImage}

- VUID-vkCmdCopyImage-srcImage-01729
  For each element of \texttt{pRegions}, if the sum of \texttt{srcOffset.y} and \texttt{extent.height} does not equal the height of the the subresource specified by \texttt{srcSubresource}, \texttt{extent.height} \textbf{must} be a multiple of the \texttt{texel block extent height} of the \texttt{VkFormat} of \texttt{srcImage}

- VUID-vkCmdCopyImage-srcImage-01730
  For each element of \texttt{pRegions}, if the sum of \texttt{srcOffset.z} and \texttt{extent.depth} does not equal the depth of the the subresource specified by \texttt{srcSubresource}, \texttt{extent.depth} \textbf{must} be a
multiple of the texel block extent depth of the VkFormat of srcImage

- VUID-vkCmdCopyImage-dstImage-01732
  For each element of pRegions, if the sum of dstOffset.x and extent.width does not equal the width of the the subresource specified by dstSubresource, extent.width must be a multiple of the texel block extent width of the VkFormat of dstImage

- VUID-vkCmdCopyImage-dstImage-01733
  For each element of pRegions, if the sum of dstOffset.y and extent.height does not equal the height of the the subresource specified by dstSubresource, extent.height must be a multiple of the texel block extent height of the VkFormat of dstImage

- VUID-vkCmdCopyImage-dstImage-01734
  For each element of pRegions, if the sum of dstOffset.z and extent.depth does not equal the depth of the the subresource specified by dstSubresource, extent.depth must be a multiple of the texel block extent depth of the VkFormat of dstImage

- VUID-vkCmdCopyImage-aspect-06662
  If the aspect member of any element of pRegions includes any flag other than VK_IMAGE_ASPECT_STENCIL_BIT or srcImage was not created with separate stencil usage, VK_IMAGE_USAGE_TRANSFER_SRC_BIT must have been included in the VkImageCreateInfo::usage used to create srcImage

- VUID-vkCmdCopyImage-aspect-06663
  If the aspect member of any element of pRegions includes any flag other than VK_IMAGE_ASPECT_STENCIL_BIT or dstImage was not created with separate stencil usage, VK_IMAGE_USAGE_TRANSFER_DST_BIT must have been included in the VkImageCreateInfo::usage used to create dstImage

- VUID-vkCmdCopyImage-aspect-06664
  If the aspect member of any element of pRegions includes VK_IMAGE_ASPECT_STENCIL_BIT, and srcImage was created with separate stencil usage, VK_IMAGE_USAGE_TRANSFER_SRC_BIT must have been included in the VkImageStencilUsageCreateInfo::stencilUsage used to create srcImage

- VUID-vkCmdCopyImage-aspect-06665
  If the aspect member of any element of pRegions includes VK_IMAGE_ASPECT_STENCIL_BIT, and dstImage was created with separate stencil usage, VK_IMAGE_USAGE_TRANSFER_DST_BIT must have been included in the VkImageStencilUsageCreateInfo::stencilUsage used to create dstImage

- VUID-vkCmdCopyImage-srcImage-07745
  srcImage and dstImage must have the same sample count

---

**Valid Usage (Implicit)**

- VUID-vkCmdCopyImage-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdCopyImage-srcImage-parameter
  srcImage must be a valid VkImage handle

- VUID-vkCmdCopyImage-srcImageLayout-parameter


**Host Synchronization**

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

---

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

The `VkImageCopy` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkImageCopy {
    VkImageSubresourceLayers srcSubresource;
    VkOffset3D srcOffset;
    VkImageSubresourceLayers dstSubresource;
} VkImageCopy;
```
VkOffset3D dstOffset;
VkExtent3D extent;
} VkImageCopy;

• `srcSubresource` and `dstSubresource` are `VkImageSubresourceLayers` structures specifying the image subresources of the images used for the source and destination image data, respectively.
• `srcOffset` and `dstOffset` select the initial \( x, y, \) and \( z \) offsets in texels of the sub-regions of the source and destination image data.
• `extent` is the size in texels of the image to copy in width, height and depth.

Valid Usage
• VUID-VkImageCopy-extent-06668
  `extent.width` must not be 0
• VUID-VkImageCopy-extent-06669
  `extent.height` must not be 0
• VUID-VkImageCopy-extent-06670
  `extent.depth` must not be 0

Valid Usage (Implicit)
• VUID-VkImageCopy-srcSubresource-parameter
  `srcSubresource` must be a valid `VkImageSubresourceLayers` structure
• VUID-VkImageCopy-dstSubresource-parameter
  `dstSubresource` must be a valid `VkImageSubresourceLayers` structure

The `VkImageSubresourceLayers` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkImageSubresourceLayers {
  VkImageAspectFlags aspectMask;
  uint32_t mipLevel;
  uint32_t baseArrayLayer;
  uint32_t layerCount;
} VkImageSubresourceLayers;
```

• `aspectMask` is a combination of `VkImageAspectFlagBits`, selecting the color, depth and/or stencil aspects to be copied.
• `mipLevel` is the mipmap level to copy
• `baseArrayLayer` and `layerCount` are the starting layer and number of layers to copy.
Valid Usage

- VUID-VkImageSubresourceLayers-aspectMask-00167
  If `aspectMask` contains `VK_IMAGE_ASPECT_COLOR_BIT`, it must not contain either of `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT`

- VUID-VkImageSubresourceLayers-aspectMask-00168
  `aspectMask` must not contain `VK_IMAGE_ASPECT_METADATA_BIT`

- VUID-VkImageSubresourceLayers-layerCount-01700
  `layerCount` must be greater than 0

Valid Usage (Implicit)

- VUID-VkImageSubresourceLayers-aspectMask-parameter
  `aspectMask` must be a valid combination of `VkImageAspectFlagBits` values

- VUID-VkImageSubresourceLayers-aspectMask-requiredbitmask
  `aspectMask` must not be 0

A more extensible version of the copy image command is defined below.

To copy data between image objects, call:

```c
// Provided by VK_VERSION_1_3
void vkCmdCopyImage2(
    VkCommandBuffer commandBuffer,
    const VkCopyImageInfo2* pCopyImageInfo);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pCopyImageInfo` is a pointer to a `VkCopyImageInfo2` structure describing the copy parameters.

This command is functionally identical to `vkCmdCopyImage`, but includes extensible sub-structures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

Valid Usage

- VUID-vkCmdCopyImage2-commandBuffer-01825
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `srcImage` must not be a protected image

- VUID-vkCmdCopyImage2-commandBuffer-01826
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `dstImage` must not be an unprotected image

- VUID-vkCmdCopyImage2-commandBuffer-01827
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `dstImage` must not be an unprotected image
Valid Usage (Implicit)

- VUID-vkCmdCopyImage2-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdCopyImage2-pCopyImageInfo-parameter
  pCopyImageInfo must be a valid pointer to a valid VkCopyImageInfo2 structure

- VUID-vkCmdCopyImage2-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdCopyImage2-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support transfer, graphics, or compute operations

- VUID-vkCmdCopyImage2-renderpass
  This command must only be called outside of a render pass instance

Host Synchronization

- Host access to commandBuffer must be externally synchronized

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics, Compute</td>
<td></td>
</tr>
</tbody>
</table>

The VkCopyImageInfo2 structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkCopyImageInfo2 {
    VkStructureType sType;
    const void* pNext;
    VkImage srcImage;
    VkImageLayout srcImageLayout;
    VkImage dstImage;
    VkImageLayout dstImageLayout;
    uint32_t regionCount;
    const VkImageCopy2* pRegions;
} VkCopyImageInfo2;
```
• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **srcImage** is the source image.
• **srcImageLayout** is the current layout of the source image subresource.
• **dstImage** is the destination image.
• **dstImageLayout** is the current layout of the destination image subresource.
• **regionCount** is the number of regions to copy.
• **pRegions** is a pointer to an array of *VkImageCopy2* structures specifying the regions to copy.

### Valid Usage

- **VUID-VkCopyImageInfo2-pRegions-00124**
  The union of all source regions, and the union of all destination regions, specified by the elements of **pRegions**, **must** not overlap in memory.

- **VUID-VkCopyImageInfo2-srcImage-01995**
  The format features of **srcImage** **must** contain **VK_FORMAT_FEATURE_TRANSFER_SRC_BIT**.

- **VUID-VkCopyImageInfo2-srcImage-01546**
  If **srcImage** is non-sparse then the image or **disjoint** plane to be copied **must** be bound completely and contiguously to a single **VkDeviceMemory** object.

- **VUID-VkCopyImageInfo2-srcImageLayout-00128**
  **srcImageLayout** **must** specify the layout of the image subresources of **srcImage** specified in **pRegions** at the time this command is executed on a **VkDevice**.

- **VUID-VkCopyImageInfo2-srcImageLayout-00129**
  **srcImageLayout** **must** be **VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL** or **VK_IMAGE_LAYOUT_GENERAL**.

- **VUID-VkCopyImageInfo2-dstImage-01996**
  The format features of **dstImage** **must** contain **VK_FORMAT_FEATURE_TRANSFER_DST_BIT**.

- **VUID-VkCopyImageInfo2-dstImage-01547**
  If **dstImage** is non-sparse then the image or **disjoint** plane that is the destination of the copy **must** be bound completely and contiguously to a single **VkDeviceMemory** object.

- **VUID-VkCopyImageInfo2-dstImageLayout-00133**
  **dstImageLayout** **must** specify the layout of the image subresources of **dstImage** specified in **pRegions** at the time this command is executed on a **VkDevice**.

- **VUID-VkCopyImageInfo2-dstImageLayout-00134**
  **dstImageLayout** **must** be **VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL** or **VK_IMAGE_LAYOUT_GENERAL**.

- **VUID-VkCopyImageInfo2-srcImage-01548**
  If the **VkFormat** of each of **srcImage** and **dstImage** is not a **multi-planar format**, the **VkFormat** of each of **srcImage** and **dstImage** **must** be size-compatible.

- **VUID-VkCopyImageInfo2-None-01549**
  In a copy to or from a plane of a **multi-planar image**, the **VkFormat** of the image and plane **must** be compatible according to the description of compatible planes for the plane being copied.
• VUID-VkCopyImageInfo2-srcImage-00136
  The sample count of srcImage and dstImage must match

• VUID-VkCopyImageInfo2-srcSubresource-01696
  The srcSubresource.mipLevel member of each element of pRegions must be less than the mipLevels specified in VkImageCreateInfo when srcImage was created

• VUID-VkCopyImageInfo2-dstSubresource-01697
  The dstSubresource.mipLevel member of each element of pRegions must be less than the mipLevels specified in VkImageCreateInfo when dstImage was created

• VUID-VkCopyImageInfo2-srcSubresource-01698
  The srcSubresource.baseArrayLayer + srcSubresource.layerCount of each element of pRegions must be less than or equal to the arrayLayers specified in VkImageCreateInfo when srcImage was created

• VUID-VkCopyImageInfo2-dstSubresource-01699
  The dstSubresource.baseArrayLayer + dstSubresource.layerCount of each element of pRegions must be less than or equal to the arrayLayers specified in VkImageCreateInfo when dstImage was created

• VUID-VkCopyImageInfo2-srcOffset-01783
  The srcOffset and extent members of each element of pRegions must respect the image transfer granularity requirements of commandBuffer’s command pool’s queue family, as described in VkQueueFamilyProperties

• VUID-VkCopyImageInfo2-dstOffset-01784
  The dstOffset and extent members of each element of pRegions must respect the image transfer granularity requirements of commandBuffer’s command pool’s queue family, as described in VkQueueFamilyProperties

• VUID-VkCopyImageInfo2-srcImage-01551
  If neither srcImage nor dstImage has a multi-planar image format then for each element of pRegions, srcSubresource.aspectMask and dstSubresource.aspectMask must match

• VUID-VkCopyImageInfo2-srcImage-01552
  If srcImage has a VkFormat with two planes then for each element of pRegions, srcSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT or VK_IMAGE_ASPECT_PLANE_1_BIT

• VUID-VkCopyImageInfo2-srcImage-01553
  If srcImage has a VkFormat with three planes then for each element of pRegions, srcSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or VK_IMAGE_ASPECT_PLANE_2_BIT

• VUID-VkCopyImageInfo2-dstImage-01554
  If dstImage has a VkFormat with two planes then for each element of pRegions, dstSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT or VK_IMAGE_ASPECT_PLANE_1_BIT

• VUID-VkCopyImageInfo2-dstImage-01555
  If dstImage has a VkFormat with three planes then for each element of pRegions, dstSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or VK_IMAGE_ASPECT_PLANE_2_BIT
• VUID-VkCopyImageInfo2-srcImage-01556
  If `srcImage` has a multi-planar image format and the `dstImage` does not have a multi-planar image format, then for each element of `pRegions`, `dstSubresource.aspectMask` must be `VK_IMAGE_ASPECT_COLOR_BIT`.

• VUID-VkCopyImageInfo2-dstImage-01557
  If `dstImage` has a multi-planar image format and the `srcImage` does not have a multi-planar image format, then for each element of `pRegions`, `srcSubresource.aspectMask` must be `VK_IMAGE_ASPECT_COLOR_BIT`.

• VUID-VkCopyImageInfo2-srcImage-04443
  If `srcImage` is of type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions`, `srcSubresource.baseArrayLayer` must be 0 and `srcSubresource.layerCount` must be 1.

• VUID-VkCopyImageInfo2-dstImage-04444
  If `dstImage` is of type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions`, `dstSubresource.baseArrayLayer` must be 0 and `dstSubresource.layerCount` must be 1.

• VUID-VkCopyImageInfo2-aspectMask-00142
  For each element of `pRegions`, `srcSubresource.aspectMask` must specify aspects present in `srcImage`.

• VUID-VkCopyImageInfo2-aspectMask-00143
  For each element of `pRegions`, `dstSubresource.aspectMask` must specify aspects present in `dstImage`.

• VUID-VkCopyImageInfo2-srcOffset-00144
  For each element of `pRegions`, `srcOffset.x` and `(extent.width + srcOffset.x)` must both be greater than or equal to 0 and less than or equal to the width of the specified `srcSubresource` of `srcImage`.

• VUID-VkCopyImageInfo2-srcOffset-00145
  For each element of `pRegions`, `srcOffset.y` and `(extent.height + srcOffset.y)` must both be greater than or equal to 0 and less than or equal to the height of the specified `srcSubresource` of `srcImage`.

• VUID-VkCopyImageInfo2-srcImage-00146
  If `srcImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `srcOffset.y` must be 0 and `extent.height` must be 1.

• VUID-VkCopyImageInfo2-dstImage-01786
  If `dstImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `dstOffset.z` must be 0 and `extent.depth` must be 1.

• VUID-VkCopyImageInfo2-srcImage-01787
  If `srcImage` is of type `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `srcOffset.z` must be 0 and `extent.depth` must be 1.

• VUID-VkCopyImageInfo2-dstImage-01785
  If `dstImage` is of type `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `dstOffset.z` must be 0 and `extent.depth` must be 1.
must be 0

- VUID-VkCopyImageInfo2-dstImage-01788
  If dstImage is of type VK_IMAGE_TYPE_2D, then for each element of pRegions, dstOffset.z must be 0

- VUID-VkCopyImageInfo2-srcImage-07743
  If srcImage and dstImage have a different VkImageType, one must be VK_IMAGE_TYPE_3D and the other must be VK_IMAGE_TYPE_2D

- VUID-VkCopyImageInfo2-srcImage-07744
  If srcImage and dstImage have the same VkImageType, the layerCount member of srcSubresource and dstSubresource in each element of pRegions must match

- VUID-VkCopyImageInfo2-srcImage-01790
  If srcImage and dstImage are both of type VK_IMAGE_TYPE_2D, then for each element of pRegions, extent.depth must be 1

- VUID-VkCopyImageInfo2-srcImage-01791
  If srcImage is of type VK_IMAGE_TYPE_2D, and dstImage is of type VK_IMAGE_TYPE_3D, then for each element of pRegions, extent.depth must equal srcSubresource.layerCount

- VUID-VkCopyImageInfo2-dstImage-01792
  If dstImage is of type VK_IMAGE_TYPE_2D, and srcImage is of type VK_IMAGE_TYPE_3D, then for each element of pRegions, extent.depth must equal dstSubresource.layerCount

- VUID-VkCopyImageInfo2-pRegions-07278
  For each element of pRegions, dstOffset.x must be a multiple of the texel block extent width of the VkFormat of srcImage

- VUID-VkCopyImageInfo2-pRegions-07279
  For each element of pRegions, dstOffset.y must be a multiple of the texel block extent height of the VkFormat of srcImage

- VUID-VkCopyImageInfo2-pRegions-07280
  For each element of pRegions, dstOffset.z must be a multiple of the texel block extent depth of the VkFormat of srcImage
• VUID-VkCopyImageInfo2-pRegions-07281
  For each element of `pRegions`, `dstOffset.x` must be a multiple of the texel block extent width of the `VkFormat` of `dstImage`

• VUID-VkCopyImageInfo2-pRegions-07282
  For each element of `pRegions`, `dstOffset.y` must be a multiple of the texel block extent height of the `VkFormat` of `dstImage`

• VUID-VkCopyImageInfo2-pRegions-07283
  For each element of `pRegions`, `dstOffset.z` must be a multiple of the texel block extent depth of the `VkFormat` of `dstImage`

• VUID-VkCopyImageInfo2-srcImage-01728
  For each element of `pRegions`, if the sum of `srcOffset.x` and `extent.width` does not equal the width of the the subresource specified by `srcSubresource`, `extent.width` must be a multiple of the texel block extent width of the `VkFormat` of `srcImage`

• VUID-VkCopyImageInfo2-srcImage-01729
  For each element of `pRegions`, if the sum of `srcOffset.y` and `extent.height` does not equal the height of the the subresource specified by `srcSubresource`, `extent.height` must be a multiple of the texel block extent height of the `VkFormat` of `srcImage`

• VUID-VkCopyImageInfo2-srcImage-01730
  For each element of `pRegions`, if the sum of `srcOffset.z` and `extent.depth` does not equal the depth of the the subresource specified by `srcSubresource`, `extent.depth` must be a multiple of the texel block extent depth of the `VkFormat` of `srcImage`

• VUID-VkCopyImageInfo2-dstImage-01732
  For each element of `pRegions`, if the sum of `dstOffset.x` and `extent.width` does not equal the width of the the subresource specified by `dstSubresource`, `extent.width` must be a multiple of the texel block extent width of the `VkFormat` of `dstImage`

• VUID-VkCopyImageInfo2-dstImage-01733
  For each element of `pRegions`, if the sum of `dstOffset.y` and `extent.height` does not equal the height of the the subresource specified by `dstSubresource`, `extent.height` must be a multiple of the texel block extent height of the `VkFormat` of `dstImage`

• VUID-VkCopyImageInfo2-dstImage-01734
  For each element of `pRegions`, if the sum of `dstOffset.z` and `extent.depth` does not equal the depth of the the subresource specified by `dstSubresource`, `extent.depth` must be a multiple of the texel block extent depth of the `VkFormat` of `dstImage`

• VUID-VkCopyImageInfo2-aspect-06662
  If the `aspect` member of any element of `pRegions` includes any flag other than `VK_IMAGE_ASPECT_STENCIL_BIT` or `srcImage` was not created with separate stencil usage, `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` must have been included in the `VkImageCreateInfo`::`usage` used to create `srcImage`

• VUID-VkCopyImageInfo2-aspect-06663
  If the `aspect` member of any element of `pRegions` includes any flag other than `VK_IMAGE_ASPECT_STENCIL_BIT` or `dstImage` was not created with separate stencil usage, `VK_IMAGE_USAGE_TRANSFER_DST_BIT` must have been included in the `VkImageCreateInfo`::`usage` used to create `dstImage`
If the `aspect` member of any element of `pRegions` includes `VK_IMAGE_ASPECT_STENCIL_BIT`, and `srcImage` was created with separate stencil usage, `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` must have been included in the `VkImageStencelUsageCreateInfo::stencilUsage` used to create `srcImage`.

If the `aspect` member of any element of `pRegions` includes `VK_IMAGE_ASPECT_STENCIL_BIT`, and `dstImage` was created with separate stencil usage, `VK_IMAGE_USAGE_TRANSFER_DST_BIT` must have been included in the `VkImageStencelUsageCreateInfo::stencilUsage` used to create `dstImage`.

`srcImage` and `dstImage` must have the same sample count.

Valid Usage (Implicit)

- **VUID-VkCopyImageInfo2-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_COPY_IMAGE_INFO_2`

- **VUID-VkCopyImageInfo2-pNext-pNext**
  
  `pNext` must be `NULL`

- **VUID-VkCopyImageInfo2-srcImage-parameter**
  
  `srcImage` must be a valid `VkImage` handle.

- **VUID-VkCopyImageInfo2-srcImageLayout-parameter**
  
  `srcImageLayout` must be a valid `VkImageLayout` value.

- **VUID-VkCopyImageInfo2-dstImage-parameter**
  
  `dstImage` must be a valid `VkImage` handle.

- **VUID-VkCopyImageInfo2-dstImageLayout-parameter**
  
  `dstImageLayout` must be a valid `VkImageLayout` value.

- **VUID-VkCopyImageInfo2-pRegions-parameter**
  
  `pRegions` must be a valid pointer to an array of `regionCount` valid `VkImageCopy2` structures.

- **VUID-VkCopyImageInfo2-regionCount-arraylength**
  
  `regionCount` must be greater than 0.

- **VUID-VkCopyImageInfo2-commonparent**
  
  Both of `dstImage`, and `srcImage` must have been created, allocated, or retrieved from the same `VkDevice`.

The `VkImageCopy2` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkImageCopy2 {
    VkStructureType sType;
    const void* pNext;
} VkImageCopy2;
```
• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• srcSubresource and dstSubresource are VkImageSubresourceLayers structures specifying the image subresources of the images used for the source and destination image data, respectively.
• srcOffset and dstOffset select the initial x, y, and z offsets in texels of the sub-regions of the source and destination image data.
• extent is the size in texels of the image to copy in width, height and depth.

Valid Usage

• VUID-VkImageCopy2-extent-06668
  extent.width must not be 0
• VUID-VkImageCopy2-extent-06669
  extent.height must not be 0
• VUID-VkImageCopy2-extent-06670
  extent.depth must not be 0

Valid Usage (Implicit)

• VUID-VkImageCopy2-sType-sType
  sType must be VK_STRUCTURE_TYPE_IMAGE_COPY_2
• VUID-VkImageCopy2-pNext-pNext
  pNext must be NULL
• VUID-VkImageCopy2-srcSubresource-parameter
  srcSubresource must be a valid VkImageSubresourceLayers structure
• VUID-VkImageCopy2-dstSubresource-parameter
  dstSubresource must be a valid VkImageSubresourceLayers structure

19.3. Copying Data Between Buffers and Images

Data can be copied between buffers and images, enabling applications to load and store data between images and user defined offsets in buffer memory.

When copying between a buffer and an image, whole texel blocks are always copied; each texel block in the specified extent in the image to be copied will be written to a region in the buffer,
specified according to the position of the texel block, and the texel block extent and size of the format being copied.

For a set of coordinates \((x,y,z,\text{layer})\), where:

\[
x \text{ is in the range } \left\lfloor \frac{x}{\text{blockWidth}} \right\rfloor \leq \text{imageOffset.x} + \frac{\text{imageExtent.width}}{\text{blockWidth}} \leq \left\lfloor \frac{x}{\text{blockWidth}} + \frac{\text{imageExtent.width}}{\text{blockWidth}} \right\rfloor,
\]

\[
y \text{ is in the range } \left\lfloor \frac{y}{\text{blockHeight}} \right\rfloor \leq \text{imageOffset.y} + \frac{\text{imageExtent.height}}{\text{blockHeight}} \leq \left\lfloor \frac{y}{\text{blockHeight}} + \frac{\text{imageExtent.height}}{\text{blockHeight}} \right\rfloor,
\]

\[
z \text{ is in the range } \left\lfloor \frac{z}{\text{blockDepth}} \right\rfloor \leq \text{imageOffset.z} + \frac{\text{imageExtent.depth}}{\text{blockDepth}} \leq \left\lfloor \frac{z}{\text{blockDepth}} + \frac{\text{imageExtent.depth}}{\text{blockDepth}} \right\rfloor,
\]

\[
\text{layer} \text{ is in the range } \left\lfloor \frac{\text{imageOffset.layer}}{\text{blockDepth}} \right\rfloor \leq \text{imageSubresource.baseArrayLayer} + \frac{\text{imageExtent.layer}}{\text{blockDepth}} \leq \left\lfloor \frac{\text{imageOffset.layer}}{\text{blockDepth}} + \frac{\text{imageExtent.layer}}{\text{blockDepth}} \right\rfloor,
\]

and where blockWidth, blockHeight, and blockDepth are the dimensions of the texel block extent of the image’s format.

For each \((x,y,z,\text{layer})\) coordinate, texels in the image layer selected by \text{layer} are accessed in the following ranges:

\[
[x \times \text{blockWidth}, \max( (x \times \text{blockWidth}) + \text{blockWidth}, \text{imageWidth} )]
\]

\[
[y \times \text{blockHeight}, \max( (y \times \text{blockHeight}) + \text{blockHeight}, \text{imageHeight} )]
\]

\[
[z \times \text{blockDepth}, \max( (z \times \text{blockDepth}) + \text{blockDepth}, \text{imageDepth} )]
\]

where imageWidth, imageHeight, and imageDepth are the dimensions of the image subresource.

For each \((x,y,z,\text{layer})\) coordinate, bytes in the buffer are accessed at offsets in the range \([\text{texelOffset}, \text{texelOffset} + \text{blockSize}]\), where:

\[
\text{texelOffset} = \text{bufferOffset} + (x \times \text{blockSize}) + (y \times \text{rowExtent}) + (z \times \text{sliceExtent}) + (\text{layer} \times \text{layerExtent})
\]

blockSize is the size of the block in bytes for the format.

\[
\text{rowExtent} = \max(\text{bufferRowLength}, \left\lfloor \frac{\text{imageExtent.width}}{\text{blockWidth}} \times \text{blockSize} \right\rfloor)
\]
sliceExtent = \max(\text{bufferImageHeight, imageExtent.height} \times \text{rowExtent})

layerExtent = imageExtent.depth \times \text{sliceExtent}

When copying between a buffer and the depth or stencil aspect of an image, data in the buffer is assumed to be laid out as separate planes rather than interleaved. Addressing calculations are thus performed for a different format than the base image, according to the aspect, as described in the following table:

Table 19. Depth/Stencil Aspect Copy Table

<table>
<thead>
<tr>
<th>Base Format</th>
<th>Depth Aspect Format</th>
<th>Stencil Aspect Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_D16_UNORM</td>
<td>VK_FORMAT_D16_UNORM</td>
<td>-</td>
</tr>
<tr>
<td>VK_FORMAT_X8_D24_UNORM_PACK32</td>
<td>VK_FORMAT_X8_D24_UNORM_PACK32</td>
<td>-</td>
</tr>
<tr>
<td>VK_FORMAT_D32_SFLOAT</td>
<td>VK_FORMAT_D32_SFLOAT</td>
<td>-</td>
</tr>
<tr>
<td>VK_FORMAT_S8_UINT</td>
<td>-</td>
<td>VK_FORMAT_S8_UINT</td>
</tr>
<tr>
<td>VK_FORMAT_D16_UNORM_S8_UINT</td>
<td>VK_FORMAT_D16_UNORM</td>
<td>VK_FORMAT_S8_UINT</td>
</tr>
<tr>
<td>VK_FORMAT_D24_UNORM_S8_UINT</td>
<td>VK_FORMAT_X8_D24_UNORM_PACK32</td>
<td>VK_FORMAT_S8_UINT</td>
</tr>
<tr>
<td>VK_FORMAT_D32_SFLOAT_S8_UINT</td>
<td>VK_FORMAT_D32_SFLOAT</td>
<td>VK_FORMAT_S8_UINT</td>
</tr>
</tbody>
</table>

When copying between a buffer and any plane of a multi-planar image, addressing calculations are performed using the compatible format for that plane, rather than the format of the multi-planar image.

Each texel block is copied from one resource to the other according to the above addressing equations.

To copy data from a buffer object to an image object, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdCopyBufferToImage(
    VkCommandBuffer commandBuffer,
    VkBuffer srcBuffer,
    VkImage dstImage,
    VkImageLayout dstImageLayout,
    uint32_t regionCount,
    const VkBufferImageCopy* pRegions);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `srcBuffer` is the source buffer.
- `dstImage` is the destination image.
- `dstImageLayout` is the layout of the destination image subresources for the copy.
- `regionCount` is the number of regions to copy.
• `pRegions` is a pointer to an array of `VkBufferImageCopy` structures specifying the regions to copy.

Each source region specified by `pRegions` is copied from the source buffer to the destination region of the destination image according to the addressing calculations for each resource. If any of the specified regions in `srcBuffer` overlaps in memory with any of the specified regions in `dstImage`, values read from those overlapping regions are undefined. If any region accesses a depth aspect in `dstImage` values copied from `srcBuffer` outside of the range [0,1] will be written as undefined values to the destination image.

Copy regions for the image **must** be aligned to a multiple of the texel block extent in each dimension, except at the edges of the image, where region extents **must** match the edge of the image.

---

**Valid Usage**

- VUID-vkCmdCopyBufferToImage-commandBuffer-01828
  
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `srcBuffer` must not be a protected buffer

- VUID-vkCmdCopyBufferToImage-commandBuffer-01829
  
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `dstImage` must not be a protected image

- VUID-vkCmdCopyBufferToImage-commandBuffer-01830
  
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `dstImage` must not be an unprotected image

- VUID-vkCmdCopyBufferToImage-commandBuffer-07737
  
  If the queue family used to create the `VkCommandPool` which `commandBuffer` was allocated from does not support `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT`, the `bufferOffset` member of any element of `pRegions` must be a multiple of 4

- VUID-vkCmdCopyBufferToImage-imageOffset-07738
  
  The `imageOffset` and `imageExtent` members of each element of `pRegions` must respect the image transfer granularity requirements of `commandBuffer`'s command pool's queue family, as described in `VkQueueFamilyProperties`  

- VUID-vkCmdCopyBufferToImage-commandBuffer-07739
  
  If the queue family used to create the `VkCommandPool` which `commandBuffer` was allocated from does not support `VK_QUEUE_GRAPHICS_BIT`, for each element of `pRegions`, the `aspectMask` member of `imageSubresource` must not be `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT`

- VUID-vkCmdCopyBufferToImage-bufferOffset-01558
  
  If `dstImage` does not have either a depth/stencil or a multi-planar format, then for each element of `pRegions`, `bufferOffset` must be a multiple of the format's texel block size

- VUID-vkCmdCopyBufferToImage-bufferOffset-01559
  
  If `dstImage` has a multi-planar format, then for each element of `pRegions`, `bufferOffset` must be a multiple of the element size of the compatible format for the format and the `aspectMask` of the `imageSubresource` as defined in Compatible formats of planes of multi-planar formats
• VUID-vkCmdCopyBufferToImage-srcImage-00199
  If \( \text{dstImage} \) is of type \( \text{VK_IMAGE_TYPE_1D} \), then for each element of \( \text{pRegions} \), \( \text{imageOffset.y} \) must be 0 and \( \text{imageExtent.height} \) must be 1.

• VUID-vkCmdCopyBufferToImage-imageOffset-00200
  For each element of \( \text{pRegions} \), \( \text{imageOffset.z} \) and \( (\text{imageExtent.depth} + \text{imageOffset.z}) \) must both be greater than or equal to 0 and less than or equal to the depth of the specified \( \text{imageSubresource} \) of \( \text{dstImage} \).

• VUID-vkCmdCopyBufferToImage-srcImage-00201
  If \( \text{dstImage} \) is of type \( \text{VK_IMAGE_TYPE_1D} \) or \( \text{VK_IMAGE_TYPE_2D} \), then for each element of \( \text{pRegions} \), \( \text{imageOffset.z} \) must be 0 and \( \text{imageExtent.depth} \) must be 1.

• VUID-vkCmdCopyBufferToImage-bufferRowLength-00203
  For each element of \( \text{pRegions} \), \( \text{bufferRowLength} \) must be a multiple of the \( \text{texel block extent width} \) of the \( \text{VkFormat} \) of \( \text{dstImage} \).

• VUID-vkCmdCopyBufferToImage-bufferImageHeight-00204
  For each element of \( \text{pRegions} \), \( \text{bufferImageHeight} \) must be a multiple of the \( \text{texel block extent height} \) of the \( \text{VkFormat} \) of \( \text{dstImage} \).

• VUID-vkCmdCopyBufferToImage-pRegions-07273
  For each element of \( \text{pRegions} \), \( \text{bufferOffset} \) must be a multiple of the \( \text{texel block size} \) of the \( \text{VkFormat} \) of \( \text{dstImage} \).

• VUID-vkCmdCopyBufferToImage-pRegions-07274
  For each element of \( \text{pRegions} \), \( \text{imageOffset.x} \) must be a multiple of the \( \text{texel block extent width} \) of the \( \text{VkFormat} \) of \( \text{dstImage} \).

• VUID-vkCmdCopyBufferToImage-pRegions-07275
  For each element of \( \text{pRegions} \), \( \text{imageOffset.y} \) must be a multiple of the \( \text{texel block extent height} \) of the \( \text{VkFormat} \) of \( \text{dstImage} \).

• VUID-vkCmdCopyBufferToImage-pRegions-07276
  For each element of \( \text{pRegions} \), \( \text{imageOffset.z} \) must be a multiple of the \( \text{texel block extent depth} \) of the \( \text{VkFormat} \) of \( \text{dstImage} \).

• VUID-vkCmdCopyBufferToImage-imageExtent-00207
  For each element of \( \text{pRegions} \), if the sum of \( \text{imageOffset.x} \) and \( \text{extent.width} \) does not equal the width of the the subresource specified by \( \text{srcSubresource} \), \( \text{extent.width} \) must be a multiple of the \( \text{texel block extent width} \) of the \( \text{VkFormat} \) of \( \text{dstImage} \).

• VUID-vkCmdCopyBufferToImage-imageExtent-00208
  For each element of \( \text{pRegions} \), if the sum of \( \text{imageOffset.y} \) and \( \text{extent.height} \) does not equal the height of the the subresource specified by \( \text{srcSubresource} \), \( \text{extent.height} \) must be a multiple of the \( \text{texel block extent height} \) of the \( \text{VkFormat} \) of \( \text{dstImage} \).

• VUID-vkCmdCopyBufferToImage-imageExtent-00209
  For each element of \( \text{pRegions} \), if the sum of \( \text{imageOffset.z} \) and \( \text{extent.depth} \) does not equal the depth of the the subresource specified by \( \text{srcSubresource} \), \( \text{extent.depth} \) must be a multiple of the \( \text{texel block extent depth} \) of the \( \text{VkFormat} \) of \( \text{dstImage} \).

• VUID-vkCmdCopyBufferToImage-aspectMask-00211
  For each element of \( \text{pRegions} \), \( \text{imageSubresource.aspectMask} \) must specify aspects present in \( \text{dstImage} \).
If \( \text{dstImage} \) has a \( \text{VkFormat} \) with two planes then for each element of \( \text{pRegions} \), \( \text{imageSubresource.aspectMask} \) must be \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) or \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \).

If \( \text{dstImage} \) has a \( \text{VkFormat} \) with three planes then for each element of \( \text{pRegions} \), \( \text{imageSubresource.aspectMask} \) must be \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \), \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \), or \( \text{VK_IMAGE_ASPECT_PLANE_2_BIT} \).

If \( \text{dstImage} \) is of type \( \text{VK_IMAGE_TYPE_3D} \), for each element of \( \text{pRegions} \), \( \text{imageSubresource.baseArrayLayer} \) must be 0 and \( \text{imageSubresource.layerCount} \) must be 1.

For each element of \( \text{pRegions} \), \( \text{bufferRowLength} \) divided by the texel block extent width and then multiplied by the texel block size of \( \text{dstImage} \) must be less than or equal to \( 2^{31} - 1 \).

If \( \text{dstImage} \) has a depth/stencil format, the \( \text{bufferOffset} \) member of any element of \( \text{pRegions} \) must be a multiple of 4.

\( \text{srcBuffer} \) must be large enough to contain all buffer locations that are accessed according to \( \text{Buffer and Image Addressing} \), for each element of \( \text{pRegions} \).

The union of all source regions, and the union of all destination regions, specified by the elements of \( \text{pRegions} \), must not overlap in memory.

\( \text{srcBuffer} \) must have been created with \( \text{VK_BUFFER_USAGE_TRANSFER_SRC_BIT} \) usage flag.

The format features of \( \text{dstImage} \) must contain \( \text{VK_FORMAT_FEATURE_TRANSFER_DST_BIT} \).

If \( \text{srcBuffer} \) is non-sparse then it must be bound completely and contiguously to a single \( \text{VkDeviceMemory} \) object.

\( \text{dstImage} \) must have been created with \( \text{VK_IMAGE_USAGE_TRANSFER_DST_BIT} \) usage flag.

If \( \text{dstImage} \) is non-sparse then it must be bound completely and contiguously to a single \( \text{VkDeviceMemory} \) object.

\( \text{dstImage} \) must have a sample count equal to \( \text{VK_SAMPLE_COUNT_1_BIT} \).

\( \text{dstImageLayout} \) must specify the layout of the image subresources of \( \text{dstImage} \) specified in \( \text{pRegions} \) at the time this command is executed on a \( \text{VkDevice} \).

\( \text{dstImageLayout} \) must be \( \text{VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL} \) or \( \text{VK_IMAGE_LAYOUT_GENERAL} \).
The `imageSubresource.mipLevel` member of each element of `pRegions` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `dstImage` was created.

The `imageSubresource.baseArrayLayer + imageSubresource.layerCount` of each element of `pRegions` must be less than or equal to the `arrayLayers` specified in ` VkImageCreateInfo` when `dstImage` was created.

For each element of `pRegions` whose `imageSubresource` contains a depth aspect, the data in `srcBuffer` must be in the range [0,1].

The image region specified by each element of `pRegions` must be contained within the specified `imageSubresource` of `dstImage`.

For each element of `pRegions`, `imageOffset.x` and `(imageExtent.width + imageOffset.x)` must both be greater than or equal to 0 and less than or equal to the width of the specified `imageSubresource` of `dstImage`.

For each element of `pRegions`, `imageOffset.y` and `(imageExtent.height + imageOffset.y)` must both be greater than or equal to 0 and less than or equal to the height of the specified `imageSubresource` of `dstImage`.

**Valid Usage (Implicit)**

- `commandBuffer` must be a valid `VkCommandBuffer` handle.
- `srcBuffer` must be a valid `VkBuffer` handle.
- `dstImage` must be a valid `VkImage` handle.
- `dstImageLayout` must be a valid `VkImageLayout` value.
- `pRegions` must be a valid pointer to an array of `regionCount` valid `VkBufferImageCopy` structures.
- `commandBuffer` must be in the recording state.
- The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations.
- This command must only be called outside of a render pass instance.
• VUID-vkCmdCopyBufferToImage-regionCount-arraylength
  regionCount must be greater than 0

• VUID-vkCmdCopyBufferToImage-commonparent
  Each of commandBuffer, dstImage, and srcBuffer must have been created, allocated, or
  retrieved from the same VkDevice

Host Synchronization

• Host access to commandBuffer must be externally synchronized

• Host access to the VkCommandPool that commandBuffer was allocated from must be externally
  synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

To copy data from an image object to a buffer object, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdCopyImageToBuffer(
    VkCommandBuffer commandBuffer,
    VkImage srcImage,
    VkImageLayout srcImageLayout,
    VkBuffer dstBuffer,
    uint32_t regionCount,
    const VkBufferImageCopy* pRegions);
```

• commandBuffer is the command buffer into which the command will be recorded.

• srcImage is the source image.

• srcImageLayout is the layout of the source image subresources for the copy.

• dstBuffer is the destination buffer.

• regionCount is the number of regions to copy.

• pRegions is a pointer to an array of VkBufferImageCopy structures specifying the regions to copy.

Each source region specified by pRegions is copied from the source image to the destination region
of the destination buffer according to the addressing calculations for each resource. If any of the
specified regions in srcImage overlaps in memory with any of the specified regions in dstBuffer,
values read from those overlapping regions are undefined.

Copy regions for the image **must** be aligned to a multiple of the texel block extent in each dimension, except at the edges of the image, where region extents **must** match the edge of the image.

---

**Valid Usage**

- **VUID-vkCmdCopyImageToBuffer-commandBuffer-01831**
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `srcImage` **must** not be a protected image.

- **VUID-vkCmdCopyImageToBuffer-commandBuffer-01832**
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `dstBuffer` **must** not be a protected buffer.

- **VUID-vkCmdCopyImageToBuffer-commandBuffer-01833**
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `dstBuffer` **must** not be an unprotected buffer.

- **VUID-vkCmdCopyImageToBuffer-commandBuffer-07746**
  If the queue family used to create the `VkCommandPool` which `commandBuffer` was allocated from does not support `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT`, the `bufferOffset` member of any element of `pRegions` **must** be a multiple of 4.

- **VUID-vkCmdCopyImageToBuffer-imageOffset-07747**
  The `imageOffset` and `imageExtent` members of each element of `pRegions` **must** respect the image transfer granularity requirements of `commandBuffer`'s command pool's queue family, as described in `VkQueueFamilyProperties`.

- **VUID-vkCmdCopyImageToBuffer-bufferOffset-01558**
  If `srcImage` does not have either a depth/stencil or a multi-planar format, then for each element of `pRegions`, `bufferOffset` **must** be a multiple of the format's texel block size.

- **VUID-vkCmdCopyImageToBuffer-bufferOffset-01559**
  If `srcImage` has a multi-planar format, then for each element of `pRegions`, `bufferOffset` **must** be a multiple of the element size of the compatible format for the format and the `aspectMask` of the `imageSubresource` as defined in `Compatible formats of planes of multi-planar formats`.

- **VUID-vkCmdCopyImageToBuffer-srcImage-00199**
  If `srcImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `imageOffset.y` **must** be 0 and `imageExtent.height` **must** be 1.

- **VUID-vkCmdCopyImageToBuffer-imageOffset-00200**
  For each element of `pRegions`, `imageOffset.z` and `(imageExtent.depth + imageOffset.z)` **must** both be greater than or equal to 0 and less than or equal to the depth of the specified `imageSubresource` of `srcImage`.

- **VUID-vkCmdCopyImageToBuffer-srcImage-00201**
  If `srcImage` is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `imageOffset.z` **must** be 0 and `imageExtent.depth` **must** be 1.
For each element of `pRegions`, `bufferRowLength` must be a multiple of the texel block extent width of the `VkFormat` of `srcImage`.

For each element of `pRegions`, `bufferImageHeight` must be a multiple of the texel block extent height of the `VkFormat` of `srcImage`.

For each element of `pRegions`, `bufferOffset` must be a multiple of the texel block size of the `VkFormat` of `srcImage`.

For each element of `pRegions`, `imageOffset.x` must be a multiple of the texel block extent width of the `VkFormat` of `srcImage`.

For each element of `pRegions`, `imageOffset.y` must be a multiple of the texel block extent height of the `VkFormat` of `srcImage`.

For each element of `pRegions`, `imageOffset.z` must be a multiple of the texel block extent depth of the `VkFormat` of `srcImage`.

For each element of `pRegions`, if the sum of `imageOffset.x` and `extent.width` does not equal the width of the the subresource specified by `srcSubresource`, `extent.width` must be a multiple of the texel block extent width of the `VkFormat` of `srcImage`.

For each element of `pRegions`, if the sum of `imageOffset.y` and `extent.height` does not equal the height of the the subresource specified by `srcSubresource`, `extent.height` must be a multiple of the texel block extent height of the `VkFormat` of `srcImage`.

For each element of `pRegions`, if the sum of `imageOffset.z` and `extent.depth` does not equal the depth of the the subresource specified by `srcSubresource`, `extent.depth` must be a multiple of the texel block extent depth of the `VkFormat` of `srcImage`.

For each element of `pRegions`, `imageSubresource.aspectMask` must specify aspects present in `srcImage`.

If `srcImage` has a `VkFormat` with two planes then for each element of `pRegions`, `imageSubresource.aspectMask` must be `VK_IMAGE_ASPECT_PLANE_0_BIT` or `VK_IMAGE_ASPECT_PLANE_1_BIT`.

If `srcImage` has a `VkFormat` with three planes then for each element of `pRegions`, `imageSubresource.aspectMask` must be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT`.

If `srcImage` is of type `VK_IMAGE_TYPE_3D`, for each element of `pRegions`,
imageSubresource.baseArrayLayer must be 0 and imageSubresource.layerCount must be 1

- VUID-vkCmdCopyImageToBuffer-pRegions-07277
  For each element of pRegions, bufferRowLength divided by the texel block extent width and then multiplied by the texel block size of srcImage must be less than or equal to 2^{31} - 1

- VUID-vkCmdCopyImageToBuffer-srcImage-04053
  If srcImage has a depth/stencil format, the bufferOffset member of any element of pRegions must be a multiple of 4

- VUID-vkCmdCopyImageToBuffer-pRegions-00183
dstBuffer must be large enough to contain all buffer locations that are accessed according to Buffer and Image Addressing, for each element of pRegions

- VUID-vkCmdCopyImageToBuffer-pRegions-00184
  The union of all source regions, and the union of all destination regions, specified by the elements of pRegions, must not overlap in memory

- VUID-vkCmdCopyImageToBuffer-srcImage-00186
  srcImage must have been created with VK_IMAGE_USAGE_TRANSFER_SRC_BIT usage flag

- VUID-vkCmdCopyImageToBuffer-srcImage-00187
  The format features of srcImage must contain VK_FORMAT_FEATURE_TRANSFER_SRC_BIT

- VUID-vkCmdCopyImageToBuffer-srcImageLayout-00189
  srcImage must have a sample count equal to VK_SAMPLE_COUNT_1_BIT

- VUID-vkCmdCopyImageToBuffer-srcImageLayout-00188
  srcImageLayout must specify the layout of the image subresources of srcImage specified in pRegions at the time this command is executed on a VkDevice

- VUID-vkCmdCopyImageToBuffer-imageSubresource-01704
  The imageSubresource.mipLevel member of each element of pRegions must be less than the mipLevels specified in VkImageCreateInfo when srcImage was created

- VUID-vkCmdCopyImageToBuffer-imageSubresource-01703
  The imageSubresource.baseArrayLayer + imageSubresource.layerCount of each element of pRegions must be less than or equal to the arrayLayers specified in VkImageCreateInfo when srcImage was created

- VUID-vkCmdCopyImageToBuffer-pRegions-06220
  The image region specified by each element of pRegions must be contained within the
specified imageSubresource of srcImage

- VUID-vkCmdCopyImageToBuffer-pRegions-06221
  For each element of pRegions, imageOffset.x and (imageExtent.width + imageOffset.x) must both be greater than or equal to 0 and less than or equal to the width of the specified imageSubresource of srcImage.

- VUID-vkCmdCopyImageToBuffer-pRegions-06222
  For each element of pRegions, imageOffset.y and (imageExtent.height + imageOffset.y) must both be greater than or equal to 0 and less than or equal to the height of the specified imageSubresource of srcImage.

### Valid Usage (Implicit)

- VUID-vkCmdCopyImageToBuffer-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle.

- VUID-vkCmdCopyImageToBuffer-srcImage-parameter
  srcImage must be a valid VkImage handle.

- VUID-vkCmdCopyImageToBuffer-srcImageLayout-parameter
  srcImageLayout must be a valid VkImageLayout value.

- VUID-vkCmdCopyImageToBuffer-dstBuffer-parameter
  dstBuffer must be a valid VkBuffer handle.

- VUID-vkCmdCopyImageToBuffer-pRegions-parameter
  pRegions must be a valid pointer to an array of regionCount valid VkBufferImageCopy structures.

- VUID-vkCmdCopyImageToBuffer-commandBuffer-recording
  commandBuffer must be in the recording state.

- VUID-vkCmdCopyImageToBuffer-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support transfer, graphics, or compute operations.

- VUID-vkCmdCopyImageToBuffer-renderpass
  This command must only be called outside of a render pass instance.

- VUID-vkCmdCopyImageToBuffer-regionCount-arraylength
  regionCount must be greater than 0.

- VUID-vkCmdCopyImageToBuffer-commonparent
  Each of commandBuffer, dstBuffer, and srcImage must have been created, allocated, or retrieved from the same VkDevice.

### Host Synchronization

- Host access to commandBuffer must be externally synchronized.

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized.
For both `vkCmdCopyBufferToImage` and `vkCmdCopyImageToBuffer`, each element of `pRegions` is a structure defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkBufferImageCopy {
    VkDeviceSize bufferOffset;
    uint32_t bufferRowLength;
    uint32_t bufferImageHeight;
    VkImageSubresourceLayers imageSubresource;
    VkOffset3D imageOffset;
    VkExtent3D imageExtent;
} VkBufferImageCopy;
```

- `bufferOffset` is the offset in bytes from the start of the buffer object where the image data is copied from or to.
- `bufferRowLength` and `bufferImageHeight` specify in texels a subregion of a larger two- or three-dimensional image in buffer memory, and control the addressing calculations. If either of these values is zero, that aspect of the buffer memory is considered to be tightly packed according to the `imageExtent`.
- `imageSubresource` is a `VkImageSubresourceLayers` used to specify the specific image subresources of the image used for the source or destination image data.
- `imageOffset` selects the initial x, y, z offsets in texels of the sub-region of the source or destination image data.
- `imageExtent` is the size in texels of the image to copy in width, height and depth.

## Valid Usage

- **VUID-VkBufferImageCopy-bufferRowLength-00195**
  `bufferRowLength must be 0, or greater than or equal to the width member of imageExtent`

- **VUID-VkBufferImageCopy-bufferImageHeight-00196**
  `bufferImageHeight must be 0, or greater than or equal to the height member of imageExtent`

- **VUID-VkBufferImageCopy-aspectMask-00212**
  The `aspectMask` member of `imageSubresource` must only have a single bit set

- **VUID-VkBufferImageCopy-imageExtent-06659**
imageExtent.width must not be 0
- VUID-VkBufferImageCopy-imageExtent-06660
imageExtent.height must not be 0
- VUID-VkBufferImageCopy-imageExtent-06661
imageExtent.depth must not be 0

Valid Usage (Implicit)

- VUID-VkBufferImageCopy-imageSubresource-parameter
imageSubresource must be a valid VkImageSubresourceLayers structure

More extensible versions of the commands to copy between buffers and images are defined below.

To copy data from a buffer object to an image object, call:

```c
// Provided by VK_VERSION_1_3
void vkCmdCopyBufferToImage2(
    VkCommandBuffer commandBuffer,
    const VkCopyBufferToImageInfo2* pCopyBufferToImageInfo);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pCopyBufferToImageInfo` is a pointer to a VkCopyBufferToImageInfo2 structure describing the copy parameters.

This command is functionally identical to `vkCmdCopyBufferToImage`, but includes extensible substructures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

Valid Usage

- VUID-vkCmdCopyBufferToImage2-commandBuffer-01828
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `srcBuffer` must not be a protected buffer
- VUID-vkCmdCopyBufferToImage2-commandBuffer-01829
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `dstImage` must not be a protected image
- VUID-vkCmdCopyBufferToImage2-commandBuffer-01830
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `dstImage` must not be an unprotected image
- VUID-vkCmdCopyBufferToImage2-commandBuffer-07737
  If the queue family used to create the VkCommandPool which `commandBuffer` was allocated from does not support `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT`, the `bufferOffset` member of any element of `pCopyBufferToImageInfo->pRegions` must be a multiple of 4
- VUID-vkCmdCopyBufferToImage2-imageOffset-07738
The `imageOffset` and `imageExtent` members of each element of `pCopyBufferToImageInfo->pRegions` must respect the image transfer granularity requirements of `commandBuffer`'s command pool's queue family, as described in `VkQueueFamilyProperties`.

- **VUID-vkCmdCopyBufferToImage2-commandBuffer-07739**
  If the queue family used to create the `VkCommandPool` which `commandBuffer` was allocated from does not support `VK_QUEUE_GRAPHICS_BIT`, for each element of `pCopyBufferToImageInfo->pRegions`, the `aspectMask` member of `imageSubresource` must not be `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT`.

## Valid Usage (Implicit)

- **VUID-vkCmdCopyBufferToImage2-commandBuffer-parameter**
  `commandBuffer` must be a valid `VkCommandBuffer` handle.

- **VUID-vkCmdCopyBufferToImage2-pCopyBufferToImageInfo-parameter**
  `pCopyBufferToImageInfo` must be a valid pointer to a valid `VkCopyBufferToImageInfo2` structure.

- **VUID-vkCmdCopyBufferToImage2-commandBuffer-recording**
  `commandBuffer` must be in the `recording state`.

- **VUID-vkCmdCopyBufferToImage2-commandBuffer-cmdpool**
  The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations.

- **VUID-vkCmdCopyBufferToImage2-renderpass**
  This command must only be called outside of a render pass instance.

## Host Synchronization

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

## Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer Graphics Compute</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkCopyBufferToImageInfo2` structure is defined as:
// Provided by VK_VERSION_1_3

typedef struct VkCopyBufferToImageInfo2 {
    VkStructureType sType;
    const void* pNext;
    VkBuffer srcBuffer;
    VkImage dstImage;
    VkImageLayout dstImageLayout;
    uint32_t regionCount;
    const VkBufferImageCopy2* pRegions;
} VkCopyBufferToImageInfo2;

• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **srcBuffer** is the source buffer.
• **dstImage** is the destination image.
• **dstImageLayout** is the layout of the destination image subresources for the copy.
• **regionCount** is the number of regions to copy.
• **pRegions** is a pointer to an array of VkBufferImageCopy2 structures specifying the regions to copy.

---

**Valid Usage**

• VUID-VkCopyBufferToImageInfo2-pRegions-00172
  The image region specified by each element of **pRegions** must be contained within the specified imageSubresource of **dstImage**

• VUID-VkCopyBufferToImageInfo2-pRegions-00171
  **srcBuffer** must be large enough to contain all buffer locations that are accessed according to **Buffer and Image Addressing**, for each element of **pRegions**

• VUID-VkCopyBufferToImageInfo2-pRegions-00173
  The union of all source regions, and the union of all destination regions, specified by the elements of **pRegions**, must not overlap in memory

• VUID-VkCopyBufferToImageInfo2-srcBuffer-00174
  **srcBuffer** must have been created with **VK_BUFFER_USAGE_TRANSFER_SRC_BIT** usage flag

• VUID-VkCopyBufferToImageInfo2-dstImage-01997
  The format features of **dstImage** must contain **VK_FORMAT_FEATURE_TRANSFER_DST_BIT**

• VUID-VkCopyBufferToImageInfo2-srcBuffer-00176
  If **srcBuffer** is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object

• VUID-VkCopyBufferToImageInfo2-dstImage-00177
  **dstImage** must have been created with **VK_IMAGE_USAGE_TRANSFER_DST_BIT** usage flag

• VUID-VkCopyBufferToImageInfo2-dstImage-00178
If \textit{dstImage} is non-sparse then it \textbf{must} be bound completely and contiguously to a single \texttt{VkDeviceMemory} object

- VUID-VkCopyBufferToImageInfo2-dstImage-00179
  \textit{dstImage} \textbf{must} have a sample count equal to \texttt{VK_SAMPLE_COUNT_1_BIT}

- VUID-VkCopyBufferToImageInfo2-dstImageLayout-00180
  \textit{dstImageLayout} \textbf{must} specify the layout of the image subresources of \textit{dstImage} specified in \textit{pRegions} at the time this command is executed on a \texttt{VkDevice}

- VUID-VkCopyBufferToImageInfo2-dstImageLayout-00181
  \textit{dstImageLayout} \textbf{must} be \texttt{VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL} or \texttt{VK_IMAGE_LAYOUT_GENERAL}

- VUID-VkCopyBufferToImageInfo2-imageSubresource-01701
  The \texttt{imageSubresource.mipLevel} member of each element of \textit{pRegions} \textbf{must} be less than the \texttt{mipLevels} specified in \texttt{VkImageCreateInfo} when \textit{dstImage} was created

- VUID-VkCopyBufferToImageInfo2-imageSubresource-01702
  The \texttt{imageSubresource.baseArrayLayer} + \texttt{imageSubresource.layerCount} of each element of \textit{pRegions} \textbf{must} be less than or equal to the \texttt{arrayLayers} specified in \texttt{VkImageCreateInfo} when \textit{dstImage} was created

- VUID-VkCopyBufferToImageInfo2-None-00214
  For each element of \textit{pRegions} whose \texttt{imageSubresource} contains a depth aspect, the data in \texttt{srcBuffer} \textbf{must} be in the range [0,1]

- VUID-VkCopyBufferToImageInfo2-pRegions-06223
  For each element of \textit{pRegions} not containing \texttt{VkCopyCommandTransformInfoQCOM} in its \texttt{pNext} chain, \texttt{imageOffset.x} and \((\texttt{imageExtent.width} + \texttt{imageOffset.x})\) \textbf{must} both be greater than or equal to 0 and less than or equal to the width of the specified \texttt{imageSubresource} of \textit{dstImage}

- VUID-VkCopyBufferToImageInfo2-pRegions-06224
  For each element of \textit{pRegions} not containing \texttt{VkCopyCommandTransformInfoQCOM} in its \texttt{pNext} chain, \texttt{imageOffset.y} and \((\texttt{imageExtent.height} + \texttt{imageOffset.y})\) \textbf{must} both be greater than or equal to 0 and less than or equal to the height of the specified \texttt{imageSubresource} of \textit{dstImage}

- VUID-VkCopyBufferToImageInfo2-bufferOffset-01558
  If \textit{dstImage} does not have either a depth/stencil or a \texttt{multi-planar format}, then for each element of \textit{pRegions}, \texttt{bufferOffset} \textbf{must} be a multiple of the format’s texel block size

- VUID-VkCopyBufferToImageInfo2-bufferOffset-01559
  If \textit{dstImage} has a \texttt{multi-planar format}, then for each element of \textit{pRegions}, \texttt{bufferOffset} \textbf{must} be a multiple of the element size of the compatible format for the format and the \texttt{aspectMask} of the \texttt{imageSubresource} as defined in \texttt{Compatible formats of planes of multi-planar formats}

- VUID-VkCopyBufferToImageInfo2-srcImage-00199
  If \textit{dstImage} is of type \texttt{VK_IMAGE_TYPE_1D}, then for each element of \textit{pRegions}, \texttt{imageOffset.y} \textbf{must} be 0 and \texttt{imageExtent.height} \textbf{must} be 1

- VUID-VkCopyBufferToImageInfo2-imageOffset-00200
  For each element of \textit{pRegions}, \texttt{imageOffset.z} and \((\texttt{imageExtent.depth} + \texttt{imageOffset.z})\) \textbf{must} both be greater than or equal to 0 and less than or equal to the depth of the specified \texttt{imageSubresource} of \textit{dstImage}
If $\text{dstImage}$ is of type $\text{VK_IMAGE_TYPE_1D}$ or $\text{VK_IMAGE_TYPE_2D}$, then for each element of $\text{pRegions}$, $\text{imageOffset.z}$ must be 0 and $\text{imageExtent.depth}$ must be 1.

For each element of $\text{pRegions}$, $\text{bufferRowLength}$ must be a multiple of the texel block extent width of the $\text{VkFormat}$ of $\text{dstImage}$.

For each element of $\text{pRegions}$, $\text{bufferImageHeight}$ must be a multiple of the texel block extent height of the $\text{VkFormat}$ of $\text{dstImage}$.

For each element of $\text{pRegions}$, $\text{bufferOffset}$ must be a multiple of the texel block size of the $\text{VkFormat}$ of $\text{dstImage}$.

For each element of $\text{pRegions}$, $\text{imageOffset.x}$ must be a multiple of the texel block extent width of the $\text{VkFormat}$ of $\text{dstImage}$.

For each element of $\text{pRegions}$, $\text{imageOffset.y}$ must be a multiple of the texel block extent height of the $\text{VkFormat}$ of $\text{dstImage}$.

For each element of $\text{pRegions}$, $\text{bufferOffset}$ must be a multiple of the texel block extend depth of the $\text{VkFormat}$ of $\text{dstImage}$.

For each element of $\text{pRegions}$, if the sum of $\text{imageOffset.x}$ and $\text{extent.width}$ does not equal the width of the the subresource specified by $\text{srcSubresource}$, $\text{extent.width}$ must be a multiple of the texel block extent width of the $\text{VkFormat}$ of $\text{dstImage}$.

For each element of $\text{pRegions}$, if the sum of $\text{imageOffset.y}$ and $\text{extent.height}$ does not equal the height of the the subresource specified by $\text{srcSubresource}$, $\text{extent.height}$ must be a multiple of the texel block extent height of the $\text{VkFormat}$ of $\text{dstImage}$.

For each element of $\text{pRegions}$, if the sum of $\text{imageOffset.z}$ and $\text{extent.depth}$ does not equal the depth of the the subresource specified by $\text{srcSubresource}$, $\text{extent.depth}$ must be a multiple of the texel block extent depth of the $\text{VkFormat}$ of $\text{dstImage}$.

For each element of $\text{pRegions}$, $\text{imageSubresource.aspectMask}$ must specify aspects present in $\text{dstImage}$.

If $\text{dstImage}$ has a $\text{VkFormat}$ with two planes then for each element of $\text{pRegions}$, $\text{imageSubresource.aspectMask}$ must be $\text{VK_IMAGE_ASPECT_PLANE_0_BIT}$ or $\text{VK_IMAGE_ASPECT_PLANE_1_BIT}$.

If $\text{dstImage}$ has a $\text{VkFormat}$ with three planes then for each element of $\text{pRegions}$, $\text{imageSubresource.aspectMask}$ must be $\text{VK_IMAGE_ASPECT_PLANE_0_BIT}$,
• VUID-VkCopyBufferToImageInfo2-baseArrayLayer-00213
  If \texttt{dstImage} is of type \texttt{VK_IMAGE_TYPE_3D}, for each element of \texttt{pRegions},
  \texttt{imageSubresource.baseArrayLayer} must be 0 and \texttt{imageSubresource.layerCount} must be 1

• VUID-VkCopyBufferToImageInfo2-pRegions-07277
  For each element of \texttt{pRegions}, \texttt{bufferRowLength} divided by the texel block extent width
  and then multiplied by the texel block size of \texttt{dstImage} must be less than or equal to \(2^{31}-1\)

• VUID-VkCopyBufferToImageInfo2-srcImage-04053
  If \texttt{dstImage} has a depth/stencil format, the \texttt{bufferOffset} member of any element of
  \texttt{pRegions} must be a multiple of 4

Valid Usage (Implicit)

• VUID-VkCopyBufferToImageInfo2-sType-sType
  \texttt{sType} must be \texttt{VK_STRUCTURE_TYPE_COPY_BUFFER_TO_IMAGE_INFO_2}

• VUID-VkCopyBufferToImageInfo2-pNext-pNext
  \texttt{pNext} must be \texttt{NULL}

• VUID-VkCopyBufferToImageInfo2-srcBuffer-parameter
  \texttt{srcBuffer} must be a valid \texttt{VkBuffer} handle

• VUID-VkCopyBufferToImageInfo2-dstImage-parameter
  \texttt{dstImage} must be a valid \texttt{VkImage} handle

• VUID-VkCopyBufferToImageInfo2-dstImageLayout-parameter
  \texttt{dstImageLayout} must be a valid \texttt{VkImageLayout} value

• VUID-VkCopyBufferToImageInfo2-pRegions-parameter
  \texttt{pRegions} must be a valid pointer to an array of \texttt{regionCount} valid \texttt{VkBufferImageCopy2}
  structures

• VUID-VkCopyBufferToImageInfo2-regionCount-arraylength
  \texttt{regionCount} must be greater than 0

• VUID-VkCopyBufferToImageInfo2-commonparent
  Both of \texttt{dstImage}, and \texttt{srcBuffer} must have been created, allocated, or retrieved from the
  same \texttt{VkDevice}

To copy data from an image object to a buffer object, call:

```c
// Provided by VK_VERSION_1_3
void vkCmdCopyImageToBuffer2(
  VkCommandBuffer commandBuffer,
  const VkCopyImageToBufferInfo2* pCopyImageToBufferInfo);
```

• \texttt{commandBuffer} is the command buffer into which the command will be recorded.
• \texttt{pCopyImageToBufferInfo} is a pointer to a \texttt{VkCopyImageToBufferInfo2} structure describing the
This command is functionally identical to `vkCmdCopyImageToBuffer`, but includes extensible substructures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

### Valid Usage

- **VUID-vkCmdCopyImageToBuffer2-commandBuffer-01831**
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `srcImage` must not be a protected image

- **VUID-vkCmdCopyImageToBuffer2-commandBuffer-01832**
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `dstBuffer` must not be a protected buffer

- **VUID-vkCmdCopyImageToBuffer2-commandBuffer-01833**
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `dstBuffer` must not be an unprotected buffer

- **VUID-vkCmdCopyImageToBuffer2-commandBuffer-07746**
  If the queue family used to create the `VkCommandPool` which `commandBuffer` was allocated from does not support `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT`, the `bufferOffset` member of any element of `pCopyImageToBufferInfo->pRegions` must be a multiple of 4

- **VUID-vkCmdCopyImageToBuffer2-imageOffset-07747**
  The `imageOffset` and `imageExtent` members of each element of `pCopyImageToBufferInfo->pRegions` must respect the image transfer granularity requirements of `commandBuffer`'s command pool's queue family, as described in `VkQueueFamilyProperties`

### Valid Usage (Implicit)

- **VUID-vkCmdCopyImageToBuffer2-commandBuffer-parameter**
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdCopyImageToBuffer2-pCopyImageToBufferInfo-parameter**
  `pCopyImageToBufferInfo` must be a valid pointer to a valid `VkCopyImageToBufferInfo2` structure

- **VUID-vkCmdCopyImageToBuffer2-commandBuffer-recording**
  `commandBuffer` must be in the `recording` state

- **VUID-vkCmdCopyImageToBuffer2-commandBuffer-cmdpool**
  The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations

- **VUID-vkCmdCopyImageToBuffer2-renderpass**
  This command must only be called outside of a render pass instance
Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

The `VkCopyImageToBufferInfo2` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkCopyImageToBufferInfo2 {  
    VkStructureType sType;
    const void* pNext;
    VkImage srcImage;
    VkImageLayout srcImageLayout;
    VkBuffer dstBuffer;
    uint32_t regionCount;
    const VkBufferImageCopy2* pRegions;
} VkCopyImageToBufferInfo2;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcImage` is the source image.
- `srcImageLayout` is the layout of the source image subresources for the copy.
- `dstBuffer` is the destination buffer.
- `regionCount` is the number of regions to copy.
- `pRegions` is a pointer to an array of `VkBufferImageCopy2` structures specifying the regions to copy.

Valid Usage

- VUID-VkCopyImageToBufferInfo2-pRegions-00182
  The image region specified by each element of `pRegions` must be contained within the specified `imageSubresource` of `srcImage`
**VUID-VkCopyImageToBufferInfo2-pRegions-00183**

`dstBuffer` must be large enough to contain all buffer locations that are accessed according to Buffer and Image Addressing, for each element of `pRegions`.

**VUID-VkCopyImageToBufferInfo2-pRegions-00184**

The union of all source regions, and the union of all destination regions, specified by the elements of `pRegions`, must not overlap in memory.

**VUID-VkCopyImageToBufferInfo2-srcImage-00186**

`srcImage` must have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage flag.

**VUID-VkCopyImageToBufferInfo2-srcImage-00187**

If `srcImage` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.

**VUID-VkCopyImageToBufferInfo2-dstBuffer-00191**

`dstBuffer` must have been created with `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage flag.

**VUID-VkCopyImageToBufferInfo2-dstBuffer-00192**

If `dstBuffer` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.

**VUID-VkCopyImageToBufferInfo2-srcImage-00188**

`srcImage` must have a sample count equal to `VK_SAMPLE_COUNT_1_BIT`.

**VUID-VkCopyImageToBufferInfo2-srcImageLayout-00189**

`srcImageLayout` must specify the layout of the image subresources of `srcImage` specified in `pRegions` at the time this command is executed on a `VkDevice`.

**VUID-VkCopyImageToBufferInfo2-srcImageLayout-00190**

`srcImageLayout` must be `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`.

**VUID-VkCopyImageToBufferInfo2-imageSubresource-01703**

The `imageSubresource.mipLevel` member of each element of `pRegions` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `srcImage` was created.

**VUID-VkCopyImageToBufferInfo2-imageSubresource-01704**

The `imageSubresource.baseArrayLayer` + `imageSubresource.layerCount` of each element of `pRegions` must be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `srcImage` was created.

**VUID-VkCopyImageToBufferInfo2-imageOffset-00197**

For each element of `pRegions` not containing `VkCopyCommandTransformInfoQCOM` in its `pNext` chain, `imageOffset.x` and `(imageExtent.width + imageOffset.x)` must both be greater than or equal to 0 and less than or equal to the width of the specified `imageSubresource` of `srcImage`.

**VUID-VkCopyImageToBufferInfo2-imageOffset-00198**

For each element of `pRegions` not containing `VkCopyCommandTransformInfoQCOM` in its `pNext` chain, `imageOffset.y` and `(imageExtent.height + imageOffset.y)` must both be greater than or equal to 0 and less than or equal to the height of the specified `imageSubresource` of `srcImage`.

**VUID-VkCopyImageToBufferInfo2-bufferOffset-01558**
If *srcImage* does not have either a depth/stencil or a *multi-planar format*, then for each element of *pRegions*, *bufferOffset* must be a multiple of the format’s texel block size

- **VUID-VkCopyImageToBufferInfo2-bufferOffset-01559**
  If *srcImage* has a *multi-planar format*, then for each element of *pRegions*, *bufferOffset* must be a multiple of the element size of the compatible format for the format and the *aspectMask* of the *imageSubresource* as defined in **Compatible formats of planes of multi-planar formats**

- **VUID-VkCopyImageToBufferInfo2-srcImage-00199**
  If *srcImage* is of type *VK_IMAGE_TYPE_1D*, then for each element of *pRegions*, *imageOffset.y* must be 0 and *imageExtent.height* must be 1

- **VUID-VkCopyImageToBufferInfo2-srcImage-00201**
  If *srcImage* is of type *VK_IMAGE_TYPE_1D* or *VK_IMAGE_TYPE_2D*, then for each element of *pRegions*, *imageOffset.z* must be 0 and *imageExtent.depth* must be 1

- **VUID-VkCopyImageToBufferInfo2-bufferRowLength-00203**
  For each element of *pRegions*, *bufferRowLength* must be a multiple of the texel block extent width of the *VkFormat* of *srcImage*

- **VUID-VkCopyImageToBufferInfo2-bufferImageHeight-00204**
  For each element of *pRegions*, *bufferImageHeight* must be a multiple of the texel block extent height of the *VkFormat* of *srcImage*

- **VUID-VkCopyImageToBufferInfo2-pRegions-07273**
  For each element of *pRegions*, *bufferRegions* must be a multiple of the texel block size of the *VkFormat* of *srcImage*

- **VUID-VkCopyImageToBufferInfo2-pRegions-07274**
  For each element of *pRegions*, *imageOffset.x* must be a multiple of the texel block extent width of the *VkFormat* of *srcImage*

- **VUID-VkCopyImageToBufferInfo2-pRegions-07275**
  For each element of *pRegions*, *imageOffset.y* must be a multiple of the texel block extent height of the *VkFormat* of *srcImage*

- **VUID-VkCopyImageToBufferInfo2-pRegions-07276**
  For each element of *pRegions*, *imageOffset.z* must be a multiple of the texel block extent depth of the *VkFormat* of *srcImage*

- **VUID-VkCopyImageToBufferInfo2-imageExtent-00207**
  For each element of *pRegions*, if the sum of *imageOffset.x* and *extent.width* does not equal the width of the the subresource specified by *srcSubresource*, *extent.width* must be a multiple of the texel block extent width of the *VkFormat* of *srcImage*

- **VUID-VkCopyImageToBufferInfo2-imageExtent-00208**
  For each element of *pRegions*, if the sum of *imageOffset.y* and *extent.height* does not equal the height of the the subresource specified by *srcSubresource*, *extent.height* must be a multiple of the texel block extent height of the *VkFormat* of *srcImage*
For each element of \( pRegions \), if the sum of \( \text{imageOffset.z} \) and \( \text{extent.depth} \) does not equal the depth of the the subresource specified by \( \text{srcSubresource} \), \( \text{extent.depth} \) must be a multiple of the \( \text{texel block extent depth} \) of the \( \text{VkFormat} \) of \( \text{srcImage} \).

For each element of \( pRegions \), \( \text{imageSubresource.aspectMask} \) must specify aspects present in \( \text{srcImage} \).

If \( \text{srcImage} \) has a \( \text{VkFormat} \) with two planes then for each element of \( pRegions \), \( \text{imageSubresource.aspectMask} \) must be \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) or \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \).

If \( \text{srcImage} \) has a \( \text{VkFormat} \) with three planes then for each element of \( pRegions \), \( \text{imageSubresource.aspectMask} \) must be \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \), \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \), or \( \text{VK_IMAGE_ASPECT_PLANE_2_BIT} \).

If \( \text{srcImage} \) is of type \( \text{VK_IMAGE_TYPE_3D} \), for each element of \( pRegions \), \( \text{imageSubresource.baseArrayLayer} \) must be 0 and \( \text{imageSubresource.layerCount} \) must be 1.

For each element of \( pRegions \), \( \text{bufferRowLength} \) divided by the \( \text{texel block extent width} \) and then multiplied by the \( \text{texel block size of srcImage} \) must be less than or equal to \( 2^{31}-1 \).

If \( \text{srcImage} \) has a depth/stencil format, the \( \text{bufferOffset} \) member of any element of \( pRegions \) must be a multiple of 4.

Valid Usage (Implicit)

\[ sType \] must be \( \text{VK_STRUCTURE_TYPE_COPY_IMAGE_TO_BUFFER_INFO_2} \)

\[ pNext \] must be NULL

\[ \text{srcImage} \] must be a valid \( \text{VkImage} \) handle

\[ \text{srcImageLayout} \] must be a valid \( \text{VkImageLayout} \) value

\[ \text{dstBuffer} \] must be a valid \( \text{VkBuffer} \) handle

\[ pRegions \] must be a valid pointer to an array of \( \text{regionCount} \) valid \( \text{VkBufferImageCopy2} \) structures

\[ \text{regionCount} \] must be greater than 0
Both of \textit{dstBuffer}, and \textit{srcImage} must have been created, allocated, or retrieved from the same \textit{VkDevice}.

For both \textit{vkCmdCopyBufferToImage2} and \textit{vkCmdCopyImageToBuffer2}, each element of \textit{pRegions} is a structure defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkBufferImageCopy2 {
    VkStructureType sType;
    const void* pNext;
    VkDeviceSize bufferOffset;
    uint32_t bufferRowLength;
    uint32_t bufferImageHeight;
    VkImageSubresourceLayers imageSubresource;
    VkOffset3D imageOffset;
    VkExtent3D imageExtent;
} VkBufferImageCopy2;
```

- \textit{sType} is the type of this structure.
- \textit{pNext} is NULL or a pointer to a structure extending this structure.
- \textit{bufferOffset} is the offset in bytes from the start of the buffer object where the image data is copied from or to.
- \textit{bufferRowLength} and \textit{bufferImageHeight} specify in texels a subregion of a larger two- or three-dimensional image in buffer memory, and control the addressing calculations. If either of these values is zero, that aspect of the buffer memory is considered to be tightly packed according to the \textit{imageExtent}.
- \textit{imageSubresource} is a \textit{VkImageSubresourceLayers} used to specify the specific image subresources of the image used for the source or destination image data.
- \textit{imageOffset} selects the initial \textit{x}, \textit{y}, \textit{z} offsets in texels of the sub-region of the source or destination image data.
- \textit{imageExtent} is the size in texels of the image to copy in width, height and depth.

This structure is functionally identical to \textit{VkBufferImageCopy}, but adds \textit{sType} and \textit{pNext} parameters, allowing it to be more easily extended.

**Valid Usage**

- VUID-VkBufferImageCopy2-bufferRowLength-00195
  \textit{bufferRowLength} must be 0, or greater than or equal to the \textit{width} member of \textit{imageExtent}

- VUID-VkBufferImageCopy2-bufferImageHeight-00196
  \textit{bufferImageHeight} must be 0, or greater than or equal to the \textit{height} member of \textit{imageExtent}

- VUID-VkBufferImageCopy2-aspectMask-00212
  The \textit{aspectMask} member of \textit{imageSubresource} must only have a single bit set
• VUID-VkBufferImageCopy2-imageExtent-06659
  imageExtent.width must not be 0
• VUID-VkBufferImageCopy2-imageExtent-06660
  imageExtent.height must not be 0
• VUID-VkBufferImageCopy2-imageExtent-06661
  imageExtent.depth must not be 0

Valid Usage (Implicit)

• VUID-VkBufferImageCopy2-sType-sType
  sType must be VK_STRUCTURE_TYPE_BUFFER_IMAGE_COPY_2
• VUID-VkBufferImageCopy2-pNext-pNext
  pNext must be NULL
• VUID-VkBufferImageCopy2-imageSubresource-parameter
  imageSubresource must be a valid VkImageSubresourceLayers structure

19.4. Image Copies with Scaling

To copy regions of a source image into a destination image, potentially performing format conversion, arbitrary scaling, and filtering, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdBlitImage(
    VkCommandBuffer commandBuffer,
    VkImage srcImage,
    VkImageLayout srcImageLayout,
    VkImage dstImage,
    VkImageLayout dstImageLayout,
    uint32_t regionCount,
    const VkImageBlit* pRegions,
    VkFilter filter);
```

• `commandBuffer` is the command buffer into which the command will be recorded.
• `srcImage` is the source image.
• `srcImageLayout` is the layout of the source image subresources for the blit.
• `dstImage` is the destination image.
• `dstImageLayout` is the layout of the destination image subresources for the blit.
• `regionCount` is the number of regions to blit.
• `pRegions` is a pointer to an array of `VkImageBlit` structures specifying the regions to blit.
• `filter` is a `VkFilter` specifying the filter to apply if the blits require scaling.
**vkCmdBlitImage** must not be used for multisampled source or destination images. Use **vkCmdResolveImage** for this purpose.

As the sizes of the source and destination extents can differ in any dimension, texels in the source extent are scaled and filtered to the destination extent. Scaling occurs via the following operations:

- For each destination texel, the integer coordinate of that texel is converted to an unnormalized texture coordinate, using the effective inverse of the equations described in **unnormalized to integer conversion**:

  \[
  u_{\text{base}} = i + \frac{1}{2} \\
  v_{\text{base}} = j + \frac{1}{2} \\
  w_{\text{base}} = k + \frac{1}{2}
  \]

- These base coordinates are then offset by the first destination offset:

  \[
  u_{\text{offset}} = u_{\text{base}} - x_{\text{dst0}} \\
  v_{\text{offset}} = v_{\text{base}} - y_{\text{dst0}} \\
  w_{\text{offset}} = w_{\text{base}} - z_{\text{dst0}} \\
  a_{\text{offset}} = a - \text{baseArrayCount}_{\text{dst}}
  \]

- The scale is determined from the source and destination regions, and applied to the offset coordinates:

  \[
  \text{scale}_u = (x_{\text{src1}} - x_{\text{src0}}) / (x_{\text{dst1}} - x_{\text{dst0}}) \\
  \text{scale}_v = (y_{\text{src1}} - y_{\text{src0}}) / (y_{\text{dst1}} - y_{\text{dst0}}) \\
  \text{scale}_w = (z_{\text{src1}} - z_{\text{src0}}) / (z_{\text{dst1}} - z_{\text{dst0}})
  \]

  \[
  u_{\text{scaled}} = u_{\text{offset}} \times \text{scale}_u
  \]
\[ v_{\text{scaled}} = v_{\text{offset}} \times \text{scale}_v \]

\[ w_{\text{scaled}} = w_{\text{offset}} \times \text{scale}_w \]

- Finally the source offset is added to the scaled coordinates, to determine the final unnormalized coordinates used to sample from srcImage:

\[ u = u_{\text{scaled}} + x_{\text{src}0} \]

\[ v = v_{\text{scaled}} + y_{\text{src}0} \]

\[ w = w_{\text{scaled}} + z_{\text{src}0} \]

\[ q = \text{mipLevel} \]

\[ a = a_{\text{offset}} + \text{baseArrayCount}_\text{src} \]

These coordinates are used to sample from the source image, as described in Image Operations chapter, with the filter mode equal to that of filter, a mipmap mode of VK_SAMPLER_MIPMAP_MODE_NEAREST and an address mode of VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE. Implementations must clamp at the edge of the source image, and may additionally clamp to the edge of the source region.

**Note**

Due to allowable rounding errors in the generation of the source texture coordinates, it is not always possible to guarantee exactly which source texels will be sampled for a given blit. As rounding errors are implementation-dependent, the exact results of a blitting operation are also implementation-dependent.

Blits are done layer by layer starting with the baseArrayLayer member of srcSubresource for the source and dstSubresource for the destination. layerCount layers are blitted to the destination image.

When blitting 3D textures, slices in the destination region bounded by dstOffsets[0].z and dstOffsets[1].z are sampled from slices in the source region bounded by srcOffsets[0].z and srcOffsets[1].z. If the filter parameter is VK_FILTER_LINEAR then the value sampled from the source image is taken by doing linear filtering using the interpolated \( z \) coordinate represented by \( w \) in the previous equations. If the filter parameter is VK_FILTER_NEAREST then the value sampled from the source image is taken from the single nearest slice, with an implementation-dependent arithmetic rounding mode.

The following filtering and conversion rules apply:
• Integer formats can only be converted to other integer formats with the same signedness.
• No format conversion is supported between depth/stencil images. The formats must match.
• Format conversions on unorm, snorm, scaled and packed float formats of the copied aspect of the image are performed by first converting the pixels to float values.
• For sRGB source formats, nonlinear RGB values are converted to linear representation prior to filtering.
• After filtering, the float values are first clamped and then cast to the destination image format. In case of sRGB destination format, linear RGB values are converted to nonlinear representation before writing the pixel to the image.

Signed and unsigned integers are converted by first clamping to the representable range of the destination format, then casting the value.

Valid Usage

• VUID-vkCmdBlitImage-commandBuffer-01834
  If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, srcImage must not be a protected image

• VUID-vkCmdBlitImage-commandBuffer-01835
  If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, dstImage must not be a protected image

• VUID-vkCmdBlitImage-commandBuffer-01836
  If commandBuffer is a protected command buffer and protectedNoFault is not supported, dstImage must not be an unprotected image

• VUID-vkCmdBlitImage-pRegions-00215
  The source region specified by each element of pRegions must be a region that is contained within srcImage

• VUID-vkCmdBlitImage-pRegions-00216
  The destination region specified by each element of pRegions must be a region that is contained within dstImage

• VUID-vkCmdBlitImage-pRegions-00217
  The union of all destination regions, specified by the elements of pRegions, must not overlap in memory with any texel that may be sampled during the blit operation

• VUID-vkCmdBlitImage-srcImage-01999
  The format features of srcImage must contain VK_FORMAT_FEATURE_BLIT_SRC_BIT

• VUID-vkCmdBlitImage-srcImage-06421
  srcImage must not use a format that requires a sampler YCbCr conversion

• VUID-vkCmdBlitImage-srcImage-00219
  srcImage must have been created with VK_IMAGE_USAGE_TRANSFER_SRC_BIT usage flag

• VUID-vkCmdBlitImage-srcImage-00220
  If srcImage is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object
*VUID-vkCmdBlitImage-srcImageLayout-00221*

srcImageLayout must specify the layout of the image subresources of srcImage specified in pRegions at the time this command is executed on a VkDevice.

*VUID-vkCmdBlitImage-srcImageLayout-00222*

srcImageLayout must be VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL.

*VUID-vkCmdBlitImage-dstImage-02000*

The format features of dstImage must contain VK_FORMAT_FEATURE_BLIT_DST_BIT.

*VUID-vkCmdBlitImage-dstImage-06422*

dstImage must not use a format that requires a sampler YC_{B}C_{R} conversion.

*VUID-vkCmdBlitImage-dstImage-00224*

dstImage must have been created with VK_IMAGE_USAGE_TRANSFER_DST_BIT usage flag.

*VUID-vkCmdBlitImage-dstImage-00225*

If dstImage is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object.

*VUID-vkCmdBlitImage-dstImageLayout-00226*

dstImageLayout must specify the layout of the image subresources of dstImage specified in pRegions at the time this command is executed on a VkDevice.

*VUID-vkCmdBlitImage-dstImageLayout-00227*

dstImageLayout must be VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL.

*VUID-vkCmdBlitImage-srcImage-00229*

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.

*VUID-vkCmdBlitImage-srcImage-00230*

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.

*VUID-vkCmdBlitImage-srcImage-00231*

If either of srcImage or dstImage was created with a depth/stencil format, the other must have exactly the same format.

*VUID-vkCmdBlitImage-srcImage-00232*

If srcImage was created with a depth/stencil format, filter must be VK_FILTER_NEAREST.

*VUID-vkCmdBlitImage-srcImage-00233*

srcImage must have been created with a samples value of VK_SAMPLE_COUNT_1_BIT.

*VUID-vkCmdBlitImage-dstImage-00234*

dstImage must have been created with a samples value of VK_SAMPLE_COUNT_1_BIT.

*VUID-vkCmdBlitImage-filter-02001*

If filter is VK_FILTER_LINEAR, then the format features of srcImage must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT.

*VUID-vkCmdBlitImage-srcSubresource-01705*

The srcSubresource.mipLevel member of each element of pRegions must be less than the mipLevels specified in VkImageCreateInfo when srcImage was created.

*VUID-vkCmdBlitImage-dstSubresource-01706*
The `dstSubresource.mipLevel` member of each element of `pRegions` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `dstImage` was created.

- **VUID-vkCmdBlitImage-srcSubresource-01707**
  The `srcSubresource.baseArrayLayer + srcSubresource.layerCount` of each element of `pRegions` must be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `srcImage` was created.

- **VUID-vkCmdBlitImage-dstSubresource-01708**
  The `dstSubresource.baseArrayLayer + dstSubresource.layerCount` of each element of `pRegions` must be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `dstImage` was created.

- **VUID-vkCmdBlitImage-srcImage-00240**
  If either `srcImage` or `dstImage` is of type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions, srcSubresource.baseArrayLayer` and `dstSubresource.baseArrayLayer` must each be `0`, and `srcSubresource.layerCount` and `dstSubresource.layerCount` must each be `1`.

- **VUID-vkCmdBlitImage-aspectMask-00241**
  For each element of `pRegions, srcSubresource.aspectMask` must specify aspects present in `srcImage`.

- **VUID-vkCmdBlitImage-aspectMask-00242**
  For each element of `pRegions, dstSubresource.aspectMask` must specify aspects present in `dstImage`.

- **VUID-vkCmdBlitImage-srcOffset-00243**
  For each element of `pRegions, srcOffsets[0].x` and `srcOffsets[1].x` must both be greater than or equal to `0` and less than or equal to the width of the specified `srcSubresource` of `srcImage`.

- **VUID-vkCmdBlitImage-srcOffset-00244**
  For each element of `pRegions, srcOffsets[0].y` and `srcOffsets[1].y` must both be greater than or equal to `0` and less than or equal to the height of the specified `srcSubresource` of `srcImage`.

- **VUID-vkCmdBlitImage-srcImage-00245**
  If `srcImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions, srcOffsets[0].y` must be `0` and `srcOffsets[1].y` must be `1`.

- **VUID-vkCmdBlitImage-srcOffset-00246**
  For each element of `pRegions, srcOffsets[0].z` and `srcOffsets[1].z` must both be greater than or equal to `0` and less than or equal to the depth of the specified `srcSubresource` of `srcImage`.

- **VUID-vkCmdBlitImage-srcImage-00247**
  If `srcImage` is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then for each element of `pRegions, srcOffsets[0].z` must be `0` and `srcOffsets[1].z` must be `1`.

- **VUID-vkCmdBlitImage-dstOffset-00248**
  For each element of `pRegions, dstOffsets[0].x` and `dstOffsets[1].x` must both be greater than or equal to `0` and less than or equal to the width of the specified `dstSubresource` of `dstImage`.

- **VUID-vkCmdBlitImage-dstOffset-00249**
  For each element of `pRegions, dstOffsets[0].y` and `dstOffsets[1].y` must both be greater than or equal to `0` and less than or equal to the height of the specified `dstSubresource` of `dstImage`.

- **VUID-vkCmdBlitImage-dstOffset-00249**
  For each element of `pRegions, dstOffsets[0].z` and `dstOffsets[1].z` must both be greater than or equal to `0` and less than or equal to the depth of the specified `dstSubresource` of `dstImage`.
For each element of pRegions, dstOffsets[0].y and dstOffsets[1].y must both be greater than or equal to 0 and less than or equal to the height of the specified dstSubresource of dstImage.

• VUID-vkCmdBlitImage-dstImage-00250
  If dstImage is of type VK_IMAGE_TYPE_1D, then for each element of pRegions, dstOffsets[0].y must be 0 and dstOffsets[1].y must be 1.

• VUID-vkCmdBlitImage-dstOffset-00251
  For each element of pRegions, dstOffsets[0].z and dstOffsets[1].z must both be greater than or equal to 0 and less than or equal to the depth of the specified dstSubresource of dstImage.

• VUID-vkCmdBlitImage-dstImage-00252
  If dstImage is of type VK_IMAGE_TYPE_1D or VK_IMAGE_TYPE_2D, then for each element of pRegions, dstOffsets[0].z must be 0 and dstOffsets[1].z must be 1.

Valid Usage (Implicit)

• VUID-vkCmdBlitImage-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle.

• VUID-vkCmdBlitImage-srcImage-parameter
  srcImage must be a valid VkImage handle.

• VUID-vkCmdBlitImage-srcImageLayout-parameter
  srcImageLayout must be a valid VkImageLayout value.

• VUID-vkCmdBlitImage-dstImage-parameter
  dstImage must be a valid VkImage handle.

• VUID-vkCmdBlitImage-dstImageLayout-parameter
  dstImageLayout must be a valid VkImageLayout value.

• VUID-vkCmdBlitImage-pRegions-parameter
  pRegions must be a valid pointer to an array of regionCount valid VkImageBlit structures.

• VUID-vkCmdBlitImage-filter-parameter
  filter must be a valid VkFilter value.

• VUID-vkCmdBlitImage-commandBuffer-recording
  commandBuffer must be in the recording state.

• VUID-vkCmdBlitImage-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations.

• VUID-vkCmdBlitImage-renderpass
  This command must only be called outside of a render pass instance.

• VUID-vkCmdBlitImage-regionCount-arraylength
  regionCount must be greater than 0.

• VUID-vkCmdBlitImage-commonparent
  Each of commandBuffer, dstImage, and srcImage must have been created, allocated, or
retrieved from the same VkDevice

**Host Synchronization**

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

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkImageBlit` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkImageBlit {
    VkImageSubresourceLayers srcSubresource; 
    VkOffset3D srcOffsets[2];
    VkImageSubresourceLayers dstSubresource; 
    VkOffset3D dstOffsets[2];
} VkImageBlit;
```

- `srcSubresource` is the subresource to blit from.
- `srcOffsets` is a pointer to an array of two `VkOffset3D` structures specifying the bounds of the source region within `srcSubresource`.
- `dstSubresource` is the subresource to blit into.
- `dstOffsets` is a pointer to an array of two `VkOffset3D` structures specifying the bounds of the destination region within `dstSubresource`.

For each element of the `pRegions` array, a blit operation is performed for the specified source and destination regions.

**Valid Usage**

- VUID-VkImageBlit-aspectMask-00238
  The `aspectMask` member of `srcSubresource` and `dstSubresource` must match
- VUID-VkImageBlit-layerCount-00239
  The `layerCount` member of `srcSubresource` and `dstSubresource` must match
A more extensible version of the blit image command is defined below.

To copy regions of a source image into a destination image, potentially performing format conversion, arbitrary scaling, and filtering, call:

```
// Provided by VK_VERSION_1_3
void vkCmdBlitImage2(
    VkCommandBuffer commandBuffer,
    const VkBlitImageInfo2* pBlitImageInfo);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pBlitImageInfo` is a pointer to a `VkBlitImageInfo2` structure describing the blit parameters.

This command is functionally identical to `vkCmdBlitImage`, but includes extensible sub-structures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

Valid Usage

- VUID-vkCmdBlitImage2-commandBuffer-01834
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `srcImage` must not be a protected image

- VUID-vkCmdBlitImage2-commandBuffer-01835
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `dstImage` must not be a protected image

- VUID-vkCmdBlitImage2-commandBuffer-01836
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `dstImage` must not be an unprotected image
• VUID-vkCmdBlitImage2-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

• VUID-vkCmdBlitImage2-renderpass
  This command must only be called outside of a render pass instance

### Host Synchronization

• Host access to commandBuffer must be externally synchronized

• Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The VkBlitImageInfo2 structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkBlitImageInfo2 {
    VkStructureType sType;
    const void* pNext;
    VkImage srcImage;
    VkImageLayout srcImageLayout;
    VkImage dstImage;
    VkImageLayout dstImageLayout;
    uint32_t regionCount;
    const VkImageBlit2* pRegions;
    VkFilter filter;
} VkBlitImageInfo2;
```

• `sType` is the type of this structure.

• `pNext` is NULL or a pointer to a structure extending this structure.

• `srcImage` is the source image.

• `srcImageLayout` is the layout of the source image subresources for the blit.

• `dstImage` is the destination image.

• `dstImageLayout` is the layout of the destination image subresources for the blit.

• `regionCount` is the number of regions to blit.
• `pRegions` is a pointer to an array of `VkImageBlit2` structures specifying the regions to blit.

• `filter` is a `VkFilter` specifying the filter to apply if the blits require scaling.

**Valid Usage**

- VUID-VkBlitImageInfo2-pRegions-00215
  The source region specified by each element of `pRegions` must be a region that is contained within `srcImage`

- VUID-VkBlitImageInfo2-pRegions-00216
  The destination region specified by each element of `pRegions` must be a region that is contained within `dstImage`

- VUID-VkBlitImageInfo2-pRegions-00217
  The union of all destination regions, specified by the elements of `pRegions`, must not overlap in memory with any texel that may be sampled during the blit operation

- VUID-VkBlitImageInfo2-srcImage-01999
  The format features of `srcImage` must contain `VK_FORMAT_FEATURE_BLIT_SRC_BIT`

- VUID-VkBlitImageInfo2-srcImage-06421
  `srcImage` must not use a format that requires a sampler Y'C'B'C'R conversion

- VUID-VkBlitImageInfo2-srcImage-00219
  `srcImage` must have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage flag

- VUID-VkBlitImageInfo2-srcImage-00220
  If `srcImage` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object

- VUID-VkBlitImageInfo2-srcImageLayout-00221
  `srcImageLayout` must specify the layout of the image subresources of `srcImage` specified in `pRegions` at the time this command is executed on a `VkDevice`

- VUID-VkBlitImageInfo2-srcImageLayout-00222
  `srcImageLayout` must be `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`

- VUID-VkBlitImageInfo2-dstImage-02000
  The format features of `dstImage` must contain `VK_FORMAT_FEATURE_BLIT_DST_BIT`

- VUID-VkBlitImageInfo2-dstImage-06422
  `dstImage` must not use a format that requires a sampler Y'C'B'C'R conversion

- VUID-VkBlitImageInfo2-dstImage-00224
  `dstImage` must have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag

- VUID-VkBlitImageInfo2-dstImage-00225
  If `dstImage` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object

- VUID-VkBlitImageInfo2-dstImageLayout-00226
  `dstImageLayout` must specify the layout of the image subresources of `dstImage` specified in `pRegions` at the time this command is executed on a `VkDevice`

- VUID-VkBlitImageInfo2-dstImageLayout-00227
  `dstImageLayout` must be `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`
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 `srcImage` was created with a `VkSampleCountFlagBits` value of `VK_SAMPLE_COUNT_1_BIT`, `samples` must be exactly the same.

If `filter` is `VK_FILTER_LINEAR`, then the `formatFeatures` of `srcImage` must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.

The `srcSubresource.mipLevel` member of each element of `pRegions` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `srcImage` was created.

The `dstSubresource.mipLevel` member of each element of `pRegions` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `dstImage` was created.

The `srcSubresource.baseArrayLayer + srcSubresource.layerCount` of each element of `pRegions` must be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `srcImage` was created.

The `dstSubresource.baseArrayLayer + dstSubresource.layerCount` of each element of `pRegions` must be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `dstImage` was created.

If either `srcImage` or `dstImage` is of type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions`, `srcSubresource.baseArrayLayer` and `dstSubresource.baseArrayLayer` must each be 0, and `srcSubresource.layerCount` and `dstSubresource.layerCount` must each be 1.

For each element of `pRegions`, `srcSubresource.aspectMask` must specify aspects present in `srcImage`.

For each element of `pRegions`, `dstSubresource.aspectMask` must specify aspects present in `dstImage`.
For each element of `pRegions`, `srcOffsets[0].x` and `srcOffsets[1].x` must both be greater than or equal to 0 and less than or equal to the width of the specified `srcSubresource` of `srcImage`.

For each element of `pRegions`, `srcOffsets[0].y` and `srcOffsets[1].y` must both be greater than or equal to 0 and less than or equal to the height of the specified `srcSubresource` of `srcImage`.

If `srcImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `srcOffsets[0].y` must be 0 and `srcOffsets[1].y` must be 1.

For each element of `pRegions`, `srcOffsets[0].z` and `srcOffsets[1].z` must both be greater than or equal to 0 and less than or equal to the depth of the specified `srcSubresource` of `srcImage`.

If `srcImage` is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `srcOffsets[0].z` must be 0 and `srcOffsets[1].z` must be 1.

For each element of `pRegions`, `dstOffsets[0].x` and `dstOffsets[1].x` must both be greater than or equal to 0 and less than or equal to the width of the specified `dstSubresource` of `dstImage`.

For each element of `pRegions`, `dstOffsets[0].y` and `dstOffsets[1].y` must both be greater than or equal to 0 and less than or equal to the height of the specified `dstSubresource` of `dstImage`.

If `dstImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `dstOffsets[0].y` must be 0 and `dstOffsets[1].y` must be 1.

For each element of `pRegions`, `dstOffsets[0].z` and `dstOffsets[1].z` must both be greater than or equal to 0 and less than or equal to the depth of the specified `dstSubresource` of `dstImage`.

If `dstImage` is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `dstOffsets[0].z` must be 0 and `dstOffsets[1].z` must be 1.

**Valid Usage (Implicit)**

- `sType` must be `VK_STRUCTURE_TYPE_BLIT_IMAGE_INFO_2`
- `pNext` must be `NULL`
• VUID-VkBlitImageInfo2-srcImage-parameter
    srcImage must be a valid VkImage handle

• VUID-VkBlitImageInfo2-srcImageLayout-parameter
    srcImageLayout must be a valid VkImageLayout value

• VUID-VkBlitImageInfo2-dstImage-parameter
    dstImage must be a valid VkImage handle

• VUID-VkBlitImageInfo2-dstImageLayout-parameter
    dstImageLayout must be a valid VkImageLayout value

• VUID-VkBlitImageInfo2-pRegions-parameter
    pRegions must be a valid pointer to an array of regionCount valid VkImageBlit2 structures

• VUID-VkBlitImageInfo2-filter-parameter
    filter must be a valid VkFilter value

• VUID-VkBlitImageInfo2-regionCount-arraylength
    regionCount must be greater than 0

• VUID-VkBlitImageInfo2-commonparent
    Both of dstImage, and srcImage must have been created, allocated, or retrieved from the same VkDevice

The VkImageBlit2 structure is defined as:

```
// Provided by VK_VERSION_1_3
typedef struct VkImageBlit2 {
    VkStructureType sType;
    const void* pNext;
    VkImageSubresourceLayers srcSubresource;
    VkOffset3D srcOffsets[2];
    VkImageSubresourceLayers dstSubresource;
    VkOffset3D dstOffsets[2];
} VkImageBlit2;
```

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• srcSubresource is the subresource to blit from.
• srcOffsets is a pointer to an array of two VkOffset3D structures specifying the bounds of the source region within srcSubresource.
• dstSubresource is the subresource to blit into.
• dstOffsets is a pointer to an array of two VkOffset3D structures specifying the bounds of the destination region within dstSubresource.

For each element of the pRegions array, a blit operation is performed for the specified source and destination regions.
Valid Usage

- VUID-VkImageBlit2-aspectMask-00238
  The aspectMask member of srcSubresource and dstSubresource must match

- VUID-VkImageBlit2-layerCount-00239
  The layerCount member of srcSubresource and dstSubresource must match

Valid Usage (Implicit)

- VUID-VkImageBlit2-sType-sType
  sType must be VK_STRUCTURE_TYPE_IMAGE_BLIT_2

- VUID-VkImageBlit2-pNext-pNext
  pNext must be NULL

- VUID-VkImageBlit2-srcSubresource-parameter
  srcSubresource must be a valid VkImageSubresourceLayers structure

- VUID-VkImageBlit2-dstSubresource-parameter
  dstSubresource must be a valid VkImageSubresourceLayers structure

19.5. Resolving Multisample Images

To resolve a multisample color image to a non-multisample color image, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdResolveImage(
  VkCommandBuffer commandBuffer,
  VkImage srcImage,
  VkImageLayout srcImageLayout,
  VkImage dstImage,
  VkImageLayout dstImageLayout,
  uint32_t regionCount,
  const VkImageResolve* pRegions);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `srcImage` is the source image.
- `srcImageLayout` is the layout of the source image subresources for the resolve.
- `dstImage` is the destination image.
- `dstImageLayout` is the layout of the destination image subresources for the resolve.
- `regionCount` is the number of regions to resolve.
- `pRegions` is a pointer to an array of VkImageResolve structures specifying the regions to resolve.

During the resolve the samples corresponding to each pixel location in the source are converted to
a single sample before being written to the destination. If the source formats are floating-point or normalized types, the sample values for each pixel are resolved in an implementation-dependent manner. If the source formats are integer types, a single sample’s value is selected for each pixel.

`srcOffset` and `dstOffset` select the initial x, y, and z offsets in texels of the sub-regions of the source and destination image data. `extent` is the size in texels of the source image to resolve in `width`, `height` and `depth`. Each element of `pRegions` must be a region that is contained within its corresponding image.

Resolves are done layer by layer starting with `baseArrayLayer` member of `srcSubresource` for the source and `dstSubresource` for the destination. `layerCount` layers are resolved to the destination image.

### Valid Usage

- **VUID-vkCmdResolveImage-commandBuffer-01837**
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `srcImage` must not be a protected image

- **VUID-vkCmdResolveImage-commandBuffer-01838**
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `dstImage` must not be a protected image

- **VUID-vkCmdResolveImage-commandBuffer-01839**
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `dstImage` must not be an unprotected image

- **VUID-vkCmdResolveImage-pRegions-00255**
  The union of all source regions, and the union of all destination regions, specified by the elements of `pRegions`, must not overlap in memory

- **VUID-vkCmdResolveImage-srcImage-00256**
  If `srcImage` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-vkCmdResolveImage-srcImage-00257**
  `srcImage` must have a sample count equal to any valid sample count value other than `VK_SAMPLE_COUNT_1_BIT`

- **VUID-vkCmdResolveImage-dstImage-00258**
  If `dstImage` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-vkCmdResolveImage-dstImage-00259**
  `dstImage` must have a sample count equal to `VK_SAMPLE_COUNT_1_BIT`

- **VUID-vkCmdResolveImage-srcImageLayout-00260**
  `srcImageLayout` must specify the layout of the image subresources of `srcImage` specified in `pRegions` at the time this command is executed on a `VkDevice`

- **VUID-vkCmdResolveImage-srcImageLayout-00261**
  `srcImageLayout` must be `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`

- **VUID-vkCmdResolveImage-dstImageLayout-00262**
  `srcImageLayout` must be `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`.

- VUID-vkCmdResolveImage-dstImageLayout-00263
  `dstImageLayout` must be `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`.

- VUID-vkCmdResolveImage-dstImage-02003
  The format features of `dstImage` must contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`.

- VUID-vkCmdResolveImage-srcImage-01386
  `srcImage` and `dstImage` must have been created with the same image format.

- VUID-vkCmdResolveImage-srcSubresource-01709
  The `srcSubresource.mipLevel` member of each element of `pRegions` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `srcImage` was created.

- VUID-vkCmdResolveImage-dstSubresource-01710
  The `dstSubresource.mipLevel` member of each element of `pRegions` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `dstImage` was created.

- VUID-vkCmdResolveImage-srcSubresource-01711
  The `srcSubresource.baseArrayLayer + srcSubresource.layerCount` of each element of `pRegions` must be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `srcImage` was created.

- VUID-vkCmdResolveImage-dstSubresource-01712
  The `dstSubresource.baseArrayLayer + dstSubresource.layerCount` of each element of `pRegions` must be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `dstImage` was created.

- VUID-vkCmdResolveImage-srcImage-04446
  If either `srcImage` or `dstImage` are of type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions`, `srcSubresource.baseArrayLayer` must be 0 and `srcSubresource.layerCount` must be 1.

- VUID-vkCmdResolveImage-srcImage-04447
  If either `srcImage` or `dstImage` are of type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions`, `dstSubresource.baseArrayLayer` must be 0 and `dstSubresource.layerCount` must be 1.

- VUID-vkCmdResolveImage-srcOffset-00269
  For each element of `pRegions`, `srcOffset.x` and `(extent.width + srcOffset.x)` must both be greater than or equal to 0 and less than or equal to the width of the specified `srcSubresource` of `srcImage`.

- VUID-vkCmdResolveImage-srcOffset-00270
  For each element of `pRegions`, `srcOffset.y` and `(extent.height + srcOffset.y)` must both be greater than or equal to 0 and less than or equal to the height of the specified `srcSubresource` of `srcImage`.

- VUID-vkCmdResolveImage-srcOffset-00271
  If `srcImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `srcOffset.y` must be 0 and `extent.height` must be 1.

- VUID-vkCmdResolveImage-srcOffset-00272
  For each element of `pRegions`, `srcOffset.z` and `(extent.depth + srcOffset.z)` must both be
greater than or equal to $0$ and less than or equal to the depth of the specified
\( srcSubresource \) of \( srcImage \)

- VUID-vkCmdResolveImage-srcImage-00273
  If \( srcImage \) is of type \( VK_IMAGE_TYPE_1D \) or \( VK_IMAGE_TYPE_2D \), then for each element of \( pRegions \), \( srcOffset.z \) must be $0$ and \( extent.depth \) must be $1$

- VUID-vkCmdResolveImage-dstOffset-00274
  For each element of \( pRegions \), \( dstOffset.x \) and \( (extent.width + dstOffset.x) \) must both be greater than or equal to $0$ and less than or equal to the width of the specified \( dstSubresource \) of \( dstImage \)

- VUID-vkCmdResolveImage-dstOffset-00275
  For each element of \( pRegions \), \( dstOffset.y \) and \( (extent.height + dstOffset.y) \) must both be greater than or equal to $0$ and less than or equal to the height of the specified \( dstSubresource \) of \( dstImage \)

- VUID-vkCmdResolveImage-dstImage-00276
  If \( dstImage \) is of type \( VK_IMAGE_TYPE_1D \), then for each element of \( pRegions \), \( dstOffset.y \) must be $0$ and \( extent.height \) must be $1$

- VUID-vkCmdResolveImage-dstOffset-00277
  For each element of \( pRegions \), \( dstOffset.z \) and \( (extent.depth + dstOffset.z) \) must both be greater than or equal to $0$ and less than or equal to the depth of the specified \( dstSubresource \) of \( dstImage \)

- VUID-vkCmdResolveImage-dstImage-00278
  If \( dstImage \) is of type \( VK_IMAGE_TYPE_1D \) or \( VK_IMAGE_TYPE_2D \), then for each element of \( pRegions \), \( dstOffset.z \) must be $0$ and \( extent.depth \) must be $1$

- VUID-vkCmdResolveImage-srcImage-06762
  \( srcImage \) must have been created with \( VK_IMAGE_USAGE_TRANSFER_SRC_BIT \) usage flag

- VUID-vkCmdResolveImage-srcImage-06763
  The format features of \( srcImage \) must contain \( VK_FORMAT_FEATURE_TRANSFER_SRC_BIT \)

- VUID-vkCmdResolveImage-dstImage-06764
  \( dstImage \) must have been created with \( VK_IMAGE_USAGE_TRANSFER_DST_BIT \) usage flag

- VUID-vkCmdResolveImage-dstImage-06765
  The format features of \( dstImage \) must contain \( VK_FORMAT_FEATURE_TRANSFER_DST_BIT \)

Valid Usage (Implicit)

- VUID-vkCmdResolveImage-commandBuffer-parameter
  \( commandBuffer \) must be a valid \( VkCommandBuffer \) handle

- VUID-vkCmdResolveImage-srcImage-parameter
  \( srcImage \) must be a valid \( VkImage \) handle

- VUID-vkCmdResolveImage-srcImageLayout-parameter
  \( srcImageLayout \) must be a valid \( VkImageLayout \) value

- VUID-vkCmdResolveImage-dstImage-parameter
  \( dstImage \) must be a valid \( VkImage \) handle
- **VUID-vkCmdResolveImage-dstImageLayout-parameter**
  `dstImageLayout` must be a valid `VkImageLayout` value.

- **VUID-vkCmdResolveImage-pRegions-parameter**
  `pRegions` must be a valid pointer to an array of `regionCount` valid `VkImageResolve` structures.

- **VUID-vkCmdResolveImage-commandBuffer-recording**
  `commandBuffer` must be in the `recording` state.

- **VUID-vkCmdResolveImage-commandBuffer-cmdpool**
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations.

- **VUID-vkCmdResolveImage-renderpass**
  This command must only be called outside of a render pass instance.

- **VUID-vkCmdResolveImage-regionCount-arraylength**
  `regionCount` must be greater than 0.

- **VUID-vkCmdResolveImage-commonparent**
  Each of `commandBuffer`, `dstImage`, and `srcImage` must have been created, allocated, or retrieved from the same `VkDevice`.

### Host Synchronization

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkImageResolve` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkImageResolve {
    VkImageSubresourceLayers srcSubresource;
    VkOffset3D srcOffset;
    VkImageSubresourceLayers dstSubresource;
    VkOffset3D dstOffset;
    VkExtent3D extent;
} VkImageResolve;
```
- `srcSubresource` and `dstSubresource` are `VkImageSubresourceLayers` structures specifying the image subresources of the images used for the source and destination image data, respectively. Resolve of depth/stencil images is not supported.

- `srcOffset` and `dstOffset` select the initial x, y, and z offsets in texels of the sub-regions of the source and destination image data.

- `extent` is the size in texels of the source image to resolve in `width`, `height` and `depth`.

### Valid Usage

- **VUID-VkImageResolve-aspectMask-00266**  
  The `aspectMask` member of `srcSubresource` and `dstSubresource` must only contain `VK_IMAGE_ASPECT_COLOR_BIT`

- **VUID-VkImageResolve-layerCount-00267**  
  The `layerCount` member of `srcSubresource` and `dstSubresource` must match

### Valid Usage (Implicit)

- **VUID-VkImageResolve-srcSubresource-parameter**  
  `srcSubresource` must be a valid `VkImageSubresourceLayers` structure

- **VUID-VkImageResolve-dstSubresource-parameter**  
  `dstSubresource` must be a valid `VkImageSubresourceLayers` structure

A more extensible version of the resolve image command is defined below.

To resolve a multisample image to a non-multisample image, call:

```c
// Provided by VK_VERSION_1_3
void vkCmdResolveImage2(
    VkCommandBuffer commandBuffer,
    const VkResolveImageInfo2* pResolveImageInfo);
```

- `commandBuffer` is the command buffer into which the command will be recorded.

- `pResolveImageInfo` is a pointer to a `VkResolveImageInfo2` structure describing the resolve parameters.

This command is functionally identical to `vkCmdResolveImage`, but includes extensible sub-structures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

### Valid Usage

- **VUID-vkCmdResolveImage2-commandBuffer-01837**  
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, 
  `srcImage` must not be a protected image
• VUID-vkCmdResolveImage2-commandBuffer-01838
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `dstImage` must not be a protected image

• VUID-vkCmdResolveImage2-commandBuffer-01839
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `dstImage` must not be an unprotected image

Valid Usage (Implicit)

• VUID-vkCmdResolveImage2-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

• VUID-vkCmdResolveImage2-pResolveImageInfo-parameter
  `pResolveImageInfo` must be a valid pointer to a valid `VkResolveImageInfo2` structure

• VUID-vkCmdResolveImage2-commandBuffer-recording
  `commandBuffer` must be in the recording state

• VUID-vkCmdResolveImage2-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

• VUID-vkCmdResolveImage2-renderpass
  This command must only be called outside of a render pass instance

Host Synchronization

• Host access to `commandBuffer` must be externally synchronized

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkResolveImageInfo2` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkResolveImageInfo2 {
    VkStructureType sType;
    const void* pNext;
    VkImage srcImage;
} VkResolveImageInfo2;
```
VkImageLayout srcImageLayout;
VkImage dstImage;
VkImageLayout dstImageLayout;
uint32_t regionCount;
const VkImageResolve2* pRegions;
}
VkResolveImageInfo2;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **srcImage** is the source image.
- **srcImageLayout** is the layout of the source image subresources for the resolve.
- **dstImage** is the destination image.
- **dstImageLayout** is the layout of the destination image subresources for the resolve.
- **regionCount** is the number of regions to resolve.
- **pRegions** is a pointer to an array of **VkImageResolve2** structures specifying the regions to resolve.

**Valid Usage**

- **VUID-VkResolveImageInfo2-pRegions-00255**
  The union of all source regions, and the union of all destination regions, specified by the elements of **pRegions**, **must** not overlap in memory

- **VUID-VkResolveImageInfo2-srcImage-00256**
  If **srcImage** is non-sparse then it **must** be bound completely and contiguously to a single **VkDeviceMemory** object

- **VUID-VkResolveImageInfo2-srcImage-00257**
  **srcImage** must have a sample count equal to any valid sample count value other than **VK_SAMPLE_COUNT_1_BIT**

- **VUID-VkResolveImageInfo2-dstImage-00258**
  If **dstImage** is non-sparse then it **must** be bound completely and contiguously to a single **VkDeviceMemory** object

- **VUID-VkResolveImageInfo2-dstImage-00259**
  **dstImage** must have a sample count equal to **VK_SAMPLE_COUNT_1_BIT**

- **VUID-VkResolveImageInfo2-srcImageLayout-00260**
  **srcImageLayout** must specify the layout of the image subresources of **srcImage** specified in **pRegions** at the time this command is executed on a **VkDevice**

- **VUID-VkResolveImageInfo2-srcImageLayout-00261**
  **srcImageLayout** must be **VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL** or **VK_IMAGE_LAYOUT_GENERAL**

- **VUID-VkResolveImageInfo2-dstImageLayout-00262**
  **dstImageLayout** **must** specify the layout of the image subresources of **dstImage** specified in **pRegions** at the time this command is executed on a **VkDevice**
- **VUID-VkResolveImageInfo2-dstImageLayout-00263**
  
  `dstImageLayout` must be `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`.

- **VUID-VkResolveImageInfo2-dstImage-02003**
  
  The format features of `dstImage` must contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`.

- **VUID-VkResolveImageInfo2-srcImage-01386**
  
  `srcImage` and `dstImage` must have been created with the same image format.

- **VUID-VkResolveImageInfo2-srcSubresource-01709**
  
  The `srcSubresource.mipLevel` member of each element of `pRegions` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `srcImage` was created.

- **VUID-VkResolveImageInfo2-dstSubresource-01710**
  
  The `dstSubresource.mipLevel` member of each element of `pRegions` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `dstImage` was created.

- **VUID-VkResolveImageInfo2-srcSubresource-01711**
  
  The `srcSubresource.baseArrayLayer` + `srcSubresource.layerCount` of each element of `pRegions` must be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `srcImage` was created.

- **VUID-VkResolveImageInfo2-dstSubresource-01712**
  
  The `dstSubresource.baseArrayLayer` + `dstSubresource.layerCount` of each element of `pRegions` must be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `dstImage` was created.

- **VUID-VkResolveImageInfo2-srcImage-04446**
  
  If either `srcImage` or `dstImage` are of type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions`, `srcSubresource.baseArrayLayer` must be 0 and `srcSubresource.layerCount` must be 1.

- **VUID-VkResolveImageInfo2-srcImage-04447**
  
  If either `srcImage` or `dstImage` are of type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions`, `dstSubresource.baseArrayLayer` must be 0 and `dstSubresource.layerCount` must be 1.

- **VUID-VkResolveImageInfo2-srcOffset-00269**
  
  For each element of `pRegions`, `srcOffset.x` and `(extent.width + srcOffset.x)` must both be greater than or equal to 0 and less than or equal to the width of the specified `srcSubresource` of `srcImage`.

- **VUID-VkResolveImageInfo2-srcOffset-00270**
  
  For each element of `pRegions`, `srcOffset.y` and `(extent.height + srcOffset.y)` must both be greater than or equal to 0 and less than or equal to the height of the specified `srcSubresource` of `srcImage`.

- **VUID-VkResolveImageInfo2-srcOffset-00271**
  
  If `srcImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `srcOffset.y` must be 0 and `extent.height` must be 1.

- **VUID-VkResolveImageInfo2-srcOffset-00272**
  
  For each element of `pRegions`, `srcOffset.z` and `(extent.depth + srcOffset.z)` must both be greater than or equal to 0 and less than or equal to the depth of the specified `srcSubresource` of `srcImage`. 
If srcImage is of type VK_IMAGE_TYPE_1D or VK_IMAGE_TYPE_2D, then for each element of pRegions, srcOffset.z must be 0 and extent.depth must be 1.

For each element of pRegions, dstOffset.x and (extent.width + dstOffset.x) must both be greater than or equal to 0 and less than or equal to the width of the specified dstSubresource of dstImage.

For each element of pRegions, dstOffset.y and (extent.height + dstOffset.y) must both be greater than or equal to 0 and less than or equal to the height of the specified dstSubresource of dstImage.

If dstImage is of type VK_IMAGE_TYPE_1D, then for each element of pRegions, dstOffset.y must be 0 and extent.height must be 1.

For each element of pRegions, dstOffset.z and (extent.depth + dstOffset.z) must both be greater than or equal to 0 and less than or equal to the depth of the specified dstSubresource of dstImage.

If dstImage is of type VK_IMAGE_TYPE_1D or VK_IMAGE_TYPE_2D, then for each element of pRegions, dstOffset.z must be 0 and extent.depth must be 1.

srcImage must have been created with VK_IMAGE_USAGE_TRANSFER_SRC_BIT usage flag.

The format features of srcImage must contain VK_FORMAT_FEATURE_TRANSFER_SRC_BIT.

dstImage must have been created with VK_IMAGE_USAGE_TRANSFER_DST_BIT usage flag.

The format features of dstImage must contain VK_FORMAT_FEATURE_TRANSFER_DST_BIT.

Valid Usage (Implicit)

sType must be VK_STRUCTURE_TYPE_RESOLVE_IMAGE_INFO_2.

pNext must be NULL.

srcImage must be a valid VkImage handle.

srcImageLayout must be a valid VkImageLayout value.

dstImage must be a valid VkImage handle.
The `VkImageResolve2` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkImageResolve2 {
    VkStructureType sType;
    const void* pNext;
    VkImageSubresourceLayers srcSubresource;
    VkOffset3D srcOffset;
    VkImageSubresourceLayers dstSubresource;
    VkOffset3D dstOffset;
    VkExtent3D extent;
} VkImageResolve2;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcSubresource` and `dstSubresource` are `VkImageSubresourceLayers` structures specifying the image subresources of the images used for the source and destination image data, respectively. Resolve of depth/stencil images is not supported.
- `srcOffset` and `dstOffset` select the initial \( x, y, \) and \( z \) offsets in texels of the sub-regions of the source and destination image data.
- `extent` is the size in texels of the source image to resolve in `width`, `height` and `depth`.

### Valid Usage

- **VUID-VkImageResolve2-aspectMask-00266**
  The `aspectMask` member of `srcSubresource` and `dstSubresource` must only contain `VK_IMAGE_ASPECT_COLOR_BIT`

- **VUID-VkImageResolve2-layerCount-00267**
  The `layerCount` member of `srcSubresource` and `dstSubresource` must match
Valid Usage (Implicit)

- VUID-VkImageResolve2-sType-sType
  
  `sType` **must** be `VK_STRUCTURE_TYPE_IMAGE_RESOLVE_2`

- VUID-VkImageResolve2-pNext-pNext
  
  `pNext` **must** be `NULL`

- VUID-VkImageResolve2-srcSubresource-parameter
  
  `srcSubresource` **must** be a valid `VkImageSubresourceLayers` structure

- VUID-VkImageResolve2-dstSubresource-parameter
  
  `dstSubresource` **must** be a valid `VkImageSubresourceLayers` structure
Chapter 20. Drawing Commands

Drawing commands (commands with Draw in the name) provoke work in a graphics pipeline. Drawing commands are recorded into a command buffer and when executed by a queue, will produce work which executes according to the bound graphics pipeline. A graphics pipeline must be bound to a command buffer before any drawing commands are recorded in that command buffer.

Each draw is made up of zero or more vertices and zero or more instances, which are processed by the device and result in the assembly of primitives. Primitives are assembled according to the pInputAssemblyState member of the VkGraphicsPipelineCreateInfo structure, which is of type VkPipelineInputAssemblyStateCreateInfo:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineInputAssemblyStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineInputAssemblyStateCreateFlags flags;
    VkPrimitiveTopology topology;
    VkBool32 primitiveRestartEnable;
} VkPipelineInputAssemblyStateCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `topology` is a VkPrimitiveTopology defining the primitive topology, as described below.
- `primitiveRestartEnable` controls whether a special vertex index value is treated as restarting the assembly of primitives. This enable only applies to indexed draws (vkCmdDrawIndexed and vkCmdDrawIndexedIndirect), and the special index value is either 0xFFFFFFFF when the `indexType` parameter of vkCmdBindIndexBuffer is equal to VK_INDEX_TYPE_UINT32, or 0xFFFF when `indexType` is equal to VK_INDEX_TYPE_UINT16. Primitive restart is not allowed for “list” topologies.

Restarting the assembly of primitives discards the most recent index values if those elements formed an incomplete primitive, and restarts the primitive assembly using the subsequent indices, but only assembling the immediately following element through the end of the originally specified elements. The primitive restart index value comparison is performed before adding the `vertexOffset` value to the index value.

Valid Usage

- VUID-VkPipelineInputAssemblyStateCreateInfo-topology-00428

  If `topology` is VK_PRIMITIVE_TOPOLOGY_POINT_LIST, VK_PRIMITIVE_TOPOLOGY_LINE_LIST, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY or VK_PRIMITIVE_TOPOLOGY_PATCH_LIST, `primitiveRestartEnable` must be VK_FALSE
If the geometryShader feature is not enabled, topology must not be any of
VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY,
VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY,
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY or
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY

If the tessellationShader feature is not enabled, topology must not be
VK_PRIMITIVE_TOPOLOGY_PATCH_LIST

Valid Usage (Implicit)

- VkPipelineInputAssemblyStateCreateInfo::sType sType must be VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO
- VkPipelineInputAssemblyStateCreateInfo::pNext pNext must be NULL
- VkPipelineInputAssemblyStateCreateInfo::flags flags must be 0
- VkPipelineInputAssemblyStateCreateInfo::topology topology must be a valid VkPrimitiveTopology value

// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineInputAssemblyStateCreateFlags;

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

To dynamically control whether a special vertex index value is treated as restarting the assembly of primitives, call:

// Provided by VK_VERSION_1_3
void vkCmdSetPrimitiveRestartEnable(
    VkCommandBuffer commandBuffer,
    VkBool32 primitiveRestartEnable);

- commandBuffer is the command buffer into which the command will be recorded.
- primitiveRestartEnable controls whether a special vertex index value is treated as restarting the assembly of primitives. It behaves in the same way as VkPipelineInputAssemblyStateCreateInfo::primitiveRestartEnable

This command sets the primitive restart enable for subsequent drawing commands when the graphics pipeline is created with VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE set in VkPipelineDynamicStateCreateInfo::pDynamicStates. Otherwise, this state is specified by the
VkPipelineInputAssemblyStateCreateInfo::primitiveRestartEnable value used to create the currently active pipeline.

Valid Usage (Implicit)

- VUID-vkCmdSetPrimitiveRestartEnable-commandBuffer-parameter commandBuffer must be a valid VkCommandBuffer handle
- VUID-vkCmdSetPrimitiveRestartEnable-commandBuffer-recording commandBuffer must be in the recording state
- VUID-vkCmdSetPrimitiveRestartEnable-commandBuffer-cmdpool The VkCommandPool that commandBuffer was allocated from must support graphics operations

Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

20.1. Primitive Topologies

Primitive topology determines how consecutive vertices are organized into primitives, and determines the type of primitive that is used at the beginning of the graphics pipeline. The effective topology for later stages of the pipeline is altered by tessellation or geometry shading (if either is in use) and depends on the execution modes of those shaders.

The primitive topologies defined by VkPrimitiveTopology are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkPrimitiveTopology {
    VK_PRIMITIVE_TOPOLOGY_POINT_LIST = 0,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST = 1,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP = 2,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST = 3,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP = 4,
};
```
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN = 5,
VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY = 6,
VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY = 7,
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY = 8,
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY = 9,
VK_PRIMITIVE_TOPOLOGY_PATCH_LIST = 10,
} VkPrimitiveTopology;

- **VK_PRIMITIVE_TOPOLOGY_POINT_LIST** specifies a series of separate point primitives.
- **VK_PRIMITIVE_TOPOLOGY_LINE_LIST** specifies a series of separate line primitives.
- **VK_PRIMITIVE_TOPOLOGY_LINE_STRIP** specifies a series of connected line primitives with consecutive lines sharing a vertex.
- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST** specifies a series of separate triangle primitives.
- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP** specifies a series of connected triangle primitives with consecutive triangles sharing an edge.
- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN** specifies a series of connected triangle primitives with all triangles sharing a common vertex.
- **VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY** specifies a series of separate line primitives with adjacency.
- **VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY** specifies a series of connected line primitives with adjacency, with consecutive primitives sharing three vertices.
- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY** specifies a series of separate triangle primitives with adjacency.
- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY** specifies connected triangle primitives with adjacency, with consecutive triangles sharing an edge.
- **VK_PRIMITIVE_TOPOLOGY_PATCH_LIST** specifies separate patch primitives.

Each primitive topology, and its construction from a list of vertices, is described in detail below with a supporting diagram, according to the following key:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>●</td>
<td>Vertex</td>
</tr>
<tr>
<td>5</td>
<td>Vertex Number</td>
</tr>
<tr>
<td>✔️</td>
<td>Provoking Vertex</td>
</tr>
<tr>
<td>————</td>
<td>Primitive Edge</td>
</tr>
<tr>
<td>————</td>
<td>Adjacency Edge</td>
</tr>
<tr>
<td>△</td>
<td>Winding Order</td>
</tr>
</tbody>
</table>
The diagrams are supported with mathematical definitions where the vertices \( v \) and primitives \( p \) are numbered starting from 0; \( v_0 \) is the first vertex in the provided data and \( p_0 \) is the first primitive in the set of primitives defined by the vertices and topology.

To dynamically set primitive topology, call:

```c
void vkCmdSetPrimitiveTopology(
    VkCommandBuffer commandBuffer,
    VkPrimitiveTopology primitiveTopology);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `primitiveTopology` specifies the primitive topology to use for drawing.

This command sets the primitive topology for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineInputAssemblyStateCreateInfo::topology` value used to create the currently active pipeline.

### Valid Usage (Implicit)

- VUID-vkCmdSetPrimitiveTopology-commandBuffer-parameter
  - `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdSetPrimitiveTopology-primitiveTopology-parameter
  - `primitiveTopology` must be a valid `VkPrimitiveTopology` value
- VUID-vkCmdSetPrimitiveTopology-commandBuffer-recording
  - `commandBuffer` must be in the recording state
- VUID-vkCmdSetPrimitiveTopology-commandBuffer-cmdpool
  - The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

### Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized
20.1.1. Topology Class

The primitive topologies are grouped into the following topology classes:

Table 20. Topology classes

<table>
<thead>
<tr>
<th>Topology Class</th>
<th>Primitive Topology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point</td>
<td>VK_PRIMITIVE_TOPOLOGY_POINT_LIST</td>
</tr>
<tr>
<td>Line</td>
<td>VK_PRIMITIVE_TOPOLOGY_LINE_LIST, VK_PRIMITIVE_TOPOLOGY_LINE_STRIP, VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY, VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY</td>
</tr>
<tr>
<td>Triangle</td>
<td>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY</td>
</tr>
<tr>
<td>Patch</td>
<td>VK_PRIMITIVE_TOPOLOGY_PATCH_LIST</td>
</tr>
</tbody>
</table>

20.1.2. Point Lists

When the topology is VK_PRIMITIVE_TOPOLOGY_POINT_LIST, each consecutive vertex defines a single point primitive, according to the equation:

\[ p_i = \{v_i\} \]

As there is only one vertex, that vertex is the provoking vertex. The number of primitives generated is equal to vertexCount.
20.1.3. Line Lists

When the primitive topology is `VK_PRIMITIVE_TOPOLOGY_LINE_LIST`, each consecutive pair of vertices defines a single line primitive, according to the equation:

\[ p_i = \{v_{2i}, v_{2i+1}\} \]

The number of primitives generated is equal to \( \lfloor \text{vertexCount}/2 \rfloor \).

The provoking vertex for \( p_i \) is \( v_{2i} \).

20.1.4. Line Strips

When the primitive topology is `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP`, one line primitive is defined by each vertex and the following vertex, according to the equation:

\[ p_i = \{v_i, v_{i+1}\} \]

The number of primitives generated is equal to \( \max(0, \text{vertexCount}-1) \).

The provoking vertex for \( p_i \) is \( v_i \).

20.1.5. Triangle Lists

When the primitive topology is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`, each consecutive set of three vertices defines a single triangle primitive, according to the equation:
\( p_i = \{v_{3i}, v_{3i+1}, v_{3i+2}\} \)

The number of primitives generated is equal to \( \lfloor \text{vertexCount}/3 \rfloor \).

The provoking vertex for \( p_i \) is \( v_{3i} \).

20.1.6. Triangle Strips

When the primitive topology is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP`, one triangle primitive is defined by each vertex and the two vertices that follow it, according to the equation:

\[
p_i = \{v_i, v_{i+(1+i\%2)}, v_{i+(2-i\%2)}\}
\]

The number of primitives generated is equal to \( \max(0, \text{vertexCount}-2) \).

The provoking vertex for \( p_i \) is \( v_i \).

\[\begin{array}{c}
0 & 1 & 2 \\
0 & 2 & 4 \\
0 & 2 & 5 \\
\end{array}\]

\[\begin{array}{c}
0 & 1 & 3 \\
0 & 2 & 3 \\
0 & 3 & 4 \\
\end{array}\]

\[\begin{array}{c}
0 & 1 & 3 \\
0 & 2 & 4 \\
0 & 2 & 5 \\
\end{array}\]

\[\begin{array}{c}
0 & 1 & 3 \\
0 & 2 & 3 \\
0 & 2 & 5 \\
\end{array}\]

**Note**

The ordering of the vertices in each successive triangle is reversed, so that the winding order is consistent throughout the strip.

20.1.7. Triangle Fans

When the primitive topology is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN`, triangle primitives are defined around a shared common vertex, according to the equation:

\( p_i = \{v_{i+1}, v_{i+2}, v_0\} \)

The number of primitives generated is equal to \( \max(0, \text{vertexCount}-2) \).

The provoking vertex for \( p_i \) is \( v_{i+1} \).
20.1.8. Line Lists With Adjacency

When the primitive topology is \texttt{VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY}, each consecutive set of four vertices defines a single line primitive with adjacency, according to the equation:

\[ p_i = \{v_{4i}, v_{4i+1}, v_{4i+2}, v_{4i+3}\} \]

A line primitive is described by the second and third vertices of the total primitive, with the remaining two vertices only accessible in a \texttt{geometry shader}.

The number of primitives generated is equal to \(\lfloor \text{vertexCount}/4 \rfloor\).

The provoking vertex for \(p_i\) is \(v_{4i+1}\).

\[ \begin{array}{cccc}
0 & 1 & 2 & 3 \\
\end{array} \]

20.1.9. Line Strips With Adjacency

When the primitive topology is \texttt{VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY}, one line primitive with adjacency is defined by each vertex and the following vertex, according to the equation:

\[ p_i = \{v_i, v_{i+1}, v_{i+2}, v_{i+3}\} \]

A line primitive is described by the second and third vertices of the total primitive, with the remaining two vertices only accessible in a \texttt{geometry shader}.

The number of primitives generated is equal to \(\max(0, \text{vertexCount}-3)\).

The provoking vertex for \(p_i\) is \(v_{i+1}\).

\[ \begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
\end{array} \]
20.1.10. Triangle Lists With Adjacency

When the primitive topology is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY`, each consecutive set of six vertices defines a single triangle primitive with adjacency, according to the equations:

\[ p_i = \{v_{6i}, v_{6i+1}, v_{6i+2}, v_{6i+3}, v_{6i+4}, v_{6i+5}\} \]

A triangle primitive is described by the first, third, and fifth vertices of the total primitive, with the remaining three vertices only accessible in a geometry shader.

The number of primitives generated is equal to \( \lfloor \text{vertexCount}/6 \rfloor \).

The provoking vertex for \( p_i \) is \( v_{6i} \).

20.1.11. Triangle Strips With Adjacency

When the primitive topology is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY`, one triangle primitive with adjacency is defined by each vertex and the following 5 vertices.

The number of primitives generated, \( n \), is equal to \( \max(0, \text{vertexCount} - 4)/2 \).

If \( n=1 \), the primitive is defined as:

\[ p = \{v_0, v_1, v_2, v_5, v_4, v_3\} \]

If \( n>1 \), the total primitive consists of different vertices according to where it is in the strip:

\[ p_i = \{v_{2i}, v_{2i+1}, v_{2i+2}, v_{2i+3}, v_{2i+4}\} \text{ when } i=0 \]

\[ p_i = \{v_{2i}, v_{2i+3}, v_{2i+4}, v_{2i+6}, v_{2i+2}, v_{2i+1}\} \text{ when } i>0, i<n-1, \text{ and } i \% 2=1 \]

\[ p_i = \{v_{2i}, v_{2i+2}, v_{2i+3}, v_{2i+6}, v_{2i+4}, v_{2i+1}\} \text{ when } i>0, i<n-1, \text{ and } i \% 2=0 \]
\[ p_i = \{v_{2i}, v_{2i+3}, v_{2i+5}, v_{2i+2}, v_{2i+1}\} \text{ when } i=n-1 \text{ and } i\%2=1 \]

\[ p_i = \{v_{2i}, v_{2i+2}, v_{2i+5}, v_{2i+4}, v_{2i+3}\} \text{ when } i=n-1 \text{ and } i\%2=0 \]

A triangle primitive is described by the first, third, and fifth vertices of the total primitive in all cases, with the remaining three vertices only accessible in a geometry shader.

**Note**

The ordering of the vertices in each successive triangle is altered so that the winding order is consistent throughout the strip.

The provoking vertex for \( p_i \) is always \( v_{2i} \).

### 20.1.12. Patch Lists

When the primitive topology is `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`, each consecutive set of \( m \) vertices defines a single patch primitive, according to the equation:

\[ p_i = \{v_{mi}, v_{mi+1}, \ldots, v_{mi+(m-2)}, v_{mi+(m-1)}\} \]
where \( m \) is equal to `VkPipelineTessellationStateCreateInfo::patchControlPoints`.

Patch lists are never passed to vertex post-processing, and as such no provoking vertex is defined for patch primitives. The number of primitives generated is equal to \( \lfloor \text{vertexCount}/m \rfloor \).

The vertices comprising a patch have no implied geometry, and are used as inputs to tessellation shaders and the fixed-function tessellator to generate new point, line, or triangle primitives.

### 20.2. Primitive Order

Primitives generated by drawing commands progress through the stages of the graphics pipeline in primitive order. Primitive order is initially determined in the following way:

1. Submission order determines the initial ordering
2. For indirect drawing commands, the order in which accessed instances of the `VkDrawIndirectCommand` are stored in `buffer`, from lower indirect buffer addresses to higher addresses.
3. If a drawing command includes multiple instances, the order in which instances are executed, from lower numbered instances to higher.
4. The order in which primitives are specified by a drawing command:
   - For non-indexed draws, from vertices with a lower numbered `vertexIndex` to a higher numbered `vertexIndex`.
   - For indexed draws, vertices sourced from a lower index buffer addresses to higher addresses.

Within this order implementations further sort primitives:

5. If tessellation shading is active, by an implementation-dependent order of new primitives generated by tessellation.
6. If geometry shading is active, by the order new primitives are generated by geometry shading.
7. If the polygon mode is not `VK_POLYGON_MODE_FILL`, by an implementation-dependent ordering of the new primitives generated within the original primitive.

Primitive order is later used to define rasterization order, which determines the order in which fragments output results to a framebuffer.

### 20.3. Programmable Primitive Shading

Once primitives are assembled, they proceed to the vertex shading stage of the pipeline. If the draw includes multiple instances, then the set of primitives is sent to the vertex shading stage multiple times, once for each instance.

It is implementation-dependent whether vertex shading occurs on vertices that are discarded as part of incomplete primitives, but if it does occur then it operates as if they were vertices in complete primitives and such invocations can have side effects.
Vertex shading receives two per-vertex inputs from the primitive assembly stage - the vertexIndex and the instanceIndex. How these values are generated is defined below, with each command.

Drawing commands fall roughly into two categories:

- Non-indexed drawing commands present a sequential vertexIndex to the vertex shader. The sequential index is generated automatically by the device (see Fixed-Function Vertex Processing for details on both specifying the vertex attributes indexed by vertexIndex, as well as binding vertex buffers containing those attributes to a command buffer). These commands are:
  - vkCmdDraw
  - vkCmdDrawIndirect
  - vkCmdDrawIndirectCount

- Indexed drawing commands read index values from an index buffer and use this to compute the vertexIndex value for the vertex shader. These commands are:
  - vkCmdDrawIndexed
  - vkCmdDrawIndexedIndirect
  - vkCmdDrawIndexedIndirectCount

To bind an index buffer to a command buffer, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdBindIndexBuffer(
    VkCommandBuffer commandBuffer,
    VkBuffer buffer,
    VkDeviceSize offset,
    VkIndexType indexType);
```

- commandBuffer is the command buffer into which the command is recorded.
- buffer is the buffer being bound.
- offset is the starting offset in bytes within buffer used in index buffer address calculations.
- indexType is a VkIndexType value specifying the size of the indices.

**Valid Usage**

- VUID-vkCmdBindIndexBuffer-offset-00431
  - offset must be less than the size of buffer

- VUID-vkCmdBindIndexBuffer-offset-00432
  - The sum of offset and the address of the range of VkDeviceMemory object that is backing buffer, must be a multiple of the type indicated by indexType

- VUID-vkCmdBindIndexBuffer-buffer-00433
  - buffer must have been created with the VK_BUFFER_USAGE_INDEX_BUFFER_BIT flag

- VUID-vkCmdBindIndexBuffer-buffer-00434
If buffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object.

Valid Usage (Implicit)

- VUID-vkCmdBindIndexBuffer-commandBuffer-parameter commandBuffer must be a valid VkCommandBuffer handle
- VUID-vkCmdBindIndexBuffer-buffer-parameter buffer must be a valid VkBuffer handle
- VUID-vkCmdBindIndexBuffer-indexType-parameter indexType must be a valid VkIndexType value
- VUID-vkCmdBindIndexBuffer-commandBuffer-recording commandBuffer must be in the recording state
- VUID-vkCmdBindIndexBuffer-commandBuffer-cmdpool The VkCommandPool that commandBuffer was allocated from must support graphics operations
- VUID-vkCmdBindIndexBuffer-commonparent Both of buffer, and commandBuffer must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Secondary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
</tbody>
</table>

Possible values of vkCmdBindIndexBuffer::indexType, specifying the size of indices, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkIndexType {
    VK_INDEX_TYPE_UINT16 = 0,
    VK_INDEX_TYPE_UINT32 = 1,
} VkIndexType;
```
• **VK_INDEX_TYPE_UINT16** specifies that indices are 16-bit unsigned integer values.

• **VK_INDEX_TYPE_UINT32** specifies that indices are 32-bit unsigned integer values.

The parameters for each drawing command are specified directly in the command or read from buffer memory, depending on the command. Drawing commands that source their parameters from buffer memory are known as *indirect* drawing commands.

All drawing commands interact with the *robustBufferAccess* feature.

To record a non-indexed draw, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdDraw(
    VkCommandBuffer
    commandBuffer,
    uint32_t
    vertexCount,
    uint32_t
    instanceCount,
    uint32_t
    firstVertex,
    uint32_t
    firstInstance);
```

• **commandBuffer** is the command buffer into which the command is recorded.

• **vertexCount** is the number of vertices to draw.

• **instanceCount** is the number of instances to draw.

• **firstVertex** is the index of the first vertex to draw.

• **firstInstance** is the instance ID of the first instance to draw.

When the command is executed, primitives are assembled using the current primitive topology and **vertexCount** consecutive vertex indices with the first **vertexIndex** value equal to **firstVertex**. The primitives are drawn **instanceCount** times with **instanceIndex** starting with **firstInstance** and increasing sequentially for each instance. The assembled primitives execute the bound graphics pipeline.

### Valid Usage

• **VUID-vkCmdDraw-magFilter-04553**
  If a **VkSampler** created with **magFilter** or **minFilter** equal to **VK_FILTER_LINEAR** and **compareEnable** equal to **VK_FALSE** is used to sample a **VkImageView** as a result of this command, then the image view’s **format features** must contain **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT**

• **VUID-vkCmdDraw-mipmapMode-04770**
  If a **VkSampler** created with **mipmapMode** equal to **VK_SAMPLER_MIPMAP_MODE_LINEAR** and **compareEnable** equal to **VK_FALSE** is used to sample a **VkImageView** as a result of this command, then the image view’s **format features** must contain **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT**

• **VUID-vkCmdDraw-None-06479**
  If a **VkImageView** is sampled with **depth comparison**, the image view’s **format features**
must contain VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_DEPTH_COMPARISON_BIT

- VUID-vkCmdDraw-None-02691
  If a VkImageView is accessed using atomic operations as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT

- VUID-vkCmdDraw-OpTypeImage-07027
  For any VkImageView being written as a storage image where the image format field of the OpTypeImage is Unknown, the view's format features must contain VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT

- VUID-vkCmdDraw-OpTypeImage-07028
  For any VkImageView being read as a storage image where the image format field of the OpTypeImage is Unknown, the view's format features must contain VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT

- VUID-vkCmdDraw-OpTypeImage-07029
  For any VkBufferView being written as a storage texel buffer where the image format field of the OpTypeImage is Unknown, the view's buffer features must contain VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT

- VUID-vkCmdDraw-OpTypeImage-07030
  Any VkBufferView being read as a storage texel buffer where the image format field of the OpTypeImage is Unknown then the view's buffer features must contain VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT

- VUID-vkCmdDraw-None-02697
  For each set n that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a descriptor set must have been bound to n at the same pipeline bind point, with a VkPipelineLayout that is compatible for set n, with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout Compatibility

- VUID-vkCmdDraw-maintenance4-06425
  If the maintenance4 feature is not enabled, then for each push constant that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a VkPipelineLayout that is compatible for push constants, with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout Compatibility

- VUID-vkCmdDraw-None-02699
  Descriptors in each bound descriptor set, specified via vkCmdBindDescriptorSets, must be valid as described by descriptor validity if they are statically used by the VkPipeline bound to the pipeline bind point used by this command

- VUID-vkCmdDraw-None-02700
  A valid pipeline must be bound to the pipeline bind point used by this command

- VUID-vkCmdDraw-commandBuffer-02701
  If the VkPipeline object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited (if the VK_NV_inherited_viewport_scissor extension is enabled) for commandBuffer, and done so after any previously bound pipeline with the corresponding state not specified as dynamic
There **must** not have been any calls to dynamic state setting commands for any state not specified as dynamic in the *VkPipeline* object bound to the pipeline bind point used by this command, since that pipeline was bound.

If the *VkPipeline* object bound to the pipeline bind point used by this command accesses a *VkSampler* object that uses unnormalized coordinates, that sampler **must** not be used to sample from any *VkImage* with a *VkImageView* of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

If the *VkPipeline* object bound to the pipeline bind point used by this command accesses a *VkSampler* object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V *OpImageSample* or *OpImageSparseSample* instructions with ImplicitLod, Dref or Proj in their name, in any shader stage.

If the *VkPipeline* object bound to the pipeline bind point used by this command accesses a *VkSampler* object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V *OpImageSample* or *OpImageSparseSample* instructions that includes a LOD bias or any offset values, in any shader stage.

If the *robustBufferAccess* feature is not enabled, and if the *VkPipeline* object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

If the *robustBufferAccess* feature is not enabled, and if the *VkPipeline* object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

If *commandBuffer* is an unprotected command buffer and *protectedNoFault* is not supported, any resource accessed by the *VkPipeline* object bound to the pipeline bind point used by this command **must** not be a protected resource.

If the *VkPipeline* object bound to the pipeline bind point used by this command accesses a *VkSampler* or *VkImageView* object that enables sampler Y’C_b conversion, that object **must** only be used with *OpImageSample* or *OpImageSparseSample* instructions.

If the *VkPipeline* object bound to the pipeline bind point used by this command accesses a *VkSampler* or *VkImageView* object that enables sampler Y’C_b conversion, that object **must** not use the *ConstOffset* and *Offset* operands.

If a *VkImageView* is accessed as a result of this command, then the image view’s *viewType*
must match the \texttt{Dim} operand of the \texttt{OpTypeImage} as described in Instruction/Sampler/Image View Validation

- VUID-vkCmdDraw-format-07753
  If a \texttt{VkImageView} is accessed as a result of this command, then the image view's \texttt{format} must match the numeric format from the \texttt{Sampled Type} operand of the \texttt{OpTypeImage} as described in the SPIR-V Sampled Type column of the Interpretation of Numeric Format table.

- VUID-vkCmdDraw-None-04115
  If a \texttt{VkImageView} is accessed using \texttt{OpImageWrite} as a result of this command, then the \texttt{Type} of the \texttt{Texel} operand of that instruction must have at least as many components as the image view's \texttt{format}.

- VUID-vkCmdDraw-OpImageWrite-04469
  If a \texttt{VkBufferView} is accessed using \texttt{OpImageWrite} as a result of this command, then the \texttt{Type} of the \texttt{Texel} operand of that instruction must have at least as many components as the buffer view's \texttt{format}.

- VUID-vkCmdDraw-None-07288
  Any shader invocation executed by this command must terminate.

- VUID-vkCmdDraw-renderPass-02684
  The current render pass must be compatible with the \texttt{renderPass} member of the \texttt{VkGraphicsPipelineCreateInfo} structure specified when creating the \texttt{VkPipeline} bound to \texttt{VK_PIPELINE_BIND_POINT_GRAPHICS}.

- VUID-vkCmdDraw-subpass-02685
  The subpass index of the current render pass must be equal to the \texttt{subpass} member of the \texttt{VkGraphicsPipelineCreateInfo} structure specified when creating the \texttt{VkPipeline} bound to \texttt{VK_PIPELINE_BIND_POINT_GRAPHICS}.

- VUID-vkCmdDraw-None-07748
  If any shader statically accesses an input attachment, a valid descriptor must be bound to the pipeline via a descriptor set.

- VUID-vkCmdDraw-OpTypeImage-07468
  If any shader executed by this pipeline accesses an \texttt{OpTypeImage} variable with a \texttt{Dim} operand of \texttt{SubpassData}, it must be decorated with an \texttt{InputAttachmentIndex} that corresponds to a valid input attachment in the current subpass.

- VUID-vkCmdDraw-None-07469
  Input attachment views accessed in a subpass must be created with the same \texttt{VkFormat} as the corresponding subpass definition, and be created with a \texttt{VkImageView} that is compatible with the attachment referenced by the subpass' \texttt{pInputAttachments[InputAttachmentIndex]} in the currently bound \texttt{VkFramebuffer} as specified by Fragment Input Attachment Compatibility.

- VUID-vkCmdDraw-None-06537
  Memory backing image subresources used as attachments in the current render pass must not be written in any way other than as an attachment by this command.

- VUID-vkCmdDraw-None-06538
  If any recorded command in the current subpass will write to an image subresource as an
attachment, this command must not read from the memory backing that image subresource in any other way than as an attachment.

- **VUID-vkCmdDraw-None-06539**
  If any recorded command in the current subpass will read from an image subresource used as an attachment in any way other than as an attachment, this command must not write to that image subresource as an attachment.

- **VUID-vkCmdDraw-None-06886**
  If the current render pass instance uses a depth/stencil attachment with a read-only layout for the depth aspect, depth writes must be disabled.

- **VUID-vkCmdDraw-None-06887**
  If the current render pass instance uses a depth/stencil attachment with a read-only layout for the stencil aspect, both front and back writeMask are not zero, and stencil test is enabled, all stencil ops must be VK_STENCIL_OP_KEEP.

- **VUID-vkCmdDraw-maxMultiviewInstanceIndex-02688**
  If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index must be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`.

- **VUID-vkCmdDraw-viewportCount-03417**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT` dynamic state enabled, then `vkCmdSetViewportWithCount` must have been called in the current command buffer prior to this drawing command, and the `viewportCount` parameter of `vkCmdSetViewportWithCount` must match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline.

- **VUID-vkCmdDraw-scissorCount-03418**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT` dynamic state enabled, then `vkCmdSetScissorWithCount` must have been called in the current command buffer prior to this drawing command, and the `scissorCount` parameter of `vkCmdSetScissorWithCount` must match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline.

- **VUID-vkCmdDraw-viewportCount-03419**
  If the bound graphics pipeline state was created with both the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT` and `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT` dynamic states enabled then both `vkCmdSetViewportWithCount` and `vkCmdSetScissorWithCount` must have been called in the current command buffer prior to this drawing command, and the `viewportCount` parameter of `vkCmdSetViewportWithCount` must match the `scissorCount` parameter of `vkCmdSetScissorWithCount`.

- **VUID-vkCmdDraw-None-04876**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE` dynamic state enabled then `vkCmdSetRasterizerDiscardEnable` must have been called in the current command buffer prior to this drawing command.
If the bound graphics pipeline state was created with the
`VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE` dynamic state enabled then
`vkCmdSetDepthBiasEnable` must have been called in the current command buffer prior
to this drawing command.

If rasterization is not disabled in the bound graphics pipeline, then for each color
attachment in the subpass, if the corresponding image view’s format features do not
contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT`, then the blendEnable member of
the corresponding element of the `pAttachments` member of `pColorBlendState` must be
`VK_FALSE`.

If rasterization is not disabled in the bound graphics pipeline, and neither the
`VK_AMD_mixed_attachment_samples` nor the `VK_NV_framebuffer_mixed_samples` extensions are
enabled, then `VkPipelineMultisampleStateCreateInfo::rasterizationSamples` must be the
same as the current subpass color and/or depth/stencil attachments.

If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView`
member of `pDepthAttachment` is not `VK_NULL_HANDLE`, and the layout member of
`pDepthAttachment` is `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, this command must
not write any values to the depth attachment.

If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView`
member of `pStencilAttachment` is not `VK_NULL_HANDLE`, and the layout member of
`pStencilAttachment` is `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, this
command must not write any values to the stencil attachment.

If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView`
member of `pDepthAttachment` is not `VK_NULL_HANDLE`, and the layout member of
`pDepthAttachment` is `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`, this
command must not write any values to the depth attachment.

If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView`
member of `pStencilAttachment` is not `VK_NULL_HANDLE`, and the layout member of
`pStencilAttachment` is `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`, this command must not
write any values to the stencil attachment.

If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView`
member of `pDepthAttachment` is not `VK_NULL_HANDLE`, and the layout member of
`pDepthAttachment` is `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, this command must not
write any values to the depth attachment.

If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView`
member of `pStencilAttachment` is not `VK_NULL_HANDLE`, and the layout member of
`pStencilAttachment` is `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`, this command must not
write any values to the stencil attachment.

If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView`
member of `pDepthAttachment` is not `VK_NULL_HANDLE`, and the layout member of
`pDepthAttachment` is `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, this
command must not write any values to the depth attachment.
write any values to the stencil attachment

• VUID-vkCmdDraw-viewMask-06178
  If the current render pass instance was begun with `vkCmdBeginRendering`, the currently bound graphics pipeline **must** have been created with a `VkPipelineRenderingCreateInfo`::`viewMask` equal to `VkRenderingInfo`::`viewMask`

• VUID-vkCmdDraw-colorAttachmentCount-06179
  If the current render pass instance was begun with `vkCmdBeginRendering`, the currently bound graphics pipeline **must** have been created with a `VkPipelineRenderingCreateInfo`::`colorAttachmentCount` equal to `VkRenderingInfo`::`colorAttachmentCount`

• VUID-vkCmdDraw-colorAttachmentCount-06180
  If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo`::`pColorAttachments->imageView` was not `VK_NULL_HANDLE` **must** have been created with a `VkFormat` equal to the corresponding element of `VkPipelineRenderingCreateInfo`::`pColorAttachmentFormats` used to create the currently bound graphics pipeline

• VUID-vkCmdDraw-colorAttachmentCount-07616
  If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo`::`pColorAttachments->imageView` was not `VK_NULL_HANDLE` **must** have been created with a `VkFormat` equal to the corresponding element of `VkPipelineRenderingCreateInfo`::`pColorAttachmentFormats` used to create the currently bound pipeline equal to `VK_FORMAT_UNDEFINED`

• VUID-vkCmdDraw-pDepthAttachment-06181
  If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo`::`pDepthAttachment->imageView` was not `VK_NULL_HANDLE`, the value of `VkPipelineRenderingCreateInfo`::`depthAttachmentFormat` used to create the currently bound graphics pipeline **must** be equal to the `VkFormat` used to create `VkRenderingInfo`::`pDepthAttachment->imageView`

• VUID-vkCmdDraw-pDepthAttachment-07617
  If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo`::`pDepthAttachment->imageView` was `VK_NULL_HANDLE`, the value of `VkPipelineRenderingCreateInfo`::`depthAttachmentFormat` used to create the currently bound graphics pipeline **must** be equal to `VK_FORMAT_UNDEFINED`

• VUID-vkCmdDraw-pStencilAttachment-06182
  If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo`::`pStencilAttachment->imageView` was not `VK_NULL_HANDLE`, the value of `VkPipelineRenderingCreateInfo`::`stencilAttachmentFormat` used to create the currently bound graphics pipeline **must** be equal to the `VkFormat` used to create `VkRenderingInfo`::`pStencilAttachment->imageView`

• VUID-vkCmdDraw-pStencilAttachment-07618
  If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo`::`pStencilAttachment->imageView` was `VK_NULL_HANDLE`, the value of `VkPipelineRenderingCreateInfo`::`stencilAttachmentFormat` used to create the currently bound graphics pipeline **must** be equal to `VK_FORMAT_UNDEFINED`
• VUID-vkCmdDraw-commandBuffer-02712
If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, any resource written to by the `VkPipeline` object bound to the pipeline bind point used by this command must not be an unprotected resource.

• VUID-vkCmdDraw-commandBuffer-02713
If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, pipeline stages other than the framebuffer-space and compute stages in the `VkPipeline` object bound to the pipeline bind point used by this command must not write to any resource.

• VUID-vkCmdDraw-None-04007
All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must have either valid or `VK_NULL_HANDLE` buffers bound.

• VUID-vkCmdDraw-None-04008
If the `nullDescriptor` feature is not enabled, all vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must not be `VK_NULL_HANDLE`.

• VUID-vkCmdDraw-None-02721
For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in Vertex Input Description.

• VUID-vkCmdDraw-primitiveTopology-03420
If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` dynamic state enabled then `vkCmdSetPrimitiveTopology` must have been called in the current command buffer prior to this drawing command, and the `primitiveTopology` parameter of `vkCmdSetPrimitiveTopology` must be of the same topology class as the pipeline `VkPipelineInputAssemblyStateCreateInfo::topology` state.

• VUID-vkCmdDraw-pStrides-04884
If the bound graphics pipeline was created with the `VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT` dynamic state enabled, then `vkCmdBindVertexBuffers2EXT` must have been called in the current command buffer prior to this drawing command, and the `pStrides` parameter of `vkCmdBindVertexBuffers2EXT` must not be `NULL`.

• VUID-vkCmdDraw-None-04879
If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE` dynamic state enabled then `vkCmdSetPrimitiveRestartEnable` must have been called in the current command buffer prior to this drawing command.

Valid Usage (Implicit)

• VUID-vkCmdDraw-commandBuffer-parameter
`commandBuffer` must be a valid `VkCommandBuffer` handle.
• **VUID-vkCmdDraw-commandBuffer-recording**
  
  *commandBuffer* must be in the *recording state*

• **VUID-vkCmdDraw-commandBuffer-cmdpool**
  
  The *VkCommandPool* that *commandBuffer* was allocated from must support graphics operations

• **VUID-vkCmdDraw-renderpass**
  
  This command must only be called inside of a render pass instance

---

### Host Synchronization

• Host access to *commandBuffer* must be externally synchronized

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

---

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To record an indexed draw, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdDrawIndexed(
  VkCommandBuffer
  commandBuffer,           // Provided by VK_VERSION_1_0
  uint32_t indexCount,     // Provided by VK_VERSION_1_0
  uint32_t instanceCount,  // Provided by VK_VERSION_1_0
  uint32_t firstIndex,     // Provided by VK_VERSION_1_0
  int32_t vertexOffset,    // Provided by VK_VERSION_1_0
  uint32_t firstInstance); // Provided by VK_VERSION_1_0
```

• *commandBuffer* is the command buffer into which the command is recorded.

• *indexCount* is the number of vertices to draw.

• *instanceCount* is the number of instances to draw.

• *firstIndex* is the base index within the index buffer.

• *vertexOffset* is the value added to the vertex index before indexing into the vertex buffer.

• *firstInstance* is the instance ID of the first instance to draw.

When the command is executed, primitives are assembled using the current primitive topology and *indexCount* vertices whose indices are retrieved from the index buffer. The index buffer is treated
as an array of tightly packed unsigned integers of size defined by the `vkCmdBindIndexBuffer ::indexType` parameter with which the buffer was bound.

The first vertex index is at an offset of `firstIndex \times indexSize + offset` within the bound index buffer, where `offset` is the offset specified by `vkCmdBindIndexBuffer` and `indexSize` is the byte size of the type specified by `indexType`. Subsequent index values are retrieved from consecutive locations in the index buffer. Indices are first compared to the primitive restart value, then zero extended to 32 bits (if the `indexType` is `VK_INDEX_TYPE_UINT16`) and have `vertexOffset` added to them, before being supplied as the `vertexIndex` value.

The primitives are drawn `instanceCount` times with `instanceIndex` starting with `firstInstance` and increasing sequentially for each instance. The assembled primitives execute the bound graphics pipeline.

---

**Valid Usage**

- **VUID-vkCmdDrawIndexed-magFilter-04553**
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's `format features` must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDrawIndexed-mipmapMode-04770**
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's `format features` must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDrawIndexed-None-06479**
  If a `VkImageView` is sampled with depth comparison, the image view's `format features` must contain `VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_DEPTH_COMPARISON_BIT`

- **VUID-vkCmdDrawIndexed-None-02691**
  If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's `format features` must contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`

- **VUID-vkCmdDrawIndexed-OpTypeImage-07027**
  For any `VkImageView` being written as a storage image where the image format field of the `OpTypeImage` is `Unknown`, the view's `format features` must contain `VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT`

- **VUID-vkCmdDrawIndexed-OpTypeImage-07028**
  For any `VkImageView` being read as a storage image where the image format field of the `OpTypeImage` is `Unknown`, the view's `format features` must contain `VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT`

- **VUID-vkCmdDrawIndexed-OpTypeImage-07029**
  For any `VkBufferView` being written as a storage texel buffer where the image format field of the `OpTypeImage` is `Unknown`, the view's `buffer features` must contain `VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT`

- **VUID-vkCmdDrawIndexed-OpTypeImage-07030**
Any `VkBufferView` being read as a storage texel buffer where the image format field of the `OpTypeImage` is `Unknown` then the view's buffer features must contain `VK_FORMAT_FEATURE_2_STORAGE_READ WITHOUT_FORMAT_BIT`.

- **VUID-vkCmdDrawIndexed-None-02697**
  For each set $n$ that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set must have been bound to $n$ at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set $n$, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility.

- **VUID-vkCmdDrawIndexed-indexed-maintenance4-06425**
  If the `maintenance4` feature is not enabled, then for each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility.

- **VUID-vkCmdDrawIndexed-None-02699**
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid as described by descriptor validity if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

- **VUID-vkCmdDrawIndexed-None-02700**
  A valid pipeline must be bound to the pipeline bind point used by this command.

- **VUID-vkCmdDrawIndexed-commandBuffer-02701**
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited (if the `VK_NV_inherited_viewport_scissor` extension is enabled) for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

- **VUID-vkCmdDrawIndexed-None-02859**
  There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

- **VUID-vkCmdDrawIndexed-None-02702**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

- **VUID-vkCmdDrawIndexed-None-02703**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

- **VUID-vkCmdDrawIndexed-None-02704**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with...
any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage

- **VUID-vkCmdDrawIndexed-None-02705**
  If the `robustBufferAccess` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point

- **VUID-vkCmdDrawIndexed-None-02706**
  If the `robustBufferAccess` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point

- **VUID-vkCmdDrawIndexed-commandBuffer-02707**
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be a protected resource

- **VUID-vkCmdDrawIndexed-None-06550**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` or `VkImageView` object that enables sampler Y’CbCr conversion, that object **must** only be used with `OpImageSample*` or `OpImageSparseSample*` instructions

- **VUID-vkCmdDrawIndexed-ConstOffset-06551**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` or `VkImageView` object that enables sampler Y’CbCr conversion, that object **must** not use the `ConstOffset` and `Offset` operands

- **VUID-vkCmdDrawIndexed-viewType-07752**
  If a `VkImageView` is accessed as a result of this command, then the image view's `viewType` **must** match the `Dim` operand of the `OpTypeImage` as described in Instruction/Sampler/Image View Validation

- **VUID-vkCmdDrawIndexed-format-07753**
  If a `VkImageView` is accessed as a result of this command, then the image view's `format` **must** match the numeric format from the `Sampled Type` operand of the `OpTypeImage` as described in the SPIR-V Sampled Type column of the Interpretation of Numeric Format table

- **VUID-vkCmdDrawIndexed-None-04115**
  If a `VkImageView` is accessed using `OpImageWrite` as a result of this command, then the `Type` of the `Texel` operand of that instruction **must** have at least as many components as the image view's format

- **VUID-vkCmdDrawIndexed-OpImageWrite-04469**
  If a `VkBufferView` is accessed using `OpImageWrite` as a result of this command, then the `Type` of the `Texel` operand of that instruction **must** have at least as many components as the buffer view's format

- **VUID-vkCmdDrawIndexed-None-07288**
  Any shader invocation executed by this command **must** terminate
The current render pass must be compatible with the renderPass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS.

The subpass index of the current render pass must be equal to the subpass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS.

If any shader statically accesses an input attachment, a valid descriptor must be bound to the pipeline via a descriptor set.

If any shader executed by this pipeline accesses an OpTypeImage variable with a Dim operand of SubpassData, it must be decorated with an InputAttachmentIndex that corresponds to a valid input attachment in the current subpass.

Input attachment views accessed in a subpass must be created with the same VkFormat as the corresponding subpass definition, and be created with a VkImageView that is compatible with the attachment referenced by the subpass' pInputAttachments [InputAttachmentIndex] in the currently bound VkFramebuffer as specified by Fragment Input Attachment Compatibility.

Memory backing image subresources used as attachments in the current render pass must not be written in any way other than as an attachment by this command.

If any recorded command in the current subpass will write to an image subresource as an attachment, this command must not read from the memory backing that image subresource in any other way than as an attachment.

If any recorded command in the current subpass will read from an image subresource used as an attachment in any way other than as an attachment, this command must not write to that image subresource as an attachment.

If the current render pass instance uses a depth/stencil attachment with a read-only layout for the depth aspect, depth writes must be disabled.

If the current render pass instance uses a depth/stencil attachment with a read-only layout for the stencil aspect, both front and back writeMask are not zero, and stencil test is enabled, all stencil ops must be VK_STENCIL_OP_KEEP.

If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index must be less than or equal to VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex.
If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT` dynamic state enabled, then `vkCmdSetViewportWithCount` must have been called in the current command buffer prior to this drawing command, and the `viewportCount` parameter of `vkCmdSetViewportWithCount` must match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT` dynamic state enabled, then `vkCmdSetScissorWithCount` must have been called in the current command buffer prior to this drawing command, and the `scissorCount` parameter of `vkCmdSetScissorWithCount` must match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline.

If the bound graphics pipeline state was created with both the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT` and `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT` dynamic states enabled then both `vkCmdSetViewportWithCount` and `vkCmdSetScissorWithCount` must have been called in the current command buffer prior to this drawing command, and the `viewportCount` parameter of `vkCmdSetViewportWithCount` must match the `scissorCount` parameter of `vkCmdSetScissorWithCount`.

If rasterization is not disabled in the bound graphics pipeline, and neither the `VK_AMD_mixed_attachment_samples` nor the `VK_NV_framebuffer_mixed_samples` extensions are enabled, then `VkPipelineMultisampleStateCreateInfo::rasterizationSamples` must be the same as the current subpass color and/or depth/stencil attachments.

If the current render pass instance was begun with `vkCmdBeginRendering`, then `imageView` must be a valid image view of the current render pass.
• VUID-vkCmdDrawIndexed-imageView-06173
  If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView` member of `pStencilAttachment` is not `VK_NULL_HANDLE`, and the `layout` member of `pStencilAttachment` is `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, this command **must** not write any values to the stencil attachment

• VUID-vkCmdDrawIndexed-imageView-06174
  If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView` member of `pDepthAttachment` is not `VK_NULL_HANDLE`, and the `layout` member of `pDepthAttachment` is `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`, this command **must** not write any values to the depth attachment

• VUID-vkCmdDrawIndexed-imageView-06175
  If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView` member of `pStencilAttachment` is not `VK_NULL_HANDLE`, and the `layout` member of `pStencilAttachment` is `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`, this command **must** not write any values to the stencil attachment

• VUID-vkCmdDrawIndexed-viewMask-06178
  If the current render pass instance was begun with `vkCmdBeginRendering`, the currently bound graphics pipeline **must** have been created with a `VkPipelineRenderingCreateInfo` `::viewMask` equal to `VkRenderingInfo` `::viewMask`

• VUID-vkCmdDrawIndexed-colorAttachmentCount-06179
  If the current render pass instance was begun with `vkCmdBeginRendering`, the currently bound graphics pipeline **must** have been created with a `VkPipelineRenderingCreateInfo` `::colorAttachmentCount` equal to `VkRenderingInfo` `::colorAttachmentCount`

• VUID-vkCmdDrawIndexed-colorAttachmentCount-06180
  If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo` `::colorAttachmentCount` greater than 0, then each element of the `VkRenderingInfo` `::pColorAttachments` array with a `imageView` not equal to `VK_NULL_HANDLE` **must** have been created with a `VkFormat` equal to the corresponding element of `VkPipelineRenderingCreateInfo` `::pColorAttachmentFormats` used to create the currently bound graphics pipeline

• VUID-vkCmdDrawIndexed-colorAttachmentCount-07616
If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo::colorAttachmentCount` greater than 0, then each element of the `VkRenderingInfo::pColorAttachments` array with a `imageView` equal to `VK_NULL_HANDLE` must have the corresponding element of `VkPipelineRenderingCreateInfo::pColorAttachmentFormats` used to create the currently bound pipeline equal to `VK_FORMAT_UNDEFINED`.

- VUID-vkCmdDrawIndexed-pDepthAttachment-06181
  If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo::pDepthAttachment->imageView` was not `VK_NULL_HANDLE`, the value of `VkPipelineRenderingCreateInfo::depthAttachmentFormat` used to create the currently bound graphics pipeline must be equal to the `VkFormat` used to create `VkRenderingInfo::pDepthAttachment->imageView`.

- VUID-vkCmdDrawIndexed-pDepthAttachment-07617
  If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo::pDepthAttachment->imageView` was `VK_NULL_HANDLE`, the value of `VkPipelineRenderingCreateInfo::depthAttachmentFormat` used to create the currently bound graphics pipeline must be equal to `VK_FORMAT_UNDEFINED`.

- VUID-vkCmdDrawIndexed-pStencilAttachment-06182
  If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo::pStencilAttachment->imageView` was not `VK_NULL_HANDLE`, the value of `VkPipelineRenderingCreateInfo::stencilAttachmentFormat` used to create the currently bound graphics pipeline must be equal to the `VkFormat` used to create `VkRenderingInfo::pStencilAttachment->imageView`.

- VUID-vkCmdDrawIndexed-pStencilAttachment-07618
  If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo::pStencilAttachment->imageView` was `VK_NULL_HANDLE`, the value of `VkPipelineRenderingCreateInfo::stencilAttachmentFormat` used to create the currently bound graphics pipeline must be equal to `VK_FORMAT_UNDEFINED`.

- VUID-vkCmdDrawIndexed-commandBuffer-02712
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, any resource written to by the `VkPipeline` object bound to the pipeline bind point used by this command must not be an unprotected resource.

- VUID-vkCmdDrawIndexed-commandBuffer-02713
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, pipeline stages other than the framebuffer-space and compute stages in the `VkPipeline` object bound to the pipeline bind point used by this command must not write to any resource.

- VUID-vkCmdDrawIndexed-None-04007
  All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must have either valid or `VK_NULL_HANDLE` buffers bound.

- VUID-vkCmdDrawIndexed-None-04008
  If the `nullDescriptor` feature is not enabled, all vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must not be
VK_NULL_HANDLE

• VUID-vkCmdDrawIndexed-None-02721
  For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in Vertex Input Description.

• VUID-vkCmdDrawIndexed-primitiveTopology-03420
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT dynamic state enabled then vkCmdSetPrimitiveTopology must have been called in the current command buffer prior to this drawing command, and the primitiveTopology parameter of vkCmdSetPrimitiveTopology must be of the same topology class as the pipeline VkPipelineInputAssemblyStateCreateInfo::topology state.

• VUID-vkCmdDrawIndexed-pStrides-04884
  If the bound graphics pipeline was created with the VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT dynamic state enabled, then vkCmdBindVertexBuffers2EXT must have been called in the current command buffer prior to this drawing command, and the pStrides parameter of vkCmdBindVertexBuffers2EXT must not be NULL.

• VUID-vkCmdDrawIndexed-None-04879
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE dynamic state enabled then vkCmdSetPrimitiveRestartEnable must have been called in the current command buffer prior to this drawing command.

• VUID-vkCmdDrawIndexed-None-07312
  An index buffer must be bound.

• VUID-vkCmdDrawIndexed-firstIndex-04932
  (indexSize × (firstIndex + indexCount) + offset) must be less than or equal to the size of the bound index buffer, with indexSize being based on the type specified by indexType, where the index buffer, indexType, and offset are specified via vkCmdBindIndexBuffer.

Valid Usage (Implicit)

• VUID-vkCmdDrawIndexed-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle.

• VUID-vkCmdDrawIndexed-commandBuffer-recording
  commandBuffer must be in the recording state.

• VUID-vkCmdDrawIndexed-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations.

• VUID-vkCmdDrawIndexed-renderpass
  This command must only be called inside of a render pass instance.
Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To record a non-indexed indirect drawing command, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdDrawIndirect(
    VkCommandBuffer commandBuffer,  
    VkBuffer buffer,                
    VkDeviceSize offset,           
    uint32_t drawCount,            
    uint32_t stride);              
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `buffer` is the buffer containing draw parameters.
- `offset` is the byte offset into `buffer` where parameters begin.
- `drawCount` is the number of draws to execute, and can be zero.
- `stride` is the byte stride between successive sets of draw parameters.

`vkCmdDrawIndirect` behaves similarly to `vkCmdDraw` except that the parameters are read by the device from a buffer during execution. `drawCount` draws are executed by the command, with parameters taken from `buffer` starting at `offset` and increasing by `stride` bytes for each successive draw. The parameters of each draw are encoded in an array of `VkDrawIndirectCommand` structures. If `drawCount` is less than or equal to one, `stride` is ignored.

Valid Usage

- VUID-vkCmdDrawIndirect-magFilter-04553
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view’s `format features` must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`
• VUID-vkCmdDrawIndirect-mipmapMode-04770
If a VkSampler created with mipmapMode equal to VK_SAMPLER_MIPMAP_MODE_LINEAR and compareEnable equal to VK_FALSE is used to sample a VkImageView as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT

• VUID-vkCmdDrawIndirect-None-06479
If a VkImageView is sampled with depth comparison, the image view's format features must contain VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_DEPTH_COMPARISON_BIT

• VUID-vkCmdDrawIndirect-None-02691
If a VkImageView is accessed using atomic operations as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT

• VUID-vkCmdDrawIndirect-OpTypeImage-07027
For any VkImageView being written as a storage image where the image format field of the OpTypeImage is Unknown, the view's format features must contain VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT

• VUID-vkCmdDrawIndirect-OpTypeImage-07028
For any VkImageView being read as a storage image where the image format field of the OpTypeImage is Unknown, the view's format features must contain VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT

• VUID-vkCmdDrawIndirect-OpTypeImage-07029
For any VkBufferView being written as a storage texel buffer where the image format field of the OpTypeImage is Unknown, the view's buffer features must contain VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT

• VUID-vkCmdDrawIndirect-OpTypeImage-07030
Any VkBufferView being read as a storage texel buffer where the image format field of the OpTypeImage is Unknown then the view's buffer features must contain VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT

• VUID-vkCmdDrawIndirect-None-02697
For each set n that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a descriptor set must have been bound to n at the same pipeline bind point, with a VkPipelineLayout that is compatible for set n, with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout Compatibility

• VUID-vkCmdDrawIndirect-None-02699
Descriptors in each bound descriptor set, specified via vkCmdBindDescriptorSets, must be valid as described by descriptor validity if they are statically used by the VkPipeline bound to the pipeline bind point used by this command

• VUID-vkCmdDrawIndirect-None-02700
A valid pipeline must be bound to the pipeline bind point used by this command
If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state **must** have been set or inherited (if the `VK_NV_inherited_viewport_scissor` extension is enabled) for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

**VUID-vkCmdDrawIndirect-None-02859**
There **must** not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

**VUID-vkCmdDrawIndirect-None-02702**
If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

**VUID-vkCmdDrawIndirect-None-02703**
If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

**VUID-vkCmdDrawIndirect-None-02704**
If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

**VUID-vkCmdDrawIndirect-None-02705**
If the robustBufferAccess feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

**VUID-vkCmdDrawIndirect-None-02706**
If the robustBufferAccess feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

**VUID-vkCmdDrawIndirect-commandBuffer-02707**
If `commandBuffer` is an unprotected command buffer and protectedNoFault is not supported, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be a protected resource.

**VUID-vkCmdDrawIndirect-None-06550**
If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` or `VkImageView` object that enables sampler Y'CbCr conversion, that object **must** only be used with `OpImageSample*` or `OpImageSparseSample*` instructions.
If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler or VkImageView object that enables sampler Y′CbCr conversion, that object must not use the ConstOffset and Offset operands.

If a VkImageView is accessed as a result of this command, then the image view's viewType must match the Dim operand of the OpTypeImage as described in Instruction/Sampler/Image View Validation.

If a VkImageView is accessed as a result of this command, then the image view's format must match the numeric format from the Sampled Type operand of the OpTypeImage as described in the SPIR-V Sampled Type column of the Interpretation of Numeric Format table.

If a VkImageView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view's format.

If a VkBufferView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the buffer view's format.

Any shader invocation executed by this command must terminate.

The current render pass must be compatible with the renderPass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS.

The subpass index of the current render pass must be equal to the subpass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS.

If any shader statically accesses an input attachment, a valid descriptor must be bound to the pipeline via a descriptor set.

If any shader executed by this pipeline accesses an OpTypeImage variable with a Dim operand of SubpassData, it must be decorated with an InputAttachmentIndex that corresponds to a valid input attachment in the current subpass.

Input attachment views accessed in a subpass must be created with the same VkFormat as the corresponding subpass definition, and be created with a VkImageView that is compatible with the attachment referenced by the subpass' pInputAttachments [InputAttachmentIndex] in the currently bound VkFramebuffer as specified by Fragment Input Attachment Compatibility.
• VUID-vkCmdDrawIndirect-None-06537
  Memory backing image subresources used as attachments in the current render pass must not be written in any way other than as an attachment by this command

• VUID-vkCmdDrawIndirect-None-06538
  If any recorded command in the current subpass will write to an image subresource as an attachment, this command must not read from the memory backing that image subresource in any other way than as an attachment

• VUID-vkCmdDrawIndirect-None-06539
  If any recorded command in the current subpass will read from an image subresource used as an attachment in any way other than as an attachment, this command must not write to that image subresource as an attachment

• VUID-vkCmdDrawIndirect-None-06886
  If the current render pass instance uses a depth/stencil attachment with a read-only layout for the depth aspect, depth writes must be disabled

• VUID-vkCmdDrawIndirect-None-06887
  If the current render pass instance uses a depth/stencil attachment with a read-only layout for the stencil aspect, both front and back writeMask are not zero, and stencil test is enabled, all stencil ops must be VK_STENCIL_OP_KEEP

• VUID-vkCmdDrawIndirect-maxMultiviewInstanceIndex-02688
  If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index must be less than or equal to VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex

• VUID-vkCmdDrawIndirect-viewportCount-03417
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT dynamic state enabled, but not the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT dynamic state enabled, then vkCmdSetViewportWithCount must have been called in the current command buffer prior to this drawing command, and the viewportCount parameter of vkCmdSetViewportWithCount must match the VkPipelineViewportStateCreateInfo::viewportCount of the pipeline

• VUID-vkCmdDrawIndirect-scissorCount-03418
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT dynamic state enabled, but not the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT dynamic state enabled, then vkCmdSetScissorWithCount must have been called in the current command buffer prior to this drawing command, and the scissorCount parameter of vkCmdSetScissorWithCount must match the VkPipelineViewportStateCreateInfo::viewportCount of the pipeline

• VUID-vkCmdDrawIndirect-viewportCount-03419
  If the bound graphics pipeline state was created with both the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT and VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT dynamic states enabled then both vkCmdSetViewportWithCount and vkCmdSetScissorWithCount must have been called in the current command buffer prior to this drawing command, and the viewportCount parameter of vkCmdSetViewportWithCount must match the scissorCount parameter of vkCmdSetScissorWithCount
If the bound graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE} dynamic state enabled then \texttt{vkCmdSetRasterizerDiscardEnable} must have been called in the current command buffer prior to this drawing command.

If the bound graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE} dynamic state enabled then \texttt{vkCmdSetDepthBiasEnable} must have been called in the current command buffer prior to this drawing command.

If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view's format features do not contain \texttt{VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT}, then the blendEnable member of the corresponding element of the pAttachments member of pColorBlendState must be \texttt{VK_FALSE}.

If rasterization is not disabled in the bound graphics pipeline, and neither the \texttt{VK_AMD_mixed_attachment_samples} nor the \texttt{VK_NV_framebuffer_mixed_samples} extensions are enabled, then \texttt{VkPipelineMultisampleStateCreateInfo::rasterizationSamples} must be the same as the current subpass color and/or depth/stencil attachments.

If the current render pass instance was begun with \texttt{vkCmdBeginRendering}, the imageView member of pDepthAttachment is not \texttt{VK_NULL_HANDLE}, and the layout member of pDepthAttachment is \texttt{VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL}, this command must not write any values to the depth attachment.

If the current render pass instance was begun with \texttt{vkCmdBeginRendering}, the imageView member of pStencilAttachment is not \texttt{VK_NULL_HANDLE}, and the layout member of pStencilAttachment is \texttt{VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL}, this command must not write any values to the stencil attachment.

If the current render pass instance was begun with \texttt{vkCmdBeginRendering}, the imageView member of pDepthAttachment is not \texttt{VK_NULL_HANDLE}, and the layout member of pDepthAttachment is \texttt{VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL}, this command must not write any values to the depth attachment.

If the current render pass instance was begun with \texttt{vkCmdBeginRendering}, the imageView member of pStencilAttachment is not \texttt{VK_NULL_HANDLE}, and the layout member of pStencilAttachment is \texttt{VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL}, this command must not write any values to the stencil attachment.

If the current render pass instance was begun with \texttt{vkCmdBeginRendering}, the imageView member of pDepthAttachment is not \texttt{VK_NULL_HANDLE}, and the layout member of pDepthAttachment is \texttt{VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL}, this command must not write any values to the depth attachment.
write any values to the depth attachment

- VUID-vkCmdDrawIndirect-imageView-06177
  If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView` member of `pStencilAttachment` is not `VK_NULL_HANDLE`, and the `layout` member of `pStencilAttachment` is `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`, this command must not write any values to the stencil attachment.

- VUID-vkCmdDrawIndirect-viewMask-06178
  If the current render pass instance was begun with `vkCmdBeginRendering`, the currently bound graphics pipeline must have been created with a `VkPipelineRenderingCreateInfo`::`viewMask` equal to `VkRenderingInfo`::`viewMask`.

- VUID-vkCmdDrawIndirect-colorAttachmentCount-06179
  If the current render pass instance was begun with `vkCmdBeginRendering`, the currently bound graphics pipeline must have been created with a `VkPipelineRenderingCreateInfo`::`colorAttachmentCount` equal to `VkRenderingInfo`::`colorAttachmentCount`.

- VUID-vkCmdDrawIndirect-colorAttachmentCount-06180
  If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo`::`colorAttachmentCount` greater than 0, then each element of the `VkRenderingInfo`::`pColorAttachments` array with a `imageView` not equal to `VK_NULL_HANDLE` must have been created with a `VkFormat` equal to the corresponding element of `VkPipelineRenderingCreateInfo`::`pColorAttachmentFormats` used to create the currently bound graphics pipeline.

- VUID-vkCmdDrawIndirect-colorAttachmentCount-07616
  If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo`::`colorAttachmentCount` greater than 0, then each element of the `VkRenderingInfo`::`pColorAttachments` array with a `imageView` equal to `VK_NULL_HANDLE` must have the corresponding element of `VkPipelineRenderingCreateInfo`::`pColorAttachmentFormats` used to create the currently bound pipeline equal to `VK_FORMAT_UNDEFINED`.

- VUID-vkCmdDrawIndirect-pDepthAttachment-06181
  If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo`::`pDepthAttachment->imageView` was not `VK_NULL_HANDLE`, the value of `VkPipelineRenderingCreateInfo`::`depthAttachmentFormat` used to create the currently bound graphics pipeline must be equal to the `VkFormat` used to create `VkRenderingInfo`::`pDepthAttachment->imageView`.

- VUID-vkCmdDrawIndirect-pDepthAttachment-07617
  If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo`::`pDepthAttachment->imageView` was `VK_NULL_HANDLE`, the value of `VkPipelineRenderingCreateInfo`::`depthAttachmentFormat` used to create the currently bound graphics pipeline must be equal to `VK_FORMAT_UNDEFINED`.

- VUID-vkCmdDrawIndirect-pStencilAttachment-06182
  If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo`::`pStencilAttachment->imageView` was not `VK_NULL_HANDLE`, the value of `VkPipelineRenderingCreateInfo`::`stencilAttachmentFormat` used to create the currently bound graphics pipeline must be equal to the `VkFormat` used to create `VkRenderingInfo`::`pStencilAttachment->imageView`.
If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo::pStencilAttachment->imageView` was `VK_NULL_HANDLE`, the value of `VkPipelineRenderingCreateInfo::stencilAttachmentFormat` used to create the currently bound graphics pipeline must be equal to `VK_FORMAT_UNDEFINED`.

All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must have either valid or `VK_NULL_HANDLE` buffers bound.

If the `nullDescriptor` feature is not enabled, all vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must not be `VK_NULL_HANDLE`.

For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in Vertex Input Description.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` dynamic state enabled then `vkCmdSetPrimitiveTopology` must have been called in the current command buffer prior to this drawing command, and the `primitiveTopology` parameter of `vkCmdSetPrimitiveTopology` must be of the same topology class as the pipeline `VkPipelineInputAssemblyStateCreateInfo::topology` state.

If the bound graphics pipeline was created with the `VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT` dynamic state enabled, then `vkCmdBindVertexBuffers2EXT` must have been called in the current command buffer prior to this drawing command, and the `pStrides` parameter of `vkCmdBindVertexBuffers2EXT` must not be `NULL`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE` dynamic state enabled then `vkCmdSetPrimitiveRestartEnable` must have been called in the current command buffer prior to this drawing command.

If `buffer` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.

`buffer` must have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set.

`offset` must be a multiple of 4.

`commandBuffer` must not be a protected command buffer.
If the multiDrawIndirect feature is not enabled, drawCount must be 0 or 1.

If drawCount is greater than 1, stride must be a multiple of 4 and must be greater than or equal to sizeof(VkDrawIndirectCommand).

If drawCount is equal to 1, (offset + sizeof(VkDrawIndirectCommand)) must be less than or equal to the size of buffer.

If drawCount is greater than 1, (stride × (drawCount - 1) + offset + sizeof(VkDrawIndirectCommand)) must be less than or equal to the size of buffer.

**Valid Usage (Implicit)**

commandBuffer must be a valid VkCommandBuffer handle.

buffer must be a valid VkBuffer handle.

commandBuffer must be in the recording state.

The VkCommandPool that commandBuffer was allocated from must support graphics operations.

This command must only be called inside of a render pass instance.

Both of buffer, and commandBuffer must have been created, allocated, or retrieved from the same VkDevice.

**Host Synchronization**

Host access to commandBuffer must be externally synchronized.

Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized.
**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkDrawIndirectCommand` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDrawIndirectCommand {
    uint32_t vertexCount;
    uint32_t instanceCount;
    uint32_t firstVertex;
    uint32_t firstInstance;
} VkDrawIndirectCommand;
```

- `vertexCount` is the number of vertices to draw.
- `instanceCount` is the number of instances to draw.
- `firstVertex` is the index of the first vertex to draw.
- `firstInstance` is the instance ID of the first instance to draw.

The members of `VkDrawIndirectCommand` have the same meaning as the similarly named parameters of `vkCmdDraw`.

**Valid Usage**

- **VUID-VkDrawIndirectCommand-None-00500**
  For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in [Vertex Input Description](#).

- **VUID-VkDrawIndirectCommand-firstInstance-00501**
  If the `drawIndirectFirstInstance` feature is not enabled, `firstInstance` must be 0

To record a non-indexed draw call with a draw call count sourced from a buffer, call:

```c
// Provided by VK_VERSION_1_2
void vkCmdDrawIndirectCount(
    VkCommandBuffer commandBuffer,
    VkBuffer buffer,
    VkDeviceSize offset,
    VkBuffer countBuffer,
    VkDeviceSize countBufferOffset,
    uint32_t maxDrawCount,
);
```
• `commandBuffer` is the command buffer into which the command is recorded.
• `buffer` is the buffer containing draw parameters.
• `offset` is the byte offset into `buffer` where parameters begin.
• `countBuffer` is the buffer containing the draw count.
• `countBufferOffset` is the byte offset into `countBuffer` where the draw count begins.
• `maxDrawCount` specifies the maximum number of draws that will be executed. The actual number of executed draw calls is the minimum of the count specified in `countBuffer` and `maxDrawCount`.
• `stride` is the byte stride between successive sets of draw parameters.

`vkCmdDrawIndirectCount` behaves similarly to `vkCmdDrawIndirect` except that the draw count is read by the device from a buffer during execution. The command will read an unsigned 32-bit integer from `countBuffer` located at `countBufferOffset` and use this as the draw count.

### Valid Usage

- **VUID-vkCmdDrawIndirectCount-magFilter-04553**
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDrawIndirectCount-mipmapMode-04770**
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDrawIndirectCount-None-06479**
  If a `VkImageView` is sampled with depth comparison, the image view's format features must contain `VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_DEPTH_COMPARISON_BIT`

- **VUID-vkCmdDrawIndirectCount-None-02691**
  If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`

- **VUID-vkCmdDrawIndirectCount-OpTypeImage-07027**
  For any `VkImageView` being written as a storage image where the image format field of the `OpTypeImage` is `Unknown`, the view's format features must contain `VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT`

- **VUID-vkCmdDrawIndirectCount-OpTypeImage-07028**
  For any `VkImageView` being read as a storage image where the image format field of the `OpTypeImage` is `Unknown`, the view's format features must contain `VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT`

- **VUID-vkCmdDrawIndirectCount-OpTypeImage-07029**
  For any `VkBufferView` being written as a storage texel buffer where the image format
field of the OpTypeImage is Unknown, the view’s buffer features must contain VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT

• VUID-vkCmdDrawIndirectCount-OpTypeImage-07030
Any VkBufferView being read as a storage texel buffer where the image format field of the OpTypeImage is Unknown then the view’s buffer features must contain VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT

• VUID-vkCmdDrawIndirectCount-None-02697
For each set \( n \) that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a descriptor set must have been bound to \( n \) at the same pipeline bind point, with a VkPipelineLayout that is compatible for set \( n \), with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout Compatibility

• VUID-vkCmdDrawIndirectCount-maintenance4-06425
If the maintenance4 feature is not enabled, then for each push constant that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a VkPipelineLayout that is compatible for push constants, with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout Compatibility

• VUID-vkCmdDrawIndirectCount-None-02699
Descriptors in each bound descriptor set, specified via vkCmdBindDescriptorSets, must be valid as described by descriptor validity if they are statically used by the VkPipeline bound to the pipeline bind point used by this command

• VUID-vkCmdDrawIndirectCount-None-02700
A valid pipeline must be bound to the pipeline bind point used by this command

• VUID-vkCmdDrawIndirectCount-commandBuffer-02701
If the VkPipeline object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited (if the VK_NV_inherited_viewport_scissor extension is enabled) for commandBuffer, and done so after any previously bound pipeline with the corresponding state not specified as dynamic

• VUID-vkCmdDrawIndirectCount-None-02859
There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the VkPipeline object bound to the pipeline bind point used by this command, since that pipeline was bound

• VUID-vkCmdDrawIndirectCount-None-02702
If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used to sample from any VkImage with a VkImageView of the type VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, VK_IMAGE_VIEW_TYPE_1D_ARRAY, VK_IMAGE_VIEW_TYPE_2D_ARRAY or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY, in any shader stage

• VUID-vkCmdDrawIndirectCount-None-02703
If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions with ImplicitLod, Dref or Proj in their name, in any shader stage
If the \textit{VkPipeline} object bound to the pipeline bind point used by this command accesses a \textit{VkSampler} object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V \textit{OpImageSample*} or \textit{OpImageSparseSample*} instructions that includes a LOD bias or any offset values, in any shader stage.

If the \textit{robustBufferAccess} feature is not enabled, and if the \textit{VkPipeline} object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

If the \textit{robustBufferAccess} feature is not enabled, and if the \textit{VkPipeline} object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

If \textit{commandBuffer} is an unprotected command buffer and \textit{protectedNoFault} is not supported, any resource accessed by the \textit{VkPipeline} object bound to the pipeline bind point used by this command must not be a protected resource.

If the \textit{VkPipeline} object bound to the pipeline bind point used by this command accesses a \textit{VkSampler} or \textit{VkImageView} object that enables \textit{sampler Y’CbCr} conversion, that object must only be used with \textit{OpImageSample*} or \textit{OpImageSparseSample*} instructions.

If a \textit{VkImage} view is accessed as a result of this command, then the image view’s \textit{viewType} must match the \textit{Dim} operand of the \textit{OpTypeImage} as described in Instruction/Sampler/Image View Validation.

If a \textit{VkImageView} is accessed as a result of this command, then the image view’s \textit{format} must match the numeric format from the \textit{Sampled Type} operand of the \textit{OpTypeImage} as described in the SPIR-V Sampled Type column of the Interpretation of Numeric Format table.

If a \textit{VkImageView} is accessed using \textit{OpImageWrite} as a result of this command, then the \textit{Type} of the \textit{Texel} operand of that instruction must have at least as many components as the image view’s format.

If a \textit{VkBufferView} is accessed using \textit{OpImageWrite} as a result of this command, then the \textit{Type} of the \textit{Texel} operand of that instruction must have at least as many components as the buffer view’s format.
Any shader invocation executed by this command must terminate.

The current render pass must be compatible with the renderPass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS.

The subpass index of the current render pass must be equal to the subpass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS.

If any shader statically accesses an input attachment, a valid descriptor must be bound to the pipeline via a descriptor set.

If any shader executed by this pipeline accesses an OpTypeImage variable with a Dim operand of SubpassData, it must be decorated with an InputAttachmentIndex that corresponds to a valid input attachment in the current subpass.

Input attachment views accessed in a subpass must be created with the same VkFormat as the corresponding subpass definition, and be created with a VkImageView that is compatible with the attachment referenced by the subpass' pInputAttachments[InputAttachmentIndex] in the currently bound VkFramebuffer as specified by Fragment Input Attachment Compatibility.

Memory backing image subresources used as attachments in the current render pass must not be written in any way other than as an attachment by this command.

If any recorded command in the current subpass will write to an image subresource as an attachment, this command must not read from the memory backing that image subresource in any other way than as an attachment.

If any recorded command in the current subpass will read from an image subresource used as an attachment in any way other than as an attachment, this command must not write to that image subresource as an attachment.

If the current render pass instance uses a depth/stencil attachment with a read-only layout for the depth aspect, depth writes must be disabled.

If the current render pass instance uses a depth/stencil attachment with a read-only layout for the stencil aspect, both front and back writeMask are not zero, and stencil test is enabled, all stencil ops must be VK_STENCIL_OP_KEEP.

If the draw is recorded in a render pass instance with multiview enabled, the maximum
instance index must be less than or equal to VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex

- VUID-vkCmdDrawIndirectCount-viewportCount-03417
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT dynamic state enabled, but not the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT dynamic state enabled, then vkCmdSetViewportWithCount must have been called in the current command buffer prior to this drawing command, and the viewportCount parameter of vkCmdSetViewportWithCount must match the VkPipelineViewportStateCreateInfo::viewportCount of the pipeline

- VUID-vkCmdDrawIndirectCount-scissorCount-03418
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT dynamic state enabled, but not the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT dynamic state enabled, then vkCmdSetScissorWithCount must have been called in the current command buffer prior to this drawing command, and the scissorCount parameter of vkCmdSetScissorWithCount must match the VkPipelineViewportStateCreateInfo::viewportCount of the pipeline

- VUID-vkCmdDrawIndirectCount-viewportCount-03419
  If the bound graphics pipeline state was created with both the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT and VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT dynamic states enabled then both vkCmdSetViewportWithCount and vkCmdSetScissorWithCount must have been called in the current command buffer prior to this drawing command, and the viewportCount parameter of vkCmdSetViewportWithCount must match the scissorCount parameter of vkCmdSetScissorWithCount

- VUID-vkCmdDrawIndirectCount-None-04876
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE dynamic state enabled then vkCmdSetRasterizerDiscardEnable must have been called in the current command buffer prior to this drawing command

- VUID-vkCmdDrawIndirectCount-None-04877
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE dynamic state enabled then vkCmdSetDepthBiasEnable must have been called in the current command buffer prior to this drawing command

- VUID-vkCmdDrawIndirectCount-blendEnable-04727
  If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view’s format features do not contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT, then the blendEnable member of the corresponding element of the pAttachments member of pColorBlendState must be VK_FALSE

- VUID-vkCmdDrawIndirectCount-rasterizationSamples-04740
  If rasterization is not disabled in the bound graphics pipeline, and neither the VK_AMD_mixed_attachment_samples nor the VK_NV_framebuffer_mixed_samples extensions are enabled, then VkPipelineMultisampleStateCreateInfo::rasterizationSamples must be the same as the current subpass color and/or depth/stencil attachments
If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView` member of `pDepthAttachment` is not `VK_NULL_HANDLE`, and the `layout` member of `pDepthAttachment` is `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, this command **must** not write any values to the depth attachment.

If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView` member of `pStencilAttachment` is not `VK_NULL_HANDLE`, and the `layout` member of `pStencilAttachment` is `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, this command **must** not write any values to the stencil attachment.

If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView` member of `pDepthAttachment` is not `VK_NULL_HANDLE`, and the `layout` member of `pDepthAttachment` is `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`, this command **must** not write any values to the depth attachment.

If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView` member of `pStencilAttachment` is not `VK_NULL_HANDLE`, and the `layout` member of `pStencilAttachment` is `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, this command **must** not write any values to the stencil attachment.

If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView` member of `pDepthAttachment` is not `VK_NULL_HANDLE`, and the `layout` member of `pDepthAttachment` is `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, this command **must** not write any values to the depth attachment.

If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView` member of `pStencilAttachment` is not `VK_NULL_HANDLE`, and the `layout` member of `pStencilAttachment` is `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`, this command **must** not write any values to the stencil attachment.

If the current render pass instance was begun with `vkCmdBeginRendering`, the currently bound graphics pipeline **must** have been created with a `VkPipelineRenderingCreateInfo`::`viewMask` equal to `VkRenderingInfo`::`viewMask`.

If the current render pass instance was begun with `vkCmdBeginRendering`, the currently bound graphics pipeline **must** have been created with a `VkPipelineRenderingCreateInfo`::`colorAttachmentCount` equal to `VkRenderingInfo`::`colorAttachmentCount`.

If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo`::`colorAttachmentCount` greater than 0, then each element of the `VkRenderingInfo`::`pColorAttachments` array with a `imageView` not equal to `VK_NULL_HANDLE` **must** have been created with a `VkFormat` equal to the corresponding element of `VkPipelineRenderingCreateInfo`::`pColorAttachmentFormats` used to create the currently bound graphics pipeline.
If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo::colorAttachmentCount` greater than 0, then each element of the `VkRenderingInfo::pColorAttachments` array with a `imageView` equal to `VK_NULL_HANDLE` must have the corresponding element of `VkPipelineRenderingCreateInfo::pColorAttachmentFormats` used to create the currently bound pipeline equal to `VK_FORMAT_UNDEFINED`.

If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo::pDepthAttachment->imageView` was not `VK_NULL_HANDLE`, the value of `VkPipelineRenderingCreateInfo::depthAttachmentFormat` used to create the currently bound graphics pipeline must be equal to the `VkFormat` used to create `VkRenderingInfo::pDepthAttachment->imageView`.

If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo::pDepthAttachment->imageView` was `VK_NULL_HANDLE`, the value of `VkPipelineRenderingCreateInfo::depthAttachmentFormat` used to create the currently bound graphics pipeline must be equal to `VK_FORMAT_UNDEFINED`.

If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo::pStencilAttachment->imageView` was not `VK_NULL_HANDLE`, the value of `VkPipelineRenderingCreateInfo::stencilAttachmentFormat` used to create the currently bound graphics pipeline must be equal to the `VkFormat` used to create `VkRenderingInfo::pStencilAttachment->imageView`.

If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo::pStencilAttachment->imageView` was `VK_NULL_HANDLE`, the value of `VkPipelineRenderingCreateInfo::stencilAttachmentFormat` used to create the currently bound graphics pipeline must be equal to `VK_FORMAT_UNDEFINED`.

All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must have either valid or `VK_NULL_HANDLE` buffers bound.

If the `nullDescriptor` feature is not enabled, all vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must not be `VK_NULL_HANDLE`.

For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in `Vertex Input Description`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` dynamic state enabled then `vkCmdSetPrimitiveTopology` must have been called in the current command buffer prior
to this drawing command, and the \texttt{primitiveTopology} parameter of \texttt{vkCmdSetPrimitiveTopology} \textbf{must} be of the same \texttt{topology} class as the pipeline \texttt{VkPipelineInputAssemblyStateCreateInfo::topology} state.

- \texttt{VUID-vkCmdDrawIndirectCount-pStrides-04884}
  If the bound graphics pipeline was created with the \texttt{VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT} dynamic state enabled, then \texttt{vkCmdBindVertexBuffers2EXT} \textbf{must} have been called in the current command buffer prior to this drawing command, and the \texttt{pStrides} parameter of \texttt{vkCmdBindVertexBuffers2EXT} \textbf{must} not be \texttt{NULL}.

- \texttt{VUID-vkCmdDrawIndirectCount-None-04879}
  If the bound graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE} dynamic state enabled then \texttt{vkCmdSetPrimitiveRestartEnable} \textbf{must} have been called in the current command buffer prior to this drawing command.

- \texttt{VUID-vkCmdDrawIndirectCount-buffer-02708}
  If \texttt{buffer} is non-sparse then it \textbf{must} be bound completely and contiguously to a single \texttt{VkDeviceMemory} object.

- \texttt{VUID-vkCmdDrawIndirectCount-buffer-02709}
  \texttt{buffer} \textbf{must} have been created with the \texttt{VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT} bit set.

- \texttt{VUID-vkCmdDrawIndirectCount-offset-02710}
  \texttt{offset} \textbf{must} be a multiple of \texttt{4}.

- \texttt{VUID-vkCmdDrawIndirectCount-commandBuffer-02711}
  \texttt{commandBuffer} \textbf{must} not be a protected command buffer.

- \texttt{VUID-vkCmdDrawIndirectCount-countBuffer-02714}
  If \texttt{countBuffer} is non-sparse then it \textbf{must} be bound completely and contiguously to a single \texttt{VkDeviceMemory} object.

- \texttt{VUID-vkCmdDrawIndirectCount-countBuffer-02715}
  \texttt{countBuffer} \textbf{must} have been created with the \texttt{VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT} bit set.

- \texttt{VUID-vkCmdDrawIndirectCount-countBufferOffset-02716}
  \texttt{countBufferOffset} \textbf{must} be a multiple of \texttt{4}.

- \texttt{VUID-vkCmdDrawIndirectCount-countBuffer-02717}
  The count stored in \texttt{countBuffer} \textbf{must} be less than or equal to \texttt{VkPhysicalDeviceLimits::maxDrawIndirectCount}.

- \texttt{VUID-vkCmdDrawIndirectCount-countBufferOffset-04129}
  \texttt{(countBufferOffset + sizeof(uint32_t))} \textbf{must} be less than or equal to the size of \texttt{countBuffer}.

- \texttt{VUID-vkCmdDrawIndirectCount-None-04445}
  If \texttt{drawIndirectCount} is not enabled this function \textbf{must} not be used.

- \texttt{VUID-vkCmdDrawIndirectCount-stride-03110}
  \texttt{stride} \textbf{must} be a multiple of \texttt{4} and \textbf{must} be greater than or equal to \texttt{sizeof(VkDrawIndirectCommand)}.

- \texttt{VUID-vkCmdDrawIndirectCount-maxDrawCount-03111}
If \( \text{maxDrawCount} \) is greater than or equal to 1, \((\text{stride} \times (\text{maxDrawCount} - 1) + \text{offset} + \text{sizeof}(\text{VkDrawIndirectCommand})) \) must be less than or equal to the size of buffer

- VUID-vkCmdDrawIndirectCount-countBuffer-03121
  If the count stored in countBuffer is equal to 1, \((\text{offset} + \text{sizeof}(\text{VkDrawIndirectCommand})) \) must be less than or equal to the size of buffer

- VUID-vkCmdDrawIndirectCount-countBuffer-03122
  If the count stored in countBuffer is greater than 1, \((\text{stride} \times (\text{drawCount} - 1) + \text{offset} + \text{sizeof}(\text{VkDrawIndirectCommand})) \) must be less than or equal to the size of buffer

Valid Usage (Implicit)

- VUID-vkCmdDrawIndirectCount-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdDrawIndirectCount-buffer-parameter
  buffer must be a valid VkBuffer handle

- VUID-vkCmdDrawIndirectCount-countBuffer-parameter
  countBuffer must be a valid VkBuffer handle

- VUID-vkCmdDrawIndirectCount-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdDrawIndirectCount-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

- VUID-vkCmdDrawIndirectCount-renderpass
  This command must only be called inside of a render pass instance

- VUID-vkCmdDrawIndirectCount-commonparent
  Each of buffer, commandBuffer, and countBuffer must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

- Host access to commandBuffer must be externally synchronized

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
To record an indexed indirect drawing command, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdDrawIndexedIndirect(
    VkCommandBuffer commandBuffer,
    VkBuffer buffer,
    VkDeviceSize offset,
    uint32_t drawCount,
    uint32_t stride);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `buffer` is the buffer containing draw parameters.
- `offset` is the byte offset into `buffer` where parameters begin.
- `drawCount` is the number of draws to execute, and can be zero.
- `stride` is the byte stride between successive sets of draw parameters.

`vkCmdDrawIndexedIndirect` behaves similarly to `vkCmdDrawIndexed` except that the parameters are read by the device from a buffer during execution. `drawCount` draws are executed by the command, with parameters taken from `buffer` starting at `offset` and increasing by `stride` bytes for each successive draw. The parameters of each draw are encoded in an array of `VkDrawIndexedIndirectCommand` structures. If `drawCount` is less than or equal to one, `stride` is ignored.

### Valid Usage

- **VUID-vkCmdDrawIndexedIndirect-magFilter-04553**
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view’s format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDrawIndexedIndirect-mipmapMode-04770**
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view’s format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDrawIndexedIndirect-None-06479**
  If a `VkImageView` is sampled with depth comparison, the image view’s format features **must** contain `VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_DEPTH_COMPARISON_BIT`

- **VUID-vkCmdDrawIndexedIndirect-None-02691**
  If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view’s format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`

- **VUID-vkCmdDrawIndexedIndirect-OpTypeImage-07027**
  For any `VkImageView` being written as a storage image where the image format field of the `OpTypeImage` is `Unknown`, the view’s format features **must** contain
For any `VkImageView` being read as a storage image where the image format field of the `OpTypeImage` is `Unknown`, the view’s format features must contain `VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT`.

For any `VkBufferView` being written as a storage texel buffer where the image format field of the `OpTypeImage` is `Unknown`, the view’s buffer features must contain `VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT`.

Any `VkBufferView` being read as a storage texel buffer where the image format field of the `OpTypeImage` is `Unknown` then the view's buffer features must contain `VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT`.

For each set `n` that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set must have been bound to `n` at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set `n`, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility.

If the `maintenance4` feature is not enabled, then for each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility.

Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid as described by descriptor validity if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

A valid pipeline must be bound to the pipeline bind point used by this command.

If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited (if the `VK_NV_inherited_viewport_scissor` extension is enabled) for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`.
VK_IMAGE_VIEW_TYPE_CUBE, VK_IMAGE_VIEW_TYPE_1D_ARRAY, VK_IMAGE_VIEW_TYPE_2D_ARRAY or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY, in any shader stage

- VUID-vkCmdDrawIndexedIndirect-None-02703
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions with ImplicitLod, Dref or Proj in their name, in any shader stage

- VUID-vkCmdDrawIndexedIndirect-None-02704
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions that includes a LOD bias or any offset values, in any shader stage

- VUID-vkCmdDrawIndexedIndirect-None-02705
  If the robustBufferAccess feature is not enabled, and if the VkPipeline object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point

- VUID-vkCmdDrawIndexedIndirect-None-02706
  If the robustBufferAccess feature is not enabled, and if the VkPipeline object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point

- VUID-vkCmdDrawIndexedIndirect-commandBuffer-02707
  If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, any resource accessed by the VkPipeline object bound to the pipeline bind point used by this command must not be a protected resource

- VUID-vkCmdDrawIndexedIndirect-None-06550
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler or VkImageView object that enables sampler Y’CbCr conversion, that object must only be used with OpImageSample* or OpImageSparseSample* instructions

- VUID-vkCmdDrawIndexedIndirect-ConstOffset-06551
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler or VkImageView object that enables sampler Y’CbCr conversion, that object must not use the ConstOffset and Offset operands

- VUID-vkCmdDrawIndexedIndirect-viewType-07752
  If a VkImageView is accessed as a result of this command, then the image view’s viewType must match the Dim operand of the OpTypeImage as described in Instruction/Sampler/Image View Validation

- VUID-vkCmdDrawIndexedIndirect-format-07753
  If a VkImageView is accessed as a result of this command, then the image view’s format must match the numeric format from the Sampled Type operand of the OpTypeImage as described in the SPIR-V Sampled Type column of the Interpretation of Numeric Format table

- VUID-vkCmdDrawIndexedIndirect-None-04115
If a VkImageView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view’s format

- VUID-vkCmdDrawIndexedIndirect-OpImageWrite-04469
  If a VkBufferView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the buffer view’s format

- VUID-vkCmdDrawIndexedIndirect-None-07288
  Any shader invocation executed by this command must terminate

- VUID-vkCmdDrawIndexedIndirect-renderPass-02684
  The current render pass must be compatible with the renderPass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS

- VUID-vkCmdDrawIndexedIndirect-subpass-02685
  The subpass index of the current render pass must be equal to the subpass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS

- VUID-vkCmdDrawIndexedIndirect-None-07748
  If any shader statically accesses an input attachment, a valid descriptor must be bound to the pipeline via a descriptor set

- VUID-vkCmdDrawIndexedIndirect-OpTypeImage-07468
  If any shader executed by this pipeline accesses an OpTypeImage variable with a Dim operand of SubpassData, it must be decorated with an InputAttachmentIndex that corresponds to a valid input attachment in the current subpass

- VUID-vkCmdDrawIndexedIndirect-None-07469
  Input attachment views accessed in a subpass must be created with the same VkFormat as the corresponding subpass definition, and be created with a VkImageView that is compatible with the attachment referenced by the subpass’ pInputAttachments [InputAttachmentIndex] in the currently bound VkFramebuffer as specified by Fragment Input Attachment Compatibility

- VUID-vkCmdDrawIndexedIndirect-None-06537
  Memory backing image subresources used as attachments in the current render pass must not be written in any way other than as an attachment by this command

- VUID-vkCmdDrawIndexedIndirect-None-06538
  If any recorded command in the current subpass will write to an image subresource as an attachment, this command must not read from the memory backing that image subresource in any other way than as an attachment

- VUID-vkCmdDrawIndexedIndirect-None-06539
  If any recorded command in the current subpass will read from an image subresource used as an attachment in any way other than as an attachment, this command must not write to that image subresource as an attachment

- VUID-vkCmdDrawIndexedIndirect-None-06886
  If the current render pass instance uses a depth/stencil attachment with a read-only
layout for the depth aspect, **depth writes must be disabled**

- **VUID-vkCmdDrawIndexedIndirect-None-06887**
  If the current render pass instance uses a depth/stencil attachment with a read-only layout for the stencil aspect, both front and back writeMask are not zero, and stencil test is enabled, **all stencil ops writeMask must be VK_STENCIL_OP_KEEP**

- **VUID-vkCmdDrawIndexedIndirect-maxMultiviewInstanceIndex-02688**
  If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must be less than or equal to VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex**

- **VUID-vkCmdDrawIndexedIndirect-viewportCount-03417**
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT dynamic state enabled, but not the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT dynamic state enabled, then `vkCmdSetViewportWithCount` **must** have been called in the current command buffer prior to this drawing command, and the `viewportCount` parameter of `vkCmdSetViewportWithCount` **must** match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline

- **VUID-vkCmdDrawIndexedIndirect-scissorCount-03418**
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT dynamic state enabled, but not the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT dynamic state enabled, then `vkCmdSetScissorWithCount` **must** have been called in the current command buffer prior to this drawing command, and the `scissorCount` parameter of `vkCmdSetScissorWithCount` **must** match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline

- **VUID-vkCmdDrawIndexedIndirect-viewportCount-03419**
  If the bound graphics pipeline state was created with both the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT and VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT dynamic states enabled then both `vkCmdSetViewportWithCount` and `vkCmdSetScissorWithCount` **must** have been called in the current command buffer prior to this drawing command, and the `viewportCount` parameter of `vkCmdSetViewportWithCount` **must** match the `scissorCount` parameter of `vkCmdSetScissorWithCount`

- **VUID-vkCmdDrawIndexedIndirect-None-04876**
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE dynamic state enabled then `vkCmdSetRasterizerDiscardEnable` **must** have been called in the current command buffer prior to this drawing command

- **VUID-vkCmdDrawIndexedIndirect-None-04877**
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE dynamic state enabled then `vkCmdSetDepthBiasEnable` **must** have been called in the current command buffer prior to this drawing command

- **VUID-vkCmdDrawIndexedIndirect-blendEnable-04727**
  If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view’s format features do not contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT, then the blendEnable member of
the corresponding element of the \texttt{pAttachments} member of \texttt{pColorBlendState} must be \texttt{VK_FALSE}

- VUID-vkCmdDrawIndexedIndirect-rasterizationSamples-04740
  If rasterization is not disabled in the bound graphics pipeline, and neither the \texttt{VK_AMD_mixed_attachment_samples} nor the \texttt{VK_NV_framebuffer_mixed_samples} extensions are enabled, then \texttt{VkPipelineMultisampleStateCreateInfo::rasterizationSamples} must be the same as the current subpass color and/or depth/stencil attachments

- VUID-vkCmdDrawIndexedIndirect-imageView-06172
  If the current render pass instance was begun with \texttt{vkCmdBeginRendering}, the \texttt{imageView} member of \texttt{pDepthAttachment} is not \texttt{VK_NULL_HANDLE}, and the layout member of \texttt{pDepthAttachment} is \texttt{VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL}, this command must not write any values to the depth attachment

- VUID-vkCmdDrawIndexedIndirect-imageView-06173
  If the current render pass instance was begun with \texttt{vkCmdBeginRendering}, the \texttt{imageView} member of \texttt{pStencilAttachment} is not \texttt{VK_NULL_HANDLE}, and the layout member of \texttt{pStencilAttachment} is \texttt{VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL}, this command must not write any values to the stencil attachment

- VUID-vkCmdDrawIndexedIndirect-imageView-06174
  If the current render pass instance was begun with \texttt{vkCmdBeginRendering}, the \texttt{imageView} member of \texttt{pDepthAttachment} is not \texttt{VK_NULL_HANDLE}, and the layout member of \texttt{pDepthAttachment} is \texttt{VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL}, this command must not write any values to the stencil attachment

- VUID-vkCmdDrawIndexedIndirect-imageView-06175
  If the current render pass instance was begun with \texttt{vkCmdBeginRendering}, the \texttt{imageView} member of \texttt{pStencilAttachment} is not \texttt{VK_NULL_HANDLE}, and the layout member of \texttt{pStencilAttachment} is \texttt{VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL}, this command must not write any values to the stencil attachment

- VUID-vkCmdDrawIndexedIndirect-viewMask-06178
  If the current render pass instance was begun with \texttt{vkCmdBeginRendering}, the currently bound graphics pipeline must have been created with a \texttt{VkPipelineRenderingCreateInfo::viewMask} equal to \texttt{VkRenderingInfo::viewMask}

- VUID-vkCmdDrawIndexedIndirect-colorAttachmentCount-06179
  If the current render pass instance was begun with \texttt{vkCmdBeginRendering}, the currently bound graphics pipeline must have been created with a \texttt{VkPipelineRenderingCreateInfo::viewMask}
::colorAttachmentCount equal to VkRenderingInfo::colorAttachmentCount

• VUID-vkCmdDrawIndexedIndirect-colorAttachmentCount-06180
  If the current render pass instance was begun with vkCmdBeginRendering and VkRenderingInfo::colorAttachmentCount greater than 0, then each element of the VkRenderingInfo::pColorAttachments array with a imageView not equal to VK_NULL_HANDLE must have been created with a VkFormat equal to the corresponding element of VkPipelineRenderingCreateInfo::pColorAttachmentFormats used to create the currently bound graphics pipeline

• VUID-vkCmdDrawIndexedIndirect-colorAttachmentCount-07616
  If the current render pass instance was begun with vkCmdBeginRendering and VkRenderingInfo::colorAttachmentCount greater than 0, then each element of the VkRenderingInfo::pColorAttachments array with a imageView equal to VK_NULL_HANDLE must have the corresponding element of VkPipelineRenderingCreateInfo::pColorAttachmentFormats used to create the currently bound pipeline equal to VK_FORMAT_UNDEFINED

• VUID-vkCmdDrawIndexedIndirect-pDepthAttachment-06181
  If the current render pass instance was begun with vkCmdBeginRendering and VkRenderingInfo::pDepthAttachment->imageView was not VK_NULL_HANDLE, the value of VkPipelineRenderingCreateInfo::depthAttachmentFormat used to create the currently bound graphics pipeline must be equal to the VkFormat used to create VkRenderingInfo::pDepthAttachment->imageView

• VUID-vkCmdDrawIndexedIndirect-pDepthAttachment-07617
  If the current render pass instance was begun with vkCmdBeginRendering and VkRenderingInfo::pDepthAttachment->imageView was VK_NULL_HANDLE, the value of VkPipelineRenderingCreateInfo::depthAttachmentFormat used to create the currently bound graphics pipeline must be equal to VK_FORMAT_UNDEFINED

• VUID-vkCmdDrawIndexedIndirect-pStencilAttachment-06182
  If the current render pass instance was begun with vkCmdBeginRendering and VkRenderingInfo::pStencilAttachment->imageView was not VK_NULL_HANDLE, the value of VkPipelineRenderingCreateInfo::stencilAttachmentFormat used to create the currently bound graphics pipeline must be equal to the VkFormat used to create VkRenderingInfo::pStencilAttachment->imageView

• VUID-vkCmdDrawIndexedIndirect-pStencilAttachment-07618
  If the current render pass instance was begun with vkCmdBeginRendering and VkRenderingInfo::pStencilAttachment->imageView was VK_NULL_HANDLE, the value of VkPipelineRenderingCreateInfo::stencilAttachmentFormat used to create the currently bound graphics pipeline must be equal to VK_FORMAT_UNDEFINED

• VUID-vkCmdDrawIndexedIndirect-None-04007
  All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must have either valid or VK_NULL_HANDLE buffers bound

• VUID-vkCmdDrawIndexedIndirect-None-04008
  If the nullDescriptor feature is not enabled, all vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must not be
For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in Vertex Input Description.

If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT dynamic state enabled then vkCmdSetPrimitiveTopology must have been called in the current command buffer prior to this drawing command, and the primitiveTopology parameter of vkCmdSetPrimitiveTopology must be of the same topology class as the pipeline VkPipelineInputAssemblyStateCreateInfo::topology state.

If the bound graphics pipeline was created with the VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT dynamic state enabled, then vkCmdBindVertexBuffers2EXT must have been called in the current command buffer prior to this drawing command, and the pStrides parameter of vkCmdBindVertexBuffers2EXT must not be NULL.

If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE dynamic state enabled then vkCmdSetPrimitiveRestartEnable must have been called in the current command buffer prior to this drawing command.

If buffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object.

buffer must have been created with the VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT bit set.

offset must be a multiple of 4.

commandBuffer must not be a protected command buffer.

If the multiDrawIndirect feature is not enabled, drawCount must be 0 or 1.

drawCount must be less than or equal to VkPhysicalDeviceLimits::maxDrawIndirectCount.

An index buffer must be bound.

If drawCount is greater than 1, stride must be a multiple of 4 and must be greater than or equal to sizeof(VkDrawIndexedIndirectCommand).

An index buffer must be bound.
If `drawCount` is equal to 1, `(offset + sizeof(VkDrawIndexedIndirectCommand))` must be less than or equal to the size of `buffer`.

- VUID-vkCmdDrawIndexedIndirect-drawCount-00540
  If `drawCount` is greater than 1, `stride × (drawCount - 1) + offset + sizeof(VkDrawIndexedIndirectCommand))` must be less than or equal to the size of `buffer`.

## Valid Usage (Implicit)

- VUID-vkCmdDrawIndexedIndirect-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle.

- VUID-vkCmdDrawIndexedIndirect-buffer-parameter
  `buffer` must be a valid `VkBuffer` handle.

- VUID-vkCmdDrawIndexedIndirect-commandBuffer-recording
  `commandBuffer` must be in the recording state.

- VUID-vkCmdDrawIndexedIndirect-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations.

- VUID-vkCmdDrawIndexedIndirect-renderpass
  This command must only be called inside of a render pass instance.

- VUID-vkCmdDrawIndexedIndirect-commonparent
  Both of `buffer`, and `commandBuffer` must have been created, allocated, or retrieved from the same `VkDevice`.

## Host Synchronization

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

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

## Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkDrawIndexedIndirectCommand` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDrawIndexedIndirectCommand {
```
uint32_t indexCount;
uint32_t instanceCount;
uint32_t firstIndex;
int32_t vertexOffset;
uint32_t firstInstance;
} VkDrawIndexedIndirectCommand;

• indexCount is the number of vertices to draw.
• instanceCount is the number of instances to draw.
• firstIndex is the base index within the index buffer.
• vertexOffset is the value added to the vertex index before indexing into the vertex buffer.
• firstInstance is the instance ID of the first instance to draw.

The members of VkDrawIndexedIndirectCommand have the same meaning as the similarly named parameters of vkCmdDrawIndexed.

**Valid Usage**

- VUID-VkDrawIndexedIndirectCommand-None-00552
  For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in Vertex Input Description

- VUID-VkDrawIndexedIndirectCommand-indexSize-00553
  (indexSize × (firstIndex + indexCount) + offset) must be less than or equal to the size of the bound index buffer, with indexSize being based on the type specified by indexType, where the index buffer, indexType, and offset are specified via vkCmdBindIndexBuffer

- VUID-VkDrawIndexedIndirectCommand-firstInstance-00554
  If the drawIndirectFirstInstance feature is not enabled, firstInstance must be 0

To record an indexed draw call with a draw call count sourced from a buffer, call:

```c
// Provided by VK_VERSION_1_2
void vkCmdDrawIndexedIndirectCount(
    VkCommandBuffer commandBuffer,  // Provided by VK_VERSION_1_2
    VkBuffer buffer,  // Provided by VK_VERSION_1_2
    VkDeviceSize offset,  // Provided by VK_VERSION_1_2
    VkBuffer countBuffer,  // Provided by VK_VERSION_1_2
    VkDeviceSize countBufferOffset,  // Provided by VK_VERSION_1_2
    uint32_t maxDrawCount,  // Provided by VK_VERSION_1_2
    uint32_t stride);
```

- commandBuffer is the command buffer into which the command is recorded.
- buffer is the buffer containing draw parameters.
- offset is the byte offset into buffer where parameters begin.
• **countBuffer** is the buffer containing the draw count.

• **countBufferOffset** is the byte offset into **countBuffer** where the draw count begins.

• **maxDrawCount** specifies the maximum number of draws that will be executed. The actual number of executed draw calls is the minimum of the count specified in **countBuffer** and **maxDrawCount**.

• **stride** is the byte stride between successive sets of draw parameters.

`vkCmdDrawIndexedIndirectCount` behaves similarly to `vkCmdDrawIndexedIndirect` except that the draw count is read by the device from a buffer during execution. The command will read an unsigned 32-bit integer from **countBuffer** located at **countBufferOffset** and use this as the draw count.

### Valid Usage

- **VUID-vkCmdDrawIndexedIndirectCount-magFilter-04553**
  If a **VkSampler** created with **magFilter** or **minFilter** equal to **VK_FILTER_LINEAR** and **compareEnable** equal to **VK_FALSE** is used to sample a **VkImageView** as a result of this command, then the image view’s **format features** **must** contain **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT**

- **VUID-vkCmdDrawIndexedIndirectCount-mipmapMode-04770**
  If a **VkSampler** created with **mipmapMode** equal to **VK_SAMPLER_MIPMAP_MODE_LINEAR** and **compareEnable** equal to **VK_FALSE** is used to sample a **VkImageView** as a result of this command, then the image view’s **format features** **must** contain **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT**

- **VUID-vkCmdDrawIndexedIndirectCount-None-06479**
  If a **VkImageView** is sampled with **depth comparison**, the image view’s **format features** **must** contain **VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_DEPTH_COMPARISON_BIT**

- **VUID-vkCmdDrawIndexedIndirectCount-None-02691**
  If a **VkImageView** is accessed using atomic operations as a result of this command, then the image view’s **format features** **must** contain **VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT**

- **VUID-vkCmdDrawIndexedIndirectCount-OpTypeImage-07027**
  For any **VkImageView** being written as a storage image where the image format field of the **OpTypeImage** is **Unknown**, the view’s **format features** **must** contain **VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT**

- **VUID-vkCmdDrawIndexedIndirectCount-OpTypeImage-07028**
  For any **VkImageView** being read as a storage image where the image format field of the **OpTypeImage** is **Unknown**, the view’s **format features** **must** contain **VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT**

- **VUID-vkCmdDrawIndexedIndirectCount-OpTypeImage-07029**
  For any **VkBufferView** being written as a storage texel buffer where the image format field of the **OpTypeImage** is **Unknown**, the view’s **buffer features** **must** contain **VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT**

- **VUID-vkCmdDrawIndexedIndirectCount-OpTypeImage-07030**
  Any **VkBufferView** being read as a storage texel buffer where the image format field of the **OpTypeImage** is **Unknown** then the view’s **buffer features** **must** contain
For each set \( n \) that is statically used by the \( \text{VkPipeline} \) bound to the pipeline bind point used by this command, a descriptor set must have been bound to \( n \) at the same pipeline bind point, with a \( \text{VkPipelineLayout} \) that is compatible for set \( n \), with the \( \text{VkPipelineLayout} \) used to create the current \( \text{VkPipeline} \), as described in Pipeline Layout Compatibility.

If the \texttt{maintenance4} feature is not enabled, then for each push constant that is statically used by the \( \text{VkPipeline} \) bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a \( \text{VkPipelineLayout} \) that is compatible for push constants, with the \( \text{VkPipelineLayout} \) used to create the current \( \text{VkPipeline} \), as described in Pipeline Layout Compatibility.

Descriptors in each bound descriptor set, specified via \texttt{vkCmdBindDescriptorSets}, must be valid as described by descriptor validity if they are statically used by the \( \text{VkPipeline} \) bound to the pipeline bind point used by this command.

A valid pipeline must be bound to the pipeline bind point used by this command.

If the \( \text{VkPipeline} \) object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited (if the \texttt{VK_NV_inherited_viewport_scissor} extension is enabled) for \( \text{commandBuffer} \), and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the \( \text{VkPipeline} \) object bound to the pipeline bind point used by this command, since that pipeline was bound.

If the \( \text{VkPipeline} \) object bound to the pipeline bind point used by this command accesses a \( \text{VkSampler} \) object that uses unnormalized coordinates, that sampler must not be used to sample from any \( \text{VkImage} \) with a \( \text{VkImageView} \) of the type \texttt{VK_IMAGE_VIEW_TYPE_3D}, \texttt{VK_IMAGE_VIEW_TYPE_CUBE}, \texttt{VK_IMAGE_VIEW_TYPE_1D_ARRAY}, \texttt{VK_IMAGE_VIEW_TYPE_2D_ARRAY} or \texttt{VK_IMAGE_VIEW_TYPE_CUBE_ARRAY}, in any shader stage.

If the \( \text{VkPipeline} \) object bound to the pipeline bind point used by this command accesses a \( \text{VkSampler} \) object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V \texttt{OpImageSample*} or \texttt{OpImageSparseSample*} instructions with \texttt{ImplicitLod}, \texttt{Dref} or \texttt{Proj} in their name, in any shader stage.

If the \( \text{VkPipeline} \) object bound to the pipeline bind point used by this command accesses a \( \text{VkSampler} \) object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V \texttt{OpImageSample*} or \texttt{OpImageSparseSample*} instructions that includes a LOD bias or any offset values, in any shader stage.
If the `robustBufferAccess` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

If the `robustBufferAccess` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command must not be a protected resource.

If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` or `VkImageView` object that enables `sampler YCbCr conversion`, that object must only be used with `OpImageSample*` or `OpImageSparseSample*` instructions.

If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` or `VkImageView` object that enables `sampler YCbCr conversion`, that object must not use the `ConstOffset` and `Offset` operands.

If a `VkImageView` is accessed as a result of this command, then the image view’s `viewType` must match the `Dim` operand of the `OpTypeImage` as described in Instruction/Sampler/Image View Validation.

If a `VkImageView` is accessed as a result of this command, then the image view’s `format` must match the numeric format from the `Sampled Type` operand of the `OpTypeImage` as described in the SPIR-V Sampled Type column of the Interpretation of Numeric Format table.

If a `VkImageView` is accessed using `OpImageWrite` as a result of this command, then the `Type` of the `Texel` operand of that instruction must have at least as many components as the image view’s `format`.

If a `VkBufferView` is accessed using `OpImageWrite` as a result of this command, then the `Type` of the `Texel` operand of that instruction must have at least as many components as the buffer view’s `format`.

Any shader invocation executed by this command must terminate.

The current render pass must be compatible with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to
VK_PIPELINE_BIND_POINT_GRAPHICS

• VUID-vkCmdDrawIndexedIndirectCount-subpass-02685
  The subpass index of the current render pass must be equal to the subpass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS

• VUID-vkCmdDrawIndexedIndirectCount-None-07748
  If any shader statically accesses an input attachment, a valid descriptor must be bound to the pipeline via a descriptor set

• VUID-vkCmdDrawIndexedIndirectCount-OpTypeImage-07468
  If any shader executed by this pipeline accesses an OpTypeImage variable with a Dim operand of SubpassData, it must be decorated with an InputAttachmentIndex that corresponds to a valid input attachment in the current subpass

• VUID-vkCmdDrawIndexedIndirectCount-None-07469
  Input attachment views accessed in a subpass must be created with the same VkFormat as the corresponding subpass definition, and be created with a VkImageView that is compatible with the attachment referenced by the subpass' pInputAttachments[InputAttachmentIndex] in the currently bound VkFramebuffer as specified by Fragment Input Attachment Compatibility

• VUID-vkCmdDrawIndexedIndirectCount-None-06537
  Memory backing image subresources used as attachments in the current render pass must not be written in any way other than as an attachment by this command

• VUID-vkCmdDrawIndexedIndirectCount-None-06538
  If any recorded command in the current subpass will write to an image subresource as an attachment, this command must not read from the memory backing that image subresource in any other way than as an attachment

• VUID-vkCmdDrawIndexedIndirectCount-None-06539
  If any recorded command in the current subpass will read from an image subresource used as an attachment in any way other than as an attachment, this command must not write to that image subresource as an attachment

• VUID-vkCmdDrawIndexedIndirectCount-None-06886
  If the current render pass instance uses a depth/stencil attachment with a read-only layout for the depth aspect, depth writes must be disabled

• VUID-vkCmdDrawIndexedIndirectCount-None-06887
  If the current render pass instance uses a depth/stencil attachment with a read-only layout for the stencil aspect, both front and back writeMask are not zero, and stencil test is enabled, all stencil ops must be VK_STENCIL_OP_KEEP

• VUID-vkCmdDrawIndexedIndirectCount-maxMultiviewInstanceIndex-02688
  If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index must be less than or equal to VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex

• VUID-vkCmdDrawIndexedIndirectCount-viewportCount-03417
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT dynamic state enabled, but not the
VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT dynamic state enabled, then
vkCmdSetViewportWithCount must have been called in the current command buffer prior to this drawing command, and the viewportCount parameter of vkCmdSetViewportWithCount must match the VkPipelineViewportStateCreateInfo::scissorCount of the pipeline

- VUID-vkCmdDrawIndexedIndirectCount-scissorCount-03418
  If the bound graphics pipeline state was created with the
  VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT dynamic state enabled, but not the
  VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT dynamic state enabled, then
  vkCmdSetScissorWithCount must have been called in the current command buffer prior to this drawing command, and the scissorCount parameter of vkCmdSetScissorWithCount must match the VkPipelineViewportStateCreateInfo::viewportCount of the pipeline

- VUID-vkCmdDrawIndexedIndirectCount-viewportCount-03419
  If the bound graphics pipeline state was created with both the
  VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT and VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT dynamic states enabled then both
  vkCmdSetViewportWithCount and vkCmdSetScissorWithCount
  must have been called in the current command buffer prior to this drawing command, and the viewportCount parameter of vkCmdSetViewportWithCount must match the scissorCount parameter of vkCmdSetScissorWithCount

- VUID-vkCmdDrawIndexedIndirectCount-None-04876
  If the bound graphics pipeline state was created with the
  VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE dynamic state enabled then
  vkCmdSetRasterizerDiscardEnable
  must have been called in the current command buffer prior to this drawing command

- VUID-vkCmdDrawIndexedIndirectCount-None-04877
  If the bound graphics pipeline state was created with the
  VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE dynamic state enabled then
  vkCmdSetDepthBiasEnable
  must have been called in the current command buffer prior to this drawing command

- VUID-vkCmdDrawIndexedIndirectCount-blendEnable-04727
  If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view's format features do not contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT, then the blendEnable member of the corresponding element of the pAttachments member of pColorBlendState must be
  VK_FALSE

- VUID-vkCmdDrawIndexedIndirectCount-rasterizationSamples-04740
  If rasterization is not disabled in the bound graphics pipeline, and neither the
  VK_AMD_mixed_attachment_samples nor the VK_NV_framebuffer_mixed_samples extensions are enabled, then
  VkPipelineMultisampleStateCreateInfo::rasterizationSamples
  must be the same as the current subpass color and/or depth/stencil attachments

- VUID-vkCmdDrawIndexedIndirectCount-imageView-06172
  If the current render pass instance was begun with
  vkCmdBeginRendering, the imageView
  member of pDepthAttachment is not
  VK_NULL_HANDLE, and the layout member of pDepthAttachment is
  VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL, this command must
  not write any values to the depth attachment
If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView` member of `pDepthAttachment` is not `VK_NULL_HANDLE`, and the `layout` member of `pDepthAttachment` is `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, this command must not write any values to the stencil attachment.

If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView` member of `pDepthAttachment` is not `VK_NULL_HANDLE`, and the `layout` member of `pDepthAttachment` is `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, this command must not write any values to the stencil attachment.

If the current render pass instance was begun with `vkCmdBeginRendering`, the `imageView` member of `pDepthAttachment` is not `VK_NULL_HANDLE`, and the `layout` member of `pDepthAttachment` is `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`, this command must not write any values to the stencil attachment.

If the current render pass instance was begun with `vkCmdBeginRendering`, the currently bound graphics pipeline must have been created with a `VkPipelineRenderingCreateInfo`::`viewMask` equal to `VkRenderingInfo`::`viewMask`.

If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo`::`colorAttachmentCount` greater than 0, then each element of the `VkRenderingInfo`::`pColorAttachments` array with an `imageView` not equal to `VK_NULL_HANDLE` must have been created with a `VkFormat` equal to the corresponding element of `VkPipelineRenderingCreateInfo`::`pColorAttachmentFormats` used to create the currently bound graphics pipeline.

If the current render pass instance was begun with `vkCmdBeginRendering` and `VkRenderingInfo`::`colorAttachmentCount` greater than 0, then each element of the `VkRenderingInfo`::`pColorAttachments` array with an `imageView` equal to `VK_NULL_HANDLE` must have the corresponding element of `VkPipelineRenderingCreateInfo`::`pColorAttachmentFormats`.
::pColorAttachmentFormats used to create the currently bound pipeline equal to VK_FORMAT_UNDEFINED

- VUID-vkCmdDrawIndexedIndirectCount-pDepthAttachment-06181
  If the current render pass instance was begun with vkCmdBeginRendering and VkRenderingInfo::pDepthAttachment->imageView was not VK_NULL_HANDLE, the value of VkPipelineRenderingCreateInfo::depthAttachmentFormat used to create the currently bound graphics pipeline must be equal to the VkFormat used to create VkRenderingInfo::pDepthAttachment->imageView

- VUID-vkCmdDrawIndexedIndirectCount-pDepthAttachment-07617
  If the current render pass instance was begun with vkCmdBeginRendering and VkRenderingInfo::pDepthAttachment->imageView was VK_NULL_HANDLE, the value of VkPipelineRenderingCreateInfo::depthAttachmentFormat used to create the currently bound graphics pipeline must be equal to VK_FORMAT_UNDEFINED

- VUID-vkCmdDrawIndexedIndirectCount-pStencilAttachment-06182
  If the current render pass instance was begun with vkCmdBeginRendering and VkRenderingInfo::pStencilAttachment->imageView was not VK_NULL_HANDLE, the value of VkPipelineRenderingCreateInfo::stencilAttachmentFormat used to create the currently bound graphics pipeline must be equal to the VkFormat used to create VkRenderingInfo::pStencilAttachment->imageView

- VUID-vkCmdDrawIndexedIndirectCount-pStencilAttachment-07618
  If the current render pass instance was begun with vkCmdBeginRendering and VkRenderingInfo::pStencilAttachment->imageView was VK_NULL_HANDLE, the value of VkPipelineRenderingCreateInfo::stencilAttachmentFormat used to create the currently bound graphics pipeline must be equal to VK_FORMAT_UNDEFINED

- VUID-vkCmdDrawIndexedIndirectCount-None-04007
  All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must have either valid or VK_NULL_HANDLE buffers bound

- VUID-vkCmdDrawIndexedIndirectCount-None-04008
  If the nullDescriptor feature is not enabled, all vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must not be VK_NULL_HANDLE

- VUID-vkCmdDrawIndexedIndirectCount-None-02721
  For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in Vertex Input Description

- VUID-vkCmdDrawIndexedIndirectCount-primitiveTopology-03420
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT dynamic state enabled then vkCmdSetPrimitiveTopology must have been called in the current command buffer prior to this drawing command, and the primitiveTopology parameter of vkCmdSetPrimitiveTopology must be of the same topology class as the pipeline VkPipelineInputAssemblyStateCreateInfo::topology state

- VUID-vkCmdDrawIndexedIndirectCount-pStrides-04884
  If the bound graphics pipeline was created with the
VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT  dynamic state enabled, then
vkCmdBindVertexBuffers2EXT  must have been called in the current command buffer prior
to this drawing command, and the  pStrides  parameter of  vkCmdBindVertexBuffers2EXT
must  not be  NULL

- VUID-vkCmdDrawIndexedIndirectCount-None-04879
  If the bound graphics pipeline state was created with the
  VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE  dynamic state enabled then
  vkCmdSetPrimitiveRestartEnable  must have been called in the current command buffer
  prior to this drawing command

- VUID-vkCmdDrawIndexedIndirectCount-buffer-02708
  If  buffer  is non-sparse then it  must  be bound completely and contiguously to a single
  VkDeviceMemory  object

- VUID-vkCmdDrawIndexedIndirectCount-buffer-02709
  buffer  must  have been created with the  VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT  bit set

- VUID-vkCmdDrawIndexedIndirectCount-offset-02710
  offset  must  be a multiple of  4

- VUID-vkCmdDrawIndexedIndirectCount-commandBuffer-02711
  commandBuffer  must  not be a protected command buffer

- VUID-vkCmdDrawIndexedIndirectCount-countBuffer-02714
  If  countBuffer  is non-sparse then it  must  be bound completely and contiguously to a single
  VkDeviceMemory  object

- VUID-vkCmdDrawIndexedIndirectCount-countBuffer-02715
  countBuffer  must  have been created with the  VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT  bit set

- VUID-vkCmdDrawIndexedIndirectCount-countBufferOffset-02716
  countBufferOffset  must  be a multiple of  4

- VUID-vkCmdDrawIndexedIndirectCount-countBuffer-02717
  The count stored in  countBuffer  must  be less than or equal to  VkPhysicalDeviceLimits
  ::maxDrawIndirectCount

- VUID-vkCmdDrawIndexedIndirectCount-countBufferOffset-04129
  (countBufferOffset + sizeof(uint32_t))  must  be less than or equal to the size of  countBuffer

- VUID-vkCmdDrawIndexedIndirectCount-None-04445
  If  drawIndirectCount  is not enabled this function  must  not be used

- VUID-vkCmdDrawIndexedIndirectCount-None-07312
  An index buffer  must  be bound

- VUID-vkCmdDrawIndexedIndirectCount-stride-03142
  stride  must  be a multiple of  4  and  must  be greater than or equal to
  sizeof(VkDrawIndexedIndirectCommand)

- VUID-vkCmdDrawIndexedIndirectCount-maxDrawCount-03143
  If  maxDrawCount  is greater than or equal to  1, (stride × (maxDrawCount - 1) + offset +
  sizeof(VkDrawIndexedIndirectCommand))  must  be less than or equal to the size of  buffer

- VUID-vkCmdDrawIndexedIndirectCount-countBuffer-03153
If count stored in countBuffer is equal to 1, (offset + sizeof(VkDrawIndexedIndirectCommand)) must be less than or equal to the size of buffer

- VUID-vkCmdDrawIndexedIndirectCount-countBuffer-03154
  If count stored in countBuffer is greater than 1, (stride × (drawCount - 1) + offset + sizeof(VkDrawIndexedIndirectCommand)) must be less than or equal to the size of buffer

### Valid Usage (Implicit)

- VUID-vkCmdDrawIndexedIndirectCount-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdDrawIndexedIndirectCount-buffer-parameter
  buffer must be a valid VkBuffer handle

- VUID-vkCmdDrawIndexedIndirectCount-countBuffer-parameter
  countBuffer must be a valid VkBuffer handle

- VUID-vkCmdDrawIndexedIndirectCount-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdDrawIndexedIndirectCount-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

- VUID-vkCmdDrawIndexedIndirectCount-renderpass
  This command must only be called inside of a render pass instance

- VUID-vkCmdDrawIndexedIndirectCount-commonparent
  Each of buffer, commandBuffer, and countBuffer must have been created, allocated, or retrieved from the same VkDevice

### Host Synchronization

- Host access to commandBuffer must be externally synchronized

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Secondary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Action</td>
</tr>
</tbody>
</table>
Chapter 21. Fixed-Function Vertex Processing

Vertex fetching is controlled via configurable state, as a logically distinct graphics pipeline stage.

21.1. Vertex Attributes

Vertex shaders can define input variables, which receive vertex attribute data transferred from one or more VkBuffer(s) by drawing commands. Vertex shader input variables are bound to buffers via an indirect binding where the vertex shader associates a vertex input attribute number with each variable, vertex input attributes are associated to vertex input bindings on a per-pipeline basis, and vertex input bindings are associated with specific buffers on a per-draw basis via the vkCmdBindVertexBuffers command. Vertex input attribute and vertex input binding descriptions also contain format information controlling how data is extracted from buffer memory and converted to the format expected by the vertex shader.

There are VkPhysicalDeviceLimits::maxVertexInputAttributes number of vertex input attributes and VkPhysicalDeviceLimits::maxVertexInputBindings number of vertex input bindings (each referred to by zero-based indices), where there are at least as many vertex input attributes as there are vertex input bindings. Applications can store multiple vertex input attributes interleaved in a single buffer, and use a single vertex input binding to access those attributes.

In GLSL, vertex shaders associate input variables with a vertex input attribute number using the location layout qualifier. The component layout qualifier associates components of a vertex shader input variable with components of a vertex input attribute.

GLSL example

```glsl
// Assign location M to variableName
layout (location=M, component=2) in vec2 variableName;

// Assign locations [N,N+L) to the array elements of variableNameArray
layout (location=N) in vec4 variableNameArray[L];
```

In SPIR-V, vertex shaders associate input variables with a vertex input attribute number using the Location decoration. The Component decoration associates components of a vertex shader input variable with components of a vertex input attribute. The Location and Component decorations are specified via the OpDecorate instruction.

SPIR-V example

```spir-v
...%1 = OpExtInstImport "GLSL.std.450"
...
OpName %9 "variableName"
OpName %15 "variableNameArray"
OpDecorate %18 BuiltIn VertexIndex
```
21.1.1. Attribute Location and Component Assignment

Vertex shaders allow Location and Component decorations on input variable declarations. The Location decoration specifies which vertex input attribute is used to read and interpret the data that a variable will consume. The Component decoration allows the location to be more finely specified for scalars and vectors, down to the individual components within a location that are consumed. The components within a location are 0, 1, 2, and 3. A variable starting at component N will consume components N, N+1, N+2, … up through its size. For single precision types, it is invalid if the sequence of components gets larger than 3.

When a vertex shader input variable declared using a 16- or 32-bit scalar or vector data type is assigned a location, its value(s) are taken from the components of the input attribute specified with the corresponding VkVertexInputAttributeDescription::location. The components used depend on the type of variable and the Component decoration specified in the variable declaration, as identified in 

<table>
<thead>
<tr>
<th>16-bit or 32-bit data type</th>
<th>Component decoration</th>
<th>Components consumed</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td>0 or unspecified</td>
<td>(x, o, o, o)</td>
</tr>
<tr>
<td>scalar</td>
<td>1</td>
<td>(o, y, o, o)</td>
</tr>
<tr>
<td>scalar</td>
<td>2</td>
<td>(o, o, z, o)</td>
</tr>
<tr>
<td>scalar</td>
<td>3</td>
<td>(o, o, o, w)</td>
</tr>
<tr>
<td>two-component vector</td>
<td>0 or unspecified</td>
<td>(x, y, o, o)</td>
</tr>
</tbody>
</table>
16-bit or 32-bit data type | Component decoration | Components consumed
---|---|---
two-component vector | 1 | (o, y, z, o)
two-component vector | 2 | (o, o, z, w)
three-component vector | 0 or unspecified | (x, y, z, o)
three-component vector | 1 | (o, y, z, w)
four-component vector | 0 or unspecified | (x, y, z, w)

Components indicated by “o” are available for use by other input variables which are sourced from the same attribute, and if used, are either filled with the corresponding component from the input format (if present), or the default value.

When a vertex shader input variable declared using a 32-bit floating point matrix type is assigned a location $i$, its values are taken from consecutive input attributes starting with the corresponding `VkVertexInputAttributeDescription::location`. Such matrices are treated as an array of column vectors with values taken from the input attributes identified in Table 22. Input attributes accessed by 32-bit input matrix variables. The `VkVertexInputAttributeDescription::format` must be specified with a `VkFormat` that corresponds to the appropriate type of column vector. The Component decoration must not be used with matrix types.

### Table 22. Input attributes accessed by 32-bit input matrix variables

<table>
<thead>
<tr>
<th>Data type</th>
<th>Column vector type</th>
<th>Locations consumed</th>
<th>Components consumed</th>
</tr>
</thead>
<tbody>
<tr>
<td>mat2</td>
<td>two-component vector</td>
<td>i, i+1</td>
<td>(x, y, o, o), (x, y, o, o)</td>
</tr>
<tr>
<td>mat2x3</td>
<td>three-component vector</td>
<td>i, i+1</td>
<td>(x, y, z, o), (x, y, z, o)</td>
</tr>
<tr>
<td>mat2x4</td>
<td>four-component vector</td>
<td>i, i+1</td>
<td>(x, y, z, w), (x, y, z, w)</td>
</tr>
<tr>
<td>mat3x2</td>
<td>two-component vector</td>
<td>i, i+1, i+2</td>
<td>(x, y, o, o), (x, y, o, o), (x, y, o, o)</td>
</tr>
<tr>
<td>mat3</td>
<td>three-component vector</td>
<td>i, i+1, i+2</td>
<td>(x, y, z, o), (x, y, z, o), (x, y, z, o)</td>
</tr>
<tr>
<td>mat3x4</td>
<td>four-component vector</td>
<td>i, i+1, i+2</td>
<td>(x, y, z, w), (x, y, z, w), (x, y, z, w)</td>
</tr>
<tr>
<td>mat4x2</td>
<td>two-component vector</td>
<td>i, i+1, i+2, i+3</td>
<td>(x, y, o, o), (x, y, o, o), (x, y, o, o), (x, y, o, o)</td>
</tr>
<tr>
<td>mat4x3</td>
<td>three-component vector</td>
<td>i, i+1, i+2, i+3</td>
<td>(x, y, z, o), (x, y, z, o), (x, y, z, o), (x, y, z, o)</td>
</tr>
<tr>
<td>mat4</td>
<td>four-component vector</td>
<td>i, i+1, i+2, i+3</td>
<td>(x, y, z, w), (x, y, z, w), (x, y, z, w), (x, y, z, w)</td>
</tr>
</tbody>
</table>

Components indicated by “o” are available for use by other input variables which are sourced from the same attribute, and if used, are either filled with the corresponding component from the input format (if present), or the default value.
When a vertex shader input variable declared using a scalar or vector 64-bit data type is assigned a location $i$, its values are taken from consecutive input attributes starting with the corresponding `VkVertexInputAttributeDescription::location`. The locations and components used depend on the type of variable and the `Component` decoration specified in the variable declaration, as identified in Input attribute locations and components accessed by 64-bit input variables. For 64-bit data types, no default attribute values are provided. Input variables must not use more components than provided by the attribute. Input attributes which have one- or two-component 64-bit formats will consume a single location. Input attributes which have three- or four-component 64-bit formats will consume two consecutive locations. A 64-bit scalar data type will consume two components, and a 64-bit two-component vector data type will consume all four components available within a location. A three- or four-component 64-bit data type must not specify a component. A three-component 64-bit data type will consume all four components of the first location and components 0 and 1 of the second location. This leaves components 2 and 3 available for other component-qualified declarations. A four-component 64-bit data type will consume all four components of the first location and all four components of the second location. It is invalid for a scalar or two-component 64-bit data type to specify a component of 1 or 3.

<table>
<thead>
<tr>
<th>Input format</th>
<th>Locations consumed</th>
<th>64-bit data type</th>
<th>Location decoration</th>
<th>Component decoration</th>
<th>32-bit component consumed</th>
</tr>
</thead>
<tbody>
<tr>
<td>R64</td>
<td>i</td>
<td>scalar</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, -, -)</td>
</tr>
<tr>
<td>R64G64</td>
<td>i</td>
<td>scalar</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, o, o)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar</td>
<td>i</td>
<td>2</td>
<td>(o, o, z, w)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>two-component vector</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, z, w)</td>
</tr>
<tr>
<td>R64G64B64</td>
<td>i, i+1</td>
<td>scalar</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, o, o), (o, o, -, -)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar</td>
<td>i</td>
<td>2</td>
<td>(o, o, z, w), (o, o, -, -)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar</td>
<td>i+1</td>
<td>0 or unspecified</td>
<td>(x, y, o, o), (x, y, -, -)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>two-component vector</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, z, w), (o, o, -, -)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>three-component vector</td>
<td>i</td>
<td>unspecified</td>
<td>(x, y, z, w), (x, y, -, -)</td>
</tr>
<tr>
<td>Input format</td>
<td>Locations consumed</td>
<td>64-bit data type</td>
<td>Location decoration</td>
<td>Component decoration</td>
<td>32-bit components consumed</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------</td>
<td>------------------</td>
<td>---------------------</td>
<td>----------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>R64G64B64A64</td>
<td>i, i+1</td>
<td>scalar</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, o, o), (o, o, o, o)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar</td>
<td>i</td>
<td>2</td>
<td>(o, o, z, w), (o, o, o, o)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar</td>
<td>i+1</td>
<td>0 or unspecified</td>
<td>(o, o, o, o), (x, y, o, o)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar</td>
<td>i+1</td>
<td>2</td>
<td>(o, o, o, o), (o, o, z, w)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>two-component vector</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, z, w), (o, o, o, o)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>two-component vector</td>
<td>i+1</td>
<td>0 or unspecified</td>
<td>(o, o, o, o), (x, y, z, w)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>three-component vector</td>
<td>i</td>
<td>unspecified</td>
<td>(x, y, z, w), (x, y, o, o)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>four-component vector</td>
<td>i</td>
<td>unspecified</td>
<td>(x, y, z, w), (x, y, z, w)</td>
</tr>
</tbody>
</table>

Components indicated by “o” are available for use by other input variables which are sourced from the same attribute. Components indicated by “-” are not available for input variables as there are no default values provided for 64-bit data types, and there is no data provided by the input format.

When a vertex shader input variable declared using a 64-bit floating-point matrix type is assigned a location \( i \), its values are taken from consecutive input attribute locations. Such matrices are treated as an array of column vectors with values taken from the input attributes as shown in Input attribute locations and components accessed by 64-bit input variables. Each column vector starts at the location immediately following the last location of the previous column vector. The number of attributes and components assigned to each matrix is determined by the matrix dimensions and ranges from two to eight locations.

When a vertex shader input variable declared using an array type is assigned a location, its values are taken from consecutive input attributes starting with the corresponding `VkVertexInputAttributeDescription::location`. The number of attributes and components assigned to each element are determined according to the data type of the array elements and Component decoration (if any) specified in the declaration of the array, as described above. Each element of the array, in order, is assigned to consecutive locations, but all at the same specified component within each location.

Only input variables declared with the data types and component decorations as specified above are supported. Location aliasing is causing two variables to have the same location number. Component aliasing is assigning the same (or overlapping) component number for two location aliases. Location aliasing is allowed only if it does not cause component aliasing. Further, when
location aliasing, the aliases sharing the location must all have the same SPIR-V floating-point component type or all have the same width integer-type components.

### 21.2. Vertex Input Description

Applications specify vertex input attribute and vertex input binding descriptions as part of graphics pipeline creation by setting the `VkGraphicsPipelineCreateInfo::pVertexInputState` pointer to a `VkPipelineVertexInputStateCreateInfo` structure.

The `VkPipelineVertexInputStateCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineVertexInputStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineVertexInputStateCreateFlags flags;
    uint32_t vertexBindingDescriptionCount;
    const VkVertexInputBindingDescription* pVertexBindingDescriptions;
    uint32_t vertexAttributeDescriptionCount;
    const VkVertexInputAttributeDescription* pVertexAttributeDescriptions;
} VkPipelineVertexInputStateCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **vertexBindingDescriptionCount** is the number of vertex binding descriptions provided in `pVertexBindingDescriptions`.
- **pVertexBindingDescriptions** is a pointer to an array of `VkVertexInputBindingDescription` structures.
- **vertexAttributeDescriptionCount** is the number of vertex attribute descriptions provided in `pVertexAttributeDescriptions`.
- **pVertexAttributeDescriptions** is a pointer to an array of `VkVertexInputAttributeDescription` structures.

### Valid Usage

- VUID-VkPipelineVertexInputStateCreateInfo-vertexBindingDescriptionCount-00613
  vertexBindingDescriptionCount must be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputBindings`

- VUID-VkPipelineVertexInputStateCreateInfo-vertexAttributeDescriptionCount-00614
  vertexAttributeDescriptionCount must be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputAttributes`

- VUID-VkPipelineVertexInputStateCreateInfo-binding-00615
  For every binding specified by each element of `pVertexAttributeDescriptions`, a
VkVertexInputBindingDescription must exist in pVertexBindingDescriptions with the same value of binding

- VUID-VkPipelineVertexInputStateCreateInfo-pVertexBindingDescriptions-00616
  All elements of pVertexBindingDescriptions must describe distinct binding numbers

- VUID-VkPipelineVertexInputStateCreateInfo-pVertexAttributeDescriptions-00617
  All elements of pVertexAttributeDescriptions must describe distinct attribute locations

Valid Usage (Implicit)

- VUID-VkPipelineVertexInputStateCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO

- VUID-VkPipelineVertexInputStateCreateInfo-pNext-pNext
  pNext must be NULL

- VUID-VkPipelineVertexInputStateCreateInfo-flags-zerobitmask
  flags must be 0

- VUID-VkPipelineVertexInputStateCreateInfo-pVertexBindingDescriptions-parameter
  If vertexBindingDescriptionCount is not 0, pVertexBindingDescriptions must be a valid pointer to an array of vertexBindingDescriptionCount valid VkVertexInputBindingDescription structures

- VUID-VkPipelineVertexInputStateCreateInfo-pVertexAttributeDescriptions-parameter
  If vertexAttributeDescriptionCount is not 0, pVertexAttributeDescriptions must be a valid pointer to an array of vertexAttributeDescriptionCount valid VkVertexInputAttributeDescription structures

// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineVertexInputStateCreateFlags;

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

Each vertex input binding is specified by the VkVertexInputBindingDescription structure, defined as:

// Provided by VK_VERSION_1_0
typedef struct VkVertexInputBindingDescription {
    uint32_t binding;
    uint32_t stride;
    VkVertexInputRate inputRate;
} VkVertexInputBindingDescription;

- binding is the binding number that this structure describes.
- stride is the byte stride between consecutive elements within the buffer.
- inputRate is a VkVertexInputRate value specifying whether vertex attribute addressing is a
function of the vertex index or of the instance index.

Valid Usage

- VUID-VkVertexInputBindingDescription-binding-00618
  binding must be less than VkPhysicalDeviceLimits::maxVertexInputBindings
- VUID-VkVertexInputBindingDescription-stride-00619
  stride must be less than or equal to VkPhysicalDeviceLimits::maxVertexInputBindingStride

Valid Usage (Implicit)

- VUID-VkVertexInputBindingDescription-inputRate-parameter
  inputRate must be a valid VkVertexInputRate value

Possible values of VkVertexInputBindingDescription::inputRate, specifying the rate at which vertex attributes are pulled from buffers, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkVertexInputRate {
    VK_VERTEX_INPUT_RATE_VERTEX = 0,
    VK_VERTEX_INPUT_RATE_INSTANCE = 1,
} VkVertexInputRate;
```

- VK_VERTEX_INPUT_RATE_VERTEX specifies that vertex attribute addressing is a function of the vertex index.
- VK_VERTEX_INPUT_RATE_INSTANCE specifies that vertex attribute addressing is a function of the instance index.

Each vertex input attribute is specified by the VkVertexInputAttributeDescription structure, defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkVertexInputAttributeDescription {
    uint32_t location;
    uint32_t binding;
    VkFormat format;
    uint32_t offset;
} VkVertexInputAttributeDescription;
```

- location is the shader input location number for this attribute.
- binding is the binding number which this attribute takes its data from.
- format is the size and type of the vertex attribute data.
- offset is a byte offset of this attribute relative to the start of an element in the vertex input
Valid Usage

- VUID-VkVertexInputAttributeDescription-location-00620
  location must be less than VkPhysicalDeviceLimits::maxVertexInputAttributes

- VUID-VkVertexInputAttributeDescription-binding-00621
  binding must be less than VkPhysicalDeviceLimits::maxVertexInputBindings

- VUID-VkVertexInputAttributeDescription-offset-00622
  offset must be less than or equal to VkPhysicalDeviceLimits::maxVertexInputAttributeOffset

- VUID-VkVertexInputAttributeDescription-format-00623
  format must be allowed as a vertex buffer format, as specified by the
  VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT flag in VkFormatProperties::bufferFeatures returned
  by vkGetPhysicalDeviceFormatProperties

Valid Usage (Implicit)

- VUID-VkVertexInputAttributeDescription-format-parameter
  format must be a valid VkFormat value

To bind vertex buffers to a command buffer for use in subsequent drawing commands, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdBindVertexBuffers(  
  VkCommandBuffer commandBuffer,  
  uint32_t firstBinding,  
  uint32_t bindingCount,  
  const VkBuffer* pBuffers,  
  const VkDeviceSize* pOffsets);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `firstBinding` is the index of the first vertex input binding whose state is updated by the command.
- `bindingCount` is the number of vertex input bindings whose state is updated by the command.
- `pBuffers` is a pointer to an array of buffer handles.
- `pOffsets` is a pointer to an array of buffer offsets.

The values taken from elements i of `pBuffers` and `pOffsets` replace the current state for the vertex input binding `firstBinding + i`, for i in [0, `bindingCount`). The vertex input binding is updated to start at the offset indicated by `pOffsets[i]` from the start of the buffer `pBuffers[i]`. All vertex input attributes that use each of these bindings will use these updated addresses in their address calculations for subsequent drawing commands.
Valid Usage

- VUID-vkCmdBindVertexBuffers-firstBinding-00624
  firstBinding must be less than VkPhysicalDeviceLimits::maxVertexInputBindings

- VUID-vkCmdBindVertexBuffers-firstBinding-00625
  The sum of firstBinding and bindingCount must be less than or equal to
  VkPhysicalDeviceLimits::maxVertexInputBindings

- VUID-vkCmdBindVertexBuffers-pOffsets-00626
  All elements of pOffsets must be less than the size of the corresponding element in
  pBuffers

- VUID-vkCmdBindVertexBuffers-pBuffers-00627
  All elements of pBuffers must have been created with the
  VK_BUFFER_USAGE_VERTEX_BUFFER_BIT flag

- VUID-vkCmdBindVertexBuffers-pBuffers-00628
  Each element of pBuffers that is non-sparse must be bound completely and contiguously
  to a single VkDeviceMemory object

- VUID-vkCmdBindVertexBuffers-pBuffers-04001
  If the nullDescriptor feature is not enabled, all elements of pBuffers must not be
  VK_NULL_HANDLE

Valid Usage (Implicit)

- VUID-vkCmdBindVertexBuffers-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdBindVertexBuffers-pBuffers-parameter
  pBuffers must be a valid pointer to an array of bindingCount valid or VK_NULL_HANDLE
  VkBuffer handles

- VUID-vkCmdBindVertexBuffers-pOffsets-parameter
  pOffsets must be a valid pointer to an array of bindingCount VkDeviceSize values

- VUID-vkCmdBindVertexBuffers-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdBindVertexBuffers-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics
  operations

- VUID-vkCmdBindVertexBuffers-bindingCount-arraylength
  bindingCount must be greater than 0

- VUID-vkCmdBindVertexBuffers-commonparent
  Both of commandBuffer, and the elements of pBuffers that are valid handles of non-ignored
  parameters must have been created, allocated, or retrieved from the same VkDevice
Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Alternatively, to bind vertex buffers, along with their sizes and strides, to a command buffer for use in subsequent drawing commands, call:

```c
// Provided by VK_VERSION_1_3
void vkCmdBindVertexBuffers2(
    VkCommandBuffer commandBuffer,
    uint32_t firstBinding,
    uint32_t bindingCount,
    const VkBuffer* pBuffers,
    const VkDeviceSize* pOffsets,
    const VkDeviceSize* pSizes,
    const VkDeviceSize* pStrides);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `firstBinding` is the index of the first vertex input binding whose state is updated by the command.
- `bindingCount` is the number of vertex input bindings whose state is updated by the command.
- `pBuffers` is a pointer to an array of buffer handles.
- `pOffsets` is a pointer to an array of buffer offsets.
- `pSizes` is `NULL` or a pointer to an array of the size in bytes of vertex data bound from `pBuffers`.
- `pStrides` is `NULL` or a pointer to an array of buffer strides.

The values taken from elements `i` of `pBuffers` and `pOffsets` replace the current state for the vertex input binding `firstBinding + i`, for `i` in `[0, bindingCount)`. The vertex input binding is updated to start at the offset indicated by `pOffsets[i]` from the start of the buffer `pBuffers[i]`. If `pSizes` is not `NULL` then `pSizes[i]` specifies the bound size of the vertex buffer starting from the corresponding elements of `pBuffers[i]` plus `pOffsets[i]`. All vertex input attributes that use each of these bindings will use these updated addresses in their address calculations for subsequent drawing commands.
This command also **dynamically sets** the byte strides between consecutive elements within buffer `pBuffers[i]` to the corresponding `pStrides[i]` value when the graphics pipeline is created with `VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, strides are specified by the `VkVertexInputBindingDescription::stride` values used to create the currently active pipeline.

### Valid Usage

- **VUID-vkCmdBindVertexBuffers2-firstBinding-03355**
  
  `firstBinding` **must** be less than `VkPhysicalDeviceLimits::maxVertexInputBindings`

- **VUID-vkCmdBindVertexBuffers2-firstBinding-03356**
  
  The sum of `firstBinding` and `bindingCount` **must** be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputBindings`

- **VUID-vkCmdBindVertexBuffers2-pOffsets-03357**
  
  All elements of `pOffsets` **must** be less than the size of the corresponding element in `pBuffers`

- **VUID-vkCmdBindVertexBuffers2-pSizes-03358**
  
  If `pSizes` is not `NULL`, all elements of `pOffsets` plus `pSizes` **must** be less than or equal to the size of the corresponding element in `pBuffers`

- **VUID-vkCmdBindVertexBuffers2-pBuffers-03359**
  
  All elements of `pBuffers` **must** have been created with the `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT` flag

- **VUID-vkCmdBindVertexBuffers2-pBuffers-03360**
  
  Each element of `pBuffers` that is non-sparse **must** be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-vkCmdBindVertexBuffers2-pBuffers-04111**
  
  If the `nullDescriptor` feature is not enabled, all elements of `pBuffers` **must** not be `VK_NULL_HANDLE`

- **VUID-vkCmdBindVertexBuffers2-pStrides-03362**
  
  If `pStrides` is not `NULL` each element of `pStrides` **must** be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputBindingStride`

- **VUID-vkCmdBindVertexBuffers2-pStrides-06209**
  
  If `pStrides` is not `NULL` each element of `pStrides` **must** be either 0 or greater than or equal to the maximum extent of all vertex input attributes fetched from the corresponding binding, where the extent is calculated as the `VkVertexInputAttributeDescription::offset` plus `VkVertexInputAttributeDescription::format` size

### Valid Usage (Implicit)

- **VUID-vkCmdBindVertexBuffers2-commandBuffer-parameter**
  
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle

- **VUID-vkCmdBindVertexBuffers2-pBuffers-parameter**
  
  `pBuffers` **must** be a valid pointer to an array of `bindingCount` valid or `VK_NULL_HANDLE`
**VkBuffer** handles

- VUID-vkCmdBindVertexBuffers2-pOffsets-parameter
  - `pOffsets` must be a valid pointer to an array of `bindingCount` `VkDeviceSize` values

- VUID-vkCmdBindVertexBuffers2-pSizes-parameter
  - If `pSizes` is not NULL, `pSizes` must be a valid pointer to an array of `bindingCount` `VkDeviceSize` values

- VUID-vkCmdBindVertexBuffers2-pStrides-parameter
  - If `pStrides` is not NULL, `pStrides` must be a valid pointer to an array of `bindingCount` `VkDeviceSize` values

- VUID-vkCmdBindVertexBuffers2-commandBuffer-recording
  - `commandBuffer` must be in the recording state

- VUID-vkCmdBindVertexBuffers2-commandBuffer-cmdpool
  - The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

- VUID-vkCmdBindVertexBuffers2-bindingCount-arraylength
  - If any of `pSizes`, or `pStrides` are not NULL, `bindingCount` must be greater than 0

- VUID-vkCmdBindVertexBuffers2-commonparent
  - Both of `commandBuffer`, and the elements of `pBuffers` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`

**Host Synchronization**

- Host access to `commandBuffer` must be externally synchronized

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

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**21.3. Vertex Input Address Calculation**

The address of each attribute for each `vertexIndex` and `instanceIndex` is calculated as follows:

- Let `attribDesc` be the member of `VkPipelineVertexInputStateCreateInfo::pVertexAttributeDescriptions` with `VkVertexInputAttributeDescription::location` equal to the vertex input attribute number.
• Let bindingDesc be the member of \( \text{VkPipelineVertexInputStateCreateInfo} \) pVertexBindingDescriptions with \( \text{VkVertexInputAttributeDescription} \) binding equal to attribDesc.binding.

• Let vertexIndex be the index of the vertex within the draw (a value between firstVertex and firstVertex+vertexCount for vkCmdDraw, or a value taken from the index buffer plus vertexOffset for vkCmdDrawIndexed), and let instanceIndex be the instance number of the draw (a value between firstInstance and firstInstance+instanceCount).

• Let offset be an array of offsets into the currently bound vertex buffers specified during vkCmdBindVertexBuffers or vkCmdBindVertexBuffers2 with pOffsets.

\[
\text{bufferBindingAddress} = \text{buffer}[\text{binding}].\text{baseAddress} + \text{offset}[\text{binding}];
\]

\[
\text{if} (\text{bindingDesc}.\text{inputRate} == \text{VK_VERTEX_INPUT_RATE_VERTEX})
\]
\[
\text{effectiveVertexOffset} = \text{vertexIndex} \times \text{bindingDesc}.\text{stride};
\]

\[
\text{else}
\]
\[
\text{effectiveVertexOffset} = \text{instanceIndex} \times \text{bindingDesc}.\text{stride};
\]

\[
\text{attribAddress} = \text{bufferBindingAddress} + \text{effectiveVertexOffset} + \text{attribDesc}.\text{offset};
\]

### 21.3.1. Vertex Input Extraction

For each attribute, raw data is extracted starting at attribAddress and is converted from the \( \text{VkVertexInputAttributeDescription} \)’s format to either floating-point, unsigned integer, or signed integer based on the base type of the format; the base type of the format must match the base type of the input variable in the shader. The input variable in the shader must be declared as a 64-bit data type if and only if format is a 64-bit data type. If format is a packed format, attribAddress must be a multiple of the size in bytes of the whole attribute data type as described in Packed Formats. Otherwise, attribAddress must be a multiple of the size in bytes of the component type indicated by format (see Formats). For attributes that are not 64-bit data types, each component is converted to the format of the input variable based on its type and size (as defined in the Format Definition section for each \( \text{VkFormat} \)), using the appropriate equations in 16-Bit Floating-Point Numbers, Unsigned 11-Bit Floating-Point Numbers, Unsigned 10-Bit Floating-Point Numbers, Fixed-Point Data Conversion, and Shared Exponent to RGB. Signed integer components smaller than 32 bits are sign-extended. Attributes that are not 64-bit data types are expanded to four components in the same way as described in conversion to RGBA. The number of components in the vertex shader input variable need not exactly match the number of components in the format. If the vertex shader has fewer components, the extra components are discarded.
Chapter 22. Tessellation

Tessellation involves three pipeline stages. First, a tessellation control shader transforms control points of a patch and can produce per-patch data. Second, a fixed-function tessellator generates multiple primitives corresponding to a tessellation of the patch in (u,v) or (u,v,w) parameter space. Third, a tessellation evaluation shader transforms the vertices of the tessellated patch, for example to compute their positions and attributes as part of the tessellated surface. The tessellator is enabled when the pipeline contains both a tessellation control shader and a tessellation evaluation shader.

22.1. Tessellator

If a pipeline includes both tessellation shaders (control and evaluation), the tessellator consumes each input patch (after vertex shading) and produces a new set of independent primitives (points, lines, or triangles). These primitives are logically produced by subdividing a geometric primitive (rectangle or triangle) according to the per-patch outer and inner tessellation levels written by the tessellation control shader. These levels are specified using the built-in variables TessLevelOuter and TessLevelInner, respectively. This subdivision is performed in an implementation-dependent manner. If no tessellation shaders are present in the pipeline, the tessellator is disabled and incoming primitives are passed through without modification.

The type of subdivision performed by the tessellator is specified by an OpExecutionMode instruction in the tessellation evaluation or tessellation control shader using one of execution modes Triangles, Quads, and IsoLines. Other tessellation-related execution modes can also be specified in either the tessellation control or tessellation evaluation shaders, and if they are specified in both then the modes must be the same.

Tessellation execution modes include:

- Triangles, Quads, and IsoLines. These control the type of subdivision and topology of the output primitives. One mode must be set in at least one of the tessellation shader stages.
- VertexOrderCw and VertexOrderCcw. These control the orientation of triangles generated by the tessellator. One mode must be set in at least one of the tessellation shader stages.
- PointMode. Controls generation of points rather than triangles or lines. This functionality defaults to disabled, and is enabled if either shader stage includes the execution mode.
- SpacingEqual, SpacingFractionalEven, and SpacingFractionalOdd. Controls the spacing of segments on the edges of tessellated primitives. One mode must be set in at least one of the tessellation shader stages.
- OutputVertices. Controls the size of the output patch of the tessellation control shader. One value must be set in at least one of the tessellation shader stages.

For triangles, the tessellator subdivides a triangle primitive into smaller triangles. For quads, the tessellator subdivides a rectangle primitive into smaller triangles. For isolines, the tessellator subdivides a rectangle primitive into a collection of line segments arranged in strips stretching across the rectangle in the u dimension (i.e. the coordinates in TessCoord are of the form (0,x) through (1,x) for all tessellation evaluation shader invocations that share a line).
Each vertex produced by the tessellator has an associated (u,v,w) or (u,v) position in a normalized parameter space, with parameter values in the range [0,1], as illustrated in figures Domain parameterization for tessellation primitive modes (upper-left origin) and Domain parameterization for tessellation primitive modes (lower-left origin). The domain space can have either an upper-left or lower-left origin, selected by the domainOrigin member of VkPipelineTessellationDomainOriginStateCreateInfo.

![Diagram of Domain Parameterization for Tessellation Primitive Modes](image)

*Figure 11. Domain parameterization for tessellation primitive modes (upper-left origin)*
Figure 12. Domain parameterization for tessellation primitive modes (lower-left origin)

Caption

In the domain parameterization diagrams, the coordinates illustrate the value of \texttt{TessCoord} at the corners of the domain. The labels on the edges indicate the inner (IL0 and IL1) and outer (OL0 through OL3) tessellation level values used to control the number of subdivisions along each edge of the domain.

For triangles, the vertex’s position is a barycentric coordinate \((u,v,w)\), where \(u + v + w = 1.0\), and indicates the relative influence of the three vertices of the triangle on the position of the vertex. For quads and isolines, the position is a \((u,v)\) coordinate indicating the relative horizontal and vertical position of the vertex relative to the subdivided rectangle. The subdivision process is explained in more detail in subsequent sections.

22.2. Tessellator Patch Discard

A patch is discarded by the tessellator if any relevant outer tessellation level is less than or equal to zero.

Patches will also be discarded if any relevant outer tessellation level corresponds to a floating-point
NaN (not a number) in implementations supporting NaN.

No new primitives are generated and the tessellation evaluation shader is not executed for patches that are discarded. For Quads, all four outer levels are relevant. For Triangles and IsoLines, only the first three or two outer levels, respectively, are relevant. Negative inner levels will not cause a patch to be discarded; they will be clamped as described below.

### 22.3. Tessellator Spacing

Each of the tessellation levels is used to determine the number and spacing of segments used to subdivide a corresponding edge. The method used to derive the number and spacing of segments is specified by an OpExecutionMode in the tessellation control or tessellation evaluation shader using one of the identifiers SpacingEqual, SpacingFractionalEven, or SpacingFractionalOdd.

If SpacingEqual is used, the floating-point tessellation level is first clamped to \([1, \text{maxLevel}]\), where \(\text{maxLevel}\) is the implementation-dependent maximum tessellation level (VkPhysicalDeviceLimits::maxTessellationGenerationLevel). The result is rounded up to the nearest integer \(n\), and the corresponding edge is divided into \(n\) segments of equal length in (u,v) space.

If SpacingFractionalEven is used, the tessellation level is first clamped to \([2, \text{maxLevel}]\) and then rounded up to the nearest even integer \(n\). If SpacingFractionalOdd is used, the tessellation level is clamped to \([1, \text{maxLevel} - 1]\) and then rounded up to the nearest odd integer \(n\). If \(n\) is one, the edge will not be subdivided. Otherwise, the corresponding edge will be divided into \(n - 2\) segments of equal length, and two additional segments of equal length that are typically shorter than the other segments. The length of the two additional segments relative to the others will decrease monotonically with \(n - f\), where \(f\) is the clamped floating-point tessellation level. When \(n - f\) is zero, the additional segments will have equal length to the other segments. As \(n - f\) approaches 2.0, the relative length of the additional segments approaches zero. The two additional segments must be placed symmetrically on opposite sides of the subdivided edge. The relative location of these two segments is implementation-dependent, but must be identical for any pair of subdivided edges with identical values of \(f\).

When tessellating triangles or quads using point mode with fractional odd spacing, the tessellator may produce interior vertices that are positioned on the edge of the patch if an inner tessellation level is less than or equal to one. Such vertices are considered distinct from vertices produced by subdividing the outer edge of the patch, even if there are pairs of vertices with identical coordinates.

### 22.4. Tessellation Primitive Ordering

Few guarantees are provided for the relative ordering of primitives produced by tessellation, as they pertain to primitive order.

- The output primitives generated from each input primitive are passed to subsequent pipeline stages in an implementation-dependent order.
- All output primitives generated from a given input primitive are passed to subsequent pipeline stages before any output primitives generated from subsequent input primitives.
22.5. Tessellator Vertex Winding Order

When the tessellator produces triangles (in the Triangles or Quads modes), the orientation of all triangles is specified with an OpExecutionMode of VertexOrderCw or VertexOrderCcw in the tessellation control or tessellation evaluation shaders. If the order is VertexOrderCw, the vertices of all generated triangles will have clockwise ordering in (u,v) or (u,v,w) space. If the order is VertexOrderCcw, the vertices will have counter-clockwise ordering in that space.

If the tessellation domain has an upper-left origin, the vertices of a triangle have counter-clockwise ordering if

\[ a = u_0 v_1 - u_1 v_0 + u_1 v_2 - u_2 v_1 + u_2 v_0 - u_0 v_2 \]

is negative, and clockwise ordering if \( a \) is positive. \( u_i \) and \( v_i \) are the \( u \) and \( v \) coordinates in normalized parameter space of the \( i \)th vertex of the triangle. If the tessellation domain has a lower-left origin, the vertices of a triangle have counter-clockwise ordering if \( a \) is positive, and clockwise ordering if \( a \) is negative.

**Note**

The value \( a \) is proportional (with a positive factor) to the signed area of the triangle.

In Triangles mode, even though the vertex coordinates have a \( w \) value, it does not participate directly in the computation of \( a \), being an affine combination of \( u \) and \( v \).

22.6. Triangle Tessellation

If the tessellation primitive mode is Triangles, an equilateral triangle is subdivided into a collection of triangles covering the area of the original triangle. First, the original triangle is subdivided into a collection of concentric equilateral triangles. The edges of each of these triangles are subdivided, and the area between each triangle pair is filled by triangles produced by joining the vertices on the subdivided edges. The number of concentric triangles and the number of subdivisions along each triangle except the outermost is derived from the first inner tessellation level. The edges of the outermost triangle are subdivided independently, using the first, second, and third outer tessellation levels to control the number of subdivisions of the \( u = 0 \) (left), \( v = 0 \) (bottom), and \( w = 0 \) (right) edges, respectively. The second inner tessellation level and the fourth outer tessellation level have no effect in this mode.

If the first inner tessellation level and all three outer tessellation levels are exactly one after clamping and rounding, only a single triangle with \((u,v,w)\) coordinates of \((0,0,1)\), \((1,0,0)\), and \((0,1,0)\) is generated. If the inner tessellation level is one and any of the outer tessellation levels is greater than one, the inner tessellation level is treated as though it were originally specified as \( 1 + \varepsilon \) and will result in a two- or three-segment subdivision depending on the tessellation spacing. When used with fractional odd spacing, the three-segment subdivision may produce inner vertices positioned on the edge of the triangle.
If any tessellation level is greater than one, tessellation begins by producing a set of concentric inner triangles and subdividing their edges. First, the three outer edges are temporarily subdivided using the clamped and rounded first inner tessellation level and the specified tessellation spacing, generating n segments. For the outermost inner triangle, the inner triangle is degenerate—a single point at the center of the triangle—if n is two. Otherwise, for each corner of the outer triangle, an inner triangle corner is produced at the intersection of two lines extended perpendicular to the corner’s two adjacent edges running through the vertex of the subdivided outer edge nearest that corner. If n is three, the edges of the inner triangle are not subdivided and it is the final triangle in the set of concentric triangles. Otherwise, each edge of the inner triangle is divided into n - 2 segments, with the n - 1 vertices of this subdivision produced by intersecting the inner edge with lines perpendicular to the edge running through the n - 1 innermost vertices of the subdivision of the outer edge. Once the outermost inner triangle is subdivided, the previous subdivision process repeats itself, using the generated triangle as an outer triangle. This subdivision process is illustrated in Inner Triangle Tessellation.

![Diagram of Inner Triangle Tessellation](image)

**Figure 13. Inner Triangle Tessellation**

**Caption**

In the Inner Triangle Tessellation diagram, inner tessellation levels of (a) four and (b) five are shown (not to scale). Solid black circles depict vertices along the edges of the concentric triangles. The edges of inner triangles are subdivided by intersecting the edge with segments perpendicular to the edge passing through each inner vertex of the subdivided outer edge. Dotted lines depict edges connecting corresponding vertices on the inner and outer triangle edges.

Once all the concentric triangles are produced and their edges are subdivided, the area between each pair of adjacent inner triangles is filled completely with a set of non-overlapping triangles. In this subdivision, two of the three vertices of each triangle are taken from adjacent vertices on a subdivided edge of one triangle; the third is one of the vertices on the corresponding edge of the other triangle. If the innermost triangle is degenerate (i.e., a point), the triangle containing it is subdivided into six triangles by connecting each of the six vertices on that triangle with the center point. If the innermost triangle is not degenerate, that triangle is added to the set of generated triangles as-is.
After the area corresponding to any inner triangles is filled, the tessellator generates triangles to cover the area between the outermost triangle and the outermost inner triangle. To do this, the temporary subdivision of the outer triangle edge above is discarded. Instead, the $u = 0$, $v = 0$, and $w = 0$ edges are subdivided according to the first, second, and third outer tessellation levels, respectively, and the tessellation spacing. The original subdivision of the first inner triangle is retained. The area between the outer and first inner triangles is completely filled by non-overlapping triangles as described above. If the first (and only) inner triangle is degenerate, a set of triangles is produced by connecting each vertex on the outer triangle edges with the center point.

After all triangles are generated, each vertex in the subdivided triangle is assigned a barycentric $(u, v, w)$ coordinate based on its location relative to the three vertices of the outer triangle.

The algorithm used to subdivide the triangular domain in $(u, v, w)$ space into individual triangles is implementation-dependent. However, the set of triangles produced will completely cover the domain, and no portion of the domain will be covered by multiple triangles.

Output triangles are generated with a topology similar to triangle lists, except that the order in which each triangle is generated, and the order in which the vertices are generated for each triangle, are implementation-dependent. However, the order of vertices in each triangle is consistent across the domain as described in Tessellator Vertex Winding Order.

### 22.7. Quad Tessellation

If the tessellation primitive mode is Quads, a rectangle is subdivided into a collection of triangles covering the area of the original rectangle. First, the original rectangle is subdivided into a regular mesh of rectangles, where the number of rectangles along the $u = 0$ and $u = 1$ (vertical) and $v = 0$ and $v = 1$ (horizontal) edges are derived from the first and second inner tessellation levels, respectively. All rectangles, except those adjacent to one of the outer rectangle edges, are decomposed into triangle pairs. The outermost rectangle edges are subdivided independently, using the first, second, third, and fourth outer tessellation levels to control the number of subdivisions of the $u = 0$ (left), $v = 0$ (bottom), $u = 1$ (right), and $v = 1$ (top) edges, respectively. The area between the inner rectangles of the mesh and the outer rectangle edges are filled by triangles produced by joining the vertices on the subdivided outer edges to the vertices on the edge of the inner rectangle mesh.

If both clamped inner tessellation levels and all four clamped outer tessellation levels are exactly one, only a single triangle pair covering the outer rectangle is generated. Otherwise, if either clamped inner tessellation level is one, that tessellation level is treated as though it was originally specified as $1 + \varepsilon$ and will result in a two- or three-segment subdivision depending on the tessellation spacing. When used with fractional odd spacing, the three-segment subdivision may produce inner vertices positioned on the edge of the rectangle.

If any tessellation level is greater than one, tessellation begins by subdividing the $u = 0$ and $u = 1$ edges of the outer rectangle into $m$ segments using the clamped and rounded first inner tessellation level and the tessellation spacing. The $v = 0$ and $v = 1$ edges are subdivided into $n$ segments using the second inner tessellation level. Each vertex on the $u = 0$ and $v = 0$ edges are joined with the corresponding vertex on the $u = 1$ and $v = 1$ edges to produce a set of vertical and horizontal lines that divide the rectangle into a grid of smaller rectangles. The primitive generator emits a pair of
non-overlapping triangles covering each such rectangle not adjacent to an edge of the outer rectangle. The boundary of the region covered by these triangles forms an inner rectangle, the edges of which are subdivided by the grid vertices that lie on the edge. If either \( m \) or \( n \) is two, the inner rectangle is degenerate, and one or both of the rectangle's edges consist of a single point. This subdivision is illustrated in Figure Inner Quad Tessellation.

Figure 14. Inner Quad Tessellation

Caption

In the Inner Quad Tessellation diagram, inner quad tessellation levels of (a) (4,2) and (b) (7,4) are shown. The regions highlighted in red in figure (b) depict the 10 inner rectangles, each of which will be subdivided into two triangles. Solid black circles depict vertices on the boundary of the outer and inner rectangles, where the inner rectangle of figure (a) is degenerate (a single line segment). Dotted lines depict the horizontal and vertical edges connecting corresponding vertices on the inner and outer rectangle edges.

After the area corresponding to the inner rectangle is filled, the tessellator must produce triangles to cover the area between the inner and outer rectangles. To do this, the subdivision of the outer rectangle edge above is discarded. Instead, the \( u = 0 \), \( v = 0 \), \( u = 1 \), and \( v = 1 \) edges are subdivided according to the first, second, third, and fourth outer tessellation levels, respectively, and the tessellation spacing. The original subdivision of the inner rectangle is retained. The area between the outer and inner rectangles is completely filled by non-overlapping triangles. Two of the three vertices of each triangle are adjacent vertices on a subdivided edge of one rectangle; the third is one of the vertices on the corresponding edge of the other rectangle. If either edge of the innermost rectangle is degenerate, the area near the corresponding outer edges is filled by connecting each vertex on the outer edge with the single vertex making up the inner edge.

The algorithm used to subdivide the rectangular domain in \((u,v)\) space into individual triangles is implementation-dependent. However, the set of triangles produced will completely cover the domain, and no portion of the domain will be covered by multiple triangles.

Output triangles are generated with a topology similar to triangle lists, except that the order in which each triangle is generated, and the order in which the vertices are generated for each triangle, are implementation-dependent. However, the order of vertices in each triangle is
consistent across the domain as described in **Tessellator Vertex Winding Order**.

### 22.8. Isoline Tessellation

If the tessellation primitive mode is **IsoLines**, a set of independent horizontal line segments is drawn. The segments are arranged into connected strips called **isolines**, where the vertices of each isoline have a constant $v$ coordinate and $u$ coordinates covering the full range $[0,1]$. The number of isolines generated is derived from the first outer tessellation level; the number of segments in each isoline is derived from the second outer tessellation level. Both inner tessellation levels and the third and fourth outer tessellation levels have no effect in this mode.

As with quad tessellation above, isoline tessellation begins with a rectangle. The $u = 0$ and $u = 1$ edges of the rectangle are subdivided according to the first outer tessellation level. For the purposes of this subdivision, the tessellation spacing mode is ignored and treated as **equal_spacing**. An isoline is drawn connecting each vertex on the $u = 0$ rectangle edge to the corresponding vertex on the $u = 1$ rectangle edge, except that no line is drawn between $(0,1)$ and $(1,1)$. If the number of isolines on the subdivided $u = 0$ and $u = 1$ edges is $n$, this process will result in $n$ equally spaced lines with constant $v$ coordinates of $0, \frac{1}{n}, \frac{2}{n}, \ldots, \frac{n-1}{n}$.

Each of the $n$ isolines is then subdivided according to the second outer tessellation level and the tessellation spacing, resulting in $m$ line segments. Each segment of each line is emitted by the tessellator. These line segments are generated with a topology similar to **line lists**, except that the order in which each line is generated, and the order in which the vertices are generated for each line segment, are implementation-dependent.

### 22.9. Tessellation Point Mode

For all primitive modes, the tessellator is capable of generating points instead of lines or triangles. If the tessellation control or tessellation evaluation shader specifies the **OpExecutionMode PointMode**, the primitive generator will generate one point for each distinct vertex produced by tessellation, rather than emitting triangles or lines. Otherwise, the tessellator will produce a collection of line segments or triangles according to the primitive mode. These points are generated with a topology similar to **point lists**, except the order in which the points are generated for each input primitive is undefined.

### 22.10. Tessellation Pipeline State

The **pTessellationState** member of **VkGraphicsPipelineCreateInfo** is a pointer to a **VkPipelineTessellationStateCreateInfo** structure.

The **VkPipelineTessellationStateCreateInfo** structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineTessellationStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineTessellationStateCreateFlags flags;
} VkPipelineTessellationStateCreateInfo;
```
```c
uint32_t patchControlPoints;
} VkPipelineTessellationStateCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `patchControlPoints` is the number of control points per patch.

### Valid Usage

- VUID-VkPipelineTessellationStateCreateInfo-patchControlPoints-01214
  `patchControlPoints` must be greater than zero and less than or equal to `VkPhysicalDeviceLimits::maxTessellationPatchSize`

### Valid Usage (Implicit)

- VUID-VkPipelineTessellationStateCreateInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO`

- VUID-VkPipelineTessellationStateCreateInfo-pNext-pNext
  `pNext` must be `NULL` or a pointer to a valid instance of `VkPipelineTessellationDomainOriginStateCreateInfo`

- VUID-VkPipelineTessellationStateCreateInfo-sType-unique
  The `sType` value of each struct in the `pNext` chain must be unique

- VUID-VkPipelineTessellationStateCreateInfo-flags-zerobitmask
  `flags` must be `0`

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineTessellationStateCreateFlags;
```

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

The `VkPipelineTessellationDomainOriginStateCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPipelineTessellationDomainOriginStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkTessellationDomainOrigin domainOrigin;
} VkPipelineTessellationDomainOriginStateCreateInfo;
```

- `sType` is the type of this structure.
• `pNext` is NULL or a pointer to a structure extending this structure.

• `domainOrigin` is a `VkTessellationDomainOrigin` value controlling the origin of the tessellation domain space.

If the `VkPipelineTessellationDomainOriginStateCreateInfo` structure is included in the `pNext` chain of `VkPipelineTessellationStateCreateInfo`, it controls the origin of the tessellation domain. If this structure is not present, it is as if `domainOrigin` was `VK_TESSELLATION_DOMAIN_ORIGIN_UPPER_LEFT`.

### Valid Usage (Implicit)

- VUID-VkPipelineTessellationDomainOriginStateCreateInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_DOMAIN_ORIGIN_STATE_CREATE_INFO`

- VUID-VkPipelineTessellationDomainOriginStateCreateInfo-domainOrigin-parameter
  `domainOrigin` must be a valid `VkTessellationDomainOrigin` value

The possible tessellation domain origins are specified by the `VkTessellationDomainOrigin` enumeration:

```
// Provided by VK_VERSION_1_1
typedef enum VkTessellationDomainOrigin {
    VK_TESSELLATION_DOMAIN_ORIGIN_UPPER_LEFT = 0,
    VK_TESSELLATION_DOMAIN_ORIGIN_LOWER_LEFT = 1,
} VkTessellationDomainOrigin;
```

- `VK_TESSELLATION_DOMAIN_ORIGIN_UPPER_LEFT` specifies that the origin of the domain space is in the upper left corner, as shown in figure Domain parameterization for tessellation primitive modes (upper-left origin).

- `VK_TESSELLATION_DOMAIN_ORIGIN_LOWER_LEFT` specifies that the origin of the domain space is in the lower left corner, as shown in figure Domain parameterization for tessellation primitive modes (lower-left origin).

This enum affects how the `VertexOrderCw` and `VertexOrderCcw` tessellation execution modes are interpreted, since the winding is defined relative to the orientation of the domain.
Chapter 23. Geometry Shading

The geometry shader operates on a group of vertices and their associated data assembled from a single input primitive, and emits zero or more output primitives and the group of vertices and their associated data required for each output primitive. Geometry shading is enabled when a geometry shader is included in the pipeline.

23.1. Geometry Shader Input Primitives

Each geometry shader invocation has access to all vertices in the primitive (and their associated data), which are presented to the shader as an array of inputs.

The input primitive type expected by the geometry shader is specified with an OpExecutionMode instruction in the geometry shader, and must match the incoming primitive type specified by either the pipeline’s primitive topology if tessellation is inactive, or the tessellation mode if tessellation is active, as follows:

- An input primitive type of InputPoints must only be used with a pipeline topology of VK_PRIMITIVE_TOPOLOGY_POINT_LIST, or with a tessellation shader specifying PointMode. The input arrays always contain one element, as described by the point list topology or tessellation in point mode.

- An input primitive type of InputLines must only be used with a pipeline topology of VK_PRIMITIVE_TOPOLOGY_LINE_LIST or VK_PRIMITIVE_TOPOLOGY_LINE_STRIP, or with a tessellation shader specifying IsoLines that does not specify PointMode. The input arrays always contain two elements, as described by the line list topology or line strip topology, or by isoline tessellation.

- An input primitive type of InputLinesAdjacency must only be used when tessellation is inactive, with a pipeline topology of VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY or VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY. The input arrays always contain four elements, as described by the line list with adjacency topology or line strip with adjacency topology.

- An input primitive type of Triangles must only be used with a pipeline topology of VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP, or VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN, or with a tessellation shader specifying Quads or Triangles that does not specify PointMode. The input arrays always contain three elements, as described by the triangle list topology, triangle strip topology, or triangle fan topology, or by triangle or quad tessellation. Vertices may be in a different absolute order than specified by the topology, but must adhere to the specified winding order.

- An input primitive type of InputTrianglesAdjacency must only be used when tessellation is inactive, with a pipeline topology of VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY or VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY. The input arrays always contain six elements, as described by the triangle list with adjacency topology or triangle strip with adjacency topology. Vertices may be in a different absolute order than specified by the topology, but must adhere to the specified winding order, and the vertices making up the main primitive must still occur at the first, third, and fifth index.
23.2. Geometry Shader Output Primitives

A geometry shader generates primitives in one of three output modes: points, line strips, or triangle strips. The primitive mode is specified in the shader using an OpExecutionMode instruction with the OutputPoints, OutputLineStrip or OutputTriangleStrip modes, respectively. Each geometry shader must include exactly one output primitive mode.

The vertices output by the geometry shader are assembled into points, lines, or triangles based on the output primitive type and the resulting primitives are then further processed as described in Rasterization. If the number of vertices emitted by the geometry shader is not sufficient to produce a single primitive, vertices corresponding to incomplete primitives are not processed by subsequent pipeline stages. The number of vertices output by the geometry shader is limited to a maximum count specified in the shader.

The maximum output vertex count is specified in the shader using an OpExecutionMode instruction with the mode set to OutputVertices and the maximum number of vertices that will be produced by the geometry shader specified as a literal. Each geometry shader must specify a maximum output vertex count.

23.3. Multiple Invocations of Geometry Shaders

Geometry shaders can be invoked more than one time for each input primitive. This is known as geometry shader instancing and is requested by including an OpExecutionMode instruction with mode specified as Invocations and the number of invocations specified as an integer literal.

In this mode, the geometry shader will execute at least n times for each input primitive, where n is the number of invocations specified in the OpExecutionMode instruction. The instance number is available to each invocation as a built-in input using InvocationId.

23.4. Geometry Shader Primitive Ordering

Limited guarantees are provided for the relative ordering of primitives produced by a geometry shader, as they pertain to primitive order.

- For instanced geometry shaders, the output primitives generated from each input primitive are passed to subsequent pipeline stages using the invocation number to order the primitives, from least to greatest.
- All output primitives generated from a given input primitive are passed to subsequent pipeline stages before any output primitives generated from subsequent input primitives.
Chapter 24. Fixed-Function Vertex Post-Processing

After pre-rasterization shader stages, the following fixed-function operations are applied to vertices of the resulting primitives:

- Flat shading (see Flat Shading).
- Primitive clipping, including client-defined half-spaces (see Primitive Clipping).
- Shader output attribute clipping (see Clipping Shader Outputs).
- Perspective division on clip coordinates (see Coordinate Transformations).
- Viewport mapping, including depth range scaling (see Controlling the Viewport).
- Front face determination for polygon primitives (see Basic Polygon Rasterization).

Next, rasterization is performed on primitives as described in chapter Rasterization.

24.1. Flat Shading

Flat shading a vertex output attribute means to assign all vertices of the primitive the same value for that output. The output values assigned are those of the provoking vertex of the primitive. Flat shading is applied to those vertex attributes that match fragment input attributes which are decorated as Flat.

If neither geometry nor tessellation shading is active, the provoking vertex is determined by the primitive topology defined by VkPipelineInputAssemblyStateCreateInfo:topology used to execute the drawing command.

If geometry shading is active, the provoking vertex is determined by the primitive topology defined by the OutputPoints, OutputLineStrips, or OutputTriangleStrips execution mode.

If tessellation shading is active but geometry shading is not, the provoking vertex may be any of the vertices in each primitive.

24.2. Primitive Clipping

Primitives are culled against the cull volume and then clipped to the clip volume. In clip coordinates, the view volume is defined by:

\[-W_c \leq x_c \leq W_c\]
\[-W_c \leq y_c \leq W_c\]
\[-z_m \leq z_c \leq W_c\]

where \(z_m\) is equal to zero.

This view volume can be further restricted by as many as VkPhysicalDeviceLimits::maxClipDistances client-defined half-spaces.
The cull volume is the intersection of up to `VkPhysicalDeviceLimits::maxCullDistances` client-defined half-spaces (if no client-defined cull half-spaces are enabled, culling against the cull volume is skipped).

A shader **must** write a single cull distance for each enabled cull half-space to elements of the `CullDistance` array. If the cull distance for any enabled cull half-space is negative for all of the vertices of the primitive under consideration, the primitive is discarded. Otherwise the primitive is clipped against the clip volume as defined below.

The clip volume is the intersection of up to `VkPhysicalDeviceLimits::maxClipDistances` client-defined half-spaces with the view volume (if no client-defined clip half-spaces are enabled, the clip volume is the view volume).

A shader **must** write a single clip distance for each enabled clip half-space to elements of the `ClipDistance` array. Clip half-space $i$ is then given by the set of points satisfying the inequality

$$c_i(P) \geq 0$$

where $c_i(P)$ is the clip distance $i$ at point $P$. For point primitives, $c_i(P)$ is simply the clip distance for the vertex in question. For line and triangle primitives, per-vertex clip distances are interpolated using a weighted mean, with weights derived according to the algorithms described in sections Basic Line Segment Rasterization and Basic Polygon Rasterization, using the perspective interpolation equations.

The number of client-defined clip and cull half-spaces that are enabled is determined by the explicit size of the built-in arrays `ClipDistance` and `CullDistance`, respectively, declared as an output in the interface of the entry point of the final shader stage before clipping.

Depth clamping is enabled or disabled via the `depthClampEnable` enable of the `VkPipelineRasterizationStateCreateInfo` structure. Depth clipping is disabled when `depthClampEnable` is `VK_TRUE`.

When depth clipping is disabled, the plane equation

$$z_m \leq z_c \leq w_c$$

(see the clip volume definition above) is ignored by view volume clipping (effectively, there is no near or far plane clipping).

If the primitive under consideration is a point or line segment, then clipping passes it unchanged if its vertices lie entirely within the clip volume.

Possible values of `VkPhysicalDevicePointClippingProperties::pointClippingBehavior`, specifying clipping behavior of a point primitive whose vertex lies outside the clip volume, are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkPointClippingBehavior {
    VK_POINT_CLIPPING_BEHAVIOR_ALL_CLIP_PLANES = 0,
}
```
VK_POINT_CLIPPING_BEHAVIOR_USER_CLIP_PLANES_ONLY = 1,
} VkPointClippingBehavior;

- **VK_POINT_CLIPPING_BEHAVIOR_ALL_CLIP_PLANES** specifies that the primitive is discarded if the vertex lies outside any clip plane, including the planes bounding the view volume.
- **VK_POINT_CLIPPING_BEHAVIOR_USER_CLIP_PLANES_ONLY** specifies that the primitive is discarded only if the vertex lies outside any user clip plane.

If either of a line segment’s vertices lie outside of the clip volume, the line segment may be clipped, with new vertex coordinates computed for each vertex that lies outside the clip volume. A clipped line segment endpoint lies on both the original line segment and the boundary of the clip volume.

This clipping produces a value, $0 \leq t \leq 1$, for each clipped vertex. If the coordinates of a clipped vertex are $P$ and the unclipped line segment’s vertex coordinates are $P_1$ and $P_2$, then $t$ satisfies the following equation

$$P = t P_1 + (1-t) P_2.$$

$t$ is used to clip vertex output attributes as described in Clipping Shader Outputs.

If the primitive is a polygon, it passes unchanged if every one of its edges lies entirely inside the clip volume, and is either clipped or discarded otherwise. If the edges of the polygon intersect the boundary of the clip volume, the intersecting edges are reconnected by new edges that lie along the boundary of the clip volume - in some cases requiring the introduction of new vertices into a polygon.

If a polygon intersects an edge of the clip volume’s boundary, the clipped polygon must include a point on this boundary edge.

Primitives rendered with user-defined half-spaces must satisfy a complementarity criterion. Suppose a series of primitives is drawn where each vertex $i$ has a single specified clip distance $d_i$ (or a number of similarly specified clip distances, if multiple half-spaces are enabled). Next, suppose that the same series of primitives are drawn again with each such clip distance replaced by $-d_i$ (and the graphics pipeline is otherwise the same). In this case, primitives must not be missing any pixels, and pixels must not be drawn twice in regions where those primitives are cut by the clip planes.

### 24.3. Clipping Shader Outputs

Next, vertex output attributes are clipped. The output values associated with a vertex that lies within the clip volume are unaffected by clipping. If a primitive is clipped, however, the output values assigned to vertices produced by clipping are clipped.

Let the output values assigned to the two vertices $P_1$ and $P_2$ of an unclipped edge be $c_1$ and $c_2$. The value of $t$ (see Primitive Clipping) for a clipped point $P$ is used to obtain the output value associated with $P$ as
\[
c = t \mathbf{c}_1 + (1-t) \mathbf{c}_2.
\]

(Multiplying an output value by a scalar means multiplying each of \(x, y, z,\) and \(w\) by the scalar.)

Since this computation is performed in clip space before division by \(w_c\), clipped output values are perspective-correct.

Polygon clipping creates a clipped vertex along an edge of the clip volume’s boundary. This situation is handled by noting that polygon clipping proceeds by clipping against one half-space at a time. Output value clipping is done in the same way, so that clipped points always occur at the intersection of polygon edges (possibly already clipped) with the clip volume’s boundary.

For vertex output attributes whose matching fragment input attributes are decorated with \textsc{NoPerspective}, the value of \(t\) used to obtain the output value associated with \(P\) will be adjusted to produce results that vary linearly in framebuffer space.

Output attributes of integer or unsigned integer type \textbf{must} always be flat shaded. Flat shaded attributes are constant over the primitive being rasterized (see Basic Line Segment Rasterization and Basic Polygon Rasterization), and no interpolation is performed. The output value \(c\) is taken from either \(c_1\) or \(c_2\), since flat shading has already occurred and the two values are identical.

### 24.4. Coordinate Transformations

Clip coordinates for a vertex result from shader execution, which yields a vertex coordinate \textit{Position}.

Perspective division on clip coordinates yields \textit{normalized device coordinates}, followed by a \textit{viewport} transformation (see Controlling the Viewport) to convert these coordinates into \textit{framebuffer coordinates}.

If a vertex in clip coordinates has a position given by

\[
\begin{pmatrix}
  x_c \\
  y_c \\
  z_c \\
  w_c
\end{pmatrix}
\]

then the vertex’s normalized device coordinates are

\[
\begin{pmatrix}
  x_d \\
  y_d \\
  z_d
\end{pmatrix} = \begin{pmatrix}
  x_c \\
  w_c \\
  y_c \\
  w_c \\
  z_c \\
  w_c
\end{pmatrix}
\]
24.5. Controlling the Viewport

The viewport transformation is determined by the selected viewport’s width and height in pixels, $p_x$ and $p_y$, respectively, and its center $(o_x, o_y)$ (also in pixels), as well as its depth range min and max determining a depth range scale value $p_z$ and a depth range bias value $o_z$ (defined below). The vertex’s framebuffer coordinates $(x_f, y_f, z_f)$ are given by

\[
x_f = \frac{p_x}{2} x_d + o_x
\]

\[
y_f = \frac{p_y}{2} y_d + o_y
\]

\[
z_f = p_z z_d + o_z
\]

Multiple viewports are available, numbered zero up to $\text{VkPhysicalDeviceLimits::maxViewports}$ minus one. The number of viewports used by a pipeline is controlled by the $\text{viewportCount}$ member of the $\text{VkPipelineViewportStateCreateInfo}$ structure used in pipeline creation.

$x_f$ and $y_f$ have limited precision, where the number of fractional bits retained is specified by $\text{VkPhysicalDeviceLimits::subPixelPrecisionBits}$.

The $\text{VkPipelineViewportStateCreateInfo}$ structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineViewportStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineViewportStateCreateFlags flags;
    uint32_t viewportCount;
    const VkViewport* pViewports;
    uint32_t scissorCount;
    const VkRect2D* pScissors;
} VkPipelineViewportStateCreateInfo;
```

- $\text{sType}$ is the type of this structure.
- $\text{pNext}$ is $\text{NULL}$ or a pointer to a structure extending this structure.
- $\text{flags}$ is reserved for future use.
- $\text{viewportCount}$ is the number of viewports used by the pipeline.
- $\text{pViewports}$ is a pointer to an array of $\text{VkViewport}$ structures, defining the viewport transforms. If the viewport state is dynamic, this member is ignored.
- $\text{scissorCount}$ is the number of $\text{scissors}$ and must match the number of viewports.
- $\text{pScissors}$ is a pointer to an array of $\text{VkRect2D}$ structures defining the rectangular bounds of the scissor for the corresponding viewport. If the scissor state is dynamic, this member is ignored.
Valid Usage

- **VUID-VkPipelineViewportStateCreateInfo-viewportCount-01216**
  If the `multiViewport` feature is not enabled, `viewportCount` must not be greater than 1.

- **VUID-VkPipelineViewportStateCreateInfo-scissorCount-01217**
  If the `multiViewport` feature is not enabled, `scissorCount` must not be greater than 1.

- **VUID-VkPipelineViewportStateCreateInfo-viewportCount-01218**
  `viewportCount` must be less than or equal to `VkPhysicalDeviceLimits::maxViewports`.

- **VUID-VkPipelineViewportStateCreateInfo-scissorCount-01219**
  `scissorCount` must be less than or equal to `VkPhysicalDeviceLimits::maxViewports`.

- **VUID-VkPipelineViewportStateCreateInfo-x-02821**
  The `x` and `y` members of `offset` member of any element of `pScissors` must be greater than or equal to 0.

- **VUID-VkPipelineViewportStateCreateInfo-offset-02822**
  Evaluation of `(offset.x + extent.width)` must not cause a signed integer addition overflow for any element of `pScissors`.

- **VUID-VkPipelineViewportStateCreateInfo-offset-02823**
  Evaluation of `(offset.y + extent.height)` must not cause a signed integer addition overflow for any element of `pScissors`.

- **VUID-VkPipelineViewportStateCreateInfo-scissorCount-04134**
  If the graphics pipeline is being created without `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT` and `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT` set then `scissorCount` and `viewportCount` must be identical.

- **VUID-VkPipelineViewportStateCreateInfo-viewportCount-04135**
  If the graphics pipeline is being created with `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT` set then `viewportCount` must be 0, otherwise it must be greater than 0.

- **VUID-VkPipelineViewportStateCreateInfo-scissorCount-04136**
  If the graphics pipeline is being created with `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT` set then `scissorCount` must be 0, otherwise it must be greater than 0.

Valid Usage (Implicit)

- **VUID-VkPipelineViewportStateCreateInfo-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO`.

- **VUID-VkPipelineViewportStateCreateInfo-pNext-pNext**
  `pNext` must be `NULL`.

- **VUID-VkPipelineViewportStateCreateInfo-flags-zerobitmask**
  `flags` must be 0.

To dynamically set the viewport count and viewports, call:
```c
void vkCmdSetViewportWithCount(
    VkCommandBuffer commandBuffer,
    uint32_t viewportCount,
    const VkViewport* pViewports);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `viewportCount` specifies the viewport count.
- `pViewports` specifies the viewports to use for drawing.

This command sets the viewport count and viewports state for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the corresponding `VkPipelineViewportStateCreateInfo::viewportCount` and `pViewports` values used to create the currently active pipeline.

### Valid Usage

- VUID-vkCmdSetViewportWithCount-viewportCount-03394
  `viewportCount` must be between 1 and `VkPhysicalDeviceLimits::maxViewports`, inclusive
- VUID-vkCmdSetViewportWithCount-viewportCount-03395
  If the multiViewport feature is not enabled, `viewportCount` must be 1

### Valid Usage (Implicit)

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

### Host Synchronization

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

---

## Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To **dynamically set** the scissor count and scissor rectangular bounds, call:

```c
// Provided by VK_VERSION_1_3
void vkCmdSetScissorWithCount(
    VkCommandBuffer commandBuffer,
    uint32_t scissorCount,
    const VkRect2D* pScissors);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `scissorCount` specifies the scissor count.
- `pScissors` specifies the scissors to use for drawing.

This command sets the scissor count and scissor rectangular bounds state for subsequence drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the corresponding `VkPipelineViewportStateCreateInfo::scissorCount` and `pScissors` values used to create the currently active pipeline.

---

## Valid Usage

- VUID-vkCmdSetScissorWithCount-scissorCount-03397
  
  `scissorCount` must be between 1 and `VkPhysicalDeviceLimits::maxViewports`, inclusive

- VUID-vkCmdSetScissorWithCount-scissorCount-03398
  
  If the `multiViewport` feature is not enabled, `scissorCount` must be 1

- VUID-vkCmdSetScissorWithCount-x-03399
  
  The x and y members of `offset` member of any element of `pScissors` must be greater than or equal to 0

- VUID-vkCmdSetScissorWithCount-offset-03400
  
  Evaluation of `(offset.x + extent.width)` must not cause a signed integer addition overflow for any element of `pScissors`

- VUID-vkCmdSetScissorWithCount-offset-03401
  
  Evaluation of `(offset.y + extent.height)` must not cause a signed integer addition
Valid Usage (Implicit)

- VUID-vkCmdSetScissorWithCount-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdSetScissorWithCount-pScissors-parameter
  `pScissors` must be a valid pointer to an array of `scissorCount` `VkRect2D` structures
- VUID-vkCmdSetScissorWithCount-commandBuffer-recording
  `commandBuffer` must be in the recording state
- VUID-vkCmdSetScissorWithCount-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations
- VUID-vkCmdSetScissorWithCount-scissorCount-arraylength
  `scissorCount` must be greater than 0

Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineViewportStateCreateFlags;

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

A pre-rasterization shader stage can direct each primitive to one of several viewports. The destination viewport for a primitive is selected by the last active pre-rasterization shader stage that has an output variable decorated with `ViewportIndex`. The viewport transform uses the viewport corresponding to the value assigned to `ViewportIndex`, and taken from an implementation-dependent vertex of each primitive. If `ViewportIndex` is outside the range zero to `viewportCount`...
minus one for a primitive, or if the last active pre-rasterization shader stage did not assign a value to ViewportIndex for all vertices of a primitive due to flow control, the values resulting from the viewport transformation of the vertices of such primitives are undefined. If the last pre-rasterization shader stage does not have an output decorated with ViewportIndex, the viewport numbered zero is used by the viewport transformation.

A single vertex can be used in more than one individual primitive, in primitives such as VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP. In this case, the viewport transformation is applied separately for each primitive.

To dynamically set the viewport transformation parameters, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetViewport(
    VkCommandBuffer commandBuffer,
    uint32_t firstViewport,
    uint32_t viewportCount,
    const VkViewport* pViewports);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `firstViewport` is the index of the first viewport whose parameters are updated by the command.
- `viewportCount` is the number of viewports whose parameters are updated by the command.
- `pViewports` is a pointer to an array of VkViewport structures specifying viewport parameters.

This command sets the viewport transformation parameters state for subsequent drawing commands when the graphics pipeline is created with VK_DYNAMIC_STATE_VIEWPORT set in VkPipelineDynamicStateCreateInfo::pDynamicStates. Otherwise, this state is specified by the VkPipelineViewportStateCreateInfo::pViewports values used to create the currently active pipeline.

The viewport parameters taken from element `i` of `pViewports` replace the current state for the viewport index `firstViewport + i`, for `i` in `[0, viewportCount)`.

### Valid Usage

- VUID-vkCmdSetViewport-firstViewport-01223
  The sum of `firstViewport` and `viewportCount` must be between `1` and `VkPhysicalDeviceLimits::maxViewports`, inclusive
- VUID-vkCmdSetViewport-firstViewport-01224
  If the `multiViewport` feature is not enabled, `firstViewport` must be `0`
- VUID-vkCmdSetViewport-viewportCount-01225
  If the `multiViewport` feature is not enabled, `viewportCount` must be `1`

### Valid Usage (Implicit)

- VUID-vkCmdSetViewport-commandBuffer-parameter
**commandBuffer** must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdSetViewport-pViewports-parameter**
  *`pViewports` must be a valid pointer to an array of `viewportCount` valid `VkViewport` structures*

- **VUID-vkCmdSetViewport-commandBuffer-recording**
  *`commandBuffer` must be in the recording state*

- **VUID-vkCmdSetViewport-commandBuffer-cmdpool**
  *The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations*

- **VUID-vkCmdSetViewport-viewportCount-arraylength**
  *`viewportCount` must be greater than 0*

---

**Host Synchronization**

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

---

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Both `VkPipelineViewportStateCreateInfo` and `vkCmdSetViewport` use `VkViewport` to set the viewport transformation parameters.

The `VkViewport` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkViewport {
    float  x;
    float  y;
    float  width;
    float  height;
    float  minDepth;
    float  maxDepth;
} VkViewport;
```

- *`x` and `y` are the viewport’s upper left corner (x,y).*
- *`width` and `height` are the viewport’s width and height, respectively.*
• **minDepth** and **maxDepth** are the depth range for the viewport.

**Note**
Despite their names, **minDepth** can be less than, equal to, or greater than **maxDepth**.

The framebuffer depth coordinate $z_f$ may be represented using either a fixed-point or floating-point representation. However, a floating-point representation **must** be used if the depth/stencil attachment has a floating-point depth component. If an $m$-bit fixed-point representation is used, we assume that it represents each value $\frac{k}{2^{m-1}}$, where $k \in \{0, 1, ..., 2^m-1\}$, as $k$ (e.g. 1.0 is represented in binary as a string of all ones).

The viewport parameters shown in the above equations are found from these values as

$$o_x = x + \frac{width}{2}$$

$$o_y = y + \frac{height}{2}$$

$$o_z = \text{minDepth}$$

$$p_x = \text{width}$$

$$p_y = \text{height}$$

$$p_z = \text{maxDepth} - \text{minDepth}$$

The application **can** specify a negative term for **height**, which has the effect of negating the y coordinate in clip space before performing the transform. When using a negative **height**, the application **should** also adjust the **y** value to point to the lower left corner of the viewport instead of the upper left corner. Using the negative **height** allows the application to avoid having to negate the y component of the **Position** output from the last **pre-rasterization shader stage**.

The width and height of the **implementation-dependent maximum viewport dimensions** **must** be greater than or equal to the width and height of the largest image which **can** be created and attached to a framebuffer.

The floating-point viewport bounds are represented with an **implementation-dependent precision**.

---

**Valid Usage**

- VUID-VkViewport-width-01770
  - width **must** be greater than 0.0
- VUID-VkViewport-width-01771
• **width must** be less than or equal to `VkPhysicalDeviceLimits::maxViewportDimensions[0]`

  - VUID-VkViewport-height-01773
    The absolute value of **height must** be less than or equal to `VkPhysicalDeviceLimits::maxViewportDimensions[1]`

  - VUID-VkViewport-x-01774
    **x must** be greater than or equal to `viewportBoundsRange[0]`

  - VUID-VkViewport-x-01232
    
    \[(x + width) must\] be less than or equal to `viewportBoundsRange[1]`

  - VUID-VkViewport-y-01775
    **y must** be greater than or equal to `viewportBoundsRange[0]`

  - VUID-VkViewport-y-01776
    **y must** be less than or equal to `viewportBoundsRange[1]`

  - VUID-VkViewport-y-01777
    \[(y + height) must\] be greater than or equal to `viewportBoundsRange[0]`

  - VUID-VkViewport-y-01233
    \[(y + height) must\] be less than or equal to `viewportBoundsRange[1]`

  - VUID-VkViewport-minDepth-02540
    **minDepth must** be between 0.0 and 1.0, inclusive

  - VUID-VkViewport-maxDepth-02541
    **maxDepth must** be between 0.0 and 1.0, inclusive
Chapter 25. Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each discrete location of this image contains associated data such as depth, color, or other attributes.

Rasterizing a primitive begins by determining which squares of an integer grid in framebuffer coordinates are occupied by the primitive, and assigning one or more depth values to each such square. This process is described below for points, lines, and polygons.

A grid square, including its (x,y) framebuffer coordinates, z (depth), and associated data added by fragment shaders, is called a fragment. A fragment is located by its upper left corner, which lies on integer grid coordinates.

Rasterization operations also refer to a fragment’s sample locations, which are offset by fractional values from its upper left corner. The rasterization rules for points, lines, and triangles involve testing whether each sample location is inside the primitive. Fragments need not actually be square, and rasterization rules are not affected by the aspect ratio of fragments. Display of non-square grids, however, will cause rasterized points and line segments to appear fatter in one direction than the other.

We assume that fragments are square, since it simplifies antialiasing and texturing. After rasterization, fragments are processed by fragment operations.

Several factors affect rasterization, including the members of VkPipelineRasterizationStateCreateInfo and VkPipelineMultisampleStateCreateInfo.

The VkPipelineRasterizationStateCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineRasterizationStateCreateInfo {
    VkStructureType sType;  // is the type of this structure.
    const void* pNext;      // is NULL or a pointer to a structure extending this structure.
    VkPipelineRasterizationStateCreateFlags flags;  // is reserved for future use.
    VkBool32 depthClampEnable;  // is reserved for future use.
    VkBool32 rasterizerDiscardEnable;  // is reserved for future use.
    VkPolygonMode polygonMode;  // is reserved for future use.
    VkCullModeFlags cullMode;  // is reserved for future use.
    VkFrontFace frontFace;  // is reserved for future use.
    VkBool32 depthBiasEnable;  // is reserved for future use.
    float depthBiasConstantFactor;  // is reserved for future use.
    float depthBiasClamp;  // is reserved for future use.
    float depthBiasSlopeFactor;  // is reserved for future use.
    float lineWidth;  // is reserved for future use.
} VkPipelineRasterizationStateCreateInfo;
```
- **depthClampEnable** controls whether to clamp the fragment's depth values as described in **Depth Test**. Enabling depth clamp will also disable clipping primitives to the z planes of the frustrum as described in **Primitive Clipping**.

- **rasterizerDiscardEnable** controls whether primitives are discarded immediately before the rasterization stage.

- **polygonMode** is the triangle rendering mode. See **VkPolygonMode**.

- **cullMode** is the triangle facing direction used for primitive culling. See **VkCullModeFlagBits**.

- **frontFace** is a **VkFrontFace** value specifying the front-facing triangle orientation to be used for culling.

- **depthBiasEnable** controls whether to bias fragment depth values.

- **depthBiasConstantFactor** is a scalar factor controlling the constant depth value added to each fragment.

- **depthBiasClamp** is the maximum (or minimum) depth bias of a fragment.

- **depthBiasSlopeFactor** is a scalar factor applied to a fragment's slope in depth bias calculations.

- **lineWidth** is the width of rasterized line segments.

---

**Valid Usage**

- VUID-VkPipelineRasterizationStateCreateInfo-depthClampEnable-00782
  If the **depthClamp** feature is not enabled, **depthClampEnable** must be **VK_FALSE**

- VUID-VkPipelineRasterizationStateCreateInfo-polygonMode-01413
  If the **fillModeNonSolid** feature is not enabled, **polygonMode** must be **VK_POLYGON_MODE_FILL**

---

**Valid Usage (Implicit)**

- VUID-VkPipelineRasterizationStateCreateInfo-sType-sType
  **sType** must be **VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO**

- VUID-VkPipelineRasterizationStateCreateInfo-pNext-pNext
  **pNext** must be **NULL**

- VUID-VkPipelineRasterizationStateCreateInfo-flags-zerobitmask
  **flags** must be **0**

- VUID-VkPipelineRasterizationStateCreateInfo-polygonMode-parameter
  **polygonMode** must be a valid **VkPolygonMode** value

- VUID-VkPipelineRasterizationStateCreateInfo-cullMode-parameter
  **cullMode** must be a valid combination of **VkCullModeFlagBits** values

- VUID-VkPipelineRasterizationStateCreateInfo-frontFace-parameter
  **frontFace** must be a valid **VkFrontFace** value

---

// Provided by VK_VERSION_1_0
**typedef** VkFlags VkPipelineRasterizationStateCreateFlags;

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

The `VkPipelineMultisampleStateCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineMultisampleStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineMultisampleStateCreateFlags flags;
    VkSampleCountFlagBits rasterizationSamples;
    VkBool32 sampleShadingEnable;
    float minSampleShading;
    const VkSampleMask* pSampleMask;
    VkBool32 alphaToCoverageEnable;
    VkBool32 alphaToOneEnable;
} VkPipelineMultisampleStateCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **rasterizationSamples** is a `VkSampleCountFlagBits` value specifying the number of samples used in rasterization.
- **sampleShadingEnable** can be used to enable **Sample Shading**.
- **minSampleShading** specifies a minimum fraction of sample shading if **sampleShadingEnable** is set to `VK_TRUE`.
- **pSampleMask** is a pointer to an array of `VkSampleMask` values used in the **sample mask test**.
- **alphaToCoverageEnable** controls whether a temporary coverage value is generated based on the alpha component of the fragment’s first color output as specified in the **Multisample Coverage** section.
- **alphaToOneEnable** controls whether the alpha component of the fragment’s first color output is replaced with one as described in **Multisample Coverage**.

Each bit in the sample mask is associated with a unique **sample index** as defined for the **coverage mask**. Each bit b for mask word w in the sample mask corresponds to sample index i, where \( i = 32 \times w + b \). **pSampleMask** has a length equal to \( \lceil \frac{\text{rasterizationSamples}}{32} \rceil \) words.

If **pSampleMask** is `NULL`, it is treated as if the mask has all bits set to `1`.

### Valid Usage

- VUID-VkPipelineMultisampleStateCreateInfo-sampleShadingEnable-00784
If the `sampleRateShading` feature is not enabled, `sampleShadingEnable` must be `VK_FALSE`.

- VUID-VkPipelineMultisampleStateCreateInfo-alphaToOneEnable-00785
  If the `alphaToOne` feature is not enabled, `alphaToOneEnable` must be `VK_FALSE`.

- VUID-VkPipelineMultisampleStateCreateInfo-minSampleShading-00786
  `minSampleShading` must be in the range [0, 1].

**Valid Usage (Implicit)**

- VUID-VkPipelineMultisampleStateCreateInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO`.

- VUID-VkPipelineMultisampleStateCreateInfo-pNext-pNext
  `pNext` must be `NULL`.

- VUID-VkPipelineMultisampleStateCreateInfo-flags-zerobitmask
  `flags` must be `0`.

- VUID-VkPipelineMultisampleStateCreateInfo-rasterizationSamples-parameter
  `rasterizationSamples` must be a valid `VkSampleCountFlagBits` value.

- VUID-VkPipelineMultisampleStateCreateInfo-pSampleMask-parameter
  If `pSampleMask` is not `NULL`, `pSampleMask` must be a valid pointer to an array of `VkSampleMask` values.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineMultisampleStateCreateFlags;
```

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

The elements of the sample mask array are of type `VkSampleMask`, each representing 32 bits of coverage information.

```c
// Provided by VK_VERSION_1_0
typedef uint32_t VkSampleMask;
```

Rasterization only generates fragments which cover one or more pixels inside the framebuffer. Pixels outside the framebuffer are never considered covered in the fragment. Fragments which would be produced by application of any of the primitive rasterization rules described below but which lie outside the framebuffer are not produced, nor are they processed by any later stage of the pipeline, including any of the fragment operations.

Surviving fragments are processed by fragment shaders. Fragment shaders determine associated data for fragments, and can also modify or replace their assigned depth values.
25.1. Discarding Primitives Before Rasterization

Primitives are discarded before rasterization if the `rasterizerDiscardEnable` member of `VkPipelineRasterizationStateCreateInfo` is enabled. When enabled, primitives are discarded after they are processed by the last active shader stage in the pipeline before rasterization.

To dynamically enable whether primitives are discarded before the rasterization stage, call:

```c
void vkCmdSetRasterizerDiscardEnable(
    VkCommandBuffer commandBuffer,       // Provided by VK_VERSION_1_3
    VkBool32 rasterizerDiscardEnable);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `rasterizerDiscardEnable` controls whether primitives are discarded immediately before the rasterization stage.

This command sets the discard enable for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the ` VkPipelineRasterizationStateCreateInfo::rasterizerDiscardEnable` value used to create the currently active pipeline.

### Valid Usage (Implicit)

- VUID-vkCmdSetRasterizerDiscardEnable-commandBuffer-parameter: `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetRasterizerDiscardEnable-commandBuffer-recording: `commandBuffer` must be in the recording state

- VUID-vkCmdSetRasterizerDiscardEnable-commandBuffer-cmdpool: The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

### Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized
25.2. Rasterization Order

Within a subpass of a render pass instance, for a given (x,y,layer,sample) sample location, the following operations are guaranteed to execute in rasterization order, for each separate primitive that includes that sample location:

1. Fragment operations, in the order defined
2. Blending, logic operations, and color writes

Execution of these operations for each primitive in a subpass occurs in primitive order.

25.3. Multisampling

Multisampling is a mechanism to antialias all Vulkan primitives: points, lines, and polygons. The technique is to sample all primitives multiple times at each pixel. Each sample in each framebuffer attachment has storage for a color, depth, and/or stencil value, such that per-fragment operations apply to each sample independently. The color sample values can be later resolved to a single color (see Resolving Multisample Images and the Render Pass chapter for more details on how to resolve multisample images to non-multisample images).

Vulkan defines rasterization rules for single-sample modes in a way that is equivalent to a multisample mode with a single sample in the center of each fragment.

Each fragment includes a coverage mask with a single bit for each sample in the fragment, and a number of depth values and associated data for each sample.

It is understood that each pixel has rasterizationSamples locations associated with it. These locations are exact positions, rather than regions or areas, and each is referred to as a sample point. The sample points associated with a pixel must be located inside or on the boundary of the unit square that is considered to bound the pixel. Furthermore, the relative locations of sample points may be identical for each pixel in the framebuffer, or they may differ.

If the current pipeline includes a fragment shader with one or more variables in its interface decorated with Sample and Input, the data associated with those variables will be assigned independently for each sample. The values for each sample must be evaluated at the location of the sample. The data associated with any other variables not decorated with Sample and Input need not be evaluated independently for each sample.

A coverage mask is generated for each fragment, based on which samples within that fragment are
determined to be within the area of the primitive that generated the fragment.

Single pixel fragments have one set of samples. Each set of samples has a number of samples determined by `VkPipelineMultisampleStateCreateInfo::rasterizationSamples`. Each sample in a set is assigned a unique sample index \( i \) in the range \([0, \text{rasterizationSamples})\).

Each sample in a fragment is also assigned a unique coverage index \( j \) in the range \([0, n \times \text{rasterizationSamples})\), where \( n \) is the number of sets in the fragment. If the fragment contains a single set of samples, the coverage index is always equal to the sample index.

The coverage mask includes \( B \) bits packed into \( W \) words, defined as:

\[
B = n \times \text{rasterizationSamples} \\
W = \lceil B/32 \rceil
\]

Bit \( b \) in coverage mask word \( w \) is 1 if the sample with coverage index \( j = 32 \times w + b \) is covered, and 0 otherwise.

If the `standardSampleLocations` member of `VkPhysicalDeviceLimits` is `VK_TRUE`, then the sample counts `VK_SAMPLE_COUNT_1_BIT`, `VK_SAMPLE_COUNT_2_BIT`, `VK_SAMPLE_COUNT_4_BIT`, `VK_SAMPLE_COUNT_8_BIT`, and `VK_SAMPLE_COUNT_16_BIT` have sample locations as listed in the following table, with the \( i \)th entry in the table corresponding to sample index \( i \). `VK_SAMPLE_COUNT_32_BIT` and `VK_SAMPLE_COUNT_64_BIT` do not have standard sample locations. Locations are defined relative to an origin in the upper left corner of the fragment.
<table>
<thead>
<tr>
<th>Sample count</th>
<th>Sample Locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_SAMPLE_COUNT_1_BIT</td>
<td>(0.5,0.5)</td>
</tr>
<tr>
<td>VK_SAMPLE_COUNT_2_BIT</td>
<td>(0.75,0.75)</td>
</tr>
<tr>
<td></td>
<td>(0.25,0.25)</td>
</tr>
<tr>
<td>VK_SAMPLE_COUNT_4_BIT</td>
<td>(0.375, 0.125)</td>
</tr>
<tr>
<td></td>
<td>(0.875, 0.375)</td>
</tr>
<tr>
<td></td>
<td>(0.125, 0.625)</td>
</tr>
<tr>
<td></td>
<td>(0.625, 0.875)</td>
</tr>
<tr>
<td>VK_SAMPLE_COUNT_8_BIT</td>
<td>(0.5625, 0.3125)</td>
</tr>
<tr>
<td></td>
<td>(0.4375, 0.6875)</td>
</tr>
<tr>
<td></td>
<td>(0.8125, 0.5625)</td>
</tr>
<tr>
<td></td>
<td>(0.3125, 0.1875)</td>
</tr>
<tr>
<td></td>
<td>(0.1875, 0.8125)</td>
</tr>
<tr>
<td></td>
<td>(0.0625, 0.4375)</td>
</tr>
<tr>
<td></td>
<td>(0.6875, 0.9375)</td>
</tr>
<tr>
<td></td>
<td>(0.9375, 0.0625)</td>
</tr>
<tr>
<td>VK_SAMPLE_COUNT_16_BIT</td>
<td>(0.5625, 0.5625)</td>
</tr>
<tr>
<td></td>
<td>(0.4375, 0.3125)</td>
</tr>
<tr>
<td></td>
<td>(0.3125, 0.625)</td>
</tr>
<tr>
<td></td>
<td>(0.75, 0.4375)</td>
</tr>
<tr>
<td></td>
<td>(0.1875, 0.375)</td>
</tr>
<tr>
<td></td>
<td>(0.625, 0.8125)</td>
</tr>
<tr>
<td></td>
<td>(0.8125, 0.6875)</td>
</tr>
<tr>
<td></td>
<td>(0.6875, 0.1875)</td>
</tr>
<tr>
<td></td>
<td>(0.375, 0.875)</td>
</tr>
<tr>
<td></td>
<td>(0.5, 0.0625)</td>
</tr>
<tr>
<td></td>
<td>(0.25, 0.125)</td>
</tr>
<tr>
<td></td>
<td>(0.125, 0.75)</td>
</tr>
<tr>
<td></td>
<td>(0.0, 0.5)</td>
</tr>
<tr>
<td></td>
<td>(0.9375, 0.25)</td>
</tr>
<tr>
<td></td>
<td>(0.875, 0.9375)</td>
</tr>
<tr>
<td></td>
<td>(0.0625, 0.0)</td>
</tr>
</tbody>
</table>
25.4. Sample Shading

Sample shading can be used to specify a minimum number of unique samples to process for each fragment. If sample shading is enabled, an implementation must invoke the fragment shader at least \( \max(\lceil \text{VkPipelineMultisampleStateCreateInfo::minSampleShading} \times \text{VkPipelineMultisampleStateCreateInfo::rasterizationSamples} \rceil, 1) \) times per fragment. If a fragment shader entry point's interface includes an input variable decorated with a BuiltIn of SampleId or SamplePosition, a value of 1.0 is used instead of minSampleShading.

Sample shading is enabled if at least one of the following conditions is true:

- \( \text{VkPipelineMultisampleStateCreateInfo::sampleShadingEnable} \) is set to VK_TRUE, or
- the fragment shader’s entry point interface includes input variables decorated with a BuiltIn of SampleId or SamplePosition built-ins.

If there are fewer invocations than covered samples, implementations may include those samples in fragment shader invocations in any manner as long as covered samples are all shaded at least once.

25.5. Points

A point is drawn by generating a set of fragments in the shape of a square centered around the vertex of the point. Each vertex has an associated point size controlling the width/height of that square. The point size is taken from the (potentially clipped) shader built-in PointSize written by:

- the geometry shader, if active;
- the tessellation evaluation shader, if active and no geometry shader is active;
- the vertex shader, otherwise

and clamped to the implementation-dependent point size range \([\text{pointSizeRange}[0], \text{pointSizeRange}[1]]\). The value written to PointSize must be greater than zero.

Not all point sizes need be supported, but the size 1.0 must be supported. The range of supported sizes and the size of evenly-spaced gradations within that range are implementation-dependent. The range and gradations are obtained from the pointSizeRange and pointSizeGranularity members of VkPhysicalDeviceLimits. If, for instance, the size range is from 0.1 to 2.0 and the gradation size is 0.1, then the sizes 0.1, 0.2, ..., 1.9, 2.0 are supported. Additional point sizes may also be supported. There is no requirement that these sizes be equally spaced. If an unsupported size is requested, the nearest supported size is used instead.

25.5.1. Basic Point Rasterization

Point rasterization produces a fragment for each fragment area group of framebuffer pixels with one or more sample points that intersect a region centered at the point’s \((x_f, y_f)\). This region is a square with side equal to the current point size. Coverage bits that correspond to sample points that intersect the region are 1, other coverage bits are 0. All fragments produced in rasterizing a point are assigned the same associated data, which are those of the vertex corresponding to the point.
However, the fragment shader built-in \texttt{PointCoord} contains point sprite texture coordinates. The $s$ and $t$ point sprite texture coordinates vary from zero to one across the point horizontally left-to-right and vertically top-to-bottom, respectively. The following formulas are used to evaluate $s$ and $t$:

\[
s = \frac{1}{2} + \frac{(x_p - x_f)}{\text{size}}
\]

\[
t = \frac{1}{2} + \frac{(y_p - y_f)}{\text{size}}
\]

where \text{size} is the point's size; $(x_p, y_p)$ is the location at which the point sprite coordinates are evaluated - this \textbf{may} be the framebuffer coordinates of the fragment center, or the location of a sample; and $(x_f, y_f)$ is the exact, unrounded framebuffer coordinate of the vertex for the point.

### 25.6. Line Segments

To \textbf{dynamically set} the line width, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetLineWidth(  
    VkCommandBuffer commandBuffer,  
    float lineWidth);
```

- \texttt{commandBuffer} is the command buffer into which the command will be recorded.
- \texttt{lineWidth} is the width of rasterized line segments.

This command sets the line width for subsequent drawing commands when the graphics pipeline is created with \texttt{VK_DYNAMIC_STATE_LINE_WIDTH} set in \texttt{VkPipelineDynamicStateCreateInfo::pDynamicStates}. Otherwise, this state is specified by the \texttt{VkPipelineRasterizationStateCreateInfo::lineWidth} value used to create the currently active pipeline.

#### Valid Usage

- \textbf{VUID-vkCmdSetLineWidth-lineWidth-00788}
  If the \texttt{wideLines} feature is not enabled, \textbf{lineWidth must be 1.0}

#### Valid Usage (Implicit)

- \textbf{VUID-vkCmdSetLineWidth-commandBuffer-parameter}
  \texttt{commandBuffer must} be a valid \texttt{VkCommandBuffer} handle
- \textbf{VUID-vkCmdSetLineWidth-commandBuffer-recording}
  \texttt{commandBuffer must} be in the \texttt{recording state}
- \textbf{VUID-vkCmdSetLineWidth-commandBuffer-cmdpool}
  The \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from \textbf{must} support graphics operations
Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Not all line widths need be supported for line segment rasterization, but width 1.0 antialiased segments must be provided. The range and gradations are obtained from the `lineWidthRange` and `lineWidthGranularity` members of `VkPhysicalDeviceLimits`. If, for instance, the size range is from 0.1 to 2.0 and the gradation size is 0.1, then the sizes 0.1, 0.2, ..., 1.9, 2.0 are supported. Additional line widths may also be supported. There is no requirement that these widths be equally spaced. If an unsupported width is requested, the nearest supported width is used instead.

25.6.1. Basic Line Segment Rasterization

Rasterized line segments produce fragments which intersect a rectangle centered on the line segment. Two of the edges are parallel to the specified line segment; each is at a distance of one-half the current width from that segment in directions perpendicular to the direction of the line. The other two edges pass through the line endpoints and are perpendicular to the direction of the specified line segment. Coverage bits that correspond to sample points that intersect the rectangle are 1, other coverage bits are 0.

Next we specify how the data associated with each rasterized fragment are obtained. Let \( p_r = (x_d, y_d) \) be the framebuffer coordinates at which associated data are evaluated. This may be the center of a fragment or the location of a sample within the fragment. When `rasterizationSamples` is `VK_SAMPLE_COUNT_1_BIT`, the fragment center must be used. Let \( p_a = (x_a, y_a) \) and \( p_b = (x_b, y_b) \) be initial and final endpoints of the line segment, respectively. Set

\[
    t = \frac{(p_r - p_a) \cdot (p_b - p_a)}{\| p_b - p_a \|^2}
\]

(Note that \( t = 0 \) at \( p_a \) and \( t = 1 \) at \( p_b \). Also note that this calculation projects the vector from \( p_a \) to \( p_r \) onto the line, and thus computes the normalized distance of the fragment along the line.)

If `strictLines` is `VK_TRUE`, line segments are rasterized using perspective or linear interpolation.

Perspective interpolation for a line segment interpolates two values in a manner that is correct when taking the perspective of the viewport into consideration, by way of the line segment’s clip
coordinates. An interpolated value $f$ can be determined by

$$f = \frac{(1 - t)f_a/w_a + tf_b/w_b}{(1 - t)/w_a + t/w_b}$$

where $f_a$ and $f_b$ are the data associated with the starting and ending endpoints of the segment, respectively; $w_a$ and $w_b$ are the clip $w$ coordinates of the starting and ending endpoints of the segment, respectively.

Linear interpolation for a line segment directly interpolates two values, and an interpolated value $f$ can be determined by

$$f = (1 - t)f_a + tf_b$$

where $f_a$ and $f_b$ are the data associated with the starting and ending endpoints of the segment, respectively.

The clip coordinate $w$ for a sample is determined using perspective interpolation. The depth value $z$ for a sample is determined using linear interpolation. Interpolation of fragment shader input values are determined by Interpolation decorations.

The above description documents the preferred method of line rasterization, and must be used when the implementation advertises the strictLines limit in VkPhysicalDeviceLimits as VK_TRUE.

When strictLines is VK_FALSE, the edges of the lines are generated as a parallelogram surrounding the original line. The major axis is chosen by noting the axis in which there is the greatest distance between the line start and end points. If the difference is equal in both directions then the X axis is chosen as the major axis. Edges 2 and 3 are aligned to the minor axis and are centered on the endpoints of the line as in Non strict lines, and each is lineWidth long. Edges 0 and 1 are parallel to the line and connect the endpoints of edges 2 and 3. Coverage bits that correspond to sample points that intersect the parallelogram are 1, other coverage bits are 0.

Samples that fall exactly on the edge of the parallelogram follow the polygon rasterization rules.

Interpolation occurs as if the parallelogram was decomposed into two triangles where each pair of vertices at each end of the line has identical attributes.
Only when `strictLines` is `VK_FALSE` implementations may deviate from the non-strict line algorithm described above in the following ways:

- Implementations may instead interpolate each fragment according to the formula in Basic Line Segment Rasterization using the original line segment endpoints.
- Rasterization of non-antialiased non-strict line segments may be performed using the rules defined in Bresenham Line Segment Rasterization.

### 25.6.2. Bresenham Line Segment Rasterization

Non-strict lines may also follow these rasterization rules for non-antialiased lines.

Line segment rasterization begins by characterizing the segment as either *x-major* or *y-major*. *x-major* line segments have slope in the closed interval [-1,1]; all other line segments are *y-major* (slope is determined by the segment's endpoints). We specify rasterization only for *x-major* segments except in cases where the modifications for *y-major* segments are not self-evident.

Ideally, Vulkan uses a *diamond-exit* rule to determine those fragments that are produced by rasterizing a line segment. For each fragment \( f \) with center at framebuffer coordinates \( x_f \) and \( y_f \), define a diamond-shaped region that is the intersection of four half planes:

\[
R_f = \{ (x, y) | |x - x_f| + |y - y_f| < \frac{1}{2} \}
\]

Essentially, a line segment starting at \( p_a \) and ending at \( p_b \) produces those fragments \( f \) for which the segment intersects \( R_c \) except if \( p_b \) is contained in \( R_c \).
To avoid difficulties when an endpoint lies on a boundary of $R_f$ we (in principle) perturb the supplied endpoints by a tiny amount. Let $p_a$ and $p_b$ have framebuffer coordinates $(x_a, y_a)$ and $(x_b, y_b)$, respectively. Obtain the perturbed endpoints $p'_a$ given by $(x_a, y_a) - (\epsilon, \epsilon)$ and $p'_b$ given by $(x_b, y_b) - (\epsilon, \epsilon)$. Rasterizing the line segment starting at $p_a$ and ending at $p_b$ produces those fragments $f$ for which the segment starting at $p'_a$ and ending on $p'_b$ intersects $R_f$, except if $p'_b$ is contained in $R_f$. $\epsilon$ is chosen to be so small that rasterizing the line segment produces the same fragments when $\delta$ is substituted for $\epsilon$ for any $0 < \delta \leq \epsilon$.

When $p_a$ and $p_b$ lie on fragment centers, this characterization of fragments reduces to Bresenham’s algorithm with one modification: lines produced in this description are “half-open”, meaning that the final fragment (corresponding to $p_b$) is not drawn. This means that when rasterizing a series of connected line segments, shared endpoints will be produced only once rather than twice (as would occur with Bresenham’s algorithm).

Implementations may use other line segment rasterization algorithms, subject to the following rules:

- The coordinates of a fragment produced by the algorithm must not deviate by more than one unit in either $x$ or $y$ framebuffer coordinates from a corresponding fragment produced by the diamond-exit rule.

- The total number of fragments produced by the algorithm must not differ from that produced by the diamond-exit rule by no more than one.

- For an $x$-major line, two fragments that lie in the same framebuffer-coordinate column must not be produced (for a $y$-major line, two fragments that lie in the same framebuffer-coordinate row must not be produced).
If two line segments share a common endpoint, and both segments are either x-major (both left-to-right or both right-to-left) or y-major (both bottom-to-top or both top-to-bottom), then rasterizing both segments must not produce duplicate fragments. Fragments also must not be omitted so as to interrupt continuity of the connected segments.

The actual width \( w \) of Bresenham lines is determined by rounding the line width to the nearest integer, clamping it to the implementation-dependent lineWidthRange (with both values rounded to the nearest integer), then clamping it to be no less than 1.

Bresenham line segments of width other than one are rasterized by offsetting them in the minor direction (for an x-major line, the minor direction is y, and for a y-major line, the minor direction is x) and producing a row or column of fragments in the minor direction. If the line segment has endpoints given by \((x_0, y_0)\) and \((x_1, y_1)\) in framebuffer coordinates, the segment with endpoints \((x_0, y_0 - \frac{w-1}{2})\) and \((x_1, y_1 - \frac{w-1}{2})\) is rasterized, but instead of a single fragment, a column of fragments of height \( w \) (a row of fragments of length \( w \) for a y-major segment) is produced at each x (y for y-major) location. The lowest fragment of this column is the fragment that would be produced by rasterizing the segment of width 1 with the modified coordinates.

The preferred method of attribute interpolation for a wide line is to generate the same attribute values for all fragments in the row or column described above, as if the adjusted line was used for interpolation and those values replicated to the other fragments, except for \text{FragCoord} which is interpolated as usual. Implementations may instead interpolate each fragment according to the formula in Basic Line Segment Rasterization, using the original line segment endpoints.

When Bresenham lines are being rasterized, sample locations may all be treated as being at the pixel center (this may affect attribute and depth interpolation).

\begin{quote}
Note
The sample locations described above are not used for determining coverage, they are only used for things like attribute interpolation. The rasterization rules that determine coverage are defined in terms of whether the line intersects pixels, as opposed to the point sampling rules used for other primitive types. So these rules are independent of the sample locations. One consequence of this is that Bresenham lines cover the same pixels regardless of the number of rasterization samples, and cover all samples in those pixels (unless masked out or killed).
\end{quote}

### 25.7. Polygons

A polygon results from the decomposition of a triangle strip, triangle fan or a series of independent triangles. Like points and line segments, polygon rasterization is controlled by several variables in the \text{VkPipelineRasterizationStateCreateInfo} structure.

#### 25.7.1. Basic Polygon Rasterization

The first step of polygon rasterization is to determine whether the triangle is \textit{back-facing} or \textit{front-facing}. This determination is made based on the sign of the (clipped or unclipped) polygon’s area computed in framebuffer coordinates. One way to compute this area is:
\[ a = -\frac{1}{2} \sum_{i=0}^{n-1} x_i^j y_i^{j+1} - x_i^{j+1} y_i^j \]

where \( x_i^j \) and \( y_i^j \) are the x and y framebuffer coordinates of the \( i \)th vertex of the \( n \)-vertex polygon (vertices are numbered starting at zero for the purposes of this computation) and \( i \oplus 1 \) is \((i + 1) \mod n\).

The interpretation of the sign of \( a \) is determined by the \texttt{VkPipelineRasterizationStateCreateInfo ::frontFace} property of the currently active pipeline. Possible values are:

\begin{verbatim}
// Provided by VK_VERSION_1_0
typedef enum VkFrontFace {
    VK_FRONT_FACE_COUNTER_CLOCKWISE = 0,
    VK_FRONT_FACE_CLOCKWISE = 1,
} VkFrontFace;
\end{verbatim}

- \texttt{VK_FRONT_FACE_COUNTER_CLOCKWISE} specifies that a triangle with positive area is considered front-facing.
- \texttt{VK_FRONT_FACE_CLOCKWISE} specifies that a triangle with negative area is considered front-facing.

Any triangle which is not front-facing is back-facing, including zero-area triangles.

To dynamically set the front face orientation, call:

\begin{verbatim}
// Provided by VK_VERSION_1_3
void vkCmdSetFrontFace(
    VkCommandBuffer commandBuffer,
    VkFrontFace frontFace);
\end{verbatim}

- \texttt{commandBuffer} is the command buffer into which the command will be recorded.
- \texttt{frontFace} is a \texttt{VkFrontFace} value specifying the front-facing triangle orientation to be used for culling.

This command sets the front face orientation for subsequent drawing commands when the graphics pipeline is created with \texttt{VK_DYNAMIC_STATE_FRONT_FACE} set in \texttt{VkPipelineDynamicStateCreateInfo::pDynamicStates}. Otherwise, this state is specified by the \texttt{VkPipelineRasterizationStateCreateInfo::frontFace} value used to create the currently active pipeline.

**Valid Usage (Implicit)**

- \texttt{VUID-vkCmdSetFrontFace-commandBuffer-parameter}
  \texttt{commandBuffer} must be a valid \texttt{VkCommandBuffer} handle
- \texttt{VUID-vkCmdSetFrontFace-frontFace-parameter}
  \texttt{frontFace} must be a valid \texttt{VkFrontFace} value
• VUID-vkCmdSetFrontFace-commandBuffer-recording
  commandBuffer must be in the recording state

• VUID-vkCmdSetFrontFace-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

**Host Synchronization**

• Host access to commandBuffer must be externally synchronized

• Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Once the orientation of triangles is determined, they are culled according to the VkPipelineRasterizationStateCreateInfo::cullMode property of the currently active pipeline. Possible values are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkCullModeFlagBits {
    VK_CULL_MODE_NONE = 0,
    VK_CULL_MODE_FRONT_BIT = 0x00000001,
    VK_CULL_MODE_BACK_BIT = 0x00000002,
    VK_CULL_MODE_FRONT_AND_BACK = 0x00000003,
} VkCullModeFlagBits;
```

• **VK_CULL_MODE_NONE** specifies that no triangles are discarded

• **VK_CULL_MODE_FRONT_BIT** specifies that front-facing triangles are discarded

• **VK_CULL_MODE_BACK_BIT** specifies that back-facing triangles are discarded

• **VK_CULL_MODE_FRONT_AND_BACK** specifies that all triangles are discarded.

Following culling, fragments are produced for any triangles which have not been discarded.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkCullModeFlags;
```
VkCullModeFlags is a bitmask type for setting a mask of zero or more VkCullModeFlagBits.

To dynamically set the cull mode, call:

```c
// Provided by VK_VERSION_1_3
void vkCmdSetCullMode(
    VkCommandBuffer commandBuffer, 
    VkCullModeFlags cullMode);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `cullMode` specifies the cull mode property to use for drawing.

This command sets the cull mode for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_CULL_MODE` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineRasterizationStateCreateInfo::cullMode` value used to create the currently active pipeline.

### Valid Usage (Implicit)

- VUID-vkCmdSetCullMode-commandBuffer-parameter `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdSetCullMode-cullMode-parameter `cullMode` must be a valid combination of `VkCullModeFlagBits` values
- VUID-vkCmdSetCullMode-commandBuffer-recording `commandBuffer` must be in the recording state
- VUID-vkCmdSetCullMode-commandBuffer-cmdpool The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

### Host Synchronization

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The rule for determining which fragments are produced by polygon rasterization is called \textit{point sampling}. The two-dimensional projection obtained by taking the x and y framebuffer coordinates of the polygon’s vertices is formed. Fragments are produced for any fragment area groups of pixels for which any sample points lie inside of this polygon. Coverage bits that correspond to sample points that satisfy the point sampling criteria are 1, other coverage bits are 0. Special treatment is given to a sample whose sample location lies on a polygon edge. In such a case, if two polygons lie on either side of a common edge (with identical endpoints) on which a sample point lies, then exactly one of the polygons \textbf{must} result in a covered sample for that fragment during rasterization. As for the data associated with each fragment produced by rasterizing a polygon, we begin by specifying how these values are produced for fragments in a triangle.

\textit{Barycentric coordinates} are a set of three numbers, a, b, and c, each in the range [0,1], with $a + b + c = 1$. These coordinates uniquely specify any point \( p \) within the triangle or on the triangle’s boundary as

\[ p = a \, p_a + b \, p_b + c \, p_c \]

where \( p_a \), \( p_b \), and \( p_c \) are the vertices of the triangle. \( a \), \( b \), and \( c \) are determined by:

\[ a = \frac{A(p \, p_b \, p_c)}{A(p_a \, p_b \, p_c)}, \quad b = \frac{A(p \, p_a \, p_c)}{A(p_a \, p_b \, p_c)}, \quad c = \frac{A(p \, p_a \, p_b)}{A(p_a \, p_b \, p_c)}, \]

where \( A(lmn) \) denotes the area in framebuffer coordinates of the triangle with vertices \( l \), \( m \), and \( n \).

Denote an associated datum at \( p_a \), \( p_b \), or \( p_c \) as \( f_a \), \( f_b \), or \( f_c \), respectively.

\textit{Perspective interpolation} for a triangle interpolates three values in a manner that is correct when taking the perspective of the viewport into consideration, by way of the triangle’s clip coordinates. An interpolated value \( f \) can be determined by

\[ f = \frac{a f_a / w_a + b f_b / w_b + c f_c / w_c}{a / w_a + b / w_b + c / w_c} \]

where \( w_a \), \( w_b \), and \( w_c \) are the clip \( w \) coordinates of \( p_a \), \( p_b \), and \( p_c \) respectively. \( a \), \( b \), and \( c \) are the barycentric coordinates of the location at which the data are produced.

\textit{Linear interpolation} for a triangle directly interpolates three values, and an interpolated value \( f \) can be determined by

\[ f = a \, f_a + b \, f_b + c \, f_c \]

where \( f_a \), \( f_b \), and \( f_c \) are the data associated with \( p_a \), \( p_b \), and \( p_c \) respectively.

The clip coordinate \( w \) for a sample is determined using perspective interpolation. The depth value \( z \) for a sample is determined using linear interpolation. Interpolation of fragment shader input values are determined by \textit{Interpolation decorations}.

For a polygon with more than three edges, such as are produced by clipping a triangle, a convex combination of the values of the datum at the polygon’s vertices \textbf{must} be used to obtain the value
assigned to each fragment produced by the rasterization algorithm. That is, it **must** be the case that at every fragment

\[ f = \sum_{i=1}^{n} a_i f_i \]

where \( n \) is the number of vertices in the polygon and \( f_i \) is the value of \( f \) at vertex \( i \). For each \( i \), \( 0 \leq a_i \leq 1 \) and \( \sum_{i=1}^{n} a_i = 1 \). The values of \( a_i \) **may** differ from fragment to fragment, but at vertex \( i \), \( a_i = 1 \) and \( a_j = 0 \) for \( j \neq i \).

---

**Note**

One algorithm that achieves the required behavior is to triangulate a polygon (without adding any vertices) and then treat each triangle individually as already discussed. A scan-line rasterizer that linearly interpolates data along each edge and then linearly interpolates data across each horizontal span from edge to edge also satisfies the restrictions (in this case the numerator and denominator of perspective interpolation are iterated independently, and a division is performed for each fragment).

### 25.7.2. Polygon Mode

Possible values of the `VkPipelineRasterizationStateCreateInfo::polygonMode` property of the currently active pipeline, specifying the method of rasterization for polygons, are:

---

```cpp
// Provided by VK_VERSION_1_0
typedef enum VkPolygonMode {
    VK_POLYGON_MODE_FILL = 0,
    VK_POLYGON_MODE_LINE = 1,
    VK_POLYGON_MODE_POINT = 2,
} VkPolygonMode;
```

- **VK_POLYGON_MODE_POINT** specifies that polygon vertices are drawn as points.
- **VK_POLYGON_MODE_LINE** specifies that polygon edges are drawn as line segments.
- **VK_POLYGON_MODE_FILL** specifies that polygons are rendered using the polygon rasterization rules in this section.

These modes affect only the final rasterization of polygons: in particular, a polygon’s vertices are shaded and the polygon is clipped and possibly culled before these modes are applied.

### 25.7.3. Depth Bias

The depth values of all fragments generated by the rasterization of a polygon **can** be biased (offset) by a single depth bias value \( \delta \) that is computed for that polygon.

**Depth Bias Enable**

The depth bias computation is enabled by the `depthBiasEnable` set with `vkCmdSetDepthBiasEnable`
or the corresponding `VkPipelineRasterizationStateCreateInfo::depthBiasEnable` value used to create the currently active pipeline. If the depth bias enable is `VK_FALSE`, no bias is applied and the fragment’s depth values are unchanged.

To **dynamically enable** whether to bias fragment depth values, call:

```c
// Provided by VK_VERSION_1_3
void vkCmdSetDepthBiasEnable(
    VkCommandBuffer commandBuffer,  // Required
    VkBool32 depthBiasEnable);      // Required
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `depthBiasEnable` controls whether to bias fragment depth values.

This command sets the depth bias enable for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineRasterizationStateCreateInfo::depthBiasEnable` value used to create the currently active pipeline.

**Valid Usage (Implicit)**

- VUID-vkCmdSetDepthBiasEnable-commandBuffer-parameter
  - `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdSetDepthBiasEnable-commandBuffer-recording
  - `commandBuffer` must be in the recording state
- VUID-vkCmdSetDepthBiasEnable-commandBuffer-cmdpool
  - The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

**Host Synchronization**

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

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Secondary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
</tbody>
</table>
**Depth Bias Computation**

The depth bias depends on three parameters:

- `depthBiasSlopeFactor` scales the maximum depth slope \( m \) of the polygon
- `depthBiasConstantFactor` scales the minimum resolvable difference \( r \) of the depth attachment
- the scaled terms are summed to produce a value which is then clamped to a minimum or maximum value specified by `depthBiasClamp`

`depthBiasSlopeFactor`, `depthBiasConstantFactor`, and `depthBiasClamp` can each be positive, negative, or zero. These parameters are set as described for `vkCmdSetDepthBias` below.

The maximum depth slope \( m \) of a triangle is

\[
m = \sqrt{\left(\frac{\partial z_f}{\partial x_f}\right)^2 + \left(\frac{\partial z_f}{\partial y_f}\right)^2}
\]

where \((x_f, y_f, z_f)\) is a point on the triangle. \( m \) may be approximated as

\[
m = \max\left(\left|\frac{\partial z_f}{\partial x_f}\right|, \left|\frac{\partial z_f}{\partial y_f}\right|\right).
\]

The minimum resolvable difference \( r \) is a parameter that depends on the depth attachment representation. It is the smallest difference in framebuffer coordinate \( z \) values that is guaranteed to remain distinct throughout polygon rasterization and in the depth attachment. All pairs of fragments generated by the rasterization of two polygons with otherwise identical vertices, but \( z_f \) values that differ by \( r \), will have distinct depth values.

For fixed-point depth attachment representations, \( r \) is constant throughout the range of the entire depth attachment. Its value is implementation-dependent but must be at most

\[
r = 2 \times 2^n
\]

for an \( n \)-bit buffer. For floating-point depth attachment, there is no single minimum resolvable difference. In this case, the minimum resolvable difference for a given polygon is dependent on the maximum exponent, \( e \), in the range of \( z \) values spanned by the primitive. If \( n \) is the number of bits in the floating-point mantissa, the minimum resolvable difference, \( r \), for the given primitive is defined as

\[
r = 2^{e_n}
\]

If no depth attachment is present, \( r \) is undefined.

The bias value \( o \) for a polygon is...
\[ o = \text{dbclamp}(m \times \text{depthBiasSlopeFactor} + r \times \text{depthBiasConstantFactor}) \]

where \( \text{dbclamp}(x) = \begin{cases} 
    x & \text{depthBiasClamp} = 0 \text{ or } NaN \\
    \min(x, \text{depthBiasClamp}) & \text{depthBiasClamp} > 0 \\
    \max(x, \text{depthBiasClamp}) & \text{depthBiasClamp} < 0 
\end{cases} \)

\( m \) is computed as described above. If the depth attachment uses a fixed-point representation, \( m \) is a function of depth values in the range \([0,1]\), and \( o \) is applied to depth values in the same range.

Depth bias is applied to triangle topology primitives received by the rasterizer regardless of polygon mode. Depth bias may also be applied to line and point topology primitives received by the rasterizer.

To dynamically set the depth bias parameters, call:

```cpp
// Provided by VK_VERSION_1_0
void vkCmdSetDepthBias(
    VkCommandBuffer commandBuffer,
    float depthBiasConstantFactor,
    float depthBiasClamp,
    float depthBiasSlopeFactor);
```

- \( \text{commandBuffer} \) is the command buffer into which the command will be recorded.
- \( \text{depthBiasConstantFactor} \) is a scalar factor controlling the constant depth value added to each fragment.
- \( \text{depthBiasClamp} \) is the maximum (or minimum) depth bias of a fragment.
- \( \text{depthBiasSlopeFactor} \) is a scalar factor applied to a fragment's slope in depth bias calculations.

This command sets the depth bias parameters for subsequent drawing commands when the graphics pipeline is created with \( \text{VK_DYNAMIC_STATE_DEPTH_BIAS} \) set in \( \text{VkPipelineDynamicStateCreateInfo::pDynamicStates} \). Otherwise, this state is specified by the corresponding \( \text{VkPipelineRasterizationStateCreateInfo::depthBiasConstantFactor, depthBiasClamp, and depthBiasSlopeFactor} \) values used to create the currently active pipeline.

### Valid Usage

- VUID-vkCmdSetDepthBias-depthBiasClamp-00790
  If the \( \text{depthBiasClamp} \) feature is not enabled, \( \text{depthBiasClamp} \) must be 0.0

### Valid Usage (Implicit)

- VUID-vkCmdSetDepthBias-commandBuffer-parameter
  \( \text{commandBuffer} \) must be a valid \( \text{VkCommandBuffer} \) handle
- VUID-vkCmdSetDepthBias-commandBuffer-recording
  \( \text{commandBuffer} \) must be in the recording state
• VUID-vkCmdSetDepthBias-commandBuffer-cmdpool

The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations.

---

**Host Synchronization**

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

---

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

970
Chapter 26. Fragment Operations

Fragments produced by rasterization go through a number of operations to determine whether or how values produced by fragment shading are written to the framebuffer.

The following fragment operations adhere to rasterization order, and are typically performed in this order:

1. Scissor test
2. Sample mask test
3. Certain Fragment shading operations:
   ◦ Sample Mask Accesses
   ◦ Depth Replacement
4. Multisample coverage
5. Depth bounds test
6. Stencil test
7. Depth test
8. Sample counting
9. Coverage reduction

The coverage mask generated by rasterization describes the initial coverage of each sample covered by the fragment. Fragment operations will update the coverage mask to add or subtract coverage where appropriate. If a fragment operation results in all bits of the coverage mask being 0, the fragment is discarded, and no further operations are performed. Fragments can also be programmatically discarded in a fragment shader by executing one of

- `OpTerminateInvocation`
- `OpDemoteToHelperInvocationEXT`
- `OpKill`.

When one of the fragment operations in this chapter is described as “replacing” a fragment shader output, that output is replaced unconditionally, even if no fragment shader previously wrote to that output.

If there is a fragment shader and it declares the EarlyFragmentTests execution mode, fragment shading and multisample coverage operations should instead be performed after sample counting.

For a pipeline with the following properties:

- a fragment shader is specified
- the fragment shader does not write to storage resources;
- the fragment shader specifies the DepthReplacing execution mode; and
- either
the fragment shader specifies the **DepthUnchanged** execution mode;

- the fragment shader specifies the **DepthGreater** execution mode and the pipeline uses a `VkPipelineDepthStencilStateCreateInfo::depthCompareOp` of **VK_COMPARE_OP_GREATER** or **VK_COMPARE_OP_GREATER_OR_EQUAL**; or

- the fragment shader specifies the **DepthLess** execution mode and the pipeline uses a `VkPipelineDepthStencilStateCreateInfo::depthCompareOp` of **VK_COMPARE_OP_LESS** or **VK_COMPARE_OP_LESS_OR_EQUAL**

the implementation may perform depth bounds test before fragment shading and perform an additional depth test immediately after that using the interpolated depth value generated by rasterization.

Once all fragment operations have completed, fragment shader outputs for covered color attachment samples pass through framebuffer operations.

## 26.1. Scissor Test

The scissor test compares the framebuffer coordinates \((x_f, y_f)\) of each sample covered by a fragment against a **scissor rectangle** at the index equal to the fragment's `ViewportIndex`.

Each scissor rectangle is defined by a `VkRect2D`. These values are either set by the `VkPipelineViewportStateCreateInfo` structure during pipeline creation, or dynamically by the `vkCmdSetScissor` command.

A given sample is considered inside a scissor rectangle if \(x_i\) is in the range \([\text{VkRect2D}::\text{offset.x, VkRect2D}::\text{offset.x} + \text{VkRect2D}::\text{extent.x}]\), and \(y_i\) is in the range \([\text{VkRect2D}::\text{offset.y, VkRect2D}::\text{offset.y} + \text{VkRect2D}::\text{extent.y}]\). Samples with coordinates outside the scissor rectangle at the corresponding `ViewportIndex` will have their coverage set to 0.

To dynamically set the scissor rectangles, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetScissor(
    VkCommandBuffer commandBuffer,   commandBuffer,
    uint32_t firstScissor,            firstScissor,
    uint32_t scissorCount,            scissorCount,
    const VkRect2D* pScissors);       pScissors);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `firstScissor` is the index of the first scissor whose state is updated by the command.
- `scissorCount` is the number of scissors whose rectangles are updated by the command.
- `pScissors` is a pointer to an array of `VkRect2D` structures defining scissor rectangles.

The scissor rectangles taken from element \(i\) of `pScissors` replace the current state for the scissor index `firstScissor + i`, for \(i\) in \([0, \text{scissorCount}]\).

This command sets the scissor rectangles for subsequent drawing commands when the graphics
pipeline is created with VK_DYNAMIC_STATE_SCISSOR set in VkPipelineDynamicStateCreateInfo::pDynamicStates. Otherwise, this state is specified by the VkPipelineViewportStateCreateInfo::pScissors values used to create the currently active pipeline.

**Valid Usage**

- **VUID-vkCmdSetScissor-firstScissor-00592**
  The sum of firstScissor and scissorCount must be between 1 and VkPhysicalDeviceLimits::maxViewports, inclusive

- **VUID-vkCmdSetScissor-firstScissor-00593**
  If the multiViewport feature is not enabled, firstScissor must be 0

- **VUID-vkCmdSetScissor-scissorCount-00594**
  If the multiViewport feature is not enabled, scissorCount must be 1

- **VUID-vkCmdSetScissor-x-00595**
  The x and y members of offset member of any element of pScissors must be greater than or equal to 0

- **VUID-vkCmdSetScissor-offset-00596**
  Evaluation of (offset.x + extent.width) must not cause a signed integer addition overflow for any element of pScissors

- **VUID-vkCmdSetScissor-offset-00597**
  Evaluation of (offset.y + extent.height) must not cause a signed integer addition overflow for any element of pScissors

**Valid Usage (Implicit)**

- **VUID-vkCmdSetScissor-commandBuffer-parameter**
  commandBuffer must be a valid VkCommandBuffer handle

- **VUID-vkCmdSetScissor-pScissors-parameter**
  pScissors must be a valid pointer to an array of scissorCount VkRect2D structures

- **VUID-vkCmdSetScissor-commandBuffer-recording**
  commandBuffer must be in the recording state

- **VUID-vkCmdSetScissor-commandBuffer-cmdpool**
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

- **VUID-vkCmdSetScissor-scissorCount-arraylength**
  scissorCount must be greater than 0

**Host Synchronization**

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally
26.2. Sample Mask Test

The sample mask test compares the coverage mask for a fragment with the sample mask defined by VkPipelineMultisampleStateCreateInfo::pSampleMask.

Each bit of the coverage mask is associated with a sample index as described in the rasterization chapter. If the bit in VkPipelineMultisampleStateCreateInfo::pSampleMask which is associated with that same sample index is set to 0, the coverage mask bit is set to 0.

26.3. Fragment Shading

Fragment shaders are invoked for each fragment, or as helper invocations.

Most operations in the fragment shader are not performed in rasterization order, with exceptions called out in the following sections.

For fragment shaders invoked by fragments, the following rules apply:

- A fragment shader must not be executed if a fragment operation that executes before fragment shading discards the fragment.

- A fragment shader may not be executed if:
  - An implementation determines that another fragment shader, invoked by a subsequent primitive in primitive order, overwrites all results computed by the shader (including writes to storage resources).
  - Any other fragment operation discards the fragment, and the shader does not write to any storage resources.

- Otherwise, at least one fragment shader must be executed.
  - If sample shading is enabled and multiple invocations per fragment are required, additional invocations must be executed as specified.
  - Each covered sample must be included in at least one fragment shader invocation.

If no fragment shader is included in the pipeline, no fragment shader is executed, and undefined values may be written to all color attachment outputs during this fragment operation.
Note

Multiple fragment shader invocations may be executed for the same fragment for any number of implementation-dependent reasons. When there is more than one fragment shader invocation per fragment, the association of samples to invocations is implementation-dependent. Stores and atomics performed by these additional invocations have the normal effect.

For example, if the subpass includes multiple views in its view mask, a fragment shader may be invoked separately for each view.

26.3.1. Sample Mask

Reading from the SampleMask built-in in the Input storage class will return the coverage mask for the current fragment as calculated by fragment operations that executed prior to fragment shading.

If sample shading is enabled, fragment shaders will only see values of 1 for samples being shaded - other bits will be 0.

Each bit of the coverage mask is associated with a sample index as described in the rasterization chapter. If the bit in SampleMask which is associated with that same sample index is set to 0, that coverage mask bit is set to 0.

Values written to the SampleMask built-in in the Output storage class will be used by the multisample coverage operation, with the same encoding as the input built-in.

26.3.2. Depth Replacement

Writing to the FragDepth built-in will replace the fragment’s calculated depth values for each sample in the input SampleMask. Depth testing performed after the fragment shader for this fragment will use this new value as zf.

26.4. Multisample Coverage

If a fragment shader is active and its entry point’s interface includes a built-in output variable decorated with SampleMask, the coverage mask is ANDed with the bits of the SampleMask built-in to generate a new coverage mask. If sample shading is enabled, bits written to SampleMask corresponding to samples that are not being shaded by the fragment shader invocation are ignored. If no fragment shader is active, or if the active fragment shader does not include SampleMask in its interface, the coverage mask is not modified.

Next, the fragment alpha value and coverage mask are modified based on the alphaToCoverageEnable and alphaToOneEnable members of the VkPipelineMultisampleStateCreateInfo structure.

All alpha values in this section refer only to the alpha component of the fragment shader output that has a Location and Index decoration of zero (see the Fragment Output Interface section). If that shader output has an integer or unsigned integer type, then these operations are skipped.
If \texttt{alphaToCoverageEnable} is enabled, a temporary coverage mask is generated where each bit is
determined by the fragment’s alpha value, which is ANDed with the fragment coverage mask.

No specific algorithm is specified for converting the alpha value to a temporary coverage mask. It is
intended that the number of 1’s in this value be proportional to the alpha value (clamped to \([0,1]\)),
with all 1’s corresponding to a value of 1.0 and all 0’s corresponding to 0.0. The algorithm \textbf{may} be
different at different framebuffer coordinates.

\begin{quote}
\textbf{Note}  
Using different algorithms at different framebuffer coordinates \textbf{may} help to avoid
artifacts caused by regular coverage sample locations.
\end{quote}

Finally, if \texttt{alphaToOneEnable} is enabled, each alpha value is replaced by the maximum representable
alpha value for fixed-point color attachments, or by 1.0 for floating-point attachments. Otherwise,
the alpha values are not changed.

\section*{26.5. Depth and Stencil Operations}

Pipeline state controlling the \texttt{depth bounds tests}, \texttt{stencil test}, and \texttt{depth test} is specified through the
members of the \texttt{VkPipelineDepthStencilStateCreateInfo} structure.

The \texttt{VkPipelineDepthStencilStateCreateInfo} structure is defined as:

\begin{verbatim}
// Provided by VK_VERSION_1_0
typedef struct VkPipelineDepthStencilStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineDepthStencilStateCreateFlags flags;
    VkBool32 depthTestEnable;
    VkBool32 depthWriteEnable;
    VkCompareOp depthCompareOp;
    VkBool32 depthBoundsTestEnable;
    VkBool32 stencilTestEnable;
    VkStencilOpState front;
    VkStencilOpState back;
    float minDepthBounds;
    float maxDepthBounds;
} VkPipelineDepthStencilStateCreateInfo;
\end{verbatim}

\begin{itemize}
\item \texttt{sType} is the type of this structure.
\item \texttt{pNext} is \texttt{NULL} or a pointer to a structure extending this structure.
\item \texttt{flags} is reserved for future use.
\item \texttt{depthTestEnable} controls whether \texttt{depth testing} is enabled.
\item \texttt{depthWriteEnable} controls whether \texttt{depth writes} are enabled when \texttt{depthTestEnable} is \texttt{VK_TRUE}.
\texttt{Depth writes are always disabled when \texttt{depthTestEnable} is \texttt{VK_FALSE}.}
\item \texttt{depthCompareOp} is a \texttt{VkCompareOp} value specifying the comparison operator to use in the \texttt{Depth}
\end{itemize}
Comparison step of the depth test.

- **depthBoundsTestEnable** controls whether depth bounds testing is enabled.
- **stencilTestEnable** controls whether stencil testing is enabled.
- **front** and **back** are **VkStencilOpState** values controlling the corresponding parameters of the stencil test.
- **minDepthBounds** is the minimum depth bound used in the depth bounds test.
- **maxDepthBounds** is the maximum depth bound used in the depth bounds test.

### Valid Usage

- VUID-VkPipelineDepthStencilStateCreateInfo-depthBoundsTestEnable-00598
  If the depthBounds feature is not enabled, **depthBoundsTestEnable** must be **VK_FALSE**

### Valid Usage (Implicit)

- VUID-VkPipelineDepthStencilStateCreateInfo-sType-sType
  **sType** must be **VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO**

- VUID-VkPipelineDepthStencilStateCreateInfo-pNext-pNext
  **pNext** must be **NULL**

- VUID-VkPipelineDepthStencilStateCreateInfo-flags-zerobitmask
  **flags** must be **0**

- VUID-VkPipelineDepthStencilStateCreateInfo-depthCompareOp-parameter
  **depthCompareOp** must be a valid **VkCompareOp** value

- VUID-VkPipelineDepthStencilStateCreateInfo-front-parameter
  **front** must be a valid **VkStencilOpState** structure

- VUID-VkPipelineDepthStencilStateCreateInfo-back-parameter
  **back** must be a valid **VkStencilOpState** structure

// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineDepthStencilStateCreateFlags;

**VkPipelineDepthStencilStateCreateFlags** is a bitmask type for setting a mask, but is currently reserved for future use.

### 26.6. Depth Bounds Test

The depth bounds test compares the depth value \( z_i \) in the depth/stencil attachment at each sample’s framebuffer coordinates \((x_i, y_i, z_i)\) and sample index \( i \) against a set of depth bounds.

The depth bounds are determined by two floating point values defining a minimum (\( \text{minDepthBounds} \)) and maximum (\( \text{maxDepthBounds} \)) depth value. These values are either set by the
A given sample is considered within the depth bounds if \( z_a \) is in the range \([\text{minDepthBounds}, \text{maxDepthBounds}]\). Samples with depth attachment values outside of the depth bounds will have their coverage set to 0.

If the depth bounds test is disabled, or if there is no depth attachment, the coverage mask is unmodified by this operation.

To **dynamically enable or disable** the depth bounds test, call:

```c
// Provided by VK_VERSION_1_3
void vkCmdSetDepthBoundsTestEnable(
    VkCommandBuffer commandBuffer,  // commandBuffer is the command buffer into which the command will be recorded.
    VkBool32 depthBoundsTestEnable   // depthBoundsTestEnable specifies if the depth bounds test is enabled.
);
```

This command sets the depth bounds enable for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_DEPTH_BOUNDS_TEST_ENABLE` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineDepthStencilStateCreateInfo::depthBoundsTestEnable` value used to create the currently active pipeline.

### Valid Usage (Implicit)

- **VUID-vkCmdSetDepthBoundsTestEnable-commandBuffer-parameter**
  - `commandBuffer` **must** be a valid `VkCommandBuffer` handle

- **VUID-vkCmdSetDepthBoundsTestEnable-commandBuffer-recording**
  - `commandBuffer` **must** be in the recording state

- **VUID-vkCmdSetDepthBoundsTestEnable-commandBuffer-cmdpool**
  - The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

### Host Synchronization

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

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

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To dynamically set the depth bounds range, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetDepthBounds(
    VkCommandBuffer commandBuffer,
    float minDepthBounds,
    float maxDepthBounds);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `minDepthBounds` is the minimum depth bound.
- `maxDepthBounds` is the maximum depth bound.

This command sets the depth bounds range for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_DEPTH_BOUNDS` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineDepthStencilStateCreateInfo::minDepthBounds` and `VkPipelineDepthStencilStateCreateInfo::maxDepthBounds` values used to create the currently active pipeline.

Valid Usage

- VUID-vkCmdSetDepthBounds-minDepthBounds-02508
  `minDepthBounds` must be between 0.0 and 1.0, inclusive

- VUID-vkCmdSetDepthBounds-maxDepthBounds-02509
  `maxDepthBounds` must be between 0.0 and 1.0, inclusive

Valid Usage (Implicit)

- VUID-vkCmdSetDepthBounds-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetDepthBounds-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdSetDepthBounds-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations
Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

26.7. Stencil Test

The stencil test compares the stencil attachment value $s_a$ in the depth/stencil attachment at each sample’s framebuffer coordinates $(x_f, y_f)$ and sample index $i$ against a stencil reference value.

If the stencil test is not enabled, as specified by `vkCmdSetStencilTestEnable` or `VkPipelineDepthStencilStateCreateInfo::stencilTestEnable`, or if there is no stencil attachment, the coverage mask is unmodified by this operation.

The stencil test is controlled by one of two sets of stencil-related state, the front stencil state and the back stencil state. Stencil tests and writes use the back stencil state when processing fragments generated by back-facing polygons, and the front stencil state when processing fragments generated by front-facing polygons or any other primitives.

The comparison operation performed is determined by the `VkCompareOp` value set by `vkCmdSetStencilOp::compareOp`, or by `VkStencilOpState::compareOp` during pipeline creation.

The compare mask $s_c$ and stencil reference value $s_r$ of the front or the back stencil state set determine arguments of the comparison operation. $s_c$ is set by the `VkPipelineDepthStencilStateCreateInfo` structure during pipeline creation, or by the `vkCmdSetStencilCompareMask` command. $s_r$ is set by `VkPipelineDepthStencilStateCreateInfo` or by `vkCmdSetStencilReference`.

$s_r$ and $s_a$ are each independently combined with $s_c$ using a bitwise AND operation to create masked reference and attachment values $s'_r$ and $s'_a$. $s'_r$ and $s'_a$ are used as the reference and test values, respectively, in the operation specified by the `VkCompareOp`.

If the comparison evaluates to false, the coverage for the sample is set to 0.

A new stencil value $s_g$ is generated according to a stencil operation defined by `VkStencilOp` parameters set by `vkCmdSetStencilOp` or `VkPipelineDepthStencilStateCreateInfo`. If the stencil test fails, `failOp` defines the stencil operation used. If the stencil test passes however, the stencil op used
is based on the depth test - if it passes, `VkPipelineDepthStencilStateCreateInfo::passOp` is used, otherwise `VkPipelineDepthStencilStateCreateInfo::depthFailOp` is used.

The stencil attachment value $s_a$ is then updated with the generated stencil value $s_g$ according to the write mask $s_w$ defined by `writeMask` in `VkPipelineDepthStencilStateCreateInfo::front` and `VkPipelineDepthStencilStateCreateInfo::back` as:

$$s_a = (s_a \& \neg s_w) \mid (s_g \& s_w)$$

To **dynamically enable or disable** the stencil test, call:

```c
// Provided by VK_VERSION_1_3
void vkCmdSetStencilTestEnable(
    VkCommandBuffer commandBuffer,
    VkBool32 stencilTestEnable);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `stencilTestEnable` specifies if the stencil test is enabled.

This command sets the stencil test enable for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_STENCIL_TEST_ENABLE` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineDepthStencilStateCreateInfo::stencilTestEnable` value used to create the currently active pipeline.

#### Valid Usage (Implicit)

- VUID-vkCmdSetStencilTestEnable-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdSetStencilTestEnable-commandBuffer-recording
  `commandBuffer` must be in the recording state
- VUID-vkCmdSetStencilTestEnable-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

#### Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized
To **dynamically set** the stencil operation, call:

```c
// Provided by VK_VERSION_1_3
void vkCmdSetStencilOp(
    VkCommandBuffer commandBuffer,
    VkStencilFaceFlags faceMask,
    VkStencilOp failOp,
    VkStencilOp passOp,
    VkStencilOp depthFailOp,
    VkCompareOp compareOp);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `faceMask` is a bitmask of `VkStencilFaceFlagBits` specifying the set of stencil state for which to update the stencil operation.
- `failOp` is a `VkStencilOp` value specifying the action performed on samples that fail the stencil test.
- `passOp` is a `VkStencilOp` value specifying the action performed on samples that pass both the depth and stencil tests.
- `depthFailOp` is a `VkStencilOp` value specifying the action performed on samples that pass the stencil test and fail the depth test.
- `compareOp` is a `VkCompareOp` value specifying the comparison operator used in the stencil test.

This command sets the stencil operation for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_STENCIL_OP` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the corresponding `VkPipelineDepthStencilStateCreateInfo::failOp`, `passOp`, `depthFailOp`, and `compareOp` values used to create the currently active pipeline, for both front and back faces.

---

### Valid Usage (Implicit)

- **VUID-vkCmdSetStencilOp-commandBuffer-parameter**
  `commandBuffer` must be a valid `VkCommandBuffer` handle
- **VUID-vkCmdSetStencilOp-faceMask-parameter**
  `faceMask` must be a valid combination of `VkStencilFaceFlagBits` values
- **VUID-vkCmdSetStencilOp-faceMask-requiredbitmask**
  `faceMask` must not be 0
• VUID-vkCmdSetStencilOp-failOp-parameter
  failOp must be a valid VkStencilOp value

• VUID-vkCmdSetStencilOp-passOp-parameter
  passOp must be a valid VkStencilOp value

• VUID-vkCmdSetStencilOp-depthFailOp-parameter
  depthFailOp must be a valid VkStencilOp value

• VUID-vkCmdSetStencilOp-compareOp-parameter
  compareOp must be a valid VkCompareOp value

• VUID-vkCmdSetStencilOp-commandBuffer-recording
  commandBuffer must be in the recording state

• VUID-vkCmdSetStencilOp-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

Host Synchronization

• Host access to commandBuffer must be externally synchronized

• Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The VkStencilOpState structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkStencilOpState {
  VkStencilOp failOp;
  VkStencilOp passOp;
  VkStencilOp depthFailOp;
  VkCompareOp compareOp;
  uint32_t compareMask;
  uint32_t writeMask;
  uint32_t reference;
} VkStencilOpState;
```

• failOp is a VkStencilOp value specifying the action performed on samples that fail the stencil test.
• **passOp** is a VkStencilOp value specifying the action performed on samples that pass both the depth and stencil tests.

• **depthFailOp** is a VkStencilOp value specifying the action performed on samples that pass the stencil test and fail the depth test.

• **compareOp** is a VkCompareOp value specifying the comparison operator used in the stencil test.

• **compareMask** selects the bits of the unsigned integer stencil values participating in the stencil test.

• **writeMask** selects the bits of the unsigned integer stencil values updated by the stencil test in the stencil framebuffer attachment.

• **reference** is an integer stencil reference value that is used in the unsigned stencil comparison.

**Valid Usage (Implicit)**

- VUID-VkStencilOpState-failOp-parameter **failOp** must be a valid VkStencilOp value

- VUID-VkStencilOpState-passOp-parameter **passOp** must be a valid VkStencilOp value

- VUID-VkStencilOpState-depthFailOp-parameter **depthFailOp** must be a valid VkStencilOp value

- VUID-VkStencilOpState-compareOp-parameter **compareOp** must be a valid VkCompareOp value

To **dynamically set** the stencil compare mask, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetStencilCompareMask(
    VkCommandBuffer commandBuffer,
    VkStencilFaceFlags faceMask,
    uint32_t compareMask);
```

- **commandBuffer** is the command buffer into which the command will be recorded.

- **faceMask** is a bitmask of VkStencilFaceFlagBits specifying the set of stencil state for which to update the compare mask.

- **compareMask** is the new value to use as the stencil compare mask.

This command sets the stencil compare mask for subsequent drawing commands when the graphics pipeline is created with VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK set in VkPipelineDynamicStateCreateInfo::pDynamicStates. Otherwise, this state is specified by the VkStencilOpState::compareMask value used to create the currently active pipeline, for both front and back faces.
Valid Usage (Implicit)

- **VUID-vkCmdSetStencilCompareMask-commandBuffer-parameter**
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdSetStencilCompareMask-faceMask-parameter**
  `faceMask` must be a valid combination of `VkStencilFaceFlagBits` values

- **VUID-vkCmdSetStencilCompareMask-faceMask-requiredbitmask**
  `faceMask` must not be 0

- **VUID-vkCmdSetStencilCompareMask-commandBuffer-recording**
  `commandBuffer` must be in the recording state

- **VUID-vkCmdSetStencilCompareMask-commandBuffer-cmdpool**
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`VkStencilFaceFlagBits` values are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkStencilFaceFlagBits {
    VK_STENCIL_FACE_FRONT_BIT = 0x00000001,
    VK_STENCIL_FACE_BACK_BIT = 0x00000002,
    VK_STENCIL_FACE_FRONT_AND_BACK = 0x00000003,
    VK_STENCIL_FRONT_AND_BACK = VK_STENCIL_FACE_FRONT_AND_BACK,
} VkStencilFaceFlagBits;
```

- **VK_STENCIL_FACE_FRONT_BIT** specifies that only the front set of stencil state is updated.
- **VK_STENCIL_FACE_BACK_BIT** specifies that only the back set of stencil state is updated.
- **VK_STENCIL_FACE_FRONT_AND_BACK** is the combination of **VK_STENCIL_FACE_FRONT_BIT** and
VK_STENCIL_FACE_BACK_BIT, and specifies that both sets of stencil state are updated.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkStencilFaceFlags;
```

VkStencilFaceFlags is a bitmask type for setting a mask of zero or more VkStencilFaceFlagBits.

To dynamically set the stencil write mask, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetStencilWriteMask(
    VkCommandBuffer commandBuffer,
    VkStencilFaceFlags faceMask,
    uint32_t writeMask);
```

- commandBuffer is the command buffer into which the command will be recorded.
- faceMask is a bitmask of VkStencilFaceFlagBits specifying the set of stencil state for which to update the write mask, as described above for vkCmdSetStencilCompareMask.
- writeMask is the new value to use as the stencil write mask.

This command sets the stencil write mask for subsequent drawing commands when the graphics pipeline is created with VK_DYNAMIC_STATE_STENCIL_WRITE_MASK set in VkPipelineDynamicStateCreateInfo::pDynamicStates. Otherwise, this state is specified by the writeMask value used to create the currently active pipeline, for both VkPipelineDepthStencilStateCreateInfo::front and VkPipelineDepthStencilStateCreateInfo::back faces.

### Valid Usage (Implicit)

- VUID-vkCmdSetStencilWriteMask-commandBuffer-parameter commandBuffer must be a valid VkCommandBuffer handle
- VUID-vkCmdSetStencilWriteMask-faceMask-parameter faceMask must be a valid combination of VkStencilFaceFlagBits values
- VUID-vkCmdSetStencilWriteMask-faceMask-requiredbitmask faceMask must not be 0
- VUID-vkCmdSetStencilWriteMask-commandBuffer-recording commandBuffer must be in the recording state
- VUID-vkCmdSetStencilWriteMask-commandBuffer-cmdpool The VkCommandPool that commandBuffer was allocated from must support graphics operations
Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To dynamically set the stencil reference value, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetStencilReference(
    VkCommandBuffer commandBuffer,
    VkStencilFaceFlags faceMask,
    uint32_t reference);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `faceMask` is a bitmask of `VkStencilFaceFlagBits` specifying the set of stencil state for which to update the reference value, as described above for `vkCmdSetStencilCompareMask`.
- `reference` is the new value to use as the stencil reference value.

This command sets the stencil reference value for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_STENCIL_REFERENCE` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineDepthStencilStateCreateInfo::reference` value used to create the currently active pipeline, for both front and back faces.

Valid Usage (Implicit)

- VUID-vkCmdSetStencilReference-commandBuffer-parameter  
  `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdSetStencilReference-faceMask-parameter  
  `faceMask` must be a valid combination of `VkStencilFaceFlagBits` values
- VUID-vkCmdSetStencilReference-faceMask-requiredbitmask  
  `faceMask` must not be 0
- VUID-vkCmdSetStencilReference-commandBuffer-recording  
  `commandBuffer` must be in the recording state
Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Possible values of the `failOp`, `passOp`, and `depthFailOp` members of `VkStencilOpState`, specifying what happens to the stored stencil value if this or certain subsequent tests fail or pass, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkStencilOp {
    VK_STENCIL_OP_KEEP = 0,
    VK_STENCIL_OP_ZERO = 1,
    VK_STENCIL_OP_REPLACE = 2,
    VK_STENCIL_OP_INCREMENT_AND_CLAMP = 3,
    VK_STENCIL_OP_DECREMENT_AND_CLAMP = 4,
    VK_STENCIL_OP_INVERT = 5,
    VK_STENCIL_OP_INCREMENT_AND_WRAP = 6,
    VK_STENCIL_OP_DECREMENT_AND_WRAP = 7,
} VkStencilOp;
```

- `VK_STENCIL_OP_KEEP` keeps the current value.
- `VK_STENCIL_OP_ZERO` sets the value to 0.
- `VK_STENCIL_OP_REPLACE` sets the value to `reference`.
- `VK_STENCIL_OP_INCREMENT_AND_CLAMP` increments the current value and clamps to the maximum representable unsigned value.
- `VK_STENCIL_OP_DECREMENT_AND_CLAMP` decrements the current value and clamps to 0.
- `VK_STENCIL_OP_INVERT` bitwise-inverts the current value.
- `VK_STENCIL_OP_INCREMENT_AND_WRAP` increments the current value and wraps to 0 when the maximum value would have been exceeded.
• **VK_STENCIL_OP_DECREMENT_AND_WRAP** decrements the current value and wraps to the maximum possible value when the value would go below 0.

For purposes of increment and decrement, the stencil bits are considered as an unsigned integer.

### 26.8. Depth Test

The depth test compares the depth value $z_a$ in the depth/stencil attachment at each sample’s framebuffer coordinates $(x_f, y_f)$ and sample index $i$ against the sample’s depth value $z_c$. If there is no depth attachment then the depth test is skipped.

The depth test occurs in three stages, as detailed in the following sections.

#### 26.8.1. Depth Clamping and Range Adjustment

If **VkPipelineRasterizationStateCreateInfo**::*depthClampEnable* is enabled, $z_i$ is clamped to $[z_{\text{min}}, z_{\text{max}}]$, where $z_{\text{min}} = \text{min}(n, f)$, $z_{\text{max}} = \text{max}(n, f)$, and $n$ and $f$ are the *minDepth* and *maxDepth* depth range values of the viewport used by this fragment, respectively.

Following depth clamping:

- If $z_i$ is not in the range $[z_{\text{min}}, z_{\text{max}}]$, then $z_i$ is undefined following this step.

#### 26.8.2. Depth Comparison

If the depth test is not enabled, as specified by **vkCmdSetDepthTestEnable** or **VkPipelineDepthStencilStateCreateInfo**::*depthTestEnable*, then this step is skipped.

The comparison operation performed is determined by the **VkCompareOp** value set by **vkCmdSetDepthCompareOp**, or by **VkPipelineDepthStencilStateCreateInfo**::*depthCompareOp* during pipeline creation. $z_i$ and $z_a$ are used as the *reference* and *test* values, respectively, in the operation specified by the **VkCompareOp**.

If the comparison evaluates to false, the coverage for the sample is set to 0.

#### 26.8.3. Depth Attachment Writes

If depth writes are enabled, as specified by **vkCmdSetDepthWriteEnable** or **VkPipelineDepthStencilStateCreateInfo**::*depthWriteEnable*, and the comparison evaluated to true, the depth attachment value $z_a$ is set to the sample’s depth value $z_c$. If there is no depth attachment, no value is written.

To dynamically enable or disable the depth test, call:

```c
// Provided by VK_VERSION_1_3
void vkCmdSetDepthTestEnable(
    VkCommandBuffer commandBuffer,
    VkBool32 depthTestEnable);
```
• `commandBuffer` is the command buffer into which the command will be recorded.
• `depthTestEnable` specifies if the depth test is enabled.

This command sets the depth test enable for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_DEPTH_TEST_ENABLE` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineDepthStencilStateCreateInfo::depthTestEnable` value used to create the currently active pipeline.

### Valid Usage (Implicit)

- VUID-vkCmdSetDepthTestEnable-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdSetDepthTestEnable-commandBuffer-recording
  `commandBuffer` must be in the recording state
- VUID-vkCmdSetDepthTestEnable-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

### Host Synchronization

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To dynamically set the depth compare operator, call:

```c
// Provided by VK_VERSION_1_3
void vkCmdSetDepthCompareOp(
    VkCommandBuffer commandBuffer,
    VkCompareOp depthCompareOp);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `depthCompareOp` is a `VkCompareOp` value specifying the comparison operator used for the Depth
Comparison step of the depth test.

This command sets the depth comparison operator for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_DEPTH_COMPARE_OP` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineDepthStencilStateCreateInfo::depthCompareOp` value used to create the currently active pipeline.

**Valid Usage (Implicit)**

- VUID-vkCmdSetDepthCompareOp-commandBuffer-parameter commandBuffer must be a valid `VkCommandBuffer` handle
- VUID-vkCmdSetDepthCompareOp-depthCompareOp-parameter depthCompareOp must be a valid `VkCompareOp` value
- VUID-vkCmdSetDepthCompareOp-commandBuffer-recording commandBuffer must be in the recording state
- VUID-vkCmdSetDepthCompareOp-commandBuffer-cmdpool The `VkCommandPool` that commandBuffer was allocated from must support graphics operations

**Host Synchronization**

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

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To dynamically set the depth write enable, call:

```c
// Provided by VK_VERSION_1_3
void vkCmdSetDepthWriteEnable(
    VkCommandBuffer commandBuffer,
    VkBool32 depthWriteEnable);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
• depthWriteEnable specifies if depth writes are enabled.

This command sets the depth write enable for subsequent drawing commands when the graphics pipeline is created with VK_DYNAMIC_STATE_DEPTH_WRITE_ENABLE set in VkPipelineDynamicStateCreateInfo::pDynamicStates. Otherwise, this state is specified by the VkPipelineDepthStencilStateCreateInfo::depthWriteEnable value used to create the currently active pipeline.

Valid Usage (Implicit)

• VUID-vkCmdSetDepthWriteEnable-commandBuffer-parameter commandBuffer must be a valid VkCommandBuffer handle
• VUID-vkCmdSetDepthWriteEnable-commandBuffer-recording commandBuffer must be in the recording state
• VUID-vkCmdSetDepthWriteEnable-commandBuffer-cmdpool The VkCommandPool that commandBuffer was allocated from must support graphics operations

Host Synchronization

• Host access to commandBuffer must be externally synchronized
• Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

26.9. Sample Counting

Occlusion queries use query pool entries to track the number of samples that pass all the per-fragment tests. The mechanism of collecting an occlusion query value is described in Occlusion Queries.

The occlusion query sample counter increments by one for each sample with a coverage value of 1 in each fragment that survives all the per-fragment tests, including scissor, sample mask, alpha to coverage, stencil, and depth tests.
### 26.10. Coverage Reduction

Coverage reduction takes the coverage information for a fragment and converts that to a boolean coverage value for each color sample in each pixel covered by the fragment.

#### 26.10.1. Pixel Coverage

Coverage for each pixel is first extracted from the total fragment coverage mask. This consists of \( \text{rasterizationSamples} \) unique coverage samples for each pixel in the fragment area, each with a unique sample index. If the fragment only contains a single pixel, coverage for the pixel is equivalent to the fragment coverage.

#### 26.10.2. Color Sample Coverage

Once pixel coverage is determined, coverage for each individual color sample corresponding to that pixel is determined.

The number of \( \text{rasterizationSamples} \) is identical to the number of samples in the color attachments. A color sample is covered if the pixel coverage sample with the same sample index \( i \) is covered.
Chapter 27. The Framebuffer

27.1. Blending

Blending combines the incoming source fragment’s R, G, B, and A values with the destination R, G, B, and A values of each sample stored in the framebuffer at the fragment’s \((x_f, y_f)\) location. Blending is performed for each color sample covered by the fragment, rather than just once for each fragment.

Source and destination values are combined according to the blend operation, quadruplets of source and destination weighting factors determined by the blend factors, and a blend constant, to obtain a new set of R, G, B, and A values, as described below.

Blending is computed and applied separately to each color attachment used by the subpass, with separate controls for each attachment.

Prior to performing the blend operation, signed and unsigned normalized fixed-point color components undergo an implied conversion to floating-point as specified by Conversion from Normalized Fixed-Point to Floating-Point. Blending computations are treated as if carried out in floating-point, and basic blend operations are performed with a precision and dynamic range no lower than that used to represent destination components.

### Note
Blending is only defined for floating-point, UNORM, SNORM, and sRGB formats. Within those formats, the implementation may only support blending on some subset of them. Which formats support blending is indicated by VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT.

The pipeline blend state is included in the VkPipelineColorBlendStateCreateInfo structure during graphics pipeline creation:

The `VkPipelineColorBlendStateCreateInfo` structure is defined as:

```
// Provided by VK_VERSION_1_0
typedef struct VkPipelineColorBlendStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineColorBlendStateCreateFlags flags;
    VkBool32 logicOpEnable;
    VkLogicOp logicOp;
    uint32_t attachmentCount;
    const VkPipelineColorBlendAttachmentState* pAttachments;
    float blendConstants[4];
} VkPipelineColorBlendStateCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
• **flags** is reserved for future use.

• **logicOpEnable** controls whether to apply Logical Operations.

• **logicOp** selects which logical operation to apply.

• **attachmentCount** is the number of `VkPipelineColorBlendAttachmentState` elements in `pAttachments`.

• **pAttachments** is a pointer to an array of `VkPipelineColorBlendAttachmentState` structures defining blend state for each color attachment.

• **blendConstants** is a pointer to an array of four values used as the R, G, B, and A components of the blend constant that are used in blending, depending on the blend factor.

**Valid Usage**

- VUID-VkPipelineColorBlendStateCreateInfo-pAttachments-00605
  If the independentBlend feature is not enabled, all elements of `pAttachments` **must** be identical

- VUID-VkPipelineColorBlendStateCreateInfo-logicOpEnable-00606
  If the logicOp feature is not enabled, `logicOpEnable` **must** be `VK_FALSE`

- VUID-VkPipelineColorBlendStateCreateInfo-logicOpEnable-00607
  If `logicOpEnable` is `VK_TRUE`, `logicOp` **must** be a valid `VkLogicOp` value

- VUID-VkPipelineColorBlendStateCreateInfo-pAttachments-07354
  If `attachmentCount` is not 0, `pAttachments` **must** be a valid pointer to an array of `attachmentCount` valid `VkPipelineColorBlendAttachmentState` structures

**Valid Usage (Implicit)**

- VUID-VkPipelineColorBlendStateCreateInfo-sType-sType
  `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO`

- VUID-VkPipelineColorBlendStateCreateInfo-pNext-pNext
  `pNext` **must** be NULL

- VUID-VkPipelineColorBlendStateCreateInfo-flags-zerobitmask
  `flags` **must** be 0

- VUID-VkPipelineColorBlendStateCreateInfo-pAttachments-parameter
  If `attachmentCount` is not 0, and `pAttachments` is not NULL, `pAttachments` **must** be a valid pointer to an array of `attachmentCount` valid `VkPipelineColorBlendAttachmentState` structures

// Provided by VK_VERSION_1_0

```c
typedef VkFlags VkPipelineColorBlendStateCreateFlags;
```

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

The `VkPipelineColorBlendAttachmentState` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineColorBlendAttachmentState {
    VkBool32 blendEnable;
    VkBlendFactor srcColorBlendFactor;
    VkBlendFactor dstColorBlendFactor;
    VkBlendOp colorBlendOp;
    VkBlendFactor srcAlphaBlendFactor;
    VkBlendFactor dstAlphaBlendFactor;
    VkBlendOp alphaBlendOp;
    VkColorComponentFlags colorWriteMask;
} VkPipelineColorBlendAttachmentState;
```

- **blendEnable** controls whether blending is enabled for the corresponding color attachment. If blending is not enabled, the source fragment's color for that attachment is passed through unmodified.
- **srcColorBlendFactor** selects which blend factor is used to determine the source factors \((S_r, S_g, S_b)\).
- **dstColorBlendFactor** selects which blend factor is used to determine the destination factors \((D_r, D_g, D_b)\).
- **colorBlendOp** selects which blend operation is used to calculate the RGB values to write to the color attachment.
- **srcAlphaBlendFactor** selects which blend factor is used to determine the source factor \(S_a\).
- **dstAlphaBlendFactor** selects which blend factor is used to determine the destination factor \(D_a\).
- **alphaBlendOp** selects which blend operation is used to calculate the alpha values to write to the color attachment.
- **colorWriteMask** is a bitmask of `VkColorComponentFlagBits` specifying which of the R, G, B, and/or A components are enabled for writing, as described for the Color Write Mask.

### Valid Usage

- **VUID-VkPipelineColorBlendAttachmentState-srcColorBlendFactor-00608**
  If the dualSrcBlend feature is not enabled, `srcColorBlendFactor` must not be `VK_BLEND_FACTOR_SRC1_COLOR`, `VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_FACTOR_SRC1_ALPHA`, or `VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA`.

- **VUID-VkPipelineColorBlendAttachmentState-dstColorBlendFactor-00609**
  If the dualSrcBlend feature is not enabled, `dstColorBlendFactor` must not be `VK_BLEND_FACTOR_SRC1_COLOR`, `VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_FACTOR_SRC1_ALPHA`, or `VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA`.

- **VUID-VkPipelineColorBlendAttachmentState-srcAlphaBlendFactor-00610**
  If the dualSrcBlend feature is not enabled, `srcAlphaBlendFactor` must not be `VK_BLEND_FACTOR_SRC1_COLOR`, `VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_FACTOR_SRC1_ALPHA`, or `VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA`.

**VK_BLEND_FACTOR_SRC1_ALPHA**, or **VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA**

- VUID-VkPipelineColorBlendAttachmentState-dstAlphaBlendFactor-00611
  If the dualSrcBlend feature is not enabled, **dstAlphaBlendFactor** must not be **VK_BLEND_FACTOR_SRC1_COLOR**, **VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR**, **VK_BLEND_FACTOR_SRC1_ALPHA**, or **VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA**

---

**Valid Usage (Implicit)**

- VUID-VkPipelineColorBlendAttachmentState-srcColorBlendFactor-parameter
  **srcColorBlendFactor** must be a valid **VkBlendFactor** value

- VUID-VkPipelineColorBlendAttachmentState-dstColorBlendFactor-parameter
  **dstColorBlendFactor** must be a valid **VkBlendFactor** value

- VUID-VkPipelineColorBlendAttachmentState-colorBlendOp-parameter
  **colorBlendOp** must be a valid **VkBlendOp** value

- VUID-VkPipelineColorBlendAttachmentState-srcAlphaBlendFactor-parameter
  **srcAlphaBlendFactor** must be a valid **VkBlendFactor** value

- VUID-VkPipelineColorBlendAttachmentState-dstAlphaBlendFactor-parameter
  **dstAlphaBlendFactor** must be a valid **VkBlendFactor** value

- VUID-VkPipelineColorBlendAttachmentState-alphaBlendOp-parameter
  **alphaBlendOp** must be a valid **VkBlendOp** value

- VUID-VkPipelineColorBlendAttachmentState-colorWriteMask-parameter
  **colorWriteMask** must be a valid combination of **VkColorComponentFlagBits** values

---

### 27.1.1. Blend Factors

The source and destination color and alpha blending factors are selected from the enum:

```c
// Provided by VK_VERSION_1_0
typedef enum VkBlendFactor {
    VK_BLEND_FACTOR_ZERO = 0,
    VK_BLEND_FACTOR_ONE = 1,
    VK_BLEND_FACTOR_SRC_COLOR = 2,
    VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR = 3,
    VK_BLEND_FACTOR_DST_COLOR = 4,
    VK_BLEND_FACTOR_ONE_MINUS_DST_COLOR = 5,
    VK_BLEND_FACTOR_SRC_ALPHA = 6,
    VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA = 7,
    VK_BLEND_FACTOR_DST_ALPHA = 8,
    VK_BLEND_FACTOR_ONE_MINUS_DST_ALPHA = 9,
    VK_BLEND_FACTOR_CONSTANT_COLOR = 10,
    VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_COLOR = 11,
    VK_BLEND_FACTOR_CONSTANT_ALPHA = 12,
    VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_ALPHA = 13,
    VK_BLEND_FACTOR_SRC_ALPHA_SATURATE = 14,
};
```
VK_BLEND_FACTOR_SRC1_COLOR = 15,
VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR = 16,
VK_BLEND_FACTOR_SRC1_ALPHA = 17,
VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA = 18,
} VkBlendFactor;

The semantics of the enum values are described in the table below:

<table>
<thead>
<tr>
<th>VkBlendFactor</th>
<th>RGB Blend Factors ((S_R,S_G,S_B)) or ((D_R,D_G,D_B))</th>
<th>Alpha Blend Factor ((S_A) or (D_A))</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_BLEND_FACTOR_ZERO</td>
<td>(0,0,0)</td>
<td>0</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE</td>
<td>(1,1,1)</td>
<td>1</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_SRC_COLOR</td>
<td>((R_{s0},G_{s0},B_{s0}))</td>
<td>(A_{s0})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR</td>
<td>((1-R_{s0},1-G_{s0},1-B_{s0}))</td>
<td>(1-A_{s0})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_DST_COLOR</td>
<td>((R_d,G_d,B_d))</td>
<td>(A_d)</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_DST_COLOR</td>
<td>((1-R_d,1-G_d,1-B_d))</td>
<td>(1-A_d)</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_SRC_ALPHA</td>
<td>((A_{s0},A_{s0},A_{s0}))</td>
<td>(A_{s0})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA</td>
<td>((1-A_{s0},1-A_{s0},1-A_{s0}))</td>
<td>(1-A_{s0})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_DST_ALPHA</td>
<td>((A_d,A_d,A_d))</td>
<td>(A_d)</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_DST_ALPHA</td>
<td>((1-A_d,1-A_d,1-A_d))</td>
<td>(1-A_d)</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_CONSTANT_COLOR</td>
<td>((R_c,G_c,B_c))</td>
<td>(A_c)</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_COLOR</td>
<td>((1-R_c,1-G_c,1-B_c))</td>
<td>(1-A_c)</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_CONSTANT_ALPHA</td>
<td>((A_c,A_c,A_c))</td>
<td>(A_c)</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_ALPHA</td>
<td>((1-A_c,1-A_c,1-A_c))</td>
<td>(1-A_c)</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_SRC_ALPHA_SATURATE</td>
<td>((f,f,f); f = \text{min}(A_{s0},1-A_{s0}))</td>
<td>1</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_SRC1_COLOR</td>
<td>((R_{s1},G_{s1},B_{s1}))</td>
<td>(A_{s1})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR</td>
<td>((1-R_{s1},1-G_{s1},1-B_{s1}))</td>
<td>(1-A_{s1})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_SRC1_ALPHA</td>
<td>((A_{s1},A_{s1},A_{s1}))</td>
<td>(A_{s1})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA</td>
<td>((1-A_{s1},1-A_{s1},1-A_{s1}))</td>
<td>(1-A_{s1})</td>
</tr>
</tbody>
</table>

In this table, the following conventions are used:

- \(R_{s0},G_{s0},B_{s0}\) and \(A_{s0}\) represent the first source color R, G, B, and A components, respectively, for the fragment output location corresponding to the color attachment being blended.
- \(R_{s1},G_{s1},B_{s1}\) and \(A_{s1}\) represent the second source color R, G, B, and A components, respectively, used in dual source blending modes, for the fragment output location corresponding to the color attachment being blended.
• \( R_d, G_d, B_d \) and \( A_d \) represent the \( R, G, B, \) and \( A \) components of the destination color. That is, the color currently in the corresponding color attachment for this fragment/sample.

• \( R_c, G_c, B_c \) and \( A_c \) represent the blend constant \( R, G, B, \) and \( A \) components, respectively.

To **dynamically set and change** the blend constants, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetBlendConstants(
    VkCommandBuffer commandBuffer, 
    const float blendConstants[4]);
```

• `commandBuffer` is the command buffer into which the command will be recorded.

• `blendConstants` is a pointer to an array of four values specifying the \( R_c, G_c, B_c, \) and \( A_c \) components of the blend constant color used in blending, depending on the **blend factor**.

This command sets blend constants for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_BLEND_CONSTANTS` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineColorBlendStateCreateInfo::blendConstants` values used to create the currently active pipeline.

---

**Valid Usage (Implicit)**

- VUID-vkCmdSetBlendConstants-commandBuffer-parameter
  - `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetBlendConstants-commandBuffer-recording
  - `commandBuffer` must be in the **recording state**

- VUID-vkCmdSetBlendConstants-commandBuffer-cmdpool
  - The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

---

**Host Synchronization**

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

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

---

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
27.1.2. Dual-Source Blending

Blend factors that use the secondary color input \((R_{s1},G_{s1},B_{s1},A_{s1})\) \((VK\_BLEND\_FACTOR\_SRC1\_COLOR, VK\_BLEND\_FACTOR\_ONE\_MINUS\_SRC1\_COLOR, VK\_BLEND\_FACTOR\_SRC1\_ALPHA,\) and \(VK\_BLEND\_FACTOR\_ONE\_MINUS\_SRC1\_ALPHA)\) may consume implementation resources that could otherwise be used for rendering to multiple color attachments. Therefore, the number of color attachments that can be used in a framebuffer may be lower when using dual-source blending.

Dual-source blending is only supported if the dualSrcBlend feature is enabled.

The maximum number of color attachments that can be used in a subpass when using dual-source blending functions is implementation-dependent and is reported as the maxFragmentDualSrcAttachments member of VkPhysicalDeviceLimits.

When using a fragment shader with dual-source blending functions, the color outputs are bound to the first and second inputs of the blender using the Index decoration, as described in Fragment Output Interface. If the second color input to the blender is not written in the shader, or if no output is bound to the second input of a blender, the result of the blending operation is not defined.

27.1.3. Blend Operations

Once the source and destination blend factors have been selected, they along with the source and destination components are passed to the blending operations. RGB and alpha components can use different operations. Possible values of VkBlendOp, specifying the operations, are:

```cpp
// Provided by VK_VERSION_1_0
typedef enum VkBlendOp {
    VK_BLEND_OP_ADD = 0,
    VK_BLEND_OP_SUBTRACT = 1,
    VK_BLEND_OP_REVERSE_SUBTRACT = 2,
    VK_BLEND_OP_MIN = 3,
    VK_BLEND_OP_MAX = 4,
} VkBlendOp;
```
The semantics of the basic blend operations are described in the table below:

<table>
<thead>
<tr>
<th>VkBlendOp</th>
<th>RGB Components</th>
<th>Alpha Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_BLEND_OP_ADD</td>
<td>( R = R_s \times S_r + R_d \times D_r )</td>
<td>( A = A_s \times S_a + A_d \times D_a )</td>
</tr>
<tr>
<td></td>
<td>( G = G_s \times S_g + G_d \times D_g )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( B = B_s \times S_b + B_d \times D_b )</td>
<td></td>
</tr>
<tr>
<td>VK_BLEND_OP_SUBTRACT</td>
<td>( R = R_s - R_d \times D_r )</td>
<td>( A = A_s \times S_a - A_d \times D_a )</td>
</tr>
<tr>
<td></td>
<td>( G = G_s - G_d \times D_g )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( B = B_s - B_d \times D_b )</td>
<td></td>
</tr>
<tr>
<td>VK_BLEND_OP_REVERSE_SUBTRACT</td>
<td>( R = R_d \times D_r - R_s \times S_r )</td>
<td>( A = A_d \times D_a \times S_s )</td>
</tr>
<tr>
<td></td>
<td>( G = G_d \times D_g - G_s \times S_g )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( B = B_d \times D_b - B_s \times S_b )</td>
<td></td>
</tr>
<tr>
<td>VK_BLEND_OP_MIN</td>
<td>( R = \min(R_s, R_d) )</td>
<td>( A = \min(A_s, A_d) )</td>
</tr>
<tr>
<td></td>
<td>( G = \min(G_s, G_d) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( B = \min(B_s, B_d) )</td>
<td></td>
</tr>
<tr>
<td>VK_BLEND_OP_MAX</td>
<td>( R = \max(R_s, R_d) )</td>
<td>( A = \max(A_s, A_d) )</td>
</tr>
<tr>
<td></td>
<td>( G = \max(G_s, G_d) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( B = \max(B_s, B_d) )</td>
<td></td>
</tr>
</tbody>
</table>

In this table, the following conventions are used:

- \( R_s, G_s, B_s, \) and \( A_s \) represent the first source color \( R, G, B, \) and \( A \) components, respectively.
- \( R_d, G_d, B_d, \) and \( A_d \) represent the \( R, G, B, \) and \( A \) components of the destination color. That is, the color currently in the corresponding color attachment for this fragment/sample.
- \( S_r, S_g, S_b, \) and \( S_a \) represent the source blend factor \( R, G, B, \) and \( A \) components, respectively.
- \( D_r, D_g, D_b, \) and \( D_a \) represent the destination blend factor \( R, G, B, \) and \( A \) components, respectively.

The blending operation produces a new set of values \( R, G, B, \) and \( A \), which are written to the framebuffer attachment. If blending is not enabled for this attachment, then \( R, G, B, \) and \( A \) are assigned \( R_s, G_s, B_s, \) and \( A_s \), respectively.

If the color attachment is fixed-point, the components of the source and destination values and blend factors are each clamped to \([0,1]\) or \([-1,1]\) respectively for an unsigned normalized or signed normalized color attachment prior to evaluating the blend operations. If the color attachment is floating-point, no clamping occurs.

If the numeric format of a framebuffer attachment uses sRGB encoding, the \( R, G, \) and \( B \) destination color values (after conversion from fixed-point to floating-point) are considered to be encoded for the sRGB color space and hence are linearized prior to their use in blending. Each \( R, G, \) and \( B \) component is converted from nonlinear to linear as described in the “sRGB EOTF” section of the Khronos Data Format Specification. If the format is not sRGB, no linearization is performed.

If the numeric format of a framebuffer attachment uses sRGB encoding, then the final \( R, G, \) and \( B \) values are converted into the nonlinear sRGB representation before being written to the framebuffer attachment as described in the “sRGB EOTF" section of the Khronos Data Format Specification.
Specification.

If the numeric format of a framebuffer color attachment is not sRGB encoded then the resulting \( c \) values for R, G and B are unmodified. The value of A is never sRGB encoded. That is, the alpha component is always stored in memory as linear.

If the framebuffer color attachment is `VK_ATTACHMENT_UNUSED`, no writes are performed through that attachment. Writes are not performed to framebuffer color attachments greater than or equal to the `VkSubpassDescription::colorAttachmentCount` or `VkSubpassDescription2::colorAttachmentCount` value.

### 27.2. Logical Operations

The application can enable a logical operation between the fragment's color values and the existing value in the framebuffer attachment. This logical operation is applied prior to updating the framebuffer attachment. Logical operations are applied only for signed and unsigned integer and normalized integer framebuffers. Logical operations are not applied to floating-point or sRGB format color attachments.

Logical operations are controlled by the `logicOpEnable` and `logicOp` members of `VkPipelineColorBlendStateCreateInfo`. If `logicOpEnable` is `VK_TRUE`, then a logical operation selected by `logicOp` is applied between each color attachment and the fragment's corresponding output value, and blending of all attachments is treated as if it were disabled. Any attachments using color formats for which logical operations are not supported simply pass through the color values unmodified. The logical operation is applied independently for each of the red, green, blue, and alpha components. The `logicOp` is selected from the following operations:

```c
// Provided by VK_VERSION_1_0
typedef enum VkLogicOp {
    VK_LOGIC_OP_CLEAR = 0,
    VK_LOGIC_OP_AND = 1,
    VK_LOGIC_OP_AND_REVERSE = 2,
    VK_LOGIC_OP_COPY = 3,
    VK_LOGIC_OP_AND_INVERTED = 4,
    VK_LOGIC_OP_NO_OP = 5,
    VK_LOGIC_OP_XOR = 6,
    VK_LOGIC_OP_OR = 7,
    VK_LOGIC_OP_NOR = 8,
    VK_LOGIC_OP_EQUIVALENT = 9,
    VK_LOGIC_OP_INVERT = 10,
    VK_LOGIC_OP_OR_REVERSE = 11,
    VK_LOGIC_OP_COPY_INVERTED = 12,
    VK_LOGIC_OP_OR_INVERTED = 13,
    VK_LOGIC_OP_NAND = 14,
    VK_LOGIC_OP_SET = 15,
} VkLogicOp;
```
The logical operations supported by Vulkan are summarized in the following table in which

- \( \neg \) is bitwise invert,
- \( \& \) is bitwise and,
- \( \lor \) is bitwise or,
- \( \oplus \) is bitwise exclusive or,
- \( s \) is the fragment's R\(_{s0}\), G\(_{s0}\), B\(_{s0}\) or A\(_{s0}\) component value for the fragment output corresponding to the color attachment being updated, and
- \( d \) is the color attachment's R, G, B or A component value:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_LOGIC_OP_CLEAR</td>
<td>0</td>
</tr>
<tr>
<td>VK_LOGIC_OP_AND</td>
<td>( s &amp; d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_AND_REVERSE</td>
<td>( s &amp; \neg d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_COPY</td>
<td>( s )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_AND_INVERTED</td>
<td>( \neg s &amp; d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_NO_OP</td>
<td>( d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_XOR</td>
<td>( s \oplus d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_OR</td>
<td>( s \lor d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_NOR</td>
<td>( \neg (s \lor d) )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_EQUIVALENT</td>
<td>( \neg (s \oplus d) )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_INVERT</td>
<td>( \neg d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_OR_REVERSE</td>
<td>( s \lor \neg d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_COPY_INVERTED</td>
<td>( \neg s )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_OR_INVERTED</td>
<td>( \neg s &amp; d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_NAND</td>
<td>( \neg (s &amp; d) )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_SET</td>
<td>all 1s</td>
</tr>
</tbody>
</table>

The result of the logical operation is then written to the color attachment as controlled by the component write mask, described in **Blend Operations**.

### 27.3. Color Write Mask

Bits which **can** be set in `VkPipelineColorBlendAttachmentState::colorWriteMask`, determining whether the final color values R, G, B and A are written to the framebuffer attachment, are:

```cpp
// Provided by VK_VERSION_1_0
typedef enum VkColorComponentFlagBits {
```
VK_COLOR_COMPONENT_R_BIT = 0x00000001,
VK_COLOR_COMPONENT_G_BIT = 0x00000002,
VK_COLOR_COMPONENT_B_BIT = 0x00000004,
VK_COLOR_COMPONENT_A_BIT = 0x00000008,
} VkColorComponentFlagBits;

- **VK_COLOR_COMPONENT_R_BIT** specifies that the R value is written to the color attachment for the appropriate sample. Otherwise, the value in memory is unmodified.
- **VK_COLOR_COMPONENT_G_BIT** specifies that the G value is written to the color attachment for the appropriate sample. Otherwise, the value in memory is unmodified.
- **VK_COLOR_COMPONENT_B_BIT** specifies that the B value is written to the color attachment for the appropriate sample. Otherwise, the value in memory is unmodified.
- **VK_COLOR_COMPONENT_A_BIT** specifies that the A value is written to the color attachment for the appropriate sample. Otherwise, the value in memory is unmodified.

The color write mask operation is applied regardless of whether blending is enabled.

// Provided by VK_VERSION_1_0
typedef VkFlags VkColorComponentFlags;

**VkColorComponentFlags** is a bitmask type for setting a mask of zero or more **VkColorComponentFlagBits**.
Chapter 28. Dispatching Commands

Dispatching commands (commands with Dispatch in the name) provoke work in a compute pipeline. Dispatching commands are recorded into a command buffer and when executed by a queue, will produce work which executes according to the bound compute pipeline. A compute pipeline must be bound to a command buffer before any dispatching commands are recorded in that command buffer.

To record a dispatch, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdDispatch(
    VkCommandBuffer commandBuffer,
    uint32_t groupCountX,
    uint32_t groupCountY,
    uint32_t groupCountZ);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `groupCountX` is the number of local workgroups to dispatch in the X dimension.
- `groupCountY` is the number of local workgroups to dispatch in the Y dimension.
- `groupCountZ` is the number of local workgroups to dispatch in the Z dimension.

When the command is executed, a global workgroup consisting of `groupCountX` × `groupCountY` × `groupCountZ` local workgroups is assembled.

Valid Usage

- **VUID-vkCmdDispatch-magFilter-04553**
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDispatch-mipmapMode-04770**
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDispatch-None-06479**
  If a `VkImageView` is sampled with depth comparison, the image view's format features must contain `VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_DEPTH_COMPARISON_BIT`

- **VUID-vkCmdDispatch-OpTypeImage-07027**
  If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`
For any `VkImageView` being written as a storage image where the image format field of the `OpTypeImage` is `Unknown`, the view's format features must contain `VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT`

- VUID-vkCmdDispatch-OpTypeImage-07028
  For any `VkImageView` being read as a storage image where the image format field of the `OpTypeImage` is `Unknown`, the view's format features must contain `VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT`

- VUID-vkCmdDispatch-OpTypeImage-07029
  For any `VkBufferView` being written as a storage texel buffer where the image format field of the `OpTypeImage` is `Unknown`, the view's buffer features must contain `VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT`

- VUID-vkCmdDispatch-OpTypeImage-07030
  Any `VkBufferView` being read as a storage texel buffer where the image format field of the `OpTypeImage` is `Unknown` then the view's buffer features must contain `VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT`

- VUID-vkCmdDispatch-None-02697
  For each set `n` that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set must have been bound to `n` at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set `n`, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility

- VUID-vkCmdDispatch-maintenance4-06425
  If the `maintenance4` feature is not enabled, then for each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility

- VUID-vkCmdDispatch-None-02699
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid as described by descriptor validity if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command

- VUID-vkCmdDispatch-None-02700
  A valid pipeline must be bound to the pipeline bind point used by this command

- VUID-vkCmdDispatch-commandBuffer-02701
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited (if the `VK_NV_inherited_viewport_scissor` extension is enabled) for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic

- VUID-vkCmdDispatch-None-02859
  There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound

- VUID-vkCmdDispatch-None-02702
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a
VkSampler object that uses unnormalized coordinates, that sampler must not be used to sample from any VkImage with a VkImageView of the type VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, VK_IMAGE_VIEW_TYPE_1D_ARRAY, VK_IMAGE_VIEW_TYPE_2D_ARRAY or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY, in any shader stage

• VUID-vkCmdDispatch-None-02703
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions with ImplicitLod, Def or Proj in their name, in any shader stage

• VUID-vkCmdDispatch-None-02704
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions that includes a LOD bias or any offset values, in any shader stage

• VUID-vkCmdDispatch-None-02705
  If the robustBufferAccess feature is not enabled, and if the VkPipeline object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point

• VUID-vkCmdDispatch-None-02706
  If the robustBufferAccess feature is not enabled, and if the VkPipeline object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point

• VUID-vkCmdDispatch-commandBuffer-02707
  If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, any resource accessed by the VkPipeline object bound to the pipeline bind point used by this command must not be a protected resource

• VUID-vkCmdDispatch-None-06550
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler or VkImageView object that enables sampler Y’CbCr conversion, that object must only be used with OpImageSample* or OpImageSparseSample* instructions

• VUID-vkCmdDispatch-ConstOffset-06551
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler or VkImageView object that enables sampler Y’CbCr conversion, that object must not use the ConstOffset and Offset operands

• VUID-vkCmdDispatch-viewType-07752
  If a VkImageView is accessed as a result of this command, then the image view’s viewType must match the Dim operand of the OpTypeImage as described in Instruction/Sampler/Image View Validation

• VUID-vkCmdDispatch-format-07753
  If a VkImageView is accessed as a result of this command, then the image view’s format must match the numeric format from the Sampled Type operand of the OpTypeImage as described in the SPIR-V Sampled Type column of the Interpretation of Numeric Format
If a `VkImageView` is accessed using `OpImageWrite` as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view's format.

If a `VkBufferView` is accessed using `OpImageWrite` as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the buffer view's format.

Any shader invocation executed by this command must terminate.

If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, any resource written to by the `VkPipeline` object bound to the pipeline bind point used by this command must not be an unprotected resource.

If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, pipeline stages other than the framebuffer-space and compute stages in the `VkPipeline` object bound to the pipeline bind point used by this command must not write to any resource.

`groupCountX` must be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0]`.

`groupCountY` must be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1]`.

`groupCountZ` must be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2]`.

Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle.
- `commandBuffer` must be in the recording state.
- The `VkCommandPool` that `commandBuffer` was allocated from must support compute operations.
- This command must only be called outside of a render pass instance.
Host Synchronization

- Host access to \textit{commandBuffer} must be externally synchronized
- Host access to the \texttt{VkCommandPool} that \textit{commandBuffer} was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Compute</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To record an indirect dispatching command, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdDispatchIndirect(
    VkCommandBuffer commandBuffer,
    VkBuffer buffer,
    VkDeviceSize offset);
```

- \textit{commandBuffer} is the command buffer into which the command will be recorded.
- \textit{buffer} is the buffer containing dispatch parameters.
- \textit{offset} is the byte offset into \textit{buffer} where parameters begin.

\texttt{vkCmdDispatchIndirect} behaves similarly to \texttt{vkCmdDispatch} except that the parameters are read by the device from a buffer during execution. The parameters of the dispatch are encoded in a \texttt{VkDispatchIndirectCommand} structure taken from \textit{buffer} starting at \textit{offset}.

Valid Usage

- VUID-vkCmdDispatchIndirect-magFilter-04553
  If a \texttt{VkSampler} created with \texttt{magFilter} or \texttt{minFilter} equal to \texttt{VK_FILTER_LINEAR} and \texttt{compareEnable} equal to \texttt{VK_FALSE} is used to sample a \texttt{VkImageView} as a result of this command, then the image view’s format features must contain \texttt{VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT}

- VUID-vkCmdDispatchIndirect-mipmapMode-04770
  If a \texttt{VkSampler} created with \texttt{.mipmapMode} equal to \texttt{VK_SAMPLER_MIPMAP_MODE_LINEAR} and \texttt{compareEnable} equal to \texttt{VK_FALSE} is used to sample a \texttt{VkImageView} as a result of this command, then the image view’s format features must contain \texttt{VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT}

- VUID-vkCmdDispatchIndirect-None-06479
If a `VkImageView` is sampled with depth comparison, the image view's format features must contain `VK_FORMAT_FEATURE_2SAMPLED_IMAGE_DEPTH_COMPARISON_BIT`

- **VUID-vkCmdDispatchIndirect-None-02691**
  If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`

- **VUID-vkCmdDispatchIndirect-OpTypeImage-07027**
  For any `VkImageView` being written as a storage image where the image format field of the `OpTypeImage` is `Unknown`, the view's format features must contain `VK_FORMAT_FEATURE_2_STORAGE_WRITEWITHOUTFORMAT_BIT`

- **VUID-vkCmdDispatchIndirect-OpTypeImage-07028**
  For any `VkImageView` being read as a storage image where the image format field of the `OpTypeImage` is `Unknown`, the view's format features must contain `VK_FORMAT_FEATURE_2_STORAGE_READONLYWITHOUTFORMAT_BIT`

- **VUID-vkCmdDispatchIndirect-OpTypeImage-07029**
  For any `VkBufferView` being written as a storage texel buffer where the image format field of the `OpTypeImage` is `Unknown`, the view's buffer features must contain `VK_FORMAT_FEATURE_2_STORAGE_WRITEWITHOUTFORMAT_BIT`

- **VUID-vkCmdDispatchIndirect-OpTypeImage-07030**
  Any `VkBufferView` being read as a storage texel buffer where the image format field of the `OpTypeImage` is `Unknown` then the view's buffer features must contain `VK_FORMAT_FEATURE_2_STORAGE_READONLYWITHOUTFORMAT_BIT`

- **VUID-vkCmdDispatchIndirect-None-02697**
  For each set `n` that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set must have been bound to `n` at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set `n`, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility

- **VUID-vkCmdDispatchIndirect-maintenance4-06425**
  If the `maintenance4` feature is not enabled, then for each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility

- **VUID-vkCmdDispatchIndirect-None-02699**
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid as described by descriptor validity if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command

- **VUID-vkCmdDispatchIndirect-None-02700**
  A valid pipeline must be bound to the pipeline bind point used by this command

- **VUID-vkCmdDispatchIndirect-commandBuffer-02701**
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited (if the `VK_NV_inherited_viewport_scissor` extension is enabled) for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic
There **must** not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGEVIEW_3D_ARAY`, in any shader stage.

If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

If the `robustBufferAccess` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

If the `robustBufferAccess` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be a protected resource.

If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` or `VkImageView` object that enables `sampler Y'CbCr` conversion, that object **must** only be used with `OpImageSample*` or `OpImageSparseSample*` instructions.

If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` or `VkImageView` object that enables `sampler Y'CbCr` conversion, that object **must** not use the `ConstOffset` and `Offset` operands.

If a `VkImageView` is accessed as a result of this command, then the image view's `viewType`
must match the \textit{Dim} operand of the \texttt{OpTypeImage} as described in \texttt{Instruction/Sampler/Image View Validation}.

- **VUID-vkCmdDispatchIndirect-format-07753**
  If a \texttt{VkImageView} is accessed as a result of this command, then the image view's \texttt{format} must match the numeric format from the \texttt{Sampled Type} operand of the \texttt{OpTypeImage} as described in the SPIR-V Sampled Type column of the \texttt{Interpretation of Numeric Format} table.

- **VUID-vkCmdDispatchIndirect-None-04115**
  If a \texttt{VkImageView} is accessed using \texttt{OpImageWrite} as a result of this command, then the \texttt{Type} of the \texttt{Texel} operand of that instruction must have at least as many components as the image view's format.

- **VUID-vkCmdDispatchIndirect-OpImageWrite-04469**
  If a \texttt{VkBufferView} is accessed using \texttt{OpImageWrite} as a result of this command, then the \texttt{Type} of the \texttt{Texel} operand of that instruction must have at least as many components as the buffer view's format.

- **VUID-vkCmdDispatchIndirect-None-07288**
  Any shader invocation executed by this command must terminate.

- **VUID-vkCmdDispatchIndirect-buffer-02708**
  If \texttt{buffer} is non-sparse then it must be bound completely and contiguously to a single \texttt{VkDeviceMemory} object.

- **VUID-vkCmdDispatchIndirect-buffer-02709**
  \texttt{buffer} must have been created with the \texttt{VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT} bit set.

- **VUID-vkCmdDispatchIndirect-offset-02710**
  \texttt{offset} must be a multiple of 4.

- **VUID-vkCmdDispatchIndirect-commandBuffer-02711**
  \texttt{commandBuffer} must not be a protected command buffer.

- **VUID-vkCmdDispatchIndirect-offset-00407**
  The sum of \texttt{offset} and the size of \texttt{VkDispatchIndirectCommand} must be less than or equal to the size of \texttt{buffer}.

---

**Valid Usage (Implicit)**

- **VUID-vkCmdDispatchIndirect-commandBuffer-parameter**
  \texttt{commandBuffer} must be a valid \texttt{VkCommandBuffer} handle.

- **VUID-vkCmdDispatchIndirect-buffer-parameter**
  \texttt{buffer} must be a valid \texttt{VkBuffer} handle.

- **VUID-vkCmdDispatchIndirect-commandBuffer-recording**
  \texttt{commandBuffer} must be in the recording state.

- **VUID-vkCmdDispatchIndirect-commandBuffer-cmdpool**
  The \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from must support compute operations.
• VUID-vkCmdDispatchIndirect-renderpass
  This command **must** only be called outside of a render pass instance

• VUID-vkCmdDispatchIndirect-commonparent
  Both of *buffer*, and *commandBuffer* **must** have been created, allocated, or retrieved from the same *VkDevice*

---

**Host Synchronization**

• Host access to *commandBuffer* **must** be externally synchronized

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

---

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Compute</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The *VkDispatchIndirectCommand* structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDispatchIndirectCommand {
    uint32_t x;
    uint32_t y;
    uint32_t z;
} VkDispatchIndirectCommand;
```

• *x* is the number of local workgroups to dispatch in the X dimension.

• *y* is the number of local workgroups to dispatch in the Y dimension.

• *z* is the number of local workgroups to dispatch in the Z dimension.

The members of *VkDispatchIndirectCommand* have the same meaning as the corresponding parameters of *vkCmdDispatch*.

---

**Valid Usage**

• VUID-VkDispatchIndirectCommand-x-00417
  *x* **must** be less than or equal to *VkPhysicalDeviceLimits::maxComputeWorkGroupCount* [0]

• VUID-VkDispatchIndirectCommand-y-00418
  *y* **must** be less than or equal to *VkPhysicalDeviceLimits::maxComputeWorkGroupCount* [1]
To record a dispatch using non-zero base values for the components of WorkgroupId, call:

```c
// Provided by VK_VERSION_1_1
void vkCmdDispatchBase(
    VkCommandBuffer commandBuffer,  
    uint32_t baseGroupX,             
    uint32_t baseGroupY,             
    uint32_t baseGroupZ,             
    uint32_t groupCountX,            
    uint32_t groupCountY,            
    uint32_t groupCountZ);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `baseGroupX` is the start value for the X component of WorkgroupId.
- `baseGroupY` is the start value for the Y component of WorkgroupId.
- `baseGroupZ` is the start value for the Z component of WorkgroupId.
- `groupCountX` is the number of local workgroups to dispatch in the X dimension.
- `groupCountY` is the number of local workgroups to dispatch in the Y dimension.
- `groupCountZ` is the number of local workgroups to dispatch in the Z dimension.

When the command is executed, a global workgroup consisting of \( \text{groupCountX} \times \text{groupCountY} \times \text{groupCountZ} \) local workgroups is assembled, with WorkgroupId values ranging from \([\text{baseGroup}^*, \text{baseGroup}^* + \text{groupCount}^*]) in each component. `vkCmdDispatch` is equivalent to `vkCmdDispatchBase(0,0,0,\text{groupCountX},\text{groupCountY},\text{groupCountZ})`.

### Valid Usage

- **VUID-vkCmdDispatchBase-magFilter-04553**
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDispatchBase-mipmapMode-04770**
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDispatchBase-None-06479**
  If a `VkImageView` is sampled with depth comparison, the image view's format features **must** contain `VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_DEPTH_COMPARISON_BIT`
If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view’s format features must contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`.

For any `VkImageView` being written as a storage image where the image format field of the `OpTypeImage` is `Unknown`, the view’s format features must contain `VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT`.

For any `VkImageView` being read as a storage image where the image format field of the `OpTypeImage` is `Unknown`, the view’s format features must contain `VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT`.

For any `VkBufferView` being written as a storage texel buffer where the image format field of the `OpTypeImage` is `Unknown`, the view’s buffer features must contain `VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT`.

For any `VkBufferView` being read as a storage texel buffer where the image format field of the `OpTypeImage` is `Unknown`, then the view’s buffer features must contain `VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT`.

For each set `n` that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set must have been bound to `n` at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set `n`, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility.

If the `maintenance4` feature is not enabled, then for each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility.

Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid as described by descriptor validity if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

A valid pipeline must be bound to the pipeline bind point used by this command.

If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited (if the `VK_NV_inherited_viewport_scissor` extension is enabled) for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

There must not have been any calls to dynamic state setting commands for any state not
specified as dynamic in the VkPipeline object bound to the pipeline bind point used by this command, since that pipeline was bound

- VUID-vkCmdDispatchBase-None-02702
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used to sample from any VkImage with a VkImageView of the type VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, VK_IMAGE_VIEW_TYPE_1D_ARRAY, VK_IMAGE_VIEW_TYPE_2D_ARRAY or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY, in any shader stage.

- VUID-vkCmdDispatchBase-None-02703
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions with ImplicitLod, Dref or Proj in their name, in any shader stage.

- VUID-vkCmdDispatchBase-None-02704
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions that includes a LOD bias or any offset values, in any shader stage.

- VUID-vkCmdDispatchBase-None-02705
  If the robustBufferAccess feature is not enabled, and if the VkPipeline object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- VUID-vkCmdDispatchBase-None-02706
  If the robustBufferAccess feature is not enabled, and if the VkPipeline object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- VUID-vkCmdDispatchBase-commandBuffer-02707
  If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, any resource accessed by the VkPipeline object bound to the pipeline bind point used by this command must not be a protected resource.

- VUID-vkCmdDispatchBase-None-06550
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler or VkImageView object that enables sampler Y’CbCr conversion, that object must only be used with OpImageSample* or OpImageSparseSample* instructions.

- VUID-vkCmdDispatchBase-ConstOffset-06551
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler or VkImageView object that enables sampler Y’CbCr conversion, that object must not use the ConstOffset and Offset operands.

- VUID-vkCmdDispatchBase-viewType-07752
  If a VkImageView is accessed as a result of this command, then the image view’s viewType must match the Dim operand of the OpTypeImage as described in Instruction/Sampler/Image View Validation.
- VUID-vkCmdDispatchBase-format-07753
If a VkImageView is accessed as a result of this command, then the image view's format must match the numeric format from the Sampled Type operand of the OpTypeImage as described in the SPIR-V Sampled Type column of the Interpretation of Numeric Format table.

- VUID-vkCmdDispatchBase-None-04115
If a VkImageView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view's format.

- VUID-vkCmdDispatchBase-OpImageWrite-04469
If a VkBufferView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the buffer view's format.

- VUID-vkCmdDispatchBase-None-07288
Any shader invocation executed by this command must terminate.

- VUID-vkCmdDispatchBase-commandBuffer-02712
If commandBuffer is a protected command buffer and protectedNoFault is not supported, any resource written to by the VkPipeline object bound to the pipeline bind point used by this command must not be an unprotected resource.

- VUID-vkCmdDispatchBase-commandBuffer-02713
If commandBuffer is a protected command buffer and protectedNoFault is not supported, pipeline stages other than the framebuffer-space and compute stages in the VkPipeline object bound to the pipeline bind point used by this command must not write to any resource.

- VUID-vkCmdDispatchBase-baseGroupX-00421
baseGroupX must be less than VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0].

- VUID-vkCmdDispatchBase-baseGroupX-00422
baseGroupY must be less than VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1].

- VUID-vkCmdDispatchBase-baseGroupX-00423
baseGroupZ must be less than VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2].

- VUID-vkCmdDispatchBase-groupCountX-00424
groupCountX must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0] minus baseGroupX.

- VUID-vkCmdDispatchBase-groupCountY-00425
groupCountY must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1] minus baseGroupY.

- VUID-vkCmdDispatchBase-groupCountZ-00426
groupCountZ must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2] minus baseGroupZ.

- VUID-vkCmdDispatchBase-baseGroupX-00427
If any of baseGroupX, baseGroupY, or baseGroupZ are not zero, then the bound compute pipeline must have been created with the VK_PIPELINE_CREATE_DISPATCH_BASE flag.
Valid Usage (Implicit)

- VUID-vkCmdDispatchBase-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdDispatchBase-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdDispatchBase-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support compute operations

- VUID-vkCmdDispatchBase-renderpass
  This command must only be called outside of a render pass instance

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary, Secondary</td>
<td>Outside</td>
<td>Compute</td>
<td>Action</td>
</tr>
</tbody>
</table>
Chapter 29. Sparse Resources

As documented in Resource Memory Association, VkBuffer and VkImage resources in Vulkan must be bound completely and contiguously to a single VkDeviceMemory object. This binding must be done before the resource is used, and the binding is immutable for the lifetime of the resource.

Sparse resources relax these restrictions and provide these additional features:

• Sparse resources can be bound non-contiguously to one or more VkDeviceMemory allocations.
• Sparse resources can be re-bound to different memory allocations over the lifetime of the resource.
• Sparse resources can have descriptors generated and used orthogonally with memory binding commands.

29.1. Sparse Resource Features

Sparse resources have several features that must be enabled explicitly at resource creation time. The features are enabled by including bits in the flags parameter of VkImageCreateInfo or VkBufferCreateInfo. Each feature also has one or more corresponding feature enables specified in VkPhysicalDeviceFeatures.

• The sparseBinding feature is the base, and provides the following capabilities:
  ◦ Resources can be bound at some defined (sparse block) granularity.
  ◦ The entire resource must be bound to memory before use regardless of regions actually accessed.
  ◦ No specific mapping of image region to memory offset is defined, i.e. the location that each texel corresponds to in memory is implementation-dependent.
  ◦ Sparse buffers have a well-defined mapping of buffer range to memory range, where an offset into a range of the buffer that is bound to a single contiguous range of memory corresponds to an identical offset within that range of memory.
  ◦ Requested via the VK_IMAGE_CREATE_SPARSE_BINDING_BIT and VK_BUFFER_CREATE_SPARSE_BINDING_BIT bits.
  ◦ A sparse image created using VK_IMAGE_CREATE_SPARSE_BINDING_BIT (but not VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT) supports all formats that non-sparse usage supports, and supports both VK_IMAGE_TILING_OPTIMAL and VK_IMAGE_TILING_LINEAR tiling.

• Sparse Residency builds on (and requires) the sparseBinding feature. It includes the following capabilities:
  ◦ Resources do not have to be completely bound to memory before use on the device.
  ◦ Images have a prescribed sparse image block layout, allowing specific rectangular regions of the image to be bound to specific offsets in memory allocations.
  ◦ Consistency of access to unbound regions of the resource is defined by the absence or presence of VkPhysicalDeviceSparseProperties::residencyNonResidentStrict. If this property is present, accesses to unbound regions of the resource are well defined and behave as if the
data bound is populated with all zeros; writes are discarded. When this property is absent, accesses are considered safe, but reads will return undefined values.

- Requested via the `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` and `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` bits.

Sparse residency support is advertised on a finer grain via the following features:

- The `sparseResidencyBuffer` feature provides support for creating `VkBuffer` objects with the `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT`.
- The `sparseResidencyImage2D` feature provides support for creating 2D single-sampled `VkImage` objects with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
- The `sparseResidencyImage3D` feature provides support for creating 3D `VkImage` objects with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
- The `sparseResidency2Samples` feature provides support for creating 2D `VkImage` objects with 2 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
- The `sparseResidency4Samples` feature provides support for creating 2D `VkImage` objects with 4 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
- The `sparseResidency8Samples` feature provides support for creating 2D `VkImage` objects with 8 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
- The `sparseResidency16Samples` feature provides support for creating 2D `VkImage` objects with 16 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.

Implementations supporting `sparseResidencyImage2D` are only **required** to support sparse 2D, single-sampled images. Support for sparse 3D and MSAA images is **optional** and can be enabled via `sparseResidencyImage3D`, `sparseResidency2Samples`, `sparseResidency4Samples`, `sparseResidency8Samples`, and `sparseResidency16Samples`.

- A sparse image created using `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` supports all non-compressed color formats with power-of-two element size that non-sparse usage supports. Additional formats **may** also be supported and **can** be queried via `vkGetPhysicalDeviceSparseImageFormatProperties`. `VK_IMAGE_TILING_LINEAR` tiling is not supported.

- The `sparseResidencyAliased` feature provides the following capability that **can** be enabled per resource:

  Allows physical memory ranges to be shared between multiple locations in the same sparse resource or between multiple sparse resources, with each binding of a memory location observing a consistent interpretation of the memory contents.

  See **Sparse Memory Aliasing** for more information.

## 29.2. Sparse Buffers and Fully-Resident Images

Both `VkBuffer` and `VkImage` objects created with the `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` or `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` bits **can** be thought of as a linear region of address space. In the `VkImage` case if `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` is not used, this linear region is entirely
opaque, meaning that there is no application-visible mapping between texel location and memory offset.

Unless `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` or `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` are also used, the entire resource must be bound to one or more `VkDeviceMemory` objects before use.

29.2.1. Sparse Buffer and Fully-Resident Image Block Size

The sparse block size in bytes for sparse buffers and fully-resident images is reported as `VkMemoryRequirements::alignment`. `alignment` represents both the memory alignment requirement and the binding granularity (in bytes) for sparse resources.

29.3. Sparse Partially-Resident Buffers

`VkBuffer` objects created with the `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` bit allow the buffer to be made only partially resident. Partially resident `VkBuffer` objects are allocated and bound identically to `VkBuffer` objects using only the `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` feature. The only difference is the ability for some regions of the buffer to be unbound during device use.

29.4. Sparse Partially-Resident Images

`VkImage` objects created with the `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` bit allow specific rectangular regions of the image called sparse image blocks to be bound to specific ranges of memory. This allows the application to manage residency at either image subresource or sparse image block granularity. Each image subresource (outside of the `mip tail`) starts on a sparse block boundary and has dimensions that are integer multiples of the corresponding dimensions of the sparse image block.

Note

Applications can use these types of images to control LOD based on total memory consumption. If memory pressure becomes an issue the application can unbind and disable specific mipmap levels of images without having to recreate resources or modify texel data of unaffected levels.

The application can also use this functionality to access subregions of the image in a “megatexture” fashion. The application can create a large image and only populate the region of the image that is currently being used in the scene.

29.4.1. Accessing Unbound Regions

The following member of `VkPhysicalDeviceSparseProperties` affects how data in unbound regions of sparse resources are handled by the implementation:

- `residencyNonResidentStrict`

If this property is not present, reads of unbound regions of the image will return undefined values. Both reads and writes are still considered safe and will not affect other resources or populated regions of the image.
If this property is present, all reads of unbound regions of the image will behave as if the region was bound to memory populated with all zeros; writes will be discarded.

**Image operations** performed on unbound memory may still alter some component values in the natural way for those accesses, e.g. substituting a value of one for alpha in formats that do not have an alpha component.

Example: Reading the alpha component of an unbacked VK_FORMAT_R8_UNORM image will return a value of 1.0f.

See Physical Device Enumeration for instructions for retrieving physical device properties.

### Implementor’s Note

For implementations that cannot natively handle access to unbound regions of a resource, the implementation may allocate and bind memory to the unbound regions. Reads and writes to unbound regions will access the implementation-managed memory instead.

Given that the values resulting from reads of unbound regions are undefined in this scenario, implementations may use the same physical memory for all unbound regions of multiple resources within the same process.

#### 29.4.2. Mip Tail Regions

Sparse images created using VK_IMAGE_CREATE_SPARSE_BINDING_BIT (without also using VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT) have no specific mapping of image region or image subresource to memory offset defined, so the entire image can be thought of as a linear opaque address region. However, images created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT do have a prescribed sparse image block layout, and hence each image subresource must start on a sparse block boundary. Within each array layer, the set of mip levels that have a smaller size than the sparse block size in bytes are grouped together into a mip tail region.

If the VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT flag is present in the flags member of VkSparseImageFormatProperties, for the image's format, then any mip level which has dimensions that are not integer multiples of the corresponding dimensions of the sparse image block, and all subsequent mip levels, are also included in the mip tail region.

The following member of VkPhysicalDeviceSparseProperties may affect how the implementation places mip levels in the mip tail region:

- residencyAlignedMipSize

Each mip tail region is bound to memory as an opaque region (i.e. must be bound using a VkSparseImageOpaqueMemoryBindInfo structure) and may be of a size greater than or equal to the sparse block size in bytes. This size is guaranteed to be an integer multiple of the sparse block size in bytes.
An implementation may choose to allow each array-layer’s mip tail region to be bound to memory independently or require that all array-layer’s mip tail regions be treated as one. This is dictated by VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT in VkSparseImageMemoryRequirements::flags.

The following diagrams depict how VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT and VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT alter memory usage and requirements.

![Sparse Image Diagram](image)

In the absence of VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT and VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT, each array layer contains a mip tail region containing texel data for all mip levels smaller than the sparse image block in any dimension.

Mip levels that are as large or larger than a sparse image block in all dimensions can be bound individually. Right-edges and bottom-edges of each level are allowed to have partially used sparse blocks. Any bound partially-used-sparse-blocks must still have their full sparse block size in bytes allocated in memory.
Figure 18. Sparse Image with Single Mip Tail

When `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` is present all array layers will share a single mip tail region.

Figure 19. Sparse Image with Aligned Mip Size

Note
The mip tail regions are presented here in 2D arrays simply for figure size reasons. Each mip tail is logically a single array of sparse blocks with an implementation-dependent mapping of texels or compressed texel blocks to sparse blocks.

When `VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT` is present the first mip level that would contain partially used sparse blocks begins the mip tail region. This level and all subsequent levels are placed in the mip tail. Only the first \( N \) mip levels whose dimensions are an exact multiple of the sparse image block dimensions can be bound and unbound on a sparse block basis.

![Image of mip tail and sparse block mapping](image.png)

**Figure 20. Sparse Image with Aligned Mip Size and Single Mip Tail**

**Note**

The mip tail region is presented here in a 2D array simply for figure size reasons. It is logically a single array of sparse blocks with an implementation-dependent mapping of texels or compressed texel blocks to sparse blocks.

When both `VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT` and `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` are present the constraints from each of these flags are in effect.

### 29.4.3. Standard Sparse Image Block Shapes

Standard sparse image block shapes define a standard set of dimensions for sparse image blocks that depend on the format of the image. Layout of texels or compressed texel blocks within a sparse image block is implementation-dependent. All currently defined standard sparse image block shapes are 64 KB in size.

For block-compressed formats (e.g. `VK_FORMAT_BC5_UNORM_BLOCK`), the texel size is the size of the compressed texel block (e.g. 128-bit for BC5) thus the dimensions of the standard sparse image block are...
shapes apply in terms of compressed texel blocks.

*Note*

For block-compressed formats, the dimensions of a sparse image block in terms of texels *can* be calculated by multiplying the sparse image block dimensions by the compressed texel block dimensions.
### Table 28. Standard Sparse Image Block Shapes (Single Sample)

<table>
<thead>
<tr>
<th>TEXEL SIZE (bits)</th>
<th>Block Shape (2D)</th>
<th>Block Shape (3D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-Bit</td>
<td>256 × 256 × 1</td>
<td>64 × 32 × 32</td>
</tr>
<tr>
<td>16-Bit</td>
<td>256 × 128 × 1</td>
<td>32 × 32 × 32</td>
</tr>
<tr>
<td>32-Bit</td>
<td>128 × 128 × 1</td>
<td>32 × 32 × 16</td>
</tr>
<tr>
<td>64-Bit</td>
<td>128 × 64 × 1</td>
<td>32 × 16 × 16</td>
</tr>
<tr>
<td>128-Bit</td>
<td>64 × 64 × 1</td>
<td>16 × 16 × 16</td>
</tr>
</tbody>
</table>

### Table 29. Standard Sparse Image Block Shapes (MSAA)

<table>
<thead>
<tr>
<th>TEXEL SIZE (bits)</th>
<th>Block Shape (2X)</th>
<th>Block Shape (4X)</th>
<th>Block Shape (8X)</th>
<th>Block Shape (16X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-Bit</td>
<td>128 × 256 × 1</td>
<td>128 × 128 × 1</td>
<td>64 × 128 × 1</td>
<td>64 × 64 × 1</td>
</tr>
<tr>
<td>16-Bit</td>
<td>128 × 128 × 1</td>
<td>128 × 64 × 1</td>
<td>64 × 64 × 1</td>
<td>64 × 32 × 1</td>
</tr>
<tr>
<td>32-Bit</td>
<td>64 × 128 × 1</td>
<td>64 × 64 × 1</td>
<td>32 × 64 × 1</td>
<td>32 × 32 × 1</td>
</tr>
<tr>
<td>64-Bit</td>
<td>64 × 64 × 1</td>
<td>64 × 32 × 1</td>
<td>32 × 32 × 1</td>
<td>32 × 16 × 1</td>
</tr>
<tr>
<td>128-Bit</td>
<td>32 × 64 × 1</td>
<td>32 × 32 × 1</td>
<td>16 × 32 × 1</td>
<td>16 × 16 × 1</td>
</tr>
</tbody>
</table>

Implementations that support the standard sparse image block shape for all formats listed in the Standard Sparse Image Block Shapes (Single Sample) and Standard Sparse Image Block Shapes (MSAA) tables may advertise the following VkPhysicalDeviceSparseProperties:

- residencyStandard2DBlockShape
- residencyStandard2DMultisampleBlockShape
- residencyStandard3DBlockShape

Reporting each of these features does not imply that all possible image types are supported as sparse. Instead, this indicates that no supported sparse image of the corresponding type will use custom sparse image block dimensions for any formats that have a corresponding standard sparse image block shape.

### 29.4.4. Custom Sparse Image Block Shapes

An implementation that does not support a standard image block shape for a particular sparse partially-resident image may choose to support a custom sparse image block shape for it instead. The dimensions of such a custom sparse image block shape are reported in VkSparseImageFormatProperties::imageGranularity. As with standard sparse image block shapes, the size in bytes of the custom sparse image block shape will be reported in VkMemoryRequirements::alignment.

Custom sparse image block dimensions are reported through vkGetPhysicalDeviceSparseImageFormatProperties and vkGetImageSparseMemoryRequirements.

An implementation must not support both the standard sparse image block shape and a custom
sparse image block shape for the same image. The standard sparse image block shape must be used if it is supported.

### 29.4.5. Multiple Aspects

Partially resident images are allowed to report separate sparse properties for different aspects of the image. One example is for depth/stencil images where the implementation separates the depth and stencil data into separate planes. Another reason for multiple aspects is to allow the application to manage memory allocation for implementation-private metadata associated with the image. See the figure below:

![Multiple Aspect Sparse Image](image)

Figure 21. Multiple Aspect Sparse Image

**Note**

The mip tail regions are presented here in 2D arrays simply for figure size reasons. Each mip tail is logically a single array of sparse blocks with an implementation-dependent mapping of texels or compressed texel blocks to sparse blocks.

In the figure above the depth, stencil, and metadata aspects all have unique sparse properties. The
per-texel stencil data is \( \frac{1}{4} \) the size of the depth data, hence the stencil sparse blocks include \( 4 \times \) the number of texels. The sparse block size in bytes for all of the aspects is identical and defined by `VkMemoryRequirements::alignment`.

**Metadata**

The metadata aspect of an image has the following constraints:

- All metadata is reported in the mip tail region of the metadata aspect.
- All metadata **must** be bound prior to device use of the sparse image.

### 29.5. Sparse Memory Aliasing

By default sparse resources have the same aliasing rules as non-sparse resources. See [Memory Aliasing](#) for more information.

`VkDevice` objects that have the `sparseResidencyAliased` feature enabled are able to use the `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT` and `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT` flags for resource creation. These flags allow resources to access physical memory bound into multiple locations within one or more sparse resources in a *data consistent* fashion. This means that reading physical memory from multiple aliased locations will return the same value.

Care **must** be taken when performing a write operation to aliased physical memory. Memory dependencies **must** be used to separate writes to one alias from reads or writes to another alias. Writes to aliased memory that are not properly guarded against accesses to different aliases will have undefined results for all accesses to the aliased memory.

Applications that wish to make use of data consistent sparse memory aliasing **must** abide by the following guidelines:

- All sparse resources that are bound to aliased physical memory **must** be created with the `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT` / `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT` flag.
- All resources that access aliased physical memory **must** interpret the memory in the same way. This implies the following:
  - Buffers and images **cannot** alias the same physical memory in a data consistent fashion. The physical memory ranges **must** be used exclusively by buffers or used exclusively by images for data consistency to be guaranteed.
  - Memory in sparse image mip tail regions **cannot** access aliased memory in a data consistent fashion.
  - Sparse images that alias the same physical memory **must** have compatible formats and be using the same sparse image block shape in order to access aliased memory in a data consistent fashion.

Failure to follow any of the above guidelines will require the application to abide by the normal, non-sparse resource aliasing rules. In this case memory **cannot** be accessed in a data consistent fashion.
Enabling sparse resource memory aliasing can be a way to lower physical memory use, but it may reduce performance on some implementations. An application developer can test on their target HW and balance the memory / performance trade-offs measured.

29.6. Sparse Resource Implementation Guidelines (Informative)

This section is Informative. It is included to aid in implementors’ understanding of sparse resources.

Device Virtual Address

The basic sparseBinding feature allows the resource to reserve its own device virtual address range at resource creation time rather than relying on a bind operation to set this. Without any other creation flags, no other constraints are relaxed compared to normal resources. All pages must be bound to physical memory before the device accesses the resource.

The sparseResidency features allow sparse resources to be used even when not all pages are bound to memory. Implementations that support access to unbound pages without causing a fault may support residencyNonResidentStrict.

Not faulting on access to unbound pages is not enough to support residencyNonResidentStrict. An implementation must also guarantee that reads after writes to unbound regions of the resource always return data for the read as if the memory contains zeros. Depending on any caching hierarchy of the implementation this may not always be possible.

Any implementation that does not fault, but does not guarantee correct read values must not support residencyNonResidentStrict.

Any implementation that cannot access unbound pages without causing a fault will require the implementation to bind the entire device virtual address range to physical memory. Any pages that the application does not bind to memory may be bound to one (or more) "placeholder" physical page(s) allocated by the implementation. Given the following properties:

- A process must not access memory from another process
- Reads return undefined values

It is sufficient for each host process to allocate these placeholder pages and use them for all resources in that process. Implementations may allocate more often (per instance, per device, or per resource).

Binding Memory

The byte size reported in VkMemoryRequirements::size must be greater than or equal to the amount of physical memory required to fully populate the resource. Some implementations
require “holes” in the device virtual address range that are never accessed. These holes may be included in the size reported for the resource.

Including or not including the device virtual address holes in the resource size will alter how the implementation provides support for VkSparseImageOpaqueMemoryBindInfo. This operation must be supported for all sparse images, even ones created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.

- If the holes are included in the size, this bind function becomes very easy. In most cases the resourceOffset is simply a device virtual address offset and the implementation can easily determine what device virtual address to bind. The cost is that the application may allocate more physical memory for the resource than it needs.

- If the holes are not included in the size, the application can allocate less physical memory than otherwise for the resource. However, in this case the implementation must account for the holes when mapping resourceOffset to the actual device virtual address intended to be mapped.

Note

If the application always uses VkSparseImageMemoryBindInfo to bind memory for the non-tail mip levels, any holes that are present in the resource size may never be bound.

Since VkSparseImageMemoryBindInfo uses texel locations to determine which device virtual addresses to bind, it is impossible to bind device virtual address holes with this operation.

Binding Metadata Memory

All metadata for sparse images have their own sparse properties and are embedded in the mip tail region for said properties. See the Multiaspect section for details.

Given that metadata is in a mip tail region, and the mip tail region must be reported as contiguous (either globally or per-array-layer), some implementations will have to resort to complicated offset → device virtual address mapping for handling VkSparseImageOpaqueMemoryBindInfo.

To make this easier on the implementation, the VK_SPARSE_MEMORY_BIND_METADATA_BIT explicitly specifies when metadata is bound with VkSparseImageOpaqueMemoryBindInfo. When this flag is not present, the resourceOffset may be treated as a strict device virtual address offset.

When VK_SPARSE_MEMORY_BIND_METADATA_BIT is present, the resourceOffset must have been derived explicitly from the imageMipTailOffset in the sparse resource properties returned for the metadata aspect. By manipulating the value returned for imageMipTailOffset, the resourceOffset does not have to correlate directly to a device virtual address offset, and may instead be whatever value makes it easiest for the implementation to derive the correct device virtual address.
29.7. Sparse Resource API

The APIs related to sparse resources are grouped into the following categories:

- Physical Device Features
- Physical Device Sparse Properties
- Sparse Image Format Properties
- Sparse Resource Creation
- Sparse Resource Memory Requirements
- Binding Resource Memory

29.7.1. Physical Device Features

Some sparse-resource related features are reported and enabled in VkPhysicalDeviceFeatures. These features must be supported and enabled on the VkDevice object before applications can use them. See Physical Device Features for information on how to get and set enabled device features, and for more detailed explanations of these features.

Sparse Physical Device Features

- **sparseBinding**: Support for creating VkBuffer and VkImage objects with the VK_BUFFER_CREATE_SPARSE_BINDING_BIT and VK_IMAGE_CREATE_SPARSE_BINDING_BIT flags, respectively.

- **sparseResidencyBuffer**: Support for creating VkBuffer objects with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT flag.

- **sparseResidencyImage2D**: Support for creating 2D single-sampled VkImage objects with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.

- **sparseResidencyImage3D**: Support for creating 3D VkImage objects with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.

- **sparseResidency2Samples**: Support for creating 2D VkImage objects with 2 samples and VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.

- **sparseResidency4Samples**: Support for creating 2D VkImage objects with 4 samples and VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.

- **sparseResidency8Samples**: Support for creating 2D VkImage objects with 8 samples and VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.

- **sparseResidency16Samples**: Support for creating 2D VkImage objects with 16 samples and VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.

- **sparseResidencyAliased**: Support for creating VkBuffer and VkImage objects with the VK_BUFFER_CREATE_SPARSE_ALIASED_BIT and VK_IMAGE_CREATE_SPARSE_ALIASED_BIT flags, respectively.

29.7.2. Physical Device Sparse Properties

Some features of the implementation are not possible to disable, and are reported to allow applications to alter their sparse resource usage accordingly. These read-only capabilities are
reported in the `VkPhysicalDeviceProperties::sparseProperties` member, which is a `VkPhysicalDeviceSparseProperties` structure.

The `VkPhysicalDeviceSparseProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPhysicalDeviceSparseProperties {
    VkBool32 residencyStandard2DBlockShape;
    VkBool32 residencyStandard2DMultisampleBlockShape;
    VkBool32 residencyStandard3DBlockShape;
    VkBool32 residencyAlignedMipSize;
    VkBool32 residencyNonResidentStrict;
} VkPhysicalDeviceSparseProperties;
```

- `residencyStandard2DBlockShape` is `VK_TRUE` if the physical device will access all single-sample 2D sparse resources using the standard sparse image block shapes (based on image format), as described in the `Standard Sparse Image Block Shapes (Single Sample)` table. If this property is not supported, the value returned in the `imageGranularity` member of the `VkSparseImageFormatProperties` structure for single-sample 2D images is not required to match the standard sparse image block dimensions listed in the table.

- `residencyStandard2DMultisampleBlockShape` is `VK_TRUE` if the physical device will access all multisample 2D sparse resources using the standard sparse image block shapes (based on image format), as described in the `Standard Sparse Image Block Shapes (MSAA)` table. If this property is not supported, the value returned in the `imageGranularity` member of the `VkSparseImageFormatProperties` structure for multisample 2D images is not required to match the standard sparse image block dimensions listed in the table.

- `residencyStandard3DBlockShape` is `VK_TRUE` if the physical device will access all 3D sparse resources using the standard sparse image block shapes (based on image format), as described in the `Standard Sparse Image Block Shapes (Single Sample)` table. If this property is not supported, the value returned in the `imageGranularity` member of the `VkSparseImageFormatProperties` structure for 3D images is not required to match the standard sparse image block dimensions listed in the table.

- `residencyAlignedMipSize` is `VK_TRUE` if images with mip level dimensions that are not integer multiples of the corresponding dimensions of the sparse image block may be placed in the mip tail. If this property is not reported, only mip levels with dimensions smaller than the `imageGranularity` member of the `VkSparseImageFormatProperties` structure will be placed in the mip tail. If this property is reported the implementation is allowed to return `VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT` in the `flags` member of `VkSparseImageFormatProperties`, indicating that mip level dimensions that are not integer multiples of the corresponding dimensions of the sparse image block will be placed in the mip tail.

- `residencyNonResidentStrict` specifies whether the physical device can consistently access nonresident regions of a resource. If this property is `VK_TRUE`, access to non-resident regions of resources will be guaranteed to return values as if the resource was populated with 0; writes to non-resident regions will be discarded.
29.7.3. Sparse Image Format Properties

Given that certain aspects of sparse image support, including the sparse image block dimensions, may be implementation-dependent, `vkGetPhysicalDeviceSparseImageFormatProperties` can be used to query for sparse image format properties prior to resource creation. This command is used to check whether a given set of sparse image parameters is supported and what the sparse image block shape will be.

Sparse Image Format Properties API

The `VkSparseImageFormatProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSparseImageFormatProperties {
    VkImageAspectFlags aspectMask;
    VkExtent3D imageGranularity;
    VkSparseImageFormatFlags flags;
} VkSparseImageFormatProperties;
```

- `aspectMask` is a bitmask `VkImageAspectFlagBits` specifying which aspects of the image the properties apply to.
- `imageGranularity` is the width, height, and depth of the sparse image block in texels or compressed texel blocks.
- `flags` is a bitmask of `VkSparseImageFormatFlagBits` specifying additional information about the sparse resource.

Bits which may be set in `VkSparseImageFormatProperties::flags`, specifying additional information about the sparse resource, are:

```c
// Provided by VK_VERSION_1_0
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;
```

- `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` specifies that the image uses a single mip tail region for all array layers.
- `VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT` specifies that the first mip level whose dimensions are not integer multiples of the corresponding dimensions of the sparse image block begins the mip tail region.
- `VK_SPARSE_IMAGE_FORMAT_NONSTANDARD_BLOCK_SIZE_BIT` specifies that the image uses non-standard sparse image block dimensions, and the `imageGranularity` values do not match the standard sparse image block dimensions for the given format.

typedef VkFlags VkSparseImageFormatFlags;

VkSparseImageFormatFlags is a bitmask type for setting a mask of zero or more
VkSparseImageFormatFlagBits.

vkGetPhysicalDeviceSparseImageFormatProperties returns an array of
VkSparseImageFormatProperties. Each element will describe properties for one set of image
aspects that are bound simultaneously in the image. This is usually one element for each aspect in
the image, but for interleaved depth/stencil images there is only one element describing the
combined aspects.

void vkGetPhysicalDeviceSparseImageFormatProperties(
    VkPhysicalDevice physicalDevice,
    VkFormat format,
    VkImageType type,
    VkSampleCountFlagBits samples,
    VkImageUsageFlags usage,
    VkImageTiling tiling,
    uint32_t* pPropertyCount,
    VkSparseImageFormatProperties* pProperties);

• physicalDevice is the physical device from which to query the sparse image format properties.
• format is the image format.
• type is the dimensionality of image.
• samples is a VkSampleCountFlagBits value specifying the number of samples per texel.
• usage is a bitmask describing the intended usage of the image.
• tiling is the tiling arrangement of the texel blocks in memory.
• pPropertyCount is a pointer to an integer related to the number of sparse format properties
  available or queried, as described below.
• pProperties is either NULL or a pointer to an array of VkSparseImageFormatProperties
  structures.

If pProperties is NULL, then the number of sparse format properties available is returned in
pPropertyCount. Otherwise, pPropertyCount must point to a variable set by the user to the number of
elements in the pProperties array, and on return the variable is overwritten with the number of
structures actually written to pProperties. If pPropertyCount is less than the number of sparse
format properties available, at most pPropertyCount structures will be written.

If VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT is not supported for the given arguments, pPropertyCount
will be set to zero upon return, and no data will be written to pProperties.

Multiple aspects are returned for depth/stencil images that are implemented as separate planes by
the implementation. The depth and stencil data planes each have unique
Depth/stencil images with depth and stencil data interleaved into a single plane will return a single `VkSparseImageFormatProperties` structure with the `aspectMask` set to `VK_IMAGE_ASPECT_DEPTH_BIT | VK_IMAGE_ASPECT_STENCIL_BIT`.

### Valid Usage

- VUID-vkGetPhysicalDeviceSparseImageFormatProperties-samples-01094
  - `samples` must be a bit value that is set in `VkImageFormatProperties::sampleCounts` returned by `vkGetPhysicalDeviceImageFormatProperties` with `format`, `type`, `tiling`, and `usage` equal to those in this command and `flags` equal to the value that is set in `VkImageCreateInfo ::flags` when the image is created.

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceSparseImageFormatProperties-physicalDevice-parameter
  - `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceSparseImageFormatProperties-format-parameter
  - `format` must be a valid `VkFormat` value
- VUID-vkGetPhysicalDeviceSparseImageFormatProperties-type-parameter
  - `type` must be a valid `VkImageType` value
- VUID-vkGetPhysicalDeviceSparseImageFormatProperties-samples-parameter
  - `samples` must be a valid `VkSampleCountFlagBits` value
- VUID-vkGetPhysicalDeviceSparseImageFormatProperties-usage-parameter
  - `usage` must be a valid combination of `VkImageUsageFlagBits` values
- VUID-vkGetPhysicalDeviceSparseImageFormatProperties-usage-requiredbitmask
  - `usage` must not be `0`
- VUID-vkGetPhysicalDeviceSparseImageFormatProperties-tiling-parameter
  - `tiling` must be a valid `VkImageTiling` value
- VUID-vkGetPhysicalDeviceSparseImageFormatProperties-pPropertyCount-parameter
  - `pPropertyCount` must be a valid pointer to a `uint32_t` value
  - If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` must be a valid pointer to an array of `pPropertyCount` `VkSparseImageFormatProperties` structures

`vkGetPhysicalDeviceSparseImageFormatProperties2` returns an array of `VkSparseImageFormatProperties2`. Each element will describe properties for one set of image aspects that are bound simultaneously in the image. This is usually one element for each aspect in the image, but for interleaved depth/stencil images there is only one element describing the combined aspects.
void vkGetPhysicalDeviceSparseImageFormatProperties2(
    VkPhysicalDevice physicalDevice,
    const VkPhysicalDeviceSparseImageFormatInfo2* pFormatInfo,
    uint32_t* pPropertyCount,
    VkSparseImageFormatProperties2* pProperties);

- **physicalDevice** is the physical device from which to query the sparse image format properties.
- **pFormatInfo** is a pointer to a `VkPhysicalDeviceSparseImageFormatInfo2` structure containing input parameters to the command.
- **pPropertyCount** is a pointer to an integer related to the number of sparse format properties available or queried, as described below.
- **pProperties** is either **NULL** or a pointer to an array of `VkSparseImageFormatProperties2` structures.

`vkGetPhysicalDeviceSparseImageFormatProperties2` behaves identically to `vkGetPhysicalDeviceSparseImageFormatProperties`, with the ability to return extended information by adding extending structures to the `pNext` chain of its `pProperties` parameter.

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceSparseImageFormatProperties2-physicalDevice-parameter
  - `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceSparseImageFormatProperties2-pFormatInfo-parameter
  - `pFormatInfo` **must** be a valid pointer to a valid `VkPhysicalDeviceSparseImageFormatInfo2` structure
- VUID-vkGetPhysicalDeviceSparseImageFormatProperties2-pPropertyCount-parameter
  - `pPropertyCount` **must** be a valid pointer to a `uint32_t` value
  - If the value referenced by `pPropertyCount` is not 0, and `pProperties` is not **NULL**, `pProperties` **must** be a valid pointer to an array of `VkSparseImageFormatProperties2` structures

The `VkPhysicalDeviceSparseImageFormatInfo2` structure is defined as:

```c
typedef struct VkPhysicalDeviceSparseImageFormatInfo2 {
    VkStructureType sType;
    const void* pNext;
    VkFormat format;
    VkImageType type;
    VkSampleCountFlagBits samples;
    VkImageUsageFlags usage;
    VkImageTiling tiling;
} VkPhysicalDeviceSparseImageFormatInfo2;
```
VkPhysicalDeviceSparseImageFormatInfo2;

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `format` is the image format.
- `type` is the dimensionality of image.
- `samples` is a `VkSampleCountFlagBits` value specifying the number of samples per texel.
- `usage` is a bitmask describing the intended usage of the image.
- `tiling` is the tiling arrangement of the texel blocks in memory.

### Valid Usage

- VUID-VkPhysicalDeviceSparseImageFormatInfo2-samples-01095
  
samples must be a bit value that is set in `VkImageFormatProperties::sampleCounts` returned by `vkGetPhysicalDeviceImageFormatProperties` with `format`, `type`, `tiling`, and `usage` equal to those in this command and `flags` equal to the value that is set in `VkImageCreateInfo::flags` when the image is created

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceSparseImageFormatInfo2-sType-sType
  
sType must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SPARSE_IMAGE_FORMAT_INFO_2`

- VUID-VkPhysicalDeviceSparseImageFormatInfo2-pNext-pNext
  
pNext must be `NULL`

- VUID-VkPhysicalDeviceSparseImageFormatInfo2-format-parameter
  
format must be a valid `VkFormat` value

- VUID-VkPhysicalDeviceSparseImageFormatInfo2-type-parameter
  
type must be a valid `VkImageType` value

- VUID-VkPhysicalDeviceSparseImageFormatInfo2-samples-parameter
  
samples must be a valid `VkSampleCountFlagBits` value

- VUID-VkPhysicalDeviceSparseImageFormatInfo2-usage-parameter
  
usage must be a valid combination of `VkImageUsageFlagBits` values

- VUID-VkPhysicalDeviceSparseImageFormatInfo2-usage-requiredbitmask
  
usage must not be `0`

- VUID-VkPhysicalDeviceSparseImageFormatInfo2-tiling-parameter
  
tiling must be a valid `VkImageTiling` value

The `VkSparseImageFormatProperties2` structure is defined as:

```cpp
// Provided by VK_VERSION_1_1
```

1038
typedef struct VkSparseImageFormatProperties2 {
    VkStructureType sType;
    void* pNext;
    VkSparseImageFormatProperties properties;
} VkSparseImageFormatProperties2;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **properties** is a VkSparseImageFormatProperties structure which is populated with the same values as in vkGetPhysicalDeviceSparseImageFormatProperties.

**Valid Usage (Implicit)**

- VUID-VkSparseImageFormatProperties2-sType-sType
  - **sType** must be VK_STRUCTURE_TYPE_SPARSE_IMAGE_FORMAT_PROPERTIES_2
- VUID-VkSparseImageFormatProperties2-pNext-pNext
  - **pNext** must be NULL

### 29.7.4. Sparse Resource Creation

Sparse resources require that one or more sparse feature flags be specified (as part of the VkPhysicalDeviceFeatures structure described previously in the Physical Device Features section) when calling vkCreateDevice. When the appropriate device features are enabled, the VK_BUFFER_CREATE_SPARSE_* and VK_IMAGE_CREATE_SPARSE_* flags can be used. See vkCreateBuffer and vkCreateImage for details of the resource creation APIs.

**Note**

Specifying VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT or VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT requires specifying VK_BUFFER_CREATE_SPARSE_BINDING_BIT or VK_IMAGE_CREATE_SPARSE_BINDING_BIT, respectively, as well. This means that resources must be created with the appropriate *_SPARSE_BINDING_BIT to be used with the sparse binding command (vkQueueBindSparse).

### 29.7.5. Sparse Resource Memory Requirements

Sparse resources have specific memory requirements related to binding sparse memory. These memory requirements are reported differently for VkBuffer objects and VkImage objects.

**Buffer and Fully- Resident Images**

Buffers (both fully and partially resident) and fully-resident images can be bound to memory using only the data from VkMemoryRequirements. For all sparse resources the VkMemoryRequirements::alignment member specifies both the bindable sparse block size in bytes and **required** alignment of VkDeviceMemory.
Partially Resident Images

Partially resident images have a different method for binding memory. As with buffers and fully resident images, the `VkMemoryRequirements::alignment` field specifies the bindable sparse block size in bytes for the image.

Requesting sparse memory requirements for `VkImage` objects using `vkGetImageSparseMemoryRequirements` will return an array of one or more `VkSparseImageMemoryRequirements` structures. Each structure describes the sparse memory requirements for a group of aspects of the image.

The sparse image must have been created using the `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` flag to retrieve valid sparse image memory requirements.

Sparse Image Memory Requirements

The `VkSparseImageMemoryRequirements` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSparseImageMemoryRequirements {
    VkSparseImageFormatProperties formatProperties;
    uint32_t imageMipTailFirstLod;
    VkDeviceSize imageMipTailSize;
    VkDeviceSize imageMipTailOffset;
    VkDeviceSize imageMipTailStride;
} VkSparseImageMemoryRequirements;
```

- `formatProperties` is a `VkSparseImageFormatProperties` structure specifying properties of the image format.
- `imageMipTailFirstLod` is the first mip level at which image subresources are included in the mip tail region.
- `imageMipTailSize` is the memory size (in bytes) of the mip tail region. If `formatProperties.flags` contains `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT`, this is the size of the whole mip tail, otherwise this is the size of the mip tail of a single array layer. This value is guaranteed to be a multiple of the sparse block size in bytes.
- `imageMipTailOffset` is the opaque memory offset used with `VkSparseImageOpaqueMemoryBindInfo` to bind the mip tail region(s).
- `imageMipTailStride` is the offset stride between each array-layer's mip tail, if `formatProperties.flags` does not contain `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` (otherwise the value is undefined).

To query sparse memory requirements for an image, call:

```c
// Provided by VK_VERSION_1_0
void vkGetImageSparseMemoryRequirements(
    VkDevice device,
    VkImage image,
    uint32_t imageCount,
    const VkSparseImageMemoryRequirements *pSparseImageMemoryRequirements,
    VkSparseImageMemoryRequirements *pSparseImageMemoryRequirementsOut);
```
uint32_t* pSparseMemoryRequirementCount,
VkSparseImageMemoryRequirements* pSparseMemoryRequirements);
To query sparse memory requirements for an image, call:

```c
// Provided by VK_VERSION_1_1
void vkGetImageSparseMemoryRequirements2(
    VkDevice device,
    const VkImageSparseMemoryRequirementsInfo2* pInfo,
    uint32_t* pSparseMemoryRequirementCount,
    VkSparseImageMemoryRequirements2* pSparseMemoryRequirements);
```

- `device` is the logical device that owns the image.
- `pInfo` is a pointer to a `VkImageSparseMemoryRequirementsInfo2` structure containing parameters required for the memory requirements query.
- `pSparseMemoryRequirementCount` is a pointer to an integer related to the number of sparse memory requirements available or queried, as described below.
- `pSparseMemoryRequirements` is either `NULL` or a pointer to an array of `VkSparseImageMemoryRequirements2` structures.

### Valid Usage (Implicit)

- **VUID-vkGetImageSparseMemoryRequirements2-device-parameter**
  `device` must be a valid `VkDevice` handle
- **VUID-vkGetImageSparseMemoryRequirements2-pInfo-parameter**
  `pInfo` must be a valid pointer to a valid `VkImageSparseMemoryRequirementsInfo2` structure
- **VUID-vkGetImageSparseMemoryRequirements2-pSparseMemoryRequirementCount-parameter**
  `pSparseMemoryRequirementCount` must be a valid pointer to a `uint32_t` value
- **VUID-vkGetImageSparseMemoryRequirements2-pSparseMemoryRequirements-parameter**
  If the value referenced by `pSparseMemoryRequirementCount` is not 0, and `pSparseMemoryRequirements` is not `NULL`, `pSparseMemoryRequirements` must be a valid pointer to an array of `pSparseMemoryRequirementCount` `VkSparseImageMemoryRequirements2` structures

To determine the sparse memory requirements for an image resource without creating an object, call:

```c
// Provided by VK_VERSION_1_3
void vkGetDeviceImageSparseMemoryRequirements(
    VkDevice device,
    const VkDeviceImageMemoryRequirements2* pInfo,
    uint32_t* pSparseMemoryRequirementCount,
    VkSparseImageMemoryRequirements2* pSparseMemoryRequirements);
```
- **device** is the logical device intended to own the image.
- **pInfo** is a pointer to a `VkDeviceImageMemoryRequirements` structure containing parameters required for the memory requirements query.
- **pSparseMemoryRequirementCount** is a pointer to an integer related to the number of sparse memory requirements available or queried, as described below.
- **pSparseMemoryRequirements** is either `NULL` or a pointer to an array of `VkSparseImageMemoryRequirements2` structures.

### Valid Usage (Implicit)

- **VUID-vkGetDeviceImageSparseMemoryRequirements-device-parameter**
  - The **device** must be a valid `VkDevice` handle.
- **VUID-vkGetDeviceImageSparseMemoryRequirements-pInfo-parameter**
  - The **pInfo** must be a valid pointer to a valid `VkDeviceImageMemoryRequirements` structure.
- **VUID-vkGetDeviceImageSparseMemoryRequirements-pSparseMemoryRequirementCount-parameter**
  - The **pSparseMemoryRequirementCount** must be a valid pointer to a `uint32_t` value.
- **VUID-vkGetDeviceImageSparseMemoryRequirements-pSparseMemoryRequirements-parameter**
  - If the value referenced by **pSparseMemoryRequirementCount** is not 0, and **pSparseMemoryRequirements** is not `NULL`, **pSparseMemoryRequirements** must be a valid pointer to an array of ` VkSparseImageMemoryRequirements2` structures.

The **VkImageSparseMemoryRequirementsInfo2** structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkImageSparseMemoryRequirementsInfo2 {
    VkStructureType sType;
    const void* pNext;
    VkImage image;
} VkImageSparseMemoryRequirementsInfo2;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **image** is the image to query.

### Valid Usage (Implicit)

- **VUID-VkImageSparseMemoryRequirementsInfo2-sType-sType**
  - The **sType** must be `VK_STRUCTURE_TYPE_IMAGE_SPARSE_MEMORY_REQUIREMENTS_INFO_2`.
- **VUID-VkImageSparseMemoryRequirementsInfo2-pNext-pNext**
  - The **pNext** must be a valid pointer.
The `VkSparseImageMemoryRequirements2` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkSparseImageMemoryRequirements2 {
    VkStructureType sType;
    void* pNext;
    VkSparseImageMemoryRequirements memoryRequirements;
} VkSparseImageMemoryRequirements2;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `memoryRequirements` is a `VkSparseImageMemoryRequirements` structure describing the memory requirements of the sparse image.

### Valid Usage (Implicit)

- VUID-VkSparseImageMemoryRequirements2-sType-sType
  - `sType` must be `VK_STRUCTURE_TYPE_SPARSE_IMAGE_MEMORY_REQUIREMENTS_2`
- VUID-VkSparseImageMemoryRequirements2-pNext-pNext
  - `pNext` must be `NULL`

### 29.7.6. Binding Resource Memory

Non-sparse resources are backed by a single physical allocation prior to device use (via `vkBindImageMemory` or `vkBindBufferMemory`), and their backing must not be changed. On the other hand, sparse resources can be bound to memory non-contiguously and these bindings can be altered during the lifetime of the resource.

**Note**

It is important to note that freeing a `VkDeviceMemory` object with `vkFreeMemory` will not cause resources (or resource regions) bound to the memory object to become unbound. Applications must not access resources bound to memory that has been freed.

Sparse memory bindings execute on a queue that includes the `VK_QUEUE_SPARSE_BINDING_BIT` bit. Applications must use synchronization primitives to guarantee that other queues do not access ranges of memory concurrently with a binding change. Applications can access other ranges of the same resource while a bind operation is executing.

**Note**
Implementations must provide a guarantee that simultaneously binding sparse blocks while another queue accesses those same sparse blocks via a sparse resource must not access memory owned by another process or otherwise corrupt the system.

While some implementations may include VK_QUEUE_SPARSE_BINDING_BIT support in queue families that also include graphics and compute support, other implementations may only expose a VK_QUEUE_SPARSE_BINDING_BIT-only queue family. In either case, applications must use synchronization primitives to explicitly request any ordering dependencies between sparse memory binding operations and other graphics/compute/transfer operations, as sparse binding operations are not automatically ordered against command buffer execution, even within a single queue.

When binding memory explicitly for the VK_IMAGE_ASPECT_METADATA_BIT the application must use the VK_SPARSE_MEMORY_BIND_METADATA_BIT in the VkSparseMemoryBind::flags field when binding memory. Binding memory for metadata is done the same way as binding memory for the mip tail, with the addition of the VK_SPARSE_MEMORY_BIND_METADATA_BIT flag.

Binding the mip tail for any aspect must only be performed using VkSparseImageOpaqueMemoryBindInfo. If formatProperties.flags contains VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT, then it can be bound with a single VkSparseMemoryBind structure, with resourceOffset = imageMipTailOffset and size = imageMipTailSize.

If formatProperties.flags does not contain VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT then the offset for the mip tail in each array layer is given as:

arrayMipTailOffset = imageMipTailOffset + arrayLayer * imageMipTailStride;

and the mip tail can be bound with layerCount VkSparseMemoryBind structures, each using size = imageMipTailSize and resourceOffset = arrayMipTailOffset as defined above.

Sparse memory binding is handled by the following APIs and related data structures.

**Sparse Memory Binding Functions**

The VkSparseMemoryBind structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSparseMemoryBind {
    VkDeviceSize resourceOffset;
    VkDeviceSize size;
    VkDeviceMemory memory;
    VkDeviceSize memoryOffset;
    VkSparseMemoryBindFlags flags;
} VkSparseMemoryBind;
```

- resourceOffset is the offset into the resource.
• **size** is the size of the memory region to be bound.

• **memory** is the **VkDeviceMemory** object that the range of the resource is bound to. If **memory** is **VK_NULL_HANDLE**, the range is unbound.

• **memoryOffset** is the offset into the **VkDeviceMemory** object to bind the resource range to. If **memory** is **VK_NULL_HANDLE**, this value is ignored.

• **flags** is a bitmask of **VkSparseMemoryBindFlagBits** specifying usage of the binding operation.

The **binding range** \([resourceOffset, resourceOffset + size]\) has different constraints based on **flags**. If **flags** contains **VK_SPARSE_MEMORY_BIND_METADATA_BIT**, the binding range **must** be within the mip tail region of the metadata aspect. This metadata region is defined by:

\[
\text{metadataRegion} = [\text{base}, \text{base} + \text{imageMipTailSize})
\]

\[
\text{base} = \text{imageMipTailOffset} + \text{imageMipTailStride} \times n
\]

and **imageMipTailOffset**, **imageMipTailSize**, and **imageMipTailStride** values are from the **VkSparseImageMemoryRequirements** corresponding to the metadata aspect of the image, and \(n\) is a valid array layer index for the image,

**imageMipTailStride** is considered to be zero for aspects where **VkSparseImageMemoryRequirements**:formatProperties.flags contains **VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT**.

If **flags** does not contain **VK_SPARSE_MEMORY_BIND_METADATA_BIT**, the binding range **must** be within the range \([0, \text{VkMemoryRequirements}:size]\).

### Valid Usage

- **VUID-VkSparseMemoryBind-memory-01096**
  If **memory** is not **VK_NULL_HANDLE**, **memory** and **memoryOffset** **must** match the memory requirements of the resource, as described in section **Resource Memory Association**

- **VUID-VkSparseMemoryBind-memory-01097**
  If **memory** is not **VK_NULL_HANDLE**, **memory** **must** not have been created with a memory type that reports **VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT** bit set

- **VUID-VkSparseMemoryBind-size-01098**
  **size** **must** be greater than 0

- **VUID-VkSparseMemoryBind-resourceOffset-01099**
  **resourceOffset** **must** be less than the size of the resource

- **VUID-VkSparseMemoryBind-size-01100**
  **size** **must** be less than or equal to the size of the resource minus **resourceOffset**

- **VUID-VkSparseMemoryBind-memoryOffset-01101**
  **memoryOffset** **must** be less than the size of **memory**

- **VUID-VkSparseMemoryBind-size-01102**
  **size** **must** be less than or equal to the size of **memory** minus **memoryOffset**
If memory was created with `VkExportMemoryAllocateInfo::handleTypes` not equal to 0, at least one handle type it contained must also have been set in `VkExternalMemoryBufferCreateInfo::handleTypes` or `VkExternalMemoryImageCreateInfo::handleTypes` when the resource was created.

If memory was created by a memory import operation, the external handle type of the imported memory must also have been set in `VkExternalMemoryBufferCreateInfo::handleTypes` or `VkExternalMemoryImageCreateInfo::handleTypes` when the resource was created.

### Valid Usage (Implicit)

- **VUID-VkSparseMemoryBind-memory-parameter**
  - If memory is not `VK_NULL_HANDLE`, memory must be a valid `VkDeviceMemory` handle.

- **VUID-VkSparseMemoryBind-flags-parameter**
  - flags must be a valid combination of `VkSparseMemoryBindFlagBits` values.

Bits which can be set in `VkSparseMemoryBind::flags`, specifying usage of a sparse memory binding operation, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkSparseMemoryBindFlagBits {
    VK_SPARSE_MEMORY_BIND_METADATA_BIT = 0x00000001,
} VkSparseMemoryBindFlagBits;
```

- **VK_SPARSE_MEMORY_BIND_METADATA_BIT** specifies that the memory being bound is only for the metadata aspect.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkSparseMemoryBindFlags;
```

`VkSparseMemoryBindFlags` is a bitmask type for setting a mask of zero or more `VkSparseMemoryBindFlagBits`.

Memory is bound to `VkBuffer` objects created with the `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` flag using the following structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSparseBufferMemoryBindInfo {
    VkBuffer       buffer;
    uint32_t       bindCount;
    const VkSparseMemoryBind* pBinds;
} VkSparseBufferMemoryBindInfo;
```
VkSparseBufferMemoryBindInfo;

- `buffer` is the `VkBuffer` object to be bound.
- `bindCount` is the number of `VkSparseMemoryBind` structures in the `pBinds` array.
- `pBinds` is a pointer to an array of `VkSparseMemoryBind` structures.

Valid Usage (Implicit)

- VUID-VkSparseBufferMemoryBindInfo-buffer-parameter
  `buffer` must be a valid `VkBuffer` handle
- VUID-VkSparseBufferMemoryBindInfo-pBinds-parameter
  `pBinds` must be a valid pointer to an array of `bindCount` valid `VkSparseMemoryBind` structures
- VUID-VkSparseBufferMemoryBindInfo-bindCount-arraylength
  `bindCount` must be greater than 0

Memory is bound to opaque regions of `VkImage` objects created with the `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` flag using the following structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSparseImageOpaqueMemoryBindInfo {
    VkImage image;
    uint32_t bindCount;
    const VkSparseMemoryBind* pBinds;
} VkSparseImageOpaqueMemoryBindInfo;
```

- `image` is the `VkImage` object to be bound.
- `bindCount` is the number of `VkSparseMemoryBind` structures in the `pBinds` array.
- `pBinds` is a pointer to an array of `VkSparseMemoryBind` structures.

Valid Usage

- VUID-VkSparseImageOpaqueMemoryBindInfo-pBinds-01103
  If the flags member of any element of `pBinds` contains `VK_SPARSE_MEMORY_BIND_METADATA_BIT`, the binding range defined must be within the mip tail region of the metadata aspect of `image`

Valid Usage (Implicit)

- VUID-VkSparseImageOpaqueMemoryBindInfo-image-parameter
  `image` must be a valid `VkImage` handle
- VUID-VkSparseImageOpaqueMemoryBindInfo-pBinds-parameter
**pBinds must** be a valid pointer to an array of `bindCount` valid `VkSparseMemoryBind` structures

- VUID-VkSparseImageOpaqueMemoryBindInfo-bindCount-arraylength

  `bindCount` must be greater than 0

---

**Note**

This operation is normally used to bind memory to fully-resident sparse images or for mip tail regions of partially resident images. However, it can also be used to bind memory for the entire binding range of partially resident images.

In case `flags` does not contain `VK_SPARSE_MEMORY_BIND_METADATA_BIT`, the `resourceOffset` is in the range [0, `VkMemoryRequirements::size`), This range includes data from all aspects of the image, including metadata. For most implementations this will probably mean that the `resourceOffset` is a simple device address offset within the resource. It is possible for an application to bind a range of memory that includes both resource data and metadata. However, the application would not know what part of the image the memory is used for, or if any range is being used for metadata.

When `flags` contains `VK_SPARSE_MEMORY_BIND_METADATA_BIT`, the binding range specified must be within the mip tail region of the metadata aspect. In this case the `resourceOffset` is not required to be a simple device address offset within the resource. However, it is defined to be within `[imageMipTailOffset, imageMipTailOffset + imageMipTailSize)` for the metadata aspect. See `VkSparseMemoryBind` for the full constraints on binding region with this flag present.

Memory can be bound to sparse image blocks of `VkImage` objects created with the `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` flag using the following structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSparseImageMemoryBindInfo {
    VkImage            image;
    uint32_t           bindCount;
    const VkSparseImageMemoryBind* pBinds;
} VkSparseImageMemoryBindInfo;
```

- `image` is the `VkImage` object to be bound
- `bindCount` is the number of `VkSparseImageMemoryBind` structures in `pBinds` array
- `pBinds` is a pointer to an array of `VkSparseImageMemoryBind` structures

---

**Valid Usage**

- VUID-VkSparseImageMemoryBindInfo-subresource-01722

  The `subresource.mipLevel` member of each element of `pBinds` must be less than the
mipLevels specified in VkImageCreateInfo when image was created

- VUID-VkSparseImageMemoryBindInfo-subresource-01723
  The subresource.arrayLayer member of each element of pBinds must be less than the arrayLayers specified in VkImageCreateInfo when image was created

- VUID-VkSparseImageMemoryBindInfo-image-02901
  image must have been created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set

### Valid Usage (Implicit)

- VUID-VkSparseImageMemoryBindInfo-image-parameter
  image must be a valid VkImage handle

- VUID-VkSparseImageMemoryBindInfo-pBinds-parameter
  pBinds must be a valid pointer to an array of bindCount valid VkSparseImageMemoryBind structures

- VUID-VkSparseImageMemoryBindInfo-bindCount-arraylength
  bindCount must be greater than 0

The VkSparseImageMemoryBind structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSparseImageMemoryBind {
    VkImageSubresource subresource;
    VkOffset3D offset;
    VkExtent3D extent;
    VkDeviceMemory memory;
    VkDeviceSize memoryOffset;
    VkSparseMemoryBindFlags flags;
} VkSparseImageMemoryBind;
```

- **subresource** is the image aspect and region of interest in the image.
- **offset** are the coordinates of the first texel within the image subresource to bind.
- **extent** is the size in texels of the region within the image subresource to bind. The extent must be a multiple of the sparse image block dimensions, except when binding sparse image blocks along the edge of an image subresource it can instead be such that any coordinate of offset + extent equals the corresponding dimensions of the image subresource.
- **memory** is the VkDeviceMemory object that the sparse image blocks of the image are bound to. If memory is VK_NULL_HANDLE, the sparse image blocks are unbound.
- **memoryOffset** is an offset into VkDeviceMemory object. If memory is VK_NULL_HANDLE, this value is ignored.
- **flags** are sparse memory binding flags.
Valid Usage

- VUID-VkSparseImageMemoryBind-memory-01104
  If the `sparseResidencyAliased` feature is not enabled, and if any other resources are bound to ranges of `memory`, the range of `memory` being bound must not overlap with those bound ranges.

- VUID-VkSparseImageMemoryBind-memory-01105
  `memory` and `memoryOffset` must match the memory requirements of the calling command's `image`, as described in section Resource Memory Association.

- VUID-VkSparseImageMemoryBind-subresource-01106
  `subresource` must be a valid image subresource for `image` (see Image Views).

- VUID-VkSparseImageMemoryBind-offset-01107
  `offset.x` must be a multiple of the sparse image block width (`VkSparseImageFormatProperties::imageGranularity.width`) of the image.

- VUID-VkSparseImageMemoryBind-extent-01108
  `extent.width` must either be a multiple of the sparse image block width of the image, or else (`extent.width + offset.x`) must equal the width of the image subresource.

- VUID-VkSparseImageMemoryBind-offset-01109
  `offset.y` must be a multiple of the sparse image block height (`VkSparseImageFormatProperties::imageGranularity.height`) of the image.

- VUID-VkSparseImageMemoryBind-extent-01110
  `extent.height` must either be a multiple of the sparse image block height of the image, or else (`extent.height + offset.y`) must equal the height of the image subresource.

- VUID-VkSparseImageMemoryBind-offset-01111
  `offset.z` must be a multiple of the sparse image block depth (`VkSparseImageFormatProperties::imageGranularity.depth`) of the image.

- VUID-VkSparseImageMemoryBind-extent-01112
  `extent.depth` must either be a multiple of the sparse image block depth of the image, or else (`extent.depth + offset.z`) must equal the depth of the image subresource.

- VUID-VkSparseImageMemoryBind-memory-02732
  If `memory` was created with `VkExportMemoryAllocateInfo::handleTypes` not equal to 0, at least one handle type it contained must also have been set in `VkExternalMemoryImageCreateInfo::handleTypes` when the image was created.

- VUID-VkSparseImageMemoryBind-memory-02733
  If `memory` was created by a memory import operation, the external handle type of the imported memory must also have been set in `VkExternalMemoryImageCreateInfo::handleTypes` when `image` was created.

Valid Usage (Implicit)

- VUID-VkSparseImageMemoryBind-subresource-parameter
  `subresource` must be a valid `VkImageSubresource` structure.
To submit sparse binding operations to a queue, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkQueueBindSparse(
    VkQueue queue,
    uint32_t bindInfoCount,
    const VkBindSparseInfo* pBindInfo,
    VkFence fence);
```

- `queue` is the queue that the sparse binding operations will be submitted to.
- `bindInfoCount` is the number of elements in the `pBindInfo` array.
- `pBindInfo` is a pointer to an array of `VkBindSparseInfo` structures, each specifying a sparse binding submission batch.
- `fence` is an optional handle to a fence to be signaled. If `fence` is not `VK_NULL_HANDLE`, it defines a fence signal operation.

`vkQueueBindSparse` is a queue submission command, with each batch defined by an element of `pBindInfo` as a `VkBindSparseInfo` structure. Batches begin execution in the order they appear in `pBindInfo`, but may complete out of order.

Within a batch, a given range of a resource must not be bound more than once. Across batches, if a range is to be bound to one allocation and offset and then to another allocation and offset, then the application must guarantee (usually using semaphores) that the binding operations are executed in the correct order, as well as to order binding operations against the execution of command buffer submissions.

As no operation to `vkQueueBindSparse` causes any pipeline stage to access memory, synchronization primitives used in this command effectively only define execution dependencies.

Additional information about fence and semaphore operation is described in the synchronization chapter.

### Valid Usage

- VUID-vkQueueBindSparse-fence-01113
  If `fence` is not `VK_NULL_HANDLE`, `fence` must be uns signaled
- VUID-vkQueueBindSparse-fence-01114
  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
- VUID-vkQueueBindSparse-pSignalSemaphores-01115
Each element of the `pSignalSemaphores` member of each element of `pBindInfo` must be unsignaled when the semaphore signal operation it defines is executed on the device.

- **VUID-vkQueueBindSparse-pWaitSemaphores-01116**
  When a semaphore wait operation referring to a binary semaphore defined by any element of the `pWaitSemaphores` member of any element of `pBindInfo` executes on `queue`, there must be no other queues waiting on the same semaphore.

- **VUID-vkQueueBindSparse-pWaitSemaphores-01117**
  All elements of the `pWaitSemaphores` member of all elements of the `pBindInfo` parameter referring to a binary semaphore must be semaphores that are signaled, or have semaphore signal operations previously submitted for execution.

- **VUID-vkQueueBindSparse-pWaitSemaphores-03245**
  All elements of the `pWaitSemaphores` member of all elements of `pBindInfo` created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_BINARY` must reference a semaphore signal operation that has been submitted for execution and any semaphore signal operations on which it depends (if any) must have also been submitted for execution.

**Valid Usage (Implicit)**

- **VUID-vkQueueBindSparse-queue-parameter**
  `queue` must be a valid `VkQueue` handle.

- **VUID-vkQueueBindSparse-pBindInfo-parameter**
  If `bindInfoCount` is not 0, `pBindInfo` must be a valid pointer to an array of `bindInfoCount` valid `VkBindSparseInfo` structures.

- **VUID-vkQueueBindSparse-fence-parameter**
  If `fence` is not `VK_NULL_HANDLE`, `fence` must be a valid `VkFence` handle.

- **VUID-vkQueueBindSparse-queuetype**
  The `queue` must support sparse binding operations.

- **VUID-vkQueueBindSparse-commonparent**
  Both of `fence`, and `queue` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`.

**Host Synchronization**

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

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Command Type</th>
</tr>
</thead>
</table>

1053
Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST

The `VkBindSparseInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkBindSparseInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t waitSemaphoreCount;
    const VkSemaphore* pWaitSemaphores;
    uint32_t bufferBindCount;
    const VkSparseBufferMemoryBindInfo* pBufferBinds;
    uint32_t imageOpaqueBindCount;
    const VkSparseImageOpaqueMemoryBindInfo* pImageOpaqueBinds;
    uint32_t imageBindCount;
    const VkSparseImageMemoryBindInfo* pImageBinds;
    uint32_t signalSemaphoreCount;
    const VkSemaphore* pSignalSemaphores;
} VkBindSparseInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `waitSemaphoreCount` is the number of semaphores upon which to wait before executing the sparse binding operations for the batch.
- `pWaitSemaphores` is a pointer to an array of semaphores upon which to wait on before the sparse binding operations for this batch begin execution. If semaphores to wait on are provided, they define a semaphore wait operation.
- `bufferBindCount` is the number of sparse buffer bindings to perform in the batch.
- `pBufferBinds` is a pointer to an array of `VkSparseBufferMemoryBindInfo` structures.
- `imageOpaqueBindCount` is the number of opaque sparse image bindings to perform.
- `pImageOpaqueBinds` is a pointer to an array of `VkSparseImageOpaqueMemoryBindInfo` structures, indicating opaque sparse image bindings to perform.
- `imageBindCount` is the number of sparse image bindings to perform.
- `pImageBinds` is a pointer to an array of `VkSparseImageMemoryBindInfo` structures, indicating sparse image bindings to perform.
• `signalSemaphoreCount` is the number of semaphores to be signaled once the sparse binding operations specified by the structure have completed execution.

• `pSignalSemaphores` is a pointer to an array of semaphores which will be signaled when the sparse binding operations for this batch have completed execution. If semaphores to be signaled are provided, they define a semaphore signal operation.

### Valid Usage

**VUID-VkBindSparseInfo-pWaitSemaphores-03246**
If any element of `pWaitSemaphores` or `pSignalSemaphores` was created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` then the `pNext` chain must include a `VkTimelineSemaphoreSubmitInfo` structure.

**VUID-VkBindSparseInfo-pNext-03247**
If the `pNext` chain of this structure includes a `VkTimelineSemaphoreSubmitInfo` structure and any element of `pWaitSemaphores` was created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` then its `waitSemaphoreValueCount` member must equal `waitSemaphoreCount`.

**VUID-VkBindSparseInfo-pNext-03248**
If the `pNext` chain of this structure includes a `VkTimelineSemaphoreSubmitInfo` structure and any element of `pSignalSemaphores` was created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` then its `signalSemaphoreValueCount` member must equal `signalSemaphoreCount`.

**VUID-VkBindSparseInfo-pSignalSemaphores-03249**
For each element of `pSignalSemaphores` created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` the corresponding element of `VkTimelineSemaphoreSubmitInfo::pSignalSemaphoreValues` must have a value greater than the current value of the semaphore when the semaphore signal operation is executed.

**VUID-VkBindSparseInfo-pWaitSemaphores-03250**
For each element of `pWaitSemaphores` created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` the corresponding element of `VkTimelineSemaphoreSubmitInfo::pWaitSemaphoreValues` must have a value which does not differ from the current value of the semaphore or from the value of any outstanding semaphore wait or signal operation on that semaphore by more than `maxTimelineSemaphoreValueDifference`.

**VUID-VkBindSparseInfo-pSignalSemaphores-03251**
For each element of `pSignalSemaphores` created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` the corresponding element of `VkTimelineSemaphoreSubmitInfo::pSignalSemaphoreValues` must have a value which does not differ from the current value of the semaphore or from the value of any outstanding semaphore wait or signal operation on that semaphore by more than `maxTimelineSemaphoreValueDifference`. 
Valid Usage (Implicit)

- VUID-VkBindSparseInfo-sType-sType
  *sType* must be VK_STRUCTURE_TYPE_BIND_SPARSE_INFO

- VUID-VkBindSparseInfo-pNext-pNext
  Each *pNext* member of any structure (including this one) in the *pNext* chain must be either NULL or a pointer to a valid instance of VkDeviceGroupBindSparseInfo or VkTimelineSemaphoreSubmitInfo

- VUID-VkBindSparseInfo-sType-unique
  The *sType* value of each struct in the *pNext* chain must be unique

- VUID-VkBindSparseInfo-pWaitSemaphores-parameter
  If *waitSemaphoreCount* is not 0, *pWaitSemaphores* must be a valid pointer to an array of *waitSemaphoreCount* valid VkSemaphore handles

- VUID-VkBindSparseInfo-pBufferBinds-parameter
  If *bufferBindCount* is not 0, *pBufferBinds* must be a valid pointer to an array of *bufferBindCount* valid VkSparseBufferMemoryBindInfo structures

- VUID-VkBindSparseInfo-pImageOpaqueBinds-parameter
  If *imageOpaqueBindCount* is not 0, *pImageOpaqueBinds* must be a valid pointer to an array of *imageOpaqueBindCount* valid VkSparseImageOpaqueMemoryBindInfo structures

- VUID-VkBindSparseInfo-pImageBinds-parameter
  If *imageBindCount* is not 0, *pImageBinds* must be a valid pointer to an array of *imageBindCount* valid VkSparseImageMemoryBindInfo structures

- VUID-VkBindSparseInfo-pSignalSemaphores-parameter
  If *signalSemaphoreCount* is not 0, *pSignalSemaphores* must be a valid pointer to an array of *signalSemaphoreCount* valid VkSemaphore handles

- VUID-VkBindSparseInfo-commonparent
  Both of the elements of *pSignalSemaphores*, and the elements of *pWaitSemaphores* that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same VkDevice

To specify the values to use when waiting for and signaling semaphores created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_TIMELINE, add a VkTimelineSemaphoreSubmitInfo structure to the *pNext* chain of the VkBindSparseInfo structure.

If the *pNext* chain of VkBindSparseInfo includes a VkDeviceGroupBindSparseInfo structure, then that structure includes device indices specifying which instance of the resources and memory are bound.

The VkDeviceGroupBindSparseInfo structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkDeviceGroupBindSparseInfo {
    VkStructureType sType;
    const void* pNext;
} VkDeviceGroupBindSparseInfo;
```
```c
uint32_t resourceDeviceIndex;
uint32_t memoryDeviceIndex;
} VkDeviceGroupBindSparseInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **resourceDeviceIndex** is a device index indicating which instance of the resource is bound.
- **memoryDeviceIndex** is a device index indicating which instance of the memory the resource instance is bound to.

These device indices apply to all buffer and image memory binds included in the batch pointing to this structure. The semaphore waits and signals for the batch are executed only by the physical device specified by the **resourceDeviceIndex**.

If this structure is not present, **resourceDeviceIndex** and **memoryDeviceIndex** are assumed to be zero.

### Valid Usage

- VUID-VkDeviceGroupBindSparseInfo-resourceDeviceIndex-01118
  ```text
  resourceDeviceIndex and memoryDeviceIndex must both be valid device indices
  ```
- VUID-VkDeviceGroupBindSparseInfo-memoryDeviceIndex-01119
  ```text
  Each memory allocation bound in this batch must have allocated an instance for memoryDeviceIndex
  ```

### Valid Usage (Implicit)

- VUID-VkDeviceGroupBindSparseInfo-sType-sType
  ```text
  sType must be VK_STRUCTURE_TYPE_DEVICE_GROUP_BIND_SPARSE_INFO
  ```
Chapter 30. Private Data

The private data extension provides a way for users to associate arbitrary user defined data with Vulkan objects. This association is accomplished by storing 64-bit unsigned integers of user defined data in private data slots. A private data slot represents a storage allocation for one data item for each child object of the device.

An application can reserve private data slots at device creation. To reserve private data slots, insert a `VkDevicePrivateDataCreateInfo` in the `pNext` chain in `VkDeviceCreateInfo` before device creation. Multiple `VkDevicePrivateDataCreateInfo` structures can be chained together, and the sum of the requested slots will be reserved. This is an exception to the specified valid usage for structure pointer chains. Reserving slots in this manner is not strictly necessary but it may improve performance.

Private data slots are represented by `VkPrivateDataSlot` handles:

```c
// Provided by VK_VERSION_1_3
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPrivateDataSlot)
```

To create a private data slot, call:

```c
// Provided by VK_VERSION_1_3
VkResult vkCreatePrivateDataSlot(
    VkDevice device, 
    const VkPrivateDataSlotCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkPrivateDataSlot* pPrivateDataSlot);
```

- `device` is the logical device associated with the creation of the object(s) holding the private data slot.
- `pCreateInfo` is a pointer to a `VkPrivateDataSlotCreateInfo`
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pPrivateDataSlot` is a pointer to a `VkPrivateDataSlot` handle in which the resulting private data slot is returned

**Valid Usage**

- VUID-vkCreatePrivateDataSlot-privateData-04564
  The `privateData` feature must be enabled

**Valid Usage (Implicit)**

- VUID-vkCreatePrivateDataSlot/device-parameter
device must be a valid `VkDevice` handle
- **VUID-vkCreatePrivateDataSlot-pCreateInfo-parameter**
  - `pCreateInfo` must be a valid pointer to a valid `VkPrivateDataSlotCreateInfo` structure.

- **VUID-vkCreatePrivateDataSlot-pAllocator-parameter**
  - If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure.

- **VUID-vkCreatePrivateDataSlot-pPrivateDataSlot-parameter**
  - `pPrivateDataSlot` must be a valid pointer to a `VkPrivateDataSlot` handle.

## Return Codes

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkPrivateDataSlotCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPrivateDataSlotCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPrivateDataSlotCreateFlags flags;
} VkPrivateDataSlotCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.

## Valid Usage (Implicit)

- **VUID-VkPrivateDataSlotCreateInfo-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_PRIVATE_DATA_SLOT_CREATE_INFO`.

- **VUID-VkPrivateDataSlotCreateInfo-pNext-pNext**
  - `pNext` must be `NULL`.

- **VUID-VkPrivateDataSlotCreateInfo-flags-zerobitmask**
  - `flags` must be `0`.

```c
// Provided by VK_VERSION_1_3
typedef VkFlags VkPrivateDataSlotCreateFlags;
```

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

To destroy a private data slot, call:

```c
// Provided by VK_VERSION_1_3
void vkDestroyPrivateDataSlot(
    VkDevice device,
    VkPrivateDataSlot privateDataSlot,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device associated with the creation of the object(s) holding the private data slot.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `privateDataSlot` is the private data slot to destroy.

### Valid Usage

- **VUID-vkDestroyPrivateDataSlot-privateDataSlot-04062**
  If `VkAllocationCallbacks` were provided when `privateDataSlot` was created, a compatible set of callbacks must be provided here.

- **VUID-vkDestroyPrivateDataSlot-privateDataSlot-04063**
  If no `VkAllocationCallbacks` were provided when `privateDataSlot` was created, `pAllocator` must be `NULL`.

### Valid Usage (Implicit)

- **VUID-vkDestroyPrivateDataSlot-device-parameter**
  `device` must be a valid `VkDevice` handle.

- **VUID-vkDestroyPrivateDataSlot-privateDataSlot-parameter**
  If `privateDataSlot` is not `VK_NULL_HANDLE`, `privateDataSlot` must be a valid `VkPrivateDataSlot` handle.

- **VUID-vkDestroyPrivateDataSlot-pAllocator-parameter**
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure.

- **VUID-vkDestroyPrivateDataSlot-privateDataSlot-parent**
  If `privateDataSlot` is a valid handle, it must have been created, allocated, or retrieved from `device`.

### Host Synchronization

- Host access to `privateDataSlot` must be externally synchronized.
To store user defined data in a slot associated with a Vulkan object, call:

```c
// Provided by VK_VERSION_1_3
VkResult vkSetPrivateData( 
    VkDevice device, 
    VkObjectType objectType, 
    uint64_t objectHandle, 
    VkPrivateDataSlot privateDataSlot, 
    uint64_t data);
```

- `device` is the device that created the object.
- `objectType` is a `VkObjectType` specifying the type of object to associate data with.
- `objectHandle` is a handle to the object to associate data with.
- `privateDataSlot` is a handle to a `VkPrivateDataSlot` specifying location of private data storage.
- `data` is user defined data to associate the object with. This data will be stored at `privateDataSlot`.

### Valid Usage

- VUID-vkSetPrivateData-objectHandle-04016
  - `objectHandle` must be `device` or a child of `device`

- VUID-vkSetPrivateData-objectHandle-04017
  - `objectHandle` must be a valid handle to an object of type `objectType`

### Valid Usage (Implicit)

- VUID-vkSetPrivateData-device-parameter
  - `device` must be a valid `VkDevice` handle

- VUID-vkSetPrivateData-objectType-parameter
  - `objectType` must be a valid `VkObjectType` value

- VUID-vkSetPrivateData-privateDataSlot-parameter
  - `privateDataSlot` must be a valid `VkPrivateDataSlot` handle

- VUID-vkSetPrivateData-privateDataSlot-parent
  - `privateDataSlot` must have been created, allocated, or retrieved from `device`

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
To retrieve user defined data from a slot associated with a Vulkan object, call:

```c
void vkGetPrivateData(
    VkDevice device,
    VkObjectType objectType,
    uint64_t objectHandle,
    VkPrivateDataSlot privateDataSlot,
    uint64_t* pData);
```

- **device** is the device that created the object
- **objectType** is a `VkObjectType` specifying the type of object data is associated with.
- **objectHandle** is a handle to the object data is associated with.
- **privateDataSlot** is a handle to a `VkPrivateDataSlot` specifying location of private data pointer storage.
- **pData** is a pointer to specify where user data is returned. 0 will be written in the absence of a previous call to `vkSetPrivateData` using the object specified by `objectHandle`.

**Note**

Due to platform details on Android, implementations might not be able to reliably return 0 from calls to `vkGetPrivateData` for `VkSwapchainKHR` objects on which `vkSetPrivateData` has not previously been called. This erratum is exclusive to the Android platform and objects of type `VkSwapchainKHR`.

### Valid Usage

- **VUID-vkGetPrivateData-objectType-04018**
  - **objectType** must be `VK_OBJECT_TYPE_DEVICE`, or an object type whose parent is `VkDevice`

### Valid Usage (Implicit)

- **VUID-vkGetPrivateData-device-parameter**
  - **device** must be a valid `VkDevice` handle
- **VUID-vkGetPrivateData-objectType-parameter**
  - **objectType** must be a valid `VkObjectType` value
- **VUID-vkGetPrivateData-privateDataSlot-parameter**
  - **privateDataSlot** must be a valid `VkPrivateDataSlot` handle
- **VUID-vkGetPrivateData-pData-parameter**
  - **pData** must be a valid pointer to a `uint64_t` value
- **VUID-vkGetPrivateData-privateDataSlot-parent**
  - **privateDataSlot** must have been created, allocated, or retrieved from `device`
Chapter 31. Extending Vulkan

New functionality may be added to Vulkan via either new extensions or new versions of the core, or new versions of an extension in some cases.

This chapter describes how Vulkan is versioned, how compatibility is affected between different versions, and compatibility rules that are followed by the Vulkan Working Group.

31.1. Instance and Device Functionality

Commands that enumerate instance properties, or that accept a VkInstance object as a parameter, are considered instance-level functionality. Commands that enumerate physical device properties, or that accept a VkDevice object or any of a device's child objects as a parameter, are considered device-level functionality.

Note

Applications usually interface to Vulkan using a loader that implements only instance-level functionality, passing device-level functionality to implementations of the full Vulkan API on the system. In some circumstances, as these may be implemented independently, it is possible that the loader and device implementations on a given installation will support different versions. To allow for this and call out when it happens, the Vulkan specification enumerates device and instance level functionality separately - they have independent version queries.

Note

Vulkan 1.0 initially specified new physical device enumeration functionality as instance-level, requiring it to be included in an instance extension. As the capabilities of device-level functionality require discovery via physical device enumeration, this led to the situation where many device extensions required an instance extension as well. To alleviate this extra work, VK_KHR_get_physical_device_properties2 (and subsequently Vulkan 1.1) redefined device-level functionality to include physical device enumeration.

31.2. Core Versions

The Vulkan Specification is regularly updated with bug fixes and clarifications. Occasionally new functionality is added to the core and at some point it is expected that there will be a desire to perform a large, breaking change to the API. In order to indicate to developers how and when these changes are made to the specification, and to provide a way to identify each set of changes, the Vulkan API maintains a version number.

31.2.1. Version Numbers

The Vulkan version number comprises four parts indicating the variant, major, minor and patch version of the Vulkan API Specification.
The variant indicates the variant of the Vulkan API supported by the implementation. This is always 0 for the Vulkan API.

Note
A non-zero variant indicates the API is a variant of the Vulkan API and applications will typically need to be modified to run against it. The variant field was a later addition to the version number, added in version 1.2.175 of the Specification. As Vulkan uses variant 0, this change is fully backwards compatible with the previous version number format for Vulkan implementations. New version number macros have been added for this change and the old macros deprecated. For existing applications using the older format and macros, an implementation with non-zero variant will decode as a very high Vulkan version. The high version number should be detectable by applications performing suitable version checking.

The major version indicates a significant change in the API, which will encompass a wholly new version of the specification.

The minor version indicates the incorporation of new functionality into the core specification.

The patch version indicates bug fixes, clarifications, and language improvements have been incorporated into the specification.

Compatibility guarantees made about versions of the API sharing any of the same version numbers are documented in Core Versions

The version number is used in several places in the API. In each such use, the version numbers are packed into a 32-bit integer as follows:

- The variant is a 3-bit integer packed into bits 31-29.
- The major version is a 7-bit integer packed into bits 28-22.
- The minor version number is a 10-bit integer packed into bits 21-12.
- The patch version number is a 12-bit integer packed into bits 11-0.

VK_API_VERSION_VARIANT extracts the API variant number from a packed version number:

```c
// Provided by VK_VERSION_1_0
#define VK_API_VERSION_VARIANT(version) ((uint32_t)(version) >> 29)
```

VK_API_VERSION_MAJOR extracts the API major version number from a packed version number:

```c
// Provided by VK_VERSION_1_0
#define VK_API_VERSION_MAJOR(version) (((uint32_t)(version) >> 22) & 0x7FU)
```

VK_VERSION_MAJOR extracts the API major version number from a packed version number:
VK_API_VERSION_MINOR extracts the API minor version number from a packed version number:

```
// Provided by VK_VERSION_1_0
#define VK_API_VERSION_MINOR(version) (((uint32_t)(version) >> 12) & 0x3FFU)
```

VK_VERSION_MINOR extracts the API minor version number from a packed version number:

```
// Provided by VK_VERSION_1_0
#define VK_VERSION_MINOR(version) (((uint32_t)(version) >> 12) & 0x3FFU)
```

VK_API_VERSION_PATCH extracts the API patch version number from a packed version number:

```
// Provided by VK_VERSION_1_0
#define VK_API_VERSION_PATCH(version) ((uint32_t)(version) & 0xFFFU)
```

VK_VERSION_PATCH extracts the API patch version number from a packed version number:

```
// Provided by VK_VERSION_1_0
#define VK_VERSION_PATCH(version) ((uint32_t)(version) & 0xFFFU)
```

VK_MAKE_API_VERSION constructs an API version number.

```
// Provided by VK_VERSION_1_0
#define VK_MAKE_API_VERSION(variant, major, minor, patch) \
    (((uint32_t)(variant)) << 29) | (((uint32_t)(major)) << 22) | \
    (((uint32_t)(minor)) << 12) | ((uint32_t)(patch))
```

- `variant` is the variant number.
- `major` is the major version number.
- `minor` is the minor version number.
- `patch` is the patch version number.

VK_MAKE_VERSION constructs an API version number.

```
// Provided by VK_VERSION_1_0
#define VK_MAKE_VERSION(variant, major, minor, patch) \
    (((uint32_t)(variant)) << 29) | (((uint32_t)(major)) << 22) | \
    (((uint32_t)(minor)) << 12) | ((uint32_t)(patch))
```
#define VK_MAKE_VERSION(major, minor, patch) 
 (((uint32_t)(major)) << 22) | (((uint32_t)(minor)) << 12) | ((uint32_t)(patch))

- **major** is the major version number.
- **minor** is the minor version number.
- **patch** is the patch version number.

**VK_API_VERSION_1_0** returns the API version number for Vulkan 1.0.0.

```c
// Provided by VK_VERSION_1_0
// Vulkan 1.0 version number
#define VK_API_VERSION_1_0 VK_MAKE_API_VERSION(0, 1, 0, 0) // Patch version should always be set to 0
```

**VK_API_VERSION_1_1** returns the API version number for Vulkan 1.1.0.

```c
// Provided by VK_VERSION_1_1
// Vulkan 1.1 version number
#define VK_API_VERSION_1_1 VK_MAKE_API_VERSION(0, 1, 1, 0) // Patch version should always be set to 0
```

**VK_API_VERSION_1_2** returns the API version number for Vulkan 1.2.0.

```c
// Provided by VK_VERSION_1_2
// Vulkan 1.2 version number
#define VK_API_VERSION_1_2 VK_MAKE_API_VERSION(0, 1, 2, 0) // Patch version should always be set to 0
```

**VK_API_VERSION_1_3** returns the API version number for Vulkan 1.3.0.

```c
// Provided by VK_VERSION_1_3
// Vulkan 1.3 version number
#define VK_API_VERSION_1_3 VK_MAKE_API_VERSION(0, 1, 3, 0) // Patch version should always be set to 0
```

### 31.2.2. Querying Version Support

The version of instance-level functionality can be queried by calling `vkEnumerateInstanceVersion`.

The version of device-level functionality can be queried by calling `vkGetPhysicalDeviceProperties` or `vkGetPhysicalDeviceProperties2`, and is returned in `VkPhysicalDeviceProperties::apiVersion`, encoded as described in Version Numbers.
31.3. Layers

When a layer is enabled, it inserts itself into the call chain for Vulkan commands the layer is interested in. Layers can be used for a variety of tasks that extend the base behavior of Vulkan beyond what is required by the specification - such as call logging, tracing, validation, or providing additional extensions.

**Note**

For example, an implementation is not expected to check that the value of enums used by the application fall within allowed ranges. Instead, a validation layer would do those checks and flag issues. This avoids a performance penalty during production use of the application because those layers would not be enabled in production.

**Note**

Vulkan layers may wrap object handles (i.e. return a different handle value to the application than that generated by the implementation). This is generally discouraged, as it increases the probability of incompatibilities with new extensions. The validation layers wrap handles in order to track the proper use and destruction of each object. See the “Architecture of the Vulkan Loader Interfaces” document for additional information.

To query the available layers, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkEnumerateInstanceLayerProperties(
    uint32_t* pPropertyCount,
    VkLayerProperties* pProperties);
```

- `pPropertyCount` is a pointer to an integer related to the number of layer properties available or queried, as described below.
- `pProperties` is either `NULL` or a pointer to an array of `VkLayerProperties` structures.

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

The list of available layers may change at any time due to actions outside of the Vulkan implementation, so two calls to `vkEnumerateInstanceLayerProperties` with the same parameters may return different results, or retrieve different `pPropertyCount` values or `pProperties` contents. Once an instance has been created, the layers enabled for that instance will continue to be enabled and valid for the lifetime of that instance, even if some of them become unavailable for future instances.
Valid Usage (Implicit)

- VUID-vkEnumerateInstanceLayerProperties-pPropertyCount-parameter
  
  `pPropertyCount` must be a valid pointer to a `uint32_t` value

- VUID-vkEnumerateInstanceLayerProperties-pProperties-parameter
  
  If the value referenced by `pPropertyCount` is not 0, and `pProperties` is not NULL, `pProperties` must be a valid pointer to an array of `pPropertyCount` `VkLayerProperties` structures.

Return Codes

Success

- VK_SUCCESS
- VK_INCOMPLETE

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkLayerProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkLayerProperties {
    char layerName[VK_MAX_EXTENSION_NAME_SIZE];
    uint32_t specVersion;
    uint32_t implementationVersion;
    char description[VK_MAX_DESCRIPTION_SIZE];
} VkLayerProperties;
```

- `layerName` is an array of `VK_MAX_EXTENSION_NAME_SIZE` `char` containing a null-terminated UTF-8 string which is the name of the layer. Use this name in the `pEnabledLayerNames` array passed in the `VkInstanceCreateInfo` structure to enable this layer for an instance.

- `specVersion` is the Vulkan version the layer was written to, encoded as described in Version Numbers.

- `implementationVersion` is the version of this layer. It is an integer, increasing with backward compatible changes.

- `description` is an array of `VK_MAX_DESCRIPTION_SIZE` `char` containing a null-terminated UTF-8 string which provides additional details that can be used by the application to identify the layer.

`VK_MAX_EXTENSION_NAME_SIZE` is the length in `char` values of an array containing a layer or extension name string, as returned in `VkLayerProperties::layerName`, `VkExtensionProperties::extensionName`, and other queries.


![Image](image_url)

**VK_MAX_DESCRIPTION_SIZE** is the length in **char** values of an array containing a string with additional descriptive information about a query, as returned in **VkLayerProperties::description** and other queries.

To enable a layer, the name of the layer **should** be added to the **ppEnabledLayerNames** member of **VkInstanceCreateInfo** when creating a **VkInstance**.

Loader implementations **may** provide mechanisms outside the Vulkan API for enabling specific layers. Layers enabled through such a mechanism are **implicitly enabled**, while layers enabled by including the layer name in the **ppEnabledLayerNames** member of **VkInstanceCreateInfo** are **explicitly enabled**. Implicitly enabled layers are loaded before explicitly enabled layers, such that implicitly enabled layers are closer to the application, and explicitly enabled layers are closer to the driver. Except where otherwise specified, implicitly enabled and explicitly enabled layers differ only in the way they are enabled, and the order in which they are loaded. Explicitly enabling a layer that is implicitly enabled results in this layer being loaded as an implicitly enabled layer; it has no additional effect.

### 31.3.1. Device Layer Deprecation

Previous versions of this specification distinguished between instance and device layers. Instance layers were only able to intercept commands that operate on **VkInstance** and **VkPhysicalDevice**, except they were not able to intercept **vkCreateDevice**. Device layers were enabled for individual devices when they were created, and could only intercept commands operating on that device or its child objects.

Device-only layers are now deprecated, and this specification no longer distinguishes between instance and device layers. Layers are enabled during instance creation, and are able to intercept all commands operating on that instance or any of its child objects. At the time of deprecation there were no known device-only layers and no compelling reason to create one.

In order to maintain compatibility with implementations released prior to device-layer deprecation, applications **should** still enumerate and enable device layers. The behavior of **vkEnumerateDeviceLayerProperties** and valid usage of the **ppEnabledLayerNames** member of **VkDeviceCreateInfo** maximizes compatibility with applications written to work with the previous requirements.

To enumerate device layers, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkEnumerateDeviceLayerProperties(  
    VkPhysicalDevice physicalDevice,  
    uint32_t* pPropertyCount,  
    uint32_t* pLayerCount);  
```
• `physicalDevice` is the physical device that will be queried.

• `pPropertyCount` is a pointer to an integer related to the number of layer properties available or queried.

• `pProperties` is either `NULL` or a pointer to an array of `VkLayerProperties` structures.

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

The list of layers enumerated by `vkEnumerateDeviceLayerProperties` must be exactly the sequence of layers enabled for the instance. The members of `VkLayerProperties` for each enumerated layer must be the same as the properties when the layer was enumerated by `vkEnumerateInstanceLayerProperties`.

```markdown
Note
Due to platform details on Android, `vkEnumerateDeviceLayerProperties` may be called with `physicalDevice` equal to `NULL` during layer discovery. This behaviour will only be observed by layer implementations, and not the underlying Vulkan driver.
```

### Valid Usage (Implicit)

- VUID-vkEnumerateDeviceLayerProperties-physicalDevice-parameter `physicalDevice` must be a valid `VkPhysicalDevice` handle

- VUID-vkEnumerateDeviceLayerProperties-pPropertyCount-parameter `pPropertyCount` must be a valid pointer to a `uint32_t` value

- VUID-vkEnumerateDeviceLayerProperties-pProperties-parameter If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` must be a valid pointer to an array of `pPropertyCount` `VkLayerProperties` structures

### Return Codes

#### Success

- `VK_SUCCESS`

- `VK_INCOMPLETE`

#### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`

- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
The `ppEnabledLayerNames` and `enabledLayerCount` members of `VkDeviceCreateInfo` are deprecated and their values **must** be ignored by implementations. However, for compatibility, only an empty list of layers or a list that exactly matches the sequence enabled at instance creation time are valid, and validation layers **should** issue diagnostics for other cases.

Regardless of the enabled layer list provided in `VkDeviceCreateInfo`, the sequence of layers active for a device will be exactly the sequence of layers enabled when the parent instance was created.

### 31.4. Extensions

Extensions **may** define new Vulkan commands, structures, and enumerants. For compilation purposes, the interfaces defined by registered extensions, including new structures and enumerants as well as function pointer types for new commands, are defined in the Khronos-supplied `vulkan_core.h` together with the core API. However, commands defined by extensions **may** not be available for static linking - in which case function pointers to these commands **should** be queried at runtime as described in **Command Function Pointers**. Extensions **may** be provided by layers as well as by a Vulkan implementation.

Because extensions **may** extend or change the behavior of the Vulkan API, extension authors **should** add support for their extensions to the Khronos validation layers. This is especially important for new commands whose parameters have been wrapped by the validation layers. See the “Architecture of the Vulkan Loader Interfaces” document for additional information.

**Note**

To enable an instance extension, the name of the extension **can** be added to the `ppEnabledExtensionNames` member of `VkInstanceCreateInfo` when creating a `VkInstance`.

To enable a device extension, the name of the extension **can** be added to the `ppEnabledExtensionNames` member of `VkDeviceCreateInfo` when creating a `VkDevice`.

Physical-Device-Level functionality does not have any enabling mechanism and **can** be used as long as the `VkPhysicalDevice` supports the device extension as determined by `vkEnumerateDeviceExtensionProperties`.

Enabling an extension (with no further use of that extension) does not change the behavior of functionality exposed by the core Vulkan API or any other extension, other than making valid the use of the commands, enums and structures defined by that extension.

Valid Usage sections for individual commands and structures do not currently contain which extensions have to be enabled in order to make their use valid, although they might do so in the future. It is defined only in the **Valid Usage for Extensions** section.

### 31.4.1. Instance Extensions

Instance extensions add new **instance-level functionality** to the API, outside of the core
specification.

To query the available instance extensions, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkEnumerateInstanceExtensionProperties(
    const char* pLayerName,
    uint32_t* pPropertyCount,
    VkExtensionProperties* pProperties);
```

- `pLayerName` is either `NULL` or a pointer to a null-terminated UTF-8 string naming the layer to retrieve extensions from.
- `pPropertyCount` is a pointer to an integer related to the number of extension properties available or queried, as described below.
- `pProperties` is either `NULL` or a pointer to an array of `VkExtensionProperties` structures.

When `pLayerName` parameter is `NULL`, only extensions provided by the Vulkan implementation or by implicitly enabled layers are returned. When `pLayerName` is the name of a layer, the instance extensions provided by that layer are returned.

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

Because the list of available layers may change externally between calls to `vkEnumerateInstanceExtensionProperties`, two calls may retrieve different results if a `pLayerName` is available in one call but not in another. The extensions supported by a layer may also change between two calls, e.g. if the layer implementation is replaced by a different version between those calls.

Implementations must not advertise any pair of extensions that cannot be enabled together due to behavioral differences, or any extension that cannot be enabled against the advertised version.

---

### Valid Usage (Implicit)

- **VUID-vkEnumerateInstanceExtensionProperties-pLayerName-parameter**
  If `pLayerName` is not `NULL`, `pLayerName` must be a null-terminated UTF-8 string.

- **VUID-vkEnumerateInstanceExtensionProperties-pPropertyCount-parameter**
  `pPropertyCount` must be a valid pointer to a `uint32_t` value.

- **VUID-vkEnumerateInstanceExtensionProperties-pProperties-parameter**
  If the value referenced by `pPropertyCount` is not 0, and `pProperties` is not `NULL`, `pProperties` must be a valid pointer to an array of `pPropertyCount` `VkExtensionProperties` structures.
31.4.2. Device Extensions

Device extensions add new device-level functionality to the API, outside of the core specification. To query the extensions available to a given physical device, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkEnumerateDeviceExtensionProperties(
    VkPhysicalDevice physicalDevice,
    const char* pLayerName,
    uint32_t* pPropertyCount,
    VkExtensionProperties* pProperties);
```

- `physicalDevice` is the physical device that will be queried.
- `pLayerName` is either NULL or a pointer to a null-terminated UTF-8 string naming the layer to retrieve extensions from.
- `pPropertyCount` is a pointer to an integer related to the number of extension properties available or queried, and is treated in the same fashion as the `vkEnumerateInstanceExtensionProperties`::`pPropertyCount` parameter.
- `pProperties` is either NULL or a pointer to an array of `VkExtensionProperties` structures.

When `pLayerName` parameter is NULL, only extensions provided by the Vulkan implementation or by implicitly enabled layers are returned. When `pLayerName` is the name of a layer, the device extensions provided by that layer are returned.

Implementations must not advertise any pair of extensions that cannot be enabled together due to behavioral differences, or any extension that cannot be enabled against the advertised version.

Implementations claiming support for the Roadmap 2022 profile must advertise the `VK_KHR_global_priority` extension in `pProperties`.

**Note**

Due to platform details on Android, `vkEnumerateDeviceExtensionProperties` may be called with `physicalDevice` equal to NULL during layer discovery. This behaviour...
will only be observed by layer implementations, and not the underlying Vulkan driver.

**Valid Usage (Implicit)**

- VUID-vkEnumerateDeviceExtensionProperties-physicalDevice-parameter
  `physicalDevice` must be a valid `VkPhysicalDevice` handle

- VUID-vkEnumerateDeviceExtensionProperties-pLayerName-parameter
  If `pLayerName` is not NULL, `pLayerName` must be a null-terminated UTF-8 string

- VUID-vkEnumerateDeviceExtensionProperties-pPropertyCount-parameter
  `pPropertyCount` must be a valid pointer to a `uint32_t` value

- VUID-vkEnumerateDeviceExtensionProperties-pProperties-parameter
  If the value referenced by `pPropertyCount` is not 0, and `pProperties` is not NULL, `pProperties` must be a valid pointer to an array of `pPropertyCount` `VkExtensionProperties` structures

**Return Codes**

**Success**
- `VK_SUCCESS`
- `VK_INCOMPLETE`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_LAYER_NOT_PRESENT`

The `VkExtensionProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkExtensionProperties {
    char extensionName[VK_MAX_EXTENSION_NAME_SIZE];
    uint32_t specVersion;
} VkExtensionProperties;
```

- `extensionName` is an array of `VK_MAX_EXTENSION_NAME_SIZE` `char` containing a null-terminated UTF-8 string which is the name of the extension.
- `specVersion` is the version of this extension. It is an integer, incremented with backward compatible changes.

**31.5. Extension Dependencies**

Some extensions are dependent on other extensions, or on specific core API versions, to function.
To enable extensions with dependencies, any required extensions must also be enabled through the same API mechanisms when creating an instance with vkCreateInstance or a device with vkCreateDevice. Each extension which has such dependencies documents them in the appendix summarizing that extension.

If an extension is supported (as queried by vkEnumerateInstanceExtensionProperties or vkEnumerateDeviceExtensionProperties), then required extensions of that extension must also be supported for the same instance or physical device.

Any device extension that has an instance extension dependency that is not enabled by vkCreateInstance is considered to be unsupported, hence it must not be returned by vkEnumerateDeviceExtensionProperties for any VkPhysicalDevice child of the instance. Instance extensions do not have dependencies on device extensions.

If a required extension has been promoted to another extension or to a core API version, then as a general rule, the dependency is also satisfied by the promoted extension or core version. This will be true so long as any features required by the original extension are also required or enabled by the promoted extension or core version. However, in some cases an extension is promoted while making some of its features optional in the promoted extension or core version. In this case, the dependency may not be satisfied. The only way to be certain is to look at the descriptions of the original dependency and the promoted version in the Layers & Extensions and Core Revisions appendices.

Note

There is metadata in vk.xml describing some aspects of promotion, especially requires, promotedto and deprecatedby attributes of <extension> tags. However, the metadata does not yet fully describe this scenario. In the future, we may extend the XML schema to describe the full set of extensions and versions satisfying a dependency.

31.6. Compatibility Guarantees (Informative)

This section is marked as informal as there is no binding responsibility on implementations of the Vulkan API - these guarantees are however a contract between the Vulkan Working Group and developers using this Specification.

31.6.1. Core Versions

Each of the major, minor, and patch versions of the Vulkan specification provide different compatibility guarantees.

Patch Versions

A difference in the patch version indicates that a set of bug fixes or clarifications have been made to the Specification. Informative enums returned by Vulkan commands that will not affect the runtime behavior of a valid application may be added in a patch version (e.g. VkVendorId).

The specification’s patch version is strictly increasing for a given major version of the specification;
any change to a specification as described above will result in the patch version being increased by 1. Patch versions are applied to all minor versions, even if a given minor version is not affected by the provoking change.

Specifications with different patch versions but the same major and minor version are *fully compatible* with each other - such that a valid application written against one will work with an implementation of another.

Note

If a patch version includes a bug fix or clarification that could have a significant impact on developer expectations, these will be highlighted in the change log. Generally the Vulkan Working Group tries to avoid these kinds of changes, instead fixing them in either an extension or core version.

Minor Versions

Changes in the minor version of the specification indicate that new functionality has been added to the core specification. This will usually include new interfaces in the header, and may also include behavior changes and bug fixes. Core functionality may be deprecated in a minor version, but will not be obsoleted or removed.

The specification's minor version is strictly increasing for a given major version of the specification; any change to a specification as described above will result in the minor version being increased by 1. Changes that can be accommodated in a patch version will not increase the minor version.

Specifications with a lower minor version are *backwards compatible* with an implementation of a specification with a higher minor version for core functionality and extensions issued with the KHR vendor tag. Vendor and multi-vendor extensions are not guaranteed to remain functional across minor versions, though in general they are with few exceptions - see Obsoletion for more information.

Major Versions

A difference in the major version of specifications indicates a large set of changes which will likely include interface changes, behavioral changes, removal of deprecated functionality, and the modification, addition, or replacement of other functionality.

The specification's major version is monotonically increasing; any change to the specification as described above will result in the major version being increased. Changes that can be accommodated in a patch or minor version will not increase the major version.

The Vulkan Working Group intends to only issue a new major version of the Specification in order to realise significant improvements to the Vulkan API that will necessarily require breaking compatibility.

A new major version will likely include a wholly new version of the specification to be issued - which could include an overhaul of the versioning semantics for the minor and patch versions. The patch and minor versions of a specification are therefore not meaningful across major versions. If a
major version of the specification includes similar versioning semantics, it is expected that the patch and the minor version will be reset to 0 for that major version.

### 31.6.2. Extensions

A KHR extension **must** be able to be enabled alongside any other KHR extension, and for any minor or patch version of the core Specification beyond the minimum version it requires. A multi-vendor extension **should** be able to be enabled alongside any KHR extension or other multi-vendor extension, and for any minor or patch version of the core Specification beyond the minimum version it requires. A vendor extension **should** be able to be enabled alongside any KHR extension, multi-vendor extension, or other vendor extension from the same vendor, and for any minor or patch version of the core Specification beyond the minimum version it requires. A vendor extension **may** be able to be enabled alongside vendor extensions from another vendor.

The one other exception to this is if a vendor or multi-vendor extension is made obsolete by either a core version or another extension, which will be highlighted in the extension appendix.

#### Promotion

Extensions, or features of an extension, **may** be promoted to a new core version of the API, or a newer extension which an equal or greater number of implementors are in favour of.

When extension functionality is promoted, minor changes **may** be introduced, limited to the following:

- Naming
- Non-intrusive parameters changes
- **Feature advertisement/enablement**
- Combining structure parameters into larger structures
- Author ID suffixes changed or removed

**Note**

If extension functionality is promoted, there is no guarantee of direct compatibility, however it should require little effort to port code from the original feature to the promoted one.

The Vulkan Working Group endeavours to ensure that larger changes are marked as either deprecated or obsoleted as appropriate, and can do so retroactively if necessary.

Extensions that are promoted are listed as being promoted in their extension appendices, with reference to where they were promoted to.

When an extension is promoted, any backwards compatibility aliases which exist in the extension will **not** be promoted.

**Note**
As a hypothetical example, if the `VK_KHR_surface` extension were promoted to part of a future core version, the `VK_COLOR_SPACE_SRGB_NONLINEAR_KHR` token defined by that extension would be promoted to `VK_COLOR_SPACE_SRGB_NONLINEAR`. However, the `VK_COLORSPACE_SRGB_NONLINEAR_KHR` token aliases `VK_COLOR_SPACE_SRGB_NONLINEAR_KHR`. The `VK_COLORSPACE_SRGB_NONLINEAR_KHR` would not be promoted, because it is a backwards compatibility alias that exists only due to a naming mistake when the extension was initially published.

**Deprecation**

Extensions **may** be marked as deprecated when the intended use cases either become irrelevant or can be solved in other ways. Generally, a new feature will become available to solve the use case in another extension or core version of the API, but it is not guaranteed.

*Note*

Features that are intended to replace deprecated functionality have no guarantees of compatibility, and applications may require drastic modification in order to make use of the new features.

Extensions that are deprecated are listed as being deprecated in their extension appendices, with an explanation of the deprecation and any features that are relevant.

**Obsoletion**

Occasionally, an extension will be marked as obsolete if a new version of the core API or a new extension is fundamentally incompatible with it. An obsoleted extension **must** not be used with the extension or core version that obsoleted it.

Extensions that are obsoleted are listed as being obsoleted in their extension appendices, with reference to what they were obsoleted by.

**Aliases**

When an extension is promoted or deprecated by a newer feature, some or all of its functionality **may** be replicated into the newer feature. Rather than duplication of all the documentation and definitions, the specification instead identifies the identical commands and types as *aliases* of one another. Each alias is mentioned together with the definition it aliases, with the older aliases marked as “equivalents”. Each alias of the same command has identical behavior, and each alias of the same type has identical meaning - they can be used interchangeably in an application with no compatibility issues.

*Note*

For promoted types, the aliased extension type is semantically identical to the new core type. The C99 headers simply *typedef* the older aliases to the promoted types.

For promoted command aliases, however, there are two separate entry point definitions, due to the fact that the C99 ABI has no way to alias command definitions without resorting to macros. Calling via either entry point definition
will produce identical behavior within the bounds of the specification, and should still invoke the same entry point in the implementation. Debug tools may use separate entry points with different debug behavior; to write the appropriate command name to an output log, for instance.

**Special Use Extensions**

Some extensions exist only to support a specific purpose or specific class of application. These are referred to as “special use extensions”. Use of these extensions in applications not meeting the special use criteria is not recommended.

Special use cases are restricted, and only those defined below are used to describe extensions:

*Table 30. Extension Special Use Cases*

<table>
<thead>
<tr>
<th>Special Use</th>
<th>XML Tag</th>
<th>Full Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAD support</td>
<td>cadsupport</td>
<td>Extension is intended to support specialized functionality used by CAD/CAM apps.</td>
</tr>
<tr>
<td>D3D support</td>
<td>d3demulation</td>
<td>Extension is intended to support D3D emulation layers, and apps ported from D3D.</td>
</tr>
<tr>
<td>Developer tools</td>
<td>devtools</td>
<td>Extension is intended to support developer tools such as capture-replay libraries.</td>
</tr>
<tr>
<td>Debugging tools</td>
<td>debugging</td>
<td>Extension is intended for use by apps when debugging.</td>
</tr>
<tr>
<td>OpenGL / ES support</td>
<td>glemulation</td>
<td>Extension is intended to support OpenGL and/or OpenGL ES emulation layers, and apps ported from those APIs, by adding functionality specific to those APIs.</td>
</tr>
</tbody>
</table>

Special use extensions are identified in the metadata for each such extension in the **Layers & Extensions** appendix, using the name in the “Special Use” column above.

Special use extensions are also identified in `vk.xml` with the short name in “XML Tag” column above, as described in the “API Extensions (extension tag)” section of the registry schema documentation.
Chapter 32. Features

Features describe functionality which is not supported on all implementations. Features are properties of the physical device. Features are optional, and must be explicitly enabled before use. Support for features is reported and enabled on a per-feature basis.

Note
Features are reported via the basic VkPhysicalDeviceFeatures structure, as well as the extensible structure VkPhysicalDeviceFeatures2, which was added in the VK_KHR_get_physical_device_properties2 extension and included in Vulkan 1.1. When new features are added in future Vulkan versions or extensions, each extension should introduce one new feature structure, if needed. This structure can be added to the pNext chain of the VkPhysicalDeviceFeatures2 structure.

For convenience, new core versions of Vulkan may introduce new unified feature structures for features promoted from extensions. At the same time, the extension’s original feature structure (if any) is also promoted to the core API, and is an alias of the extension’s structure. This results in multiple names for the same feature: in the original extension’s feature structure and the promoted structure alias, in the unified feature structure. When a feature was implicitly supported and enabled in the extension, but an explicit name was added during promotion, then the extension itself acts as an alias for the feature as listed in the table below.

All aliases of the same feature in the core API must be reported consistently: either all must be reported as supported, or none of them. When a promoted extension is available, any corresponding feature aliases must be supported.

Table 31. Extension Feature Aliases

<table>
<thead>
<tr>
<th>Extension</th>
<th>Feature(s)</th>
</tr>
</thead>
</table>

To query supported features, call:

```c
// Provided by VK_VERSION_1_0
void vkGetPhysicalDeviceFeatures(
    VkPhysicalDevice physicalDevice,
    VkPhysicalDeviceFeatures* pFeatures);
```

- `physicalDevice` is the physical device from which to query the supported features.
- `pFeatures` is a pointer to a VkPhysicalDeviceFeatures structure in which the physical device features are returned. For each feature, a value of VK_TRUE specifies that the feature is supported on this physical device, and VK_FALSE specifies that the feature is not supported.

Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceFeatures-physicalDevice-parameter
  `physicalDevice` must be a valid VkPhysicalDevice handle
Fine-grained features used by a logical device **must** be enabled at VkDevice creation time. If a feature is enabled that the physical device does not support, VkDevice creation will fail and return VK_ERROR_FEATURE_NOT_PRESENT.

The fine-grained features are enabled by passing a pointer to the VkPhysicalDeviceFeatures structure via the pEnabledFeatures member of the VkDeviceCreateInfo structure that is passed into the vkCreateDevice call. If a member of pEnabledFeatures is set to VK_TRUE or VK_FALSE, then the device will be created with the indicated feature enabled or disabled, respectively. Features can also be enabled by using the VkPhysicalDeviceFeatures2 structure.

If an application wishes to enable all features supported by a device, it can simply pass in the VkPhysicalDeviceFeatures structure that was previously returned by vkGetPhysicalDeviceFeatures. To disable an individual feature, the application can set the desired member to VK_FALSE in the same structure. Setting pEnabledFeatures to NULL and not including a VkPhysicalDeviceFeatures2 in the pNext chain of VkDeviceCreateInfo is equivalent to setting all members of the structure to VK_FALSE.

**Note**

Some features, such as robustBufferAccess, **may** incur a runtime performance cost. Application writers **should** carefully consider the implications of enabling all supported features.

To query supported features defined by the core or extensions, call:

```c
// Provided by VK_VERSION_1_1
void vkGetPhysicalDeviceFeatures2(
    VkPhysicalDevice physicalDevice,
    VkPhysicalDeviceFeatures2* pFeatures);
```

- **physicalDevice** is the physical device from which to query the supported features.
- **pFeatures** is a pointer to a VkPhysicalDeviceFeatures2 structure in which the physical device features are returned.

Each structure in pFeatures and its pNext chain contains members corresponding to fine-grained features. vkGetPhysicalDeviceFeatures2 writes each member to a boolean value indicating whether that feature is supported.

### Valid Usage (Implicit)

- **VUID-vkGetPhysicalDeviceFeatures2-physicalDevice-parameter**
  - `physicalDevice` **must** be a valid VkPhysicalDevice handle
- **VUID-vkGetPhysicalDeviceFeatures2-pFeatures-parameter**
  - `pFeatures` **must** be a valid pointer to a VkPhysicalDeviceFeatures2 structure
The `VkPhysicalDeviceFeatures2` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceFeatures2 {
    VkStructureType sType;
    void* pNext;
    VkPhysicalDeviceFeatures features;
} VkPhysicalDeviceFeatures2;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `features` is a `VkPhysicalDeviceFeatures` structure describing the fine-grained features of the Vulkan 1.0 API.

The `pNext` chain of this structure is used to extend the structure with features defined by extensions. This structure can be used in `vkGetPhysicalDeviceFeatures2` or can be included in the `pNext` chain of a `VkDeviceCreateInfo` structure, in which case it controls which features are enabled on the device in lieu of `pEnabledFeatures`.

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceFeatures2-sType-sType`  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2`

The `VkPhysicalDeviceFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_0
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;
} VkPhysicalDeviceFeatures;
```
This structure describes the following features:

- **robustBufferAccess** specifies that accesses to buffers are bounds-checked against the range of the buffer descriptor (as determined by VkDescriptorBufferInfo::range, VkBufferViewCreateInfo::range, or the size of the buffer). Out of bounds accesses must not cause application termination, and the effects of shader loads, stores, and atomics must conform to an implementation-dependent behavior as described below.

  A buffer access is considered to be out of bounds if any of the following are true:

  - The pointer was formed by OpImageTexelPointer and the coordinate is less than zero or greater than or equal to the number of whole elements in the bound range.
  - The pointer was not formed by OpImageTexelPointer and the object pointed to is not
wholly contained within the bound range. This includes accesses performed via variable pointers where the buffer descriptor being accessed cannot be statically determined. Uninitialized pointers and pointers equal to OpConstantNull are treated as pointing to a zero-sized object, so all accesses through such pointers are considered to be out of bounds. Buffer accesses through buffer device addresses are not bounds-checked.

**Note**

If a SPIR-V OpLoad instruction loads a structure and the tail end of the structure is out of bounds, then all members of the structure are considered out of bounds even if the members at the end are not statically used.

- If any buffer access is determined to be out of bounds, then any other access of the same type (load, store, or atomic) to the same buffer that accesses an address less than 16 bytes away from the out of bounds address may also be considered out of bounds.
- If the access is a load that reads from the same memory locations as a prior store in the same shader invocation, with no other intervening accesses to the same memory locations in that shader invocation, then the result of the load may be the value stored by the store instruction, even if the access is out of bounds. If the load is Volatile, then an out of bounds load must return the appropriate out of bounds value.

- **Out-of-bounds buffer loads will return any of the following values:**
  - Values from anywhere within the memory range(s) bound to the buffer (possibly including bytes of memory past the end of the buffer, up to the end of the bound range).
  - Zero values, or (0,0,0,x) vectors for vector reads where x is a valid value represented in the type of the vector components and may be any of:
    - 0, 1, or the maximum representable positive integer value, for signed or unsigned integer components
    - 0.0 or 1.0, for floating-point components
  - Out-of-bounds writes may modify values within the memory range(s) bound to the buffer, but must not modify any other memory.
  - Out-of-bounds atomics may modify values within the memory range(s) bound to the buffer, but must not modify any other memory, and return an undefined value.
  - Vertex input attributes are considered out of bounds if the offset of the attribute in the bound vertex buffer range plus the size of the attribute is greater than either:
    - vertexBufferRangeSize, if bindingStride == 0; or
    - (vertexBufferRangeSize - (vertexBufferRangeSize % bindingStride))

where vertexBufferRangeSize is the byte size of the memory range bound to the vertex buffer binding and bindingStride is the byte stride of the corresponding vertex input binding. Further, if any vertex input attribute using a specific vertex input binding is out of bounds, then all vertex input attributes using that vertex input binding for that vertex shader invocation are considered out of bounds.

- If a vertex input attribute is out of bounds, it will be assigned one of the following values:
  - zero values
  - (0,0,0,x) vectors for vector reads where x is a valid value represented in the type of the vector components
values:

- Values from anywhere within the memory range(s) bound to the buffer, converted according to the format of the attribute.
- Zero values, format converted according to the format of the attribute.
- Zero values, or (0,0,0,x) vectors, as described above.

If robustBufferAccess is not enabled, applications must not perform out of bounds accesses.

- fullDrawIndexUint32 specifies the full 32-bit range of indices is supported for indexed draw calls when using a VkIndexType of VK_INDEX_TYPE_UINT32. maxDrawIndexedIndexValue is the maximum index value that may be used (aside from the primitive restart index, which is always 2^31-1 when the VkIndexType is VK_INDEX_TYPE_UINT32). If this feature is supported, maxDrawIndexedIndexValue must be 2^31-1; otherwise it must be no smaller than 2^24-1. See maxDrawIndexedIndexValue.

- imageCubeArray specifies whether image views with a VkImageViewType of VK_IMAGE_VIEW_TYPE_CUBE_ARRAY can be created, and that the corresponding SampledCubeArray and ImageCubeArray SPIR-V capabilities can be used in shader code.

- independentBlend specifies whether the VkPipelineColorBlendAttachmentState settings are controlled independently per-attachment. If this feature is not enabled, the VkPipelineColorBlendAttachmentState settings for all color attachments must be identical. Otherwise, a different VkPipelineColorBlendAttachmentState can be provided for each bound color attachment.

- geometryShader specifies whether geometry shaders are supported. If this feature is not enabled, the VK_SHADER_STAGE_GEOMETRY_BIT and VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT enum values must not be used. This also specifies whether shader modules can declare the Geometry capability.

- tessellationShader specifies whether tessellation control and evaluation shaders are supported. If this feature is not enabled, the VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT, VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT, VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT, VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT, and VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO enum values must not be used. This also specifies whether shader modules can declare the Tessellation capability.

- sampleRateShading specifies whether Sample Shading and multisample interpolation are supported. If this feature is not enabled, the sampleShadingEnable member of the VkPipelineMultisampleStateCreateInfo structure must be set to VK_FALSE and the minSampleShading member is ignored. This also specifies whether shader modules can declare the SampleRateShading capability.

- dualSrcBlend specifies whether blend operations which take two sources are supported. If this feature is not enabled, the VK_BLEND_FACTOR_SRC1_COLOR, VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR, VK_BLEND_FACTOR_SRC1_ALPHA, and VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA enum values must not be used as source or destination blending factors. See Dual-Source Blending.

- logicOp specifies whether logic operations are supported. If this feature is not enabled, the logicOpEnable member of the VkPipelineColorBlendStateCreateInfo structure must be set to VK_FALSE, and the logicOp member is ignored.

- multiDrawIndirect specifies whether multiple draw indirect is supported. If this feature is not
enabled, the `drawCount` parameter to the `vkCmdDrawIndirect` and `vkCmdDrawIndexedIndirect` commands **must** be 0 or 1. The `maxDrawIndirectCount` member of the `VkPhysicalDeviceLimits` structure **must** also be 1 if this feature is not supported. See `maxDrawIndirectCount`.

- **drawIndirectFirstInstance** specifies whether indirect drawing calls support the `firstInstance` parameter. If this feature is not enabled, the `firstInstance` member of all `VkDrawIndirectCommand` and `VkDrawIndexedIndirectCommand` structures that are provided to the `vkCmdDrawIndirect` and `vkCmdDrawIndexedIndirect` commands **must** be 0.

- **depthClamp** specifies whether depth clamping is supported. If this feature is not enabled, the `depthClampEnable` member of the `VkPipelineRasterizationStateCreateInfo` structure **must** be set to `VK_FALSE`. Otherwise, setting `depthClampEnable` to `VK_TRUE` will enable depth clamping.

- **depthBiasClamp** specifies whether depth bias clamping is supported. If this feature is not enabled, the `depthBiasClamp` member of the `VkPipelineRasterizationStateCreateInfo` structure **must** be set to 0.0 unless the `VK_DYNAMIC_STATE_DEPTH_BIAS` dynamic state is enabled, and the `depthBiasClamp` parameter to `vkCmdSetDepthBias` **must** be set to 0.0.

- **fillModeNonSolid** specifies whether point and wireframe fill modes are supported. If this feature is not enabled, the `VK_POLYGON_MODE_POINT` and `VK_POLYGON_MODE_LINE` enum values **must** not be used.

- **depthBounds** specifies whether depth bounds tests are supported. If this feature is not enabled, the `depthBoundsTestEnable` member of the `VkPipelineDepthStencilStateCreateInfo` structure **must** be set to `VK_FALSE`. When `depthBoundsTestEnable` is set to `VK_FALSE`, the `minDepthBounds` and `maxDepthBounds` members of the `VkPipelineDepthStencilStateCreateInfo` structure are ignored.

- **wideLines** specifies whether lines with width other than 1.0 are supported. If this feature is not enabled, the `lineWidth` member of the `VkPipelineRasterizationStateCreateInfo` structure **must** be set to 1.0 unless the `VK_DYNAMIC_STATE_LINE_WIDTH` dynamic state is enabled, and the `lineWidth` parameter to `vkCmdSetLineWidth` **must** be set to 1.0. When this feature is supported, the range and granularity of supported line widths are indicated by the `lineWidthRange` and `lineWidthGranularity` members of the `VkPhysicalDeviceLimits` structure, respectively.

- **largePoints** specifies whether points with size greater than 1.0 are supported. If this feature is not enabled, only a point size of 1.0 written by a shader is supported. The range and granularity of supported point sizes are indicated by the `pointSizeRange` and `pointSizeGranularity` members of the `VkPhysicalDeviceLimits` structure, respectively.

- **alphaToOne** specifies whether the implementation is able to replace the alpha value of the fragment shader color output in the `Multisample Coverage` fragment operation. If this feature is not enabled, then the `alphaToOneEnable` member of the `VkPipelineMultisampleStateCreateInfo` structure **must** be set to `VK_FALSE`. Otherwise setting `alphaToOneEnable` to `VK_TRUE` will enable alpha-to-one behavior.

- **multiViewport** specifies whether more than one viewport is supported. If this feature is not enabled:
  - The `viewportCount` and `scissorCount` members of the `VkPipelineViewportStateCreateInfo` structure **must** be set to 1.
  - The `firstViewport` and `viewportCount` parameters to the `vkCmdSetViewport` command **must** be set to 0 and 1, respectively.
  - The `firstScissor` and `scissorCount` parameters to the `vkCmdSetScissor` command **must** be set
to 0 and 1, respectively.

- **samplerAnisotropy** specifies whether anisotropic filtering is supported. If this feature is not enabled, the `anisotropyEnable` member of the `VkSamplerCreateInfo` structure must be `VK_FALSE`.

- **textureCompressionETC2** specifies whether all of the ETC2 and EAC compressed texture formats are supported. If this feature is enabled, then the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `VK_FORMAT_FEATURE_BLIT_SRC_BIT` and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` features must be supported in `optimalTilingFeatures` for the following formats:

  - `VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK`
  - `VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK`
  - `VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK`
  - `VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK`
  - `VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK`
  - `VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK`
  - `VK_FORMAT_EAC_R11_UNORM_BLOCK`
  - `VK_FORMAT_EAC_R11_SNORM_BLOCK`
  - `VK_FORMAT_EAC_R11G11_UNORM_BLOCK`
  - `VK_FORMAT_EAC_R11G11_SNORM_BLOCK`

To query for additional properties, or if the feature is not enabled, `vkGetPhysicalDeviceFormatProperties` and `vkGetPhysicalDeviceImageFormatProperties` can be used to check for supported properties of individual formats as normal.

- **textureCompressionASTC_LDR** specifies whether all of the ASTC LDR compressed texture formats are supported. If this feature is enabled, then the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `VK_FORMAT_FEATURE_BLIT_SRC_BIT` and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` features must be supported in `optimalTilingFeatures` for the following formats:

  - `VK_FORMAT_ASTC_4x4_UNORM_BLOCK`
  - `VK_FORMAT_ASTC_4x4_SRGB_BLOCK`
  - `VK_FORMAT_ASTC_5x4_UNORM_BLOCK`
  - `VK_FORMAT_ASTC_5x4_SRGB_BLOCK`
  - `VK_FORMAT_ASTC_5x5_UNORM_BLOCK`
  - `VK_FORMAT_ASTC_5x5_SRGB_BLOCK`
  - `VK_FORMAT_ASTC_6x5_UNORM_BLOCK`
  - `VK_FORMAT_ASTC_6x5_SRGB_BLOCK`
  - `VK_FORMAT_ASTC_6x6_UNORM_BLOCK`
  - `VK_FORMAT_ASTC_6x6_SRGB_BLOCK`
  - `VK_FORMAT_ASTC_8x5_UNORM_BLOCK`
  - `VK_FORMAT_ASTC_8x5_SRGB_BLOCK`
  - `VK_FORMAT_ASTC_8x6_UNORM_BLOCK`
  - `VK_FORMAT_ASTC_8x6_SRGB_BLOCK`
  - `VK_FORMAT_ASTC_8x8_UNORM_BLOCK`
  - `VK_FORMAT_ASTC_8x8_SRGB_BLOCK`
- VK_FORMAT_ASTC_8x6_UNORM_BLOCK
- VK_FORMAT_ASTC_8x6_SRGB_BLOCK
- VK_FORMAT_ASTC_8x8_UNORM_BLOCK
- VK_FORMAT_ASTC_8x8_SRGB_BLOCK
- VK_FORMAT_ASTC_10x5_UNORM_BLOCK
- VK_FORMAT_ASTC_10x5_SRGB_BLOCK
- VK_FORMAT_ASTC_10x6_UNORM_BLOCK
- VK_FORMAT_ASTC_10x6_SRGB_BLOCK
- VK_FORMAT_ASTC_10x8_UNORM_BLOCK
- VK_FORMAT_ASTC_10x8_SRGB_BLOCK
- VK_FORMAT_ASTC_10x10_UNORM_BLOCK
- VK_FORMAT_ASTC_10x10_SRGB_BLOCK
- VK_FORMAT_ASTC_12x10_UNORM_BLOCK
- VK_FORMAT_ASTC_12x10_SRGB_BLOCK
- VK_FORMAT_ASTC_12x12_UNORM_BLOCK
- VK_FORMAT_ASTC_12x12_SRGB_BLOCK

To query for additional properties, or if the feature is not enabled, `vkGetPhysicalDeviceFormatProperties` and `vkGetPhysicalDeviceImageFormatProperties` can be used to check for supported properties of individual formats as normal.

- `textureCompressionBC` specifies whether all of the BC compressed texture formats are supported. If this feature is enabled, then the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `VK_FORMAT_FEATURE_BLIT_SRC_BIT` and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` features must be supported in `optimalTilingFeatures` for the following formats:

  - VK_FORMAT_BC1_RGB_UNORM_BLOCK
  - VK_FORMAT_BC1_RGB_SRGB_BLOCK
  - VK_FORMAT_BC1_RGBA_UNORM_BLOCK
  - VK_FORMAT_BC1_RGBA_SRGB_BLOCK
  - VK_FORMAT_BC2_UNORM_BLOCK
  - VK_FORMAT_BC2_SRGB_BLOCK
  - VK_FORMAT_BC3_UNORM_BLOCK
  - VK_FORMAT_BC3_SRGB_BLOCK
  - VK_FORMAT_BC4_UNORM_BLOCK
  - VK_FORMAT_BC4_SNORM_BLOCK
  - VK_FORMAT_BC5_UNORM_BLOCK
  - VK_FORMAT_BC5_SNORM_BLOCK
To query for additional properties, or if the feature is not enabled, 
vkGetPhysicalDeviceFormatProperties and vkGetPhysicalDeviceImageFormatProperties can be used to check for supported properties of individual formats as normal.

- **occlusionQueryPrecise** specifies whether occlusion queries returning actual sample counts are supported. Occlusion queries are created in a VkQueryPool by specifying the queryType of VK_QUERY_TYPE_OCCLUSION in the VkQueryPoolCreateInfo structure which is passed to vkCreateQueryPool. If this feature is enabled, queries of this type can enable VK_QUERY_CONTROL_PRECISE_BIT in the flags parameter to vkCmdBeginQuery. If this feature is not supported, the implementation supports only boolean occlusion queries. When any samples are passed, boolean queries will return a non-zero result value, otherwise a result value of zero is returned. When this feature is enabled and VK_QUERY_CONTROL_PRECISE_BIT is set, occlusion queries will report the actual number of samples passed.

- **pipelineStatisticsQuery** specifies whether the pipeline statistics queries are supported. If this feature is not enabled, queries of type VK_QUERY_TYPE_PIPELINE_STATISTICS cannot be created, and none of the VkQueryPipelineStatisticFlagBits bits can be set in the pipelineStatistics member of the VkQueryPoolCreateInfo structure.

- **vertexPipelineStoresAndAtomics** specifies whether storage buffers and images support stores and atomic operations in the vertex, tessellation, and geometry shader stages. If this feature is not enabled, all storage image, storage texel buffer, and storage buffer variables used by these stages in shader modules must be decorated with the NonWritable decoration (or the readonly memory qualifier in GLSL).

- **fragmentStoresAndAtomics** specifies whether storage buffers and images support stores and atomic operations in the fragment shader stage. If this feature is not enabled, all storage image, storage texel buffer, and storage buffer variables used by the fragment stage in shader modules must be decorated with the NonWritable decoration (or the readonly memory qualifier in GLSL).

- **shaderTessellationAndGeometryPointSize** specifies whether the PointSize built-in decoration is available in the tessellation control, tessellation evaluation, and geometry shader stages. If this feature is not enabled, members decorated with the PointSize built-in decoration must not be read from or written to and all points written from a tessellation or geometry shader will have a size of 1.0. This also specifies whether shader modules can declare the TessellationPointSize capability for tessellation control and evaluation shaders, or if the shader modules can declare the GeometryPointSize capability for geometry shaders. An implementation supporting this feature must also support one or both of the tessellationShader or geometryShader features.

- **shaderImageGatherExtended** specifies whether the extended set of image gather instructions are available in shader code. If this feature is not enabled, the OpImage*Gather instructions do not support the Offset and ConstOffsets operands. This also specifies whether shader modules can declare the ImageGatherExtended capability.

- **shaderStorageImageExtendedFormats** specifies whether all the “storage image extended formats” below are supported; if this feature is supported, then the VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT
must be supported in `optimalTilingFeatures` for the following formats:

- `VK_FORMAT_R16G16_SFLOAT`
- `VK_FORMAT_B10G11R11_UFLOAT_PACK32`
- `VK_FORMAT_R16_SFLOAT`
- `VK_FORMAT_R16G16B16A16_UNORM`
- `VK_FORMAT_A2B10G10R10_UNORM_PACK32`
- `VK_FORMAT_R16G16_UNORM`
- `VK_FORMAT_R8G8_UNORM`
- `VK_FORMAT_R16_UNORM`
- `VK_FORMAT_R8_UNORM`
- `VK_FORMAT_R16G16B16A16_SNORM`
- `VK_FORMAT_R16G16_SNORM`
- `VK_FORMAT_R8G8_SNORM`
- `VK_FORMAT_R16_SNORM`
- `VK_FORMAT_R8_SNORM`
- `VK_FORMAT_R16G16_SINT`
- `VK_FORMAT_R8G8_SINT`
- `VK_FORMAT_R16_SINT`
- `VK_FORMAT_R8_SINT`
- `VK_FORMAT_A2B10G10R10_UINT_PACK32`
- `VK_FORMAT_R16G16_UINT`
- `VK_FORMAT_R8G8_UINT`
- `VK_FORMAT_R16_UINT`
- `VK_FORMAT_R8_UINT`

**Note**

`shaderStorageImageExtendedFormats` feature only adds a guarantee of format support, which is specified for the whole physical device. Therefore enabling or disabling the feature via `vkCreateDevice` has no practical effect.

To query for additional properties, or if the feature is not supported, `vkGetPhysicalDeviceFormatProperties` and `vkGetPhysicalDeviceImageFormatProperties` can be used to check for supported properties of individual formats, as usual rules allow.

`VK_FORMAT_R32G32_UINT`, `VK_FORMAT_R32G32_SINT`, and `VK_FORMAT_R32G32_SFLOAT` from `StorageImageExtendedFormats` SPIR-V capability, are already covered by core Vulkan mandatory format support.
• shaderStorageImageMultisample specifies whether multisampled storage images are supported. If this feature is not enabled, images that are created with a usage that includes VK_IMAGE_USAGE_STORAGE_BIT must be created with samples equal to VK_SAMPLE_COUNT_1_BIT. This also specifies whether shader modules can declare the StorageImageMultisample and ImageMSArray capabilities.

• shaderStorageImageReadWithoutFormat specifies whether storage images require a format qualifier to be specified when reading. shaderStorageImageReadWithoutFormat applies only to formats listed in the storage without format list.

• shaderStorageImageWriteWithoutFormat specifies whether storage images require a format qualifier to be specified when writing. shaderStorageImageWriteWithoutFormat applies only to formats listed in the storage without format list.

• shaderUniformBufferArrayDynamicIndexing specifies whether arrays of uniform buffers can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also specifies whether shader modules can declare the UniformBufferArrayDynamicIndexing capability.

• shaderSampledImageArrayDynamicIndexing specifies whether arrays of samplers or sampled images can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_SAMPLER, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, or VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also specifies whether shader modules can declare the SampledImageArrayDynamicIndexing capability.

• shaderStorageBufferArrayDynamicIndexing specifies whether arrays of storage buffers can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also specifies whether shader modules can declare the StorageBufferArrayDynamicIndexing capability.

• shaderStorageImageArrayDynamicIndexing specifies whether arrays of storage images can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_STORAGE_IMAGE must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also specifies whether shader modules can declare the StorageImageArrayDynamicIndexing capability.

• shaderClipDistance specifies whether clip distances are supported in shader code. If this feature is not enabled, any members decorated with the ClipDistance built-in decoration must not be read from or written to in shader modules. This also specifies whether shader modules can declare the ClipDistance capability.

• shaderCullDistance specifies whether cull distances are supported in shader code. If this feature is not enabled, any members decorated with the CullDistance built-in decoration must not be read from or written to in shader modules. This also specifies whether shader modules can declare the CullDistance capability.

• shaderFloat64 specifies whether 64-bit floats (doubles) are supported in shader code. If this
feature is not enabled, 64-bit floating-point types must not be used in shader code. This also specifies whether shader modules can declare the float64 capability. Declaring and using 64-bit floats is enabled for all storage classes that SPIR-V allows with the float64 capability.

- shaderInt64 specifies whether 64-bit integers (signed and unsigned) are supported in shader code. If this feature is not enabled, 64-bit integer types must not be used in shader code. This also specifies whether shader modules can declare the Int64 capability. Declaring and using 64-bit integers is enabled for all storage classes that SPIR-V allows with the Int64 capability.

- shaderInt16 specifies whether 16-bit integers (signed and unsigned) are supported in shader code. If this feature is not enabled, 16-bit integer types must not be used in shader code. This also specifies whether shader modules can declare the Int16 capability. However, this only enables a subset of the storage classes that SPIR-V allows for the Int16 SPIR-V capability: Declaring and using 16-bit integers in the Private, Workgroup, and Function storage classes is enabled, while declaring them in the interface storage classes (e.g., UniformConstant, Uniform, StorageBuffer, Input, Output, and PushConstant) is not enabled.

- shaderResourceResidency specifies whether image operations that return resource residency information are supported in shader code. If this feature is not enabled, the OpImageSparse* instructions must not be used in shader code. This also specifies whether shader modules can declare the SparseResidency capability. The feature requires at least one of the sparseResidency* features to be supported.

- shaderResourceMinLod specifies whether image operations specifying the minimum resource LOD are supported in shader code. If this feature is not enabled, the MinLod image operand must not be used in shader code. This also specifies whether shader modules can declare the MinLod capability.

- sparseBinding specifies whether resource memory can be managed at opaque sparse block level instead of at the object level. If this feature is not enabled, resource memory must be bound only on a per-object basis using the vkBindBufferMemory and vkBindImageMemory commands. In this case, buffers and images must not be created with VK_BUFFER_CREATE_SPARSE_BINDING_BIT and VK_IMAGE_CREATE_SPARSE_BINDING_BIT set in the flags member of the VkBufferCreateInfo and VkImageCreateInfo structures, respectively. Otherwise resource memory can be managed as described in Sparse Resource Features.

- sparseResidencyBuffer specifies whether the device can access partially resident buffers. If this feature is not enabled, buffers must not be created with VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT set in the flags member of the VkBufferCreateInfo structure.

- sparseResidencyImage2D specifies whether the device can access partially resident 2D images with 1 sample per pixel. If this feature is not enabled, images with an imageType of VK_IMAGE_TYPE_2D and samples set to VK_SAMPLE_COUNT_1_BIT must not be created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set in the flags member of the VkImageCreateInfo structure.

-sparseResidencyImage3D specifies whether the device can access partially resident 3D images. If this feature is not enabled, images with an imageType of VK_IMAGE_TYPE_3D must not be created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set in the flags member of the VkImageCreateInfo structure.

- sparseResidency2Samples specifies whether the physical device can access partially resident 2D images with 2 samples per pixel. If this feature is not enabled, images with an imageType of
VK_IMAGE_TYPE_2D and samples set to VK_SAMPLE_COUNT_2_BIT must not be created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set in the flags member of the VkImageCreateInfo structure.

- sparseResidency4Samples specifies whether the physical device can access partially resident 2D images with 4 samples per pixel. If this feature is not enabled, images with an imageType of VK_IMAGE_TYPE_2D and samples set to VK_SAMPLE_COUNT_4_BIT must not be created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set in the flags member of the VkImageCreateInfo structure.

- sparseResidency8Samples specifies whether the physical device can access partially resident 2D images with 8 samples per pixel. If this feature is not enabled, images with an imageType of VK_IMAGE_TYPE_2D and samples set to VK_SAMPLE_COUNT_8_BIT must not be created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set in the flags member of the VkImageCreateInfo structure.

- sparseResidency16Samples specifies whether the physical device can access partially resident 2D images with 16 samples per pixel. If this feature is not enabled, images with an imageType of VK_IMAGE_TYPE_2D and samples set to VK_SAMPLE_COUNT_16_BIT must not be created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set in the flags member of the VkImageCreateInfo structure.

- sparseResidencyAliased specifies whether the physical device can correctly access data aliased into multiple locations. If this feature is not enabled, the VK_BUFFER_CREATE_SPARSE_ALIASED_BIT and VK_IMAGE_CREATE_SPARSE_ALIASED_BIT enum values must not be used in flags members of the VkBufferCreateInfo and VkImageCreateInfo structures, respectively.

- variableMultisampleRate specifies whether all pipelines that will be bound to a command buffer during a subpass which uses no attachments must have the same value for VkPipelineMultisampleStateCreateInfo::rasterizationSamples. If set to VK_TRUE, the implementation supports variable multisample rates in a subpass which uses no attachments. If set to VK_FALSE, then all pipelines bound in such a subpass must have the same multisample rate. This has no effect in situations where a subpass uses any attachments.

- inheritedQueries specifies whether a secondary command buffer may be executed while a query is active.

The VkPhysicalDeviceVulkan11Features structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceVulkan11Features {
    VkStructureType sType;
    void* pNext;
    VkBool32 storageBuffer16BitAccess;
    VkBool32 uniformAndStorageBuffer16BitAccess;
    VkBool32 storagePushConstant16;
    VkBool32 storageInputOutput16;
    VkBool32 multiview;
    VkBool32 multiviewGeometryShader;
    VkBool32 multiviewTessellationShader;
    VkBool32 variablePointersStorageBuffer;
    VkBool32 variablePointers;
} VkPhysicalDeviceVulkan11Features;
```
This structure describes the following features:

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

- `storageBuffer16BitAccess` specifies whether objects in the StorageBuffer, or PhysicalStorageBuffer storage class with the Block decoration can have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members must not be used in such objects. This also specifies whether shader modules can declare the StorageBuffer16BitAccess capability.

- `uniformAndStorageBuffer16BitAccess` specifies whether objects in the Uniform storage class with the Block decoration can have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members must not be used in such objects. This also specifies whether shader modules can declare the UniformAndStorageBuffer16BitAccess capability.

- `storagePushConstant16` specifies whether objects in the PushConstant storage class can have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or floating-point members must not be used in such objects. This also specifies whether shader modules can declare the StoragePushConstant16 capability.

- `storageInputOutput16` specifies whether objects in the Input and Output storage classes can have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members must not be used in such objects. This also specifies whether shader modules can declare the StorageInputOutput16 capability.

- `multiview` specifies whether the implementation supports multiview rendering within a render pass. If this feature is not enabled, the view mask of each subpass must always be zero.

- `multiviewGeometryShader` specifies whether the implementation supports multiview rendering within a render pass, with geometry shaders. If this feature is not enabled, then a pipeline compiled against a subpass with a non-zero view mask must not include a geometry shader.

- `multiviewTessellationShader` specifies whether the implementation supports multiview rendering within a render pass, with tessellation shaders. If this feature is not enabled, then a pipeline compiled against a subpass with a non-zero view mask must not include any tessellation shaders.

- `variablePointersStorageBuffer` specifies whether the implementation supports the SPIR-V VariablePointersStorageBuffer capability. When this feature is not enabled, shader modules must not declare the SPV_KHR_variable_pointers extension or the VariablePointersStorageBuffer capability.

- `variablePointers` specifies whether the implementation supports the SPIR-V VariablePointers capability. When this feature is not enabled, shader modules must not declare the VariablePointers capability.
• **protectedMemory** specifies whether **protected memory** is supported.

• **samplerYcbcrConversion** specifies whether the implementation supports **sampler Y’C_bC_r conversion**. If **samplerYcbcrConversion** is `VK_FALSE`, sampler Y’C_bC_r conversion is not supported, and samplers using sampler Y’C_bC_r conversion must not be used.

• **shaderDrawParameters** specifies whether the implementation supports the **SPIR-V DrawParameters capability**. When this feature is not enabled, shader modules must not declare the **SPV_KHR_shader_draw_parameters extension** or the **DrawParameters capability**.

If the **VkPhysicalDeviceVulkan11Features** structure is included in the **pNext** chain of the **VkPhysicalDeviceFeatures2** structure passed to **vkGetPhysicalDeviceFeatures2**, it is filled in to indicate whether each corresponding feature is supported. **VkPhysicalDeviceVulkan11Features** can also be used in the **pNext** chain of **VkDeviceCreateInfo** to selectively enable these features.

---

**Valid Usage (Implicit)**

- **VUID-VkPhysicalDeviceVulkan11Features-sType-sType**

  **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_1_FEATURES**

---

The **VkPhysicalDeviceVulkan12Features** structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceVulkan12Features {
    VkStructureType sType;
    void* pNext;
    VkBool32 samplerMirrorClampToEdge;
    VkBool32 drawIndirectCount;
    VkBool32 storageBuffer8BitAccess;
    VkBool32 uniformAndStorageBuffer8BitAccess;
    VkBool32 storagePushConstant8;
    VkBool32 shaderBufferInt64Atomics;
    VkBool32 shaderSharedInt64Atomics;
    VkBool32 shaderFloat16;
    VkBool32 shaderInt8;
    VkBool32 descriptorIndexing;
    VkBool32 shaderInputAttachmentArrayDynamicIndexing;
    VkBool32 shaderUniformTexelBufferArrayDynamicIndexing;
    VkBool32 shaderStorageTexelBufferArrayDynamicIndexing;
    VkBool32 shaderUniformBufferArrayNonUniformIndexing;
    VkBool32 shaderSampledImageArrayNonUniformIndexing;
    VkBool32 shaderStorageBufferArrayNonUniformIndexing;
    VkBool32 shaderStorageImageArrayNonUniformIndexing;
    VkBool32 shaderInputAttachmentArrayNonUniformIndexing;
    VkBool32 shaderUniformTexelBufferArrayNonUniformIndexing;
    VkBool32 shaderStorageTexelBufferArrayNonUniformIndexing;
    VkBool32 descriptorBindingUniformBufferUpdateAfterBind;
    VkBool32 descriptorBindingSampledImageUpdateAfterBind;
    VkBool32 descriptorBindingStorageImageUpdateAfterBind;
    VkBool32 descriptorBindingStorageBufferUpdateAfterBind;
};
```
This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.

- **samplerMirrorClampToEdge** indicates whether the implementation supports the `VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE` sampler address mode. If this feature is not enabled, the `VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE` sampler address mode must not be used.

- **drawIndirectCount** indicates whether the implementation supports the `vkCmdDrawIndirectCount` and `vkCmdDrawIndexedIndirectCount` functions. If this feature is not enabled, these functions must not be used.

- **storageBuffer8BitAccess** indicates whether objects in the `StorageBuffer`, or `PhysicalStorageBuffer` storage class with the `Block` decoration can have 8-bit integer members. If this feature is not enabled, 8-bit integer members must not be used in such objects. This also indicates whether shader modules can declare the `StorageBuffer8BitAccess` capability.

- **uniformAndStorageBuffer8BitAccess** indicates whether objects in the `Uniform` storage class with the `Block` decoration can have 8-bit integer members. If this feature is not enabled, 8-bit integer members must not be used in such objects. This also indicates whether shader modules can declare the `UniformAndStorageBuffer8BitAccess` capability.

- **storagePushConstant8** indicates whether objects in the `PushConstant` storage class can have 8-bit integer members. If this feature is not enabled, 8-bit integer members must not be used in such
objects. This also indicates whether shader modules can declare the `StoragePushConstant8` capability.

- `shaderBufferInt64Atomics` indicates whether shaders can perform 64-bit unsigned and signed integer atomic operations on buffers.

- `shaderSharedInt64Atomics` indicates whether shaders can perform 64-bit unsigned and signed integer atomic operations on shared memory.

- `shaderFloat16` indicates whether 16-bit floats (halves) are supported in shader code. This also indicates whether shader modules can declare the `Float16` capability. However, this only enables a subset of the storage classes that SPIR-V allows for the `Float16` SPIR-V capability: Declaring and using 16-bit floats in the `Private`, `Workgroup`, and `Function` storage classes is enabled, while declaring them in the interface storage classes (e.g., `UniformConstant`, `Uniform`, `StorageBuffer`, `Input`, `Output`, and `PushConstant`) is not enabled.

- `shaderInt8` indicates whether 8-bit integers (signed and unsigned) are supported in shader code. This also indicates whether shader modules can declare the `Int8` capability. However, this only enables a subset of the storage classes that SPIR-V allows for the `Int8` SPIR-V capability: Declaring and using 8-bit integers in the `Private`, `Workgroup`, and `Function` storage classes is enabled, while declaring them in the interface storage classes (e.g., `UniformConstant`, `Uniform`, `StorageBuffer`, `Input`, `Output`, and `PushConstant`) is not enabled.

- `descriptorIndexing` indicates whether the implementation supports the minimum set of descriptor indexing features as described in the `Feature Requirements` section. Enabling the `descriptorIndexing` member when `vkCreateDevice` is called does not imply the other minimum descriptor indexing features are also enabled. Those other descriptor indexing features must be enabled individually as needed by the application.

- `shaderInputAttachmentArrayDynamicIndexing` indicates whether arrays of input attachments can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `InputAttachmentArrayDynamicIndexing` capability.

- `shaderUniformTexelBufferArrayDynamicIndexing` indicates whether arrays of uniform texel buffers can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `UniformTexelBufferArrayDynamicIndexing` capability.

- `shaderStorageTexelBufferArrayDynamicIndexing` indicates whether arrays of storage texel buffers can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `StorageTexelBufferArrayDynamicIndexing` capability.

- `shaderUniformBufferArrayNonUniformIndexing` indicates whether arrays of uniform buffers can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or...
VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the UniformBufferArrayNonUniformIndexing capability.

- shaderSampledImageArrayNonUniformIndexing indicates whether arrays of samplers or sampled images can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_SAMPLER, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, or VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the SampledImageArrayNonUniformIndexing capability.

- shaderStorageBufferArrayNonUniformIndexing indicates whether arrays of storage buffers can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the StorageBufferArrayNonUniformIndexing capability.

- shaderStorageImageArrayNonUniformIndexing indicates whether arrays of storage images can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_STORAGE_IMAGE must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the StorageImageArrayNonUniformIndexing capability.

- shaderInputAttachmentArrayNonUniformIndexing indicates whether arrays of input attachments can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the InputAttachmentArrayNonUniformIndexing capability.

- shaderUniformTexelBufferArrayNonUniformIndexing indicates whether arrays of uniform texel buffers can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the UniformTexelBufferArrayNonUniformIndexing capability.

- shaderStorageTexelBufferArrayNonUniformIndexing indicates whether arrays of storage texel buffers can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the StorageTexelBufferArrayNonUniformIndexing capability.

- descriptorBindingUniformBufferUpdateAfterBind indicates whether the implementation supports updating uniform buffer descriptors after a set is bound. If this feature is not enabled, VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT must not be used with VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER.

- descriptorBindingSampledImageUpdateAfterBind indicates whether the implementation supports
updating sampled image descriptors after a set is bound. If this feature is not enabled, 
VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT must not be used with
VK_DESCRIPTOR_TYPE_SAMPLER, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, or
VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE.

- descriptorBindingStorageImageUpdateAfterBind indicates whether the implementation supports
  updating storage image descriptors after a set is bound. If this feature is not enabled, 
  VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT must not be used with
  VK_DESCRIPTOR_TYPE_STORAGE_IMAGE.

- descriptorBindingStorageBufferUpdateAfterBind indicates whether the implementation supports
  updating storage buffer descriptors after a set is bound. If this feature is not enabled, 
  VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT must not be used with
  VK_DESCRIPTOR_TYPE_STORAGE_BUFFER.

- descriptorBindingUniformTexelBufferUpdateAfterBind indicates whether the implementation
  supports updating uniform texel buffer descriptors after a set is bound. If this feature is not
  enabled, VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT must not be used with
  VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER.

- descriptorBindingStorageTexelBufferUpdateAfterBind indicates whether the implementation
  supports updating storage texel buffer descriptors after a set is bound. If this feature is not
  enabled, VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT must not be used with
  VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER.

- descriptorBindingUpdateUnusedWhilePending indicates whether the implementation supports
  updating descriptors while the set is in use. If this feature is not enabled,
  VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT must not be used.

- descriptorBindingPartiallyBound indicates whether the implementation supports statically
  using a descriptor set binding in which some descriptors are not valid. If this feature is not
  enabled, VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT must not be used.

- descriptorBindingVariableDescriptorCount indicates whether the implementation supports
  descriptor sets with a variable-sized last binding. If this feature is not enabled, 
  VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT must not be used.

- runtimeDescriptorArray indicates whether the implementation supports the SPIR-V
  RuntimeDescriptorArray capability. If this feature is not enabled, descriptors must not be
  declared in runtime arrays.

- samplerFilterMinmax indicates whether the implementation supports a minimum set of required
  formats supporting min/max filtering as defined by the filterMinmaxSingleComponentFormats
  property minimum requirements. If this feature is not enabled, then
  VkSamplerReductionModeCreateInfo must only use
  VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE.

- scalarBlockLayout indicates that the implementation supports the layout of resource blocks in
  shaders using scalar alignment.

- imagelessFramebuffer indicates that the implementation supports specifying the image view for
  attachments at render pass begin time via VkRenderPassAttachmentBeginInfo.

- uniformBufferStandardLayout indicates that the implementation supports the same layouts for
  uniform buffers as for storage and other kinds of buffers. See Standard Buffer Layout.
• **shaderSubgroupExtendedTypes** is a boolean specifying whether subgroup operations can use 8-bit integer, 16-bit integer, 64-bit integer, 16-bit floating-point, and vectors of these types in **group operations** with **subgroup scope**, if the implementation supports the types.

• **separateDepthStencilLayouts** indicates whether the implementation supports a **VkImageMemoryBarrier** for a depth/stencil image with only one of **VK_IMAGE_ASPECT_DEPTH_BIT** or **VK_IMAGE_ASPECT_STENCIL_BIT** set, and whether **VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL**, **VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL**, **VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL**, or **VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL** can be used.

• **hostQueryReset** indicates that the implementation supports resetting queries from the host with **vkResetQueryPool**.

• **timelineSemaphore** indicates whether semaphores created with a **VkSemaphoreType** of **VK_SEMAPHORE_TYPE_TIMELINE** are supported.

• **bufferDeviceAddress** indicates that the implementation supports accessing buffer memory in shaders as storage buffers via an address queried from **vkGetBufferDeviceAddress**.

• **bufferDeviceAddressCaptureReplay** indicates that the implementation supports saving and reusing buffer and device addresses, e.g. for trace capture and replay.

• **bufferDeviceAddressMultiDevice** indicates that the implementation supports the **bufferDeviceAddress** feature for logical devices created with multiple physical devices. If this feature is not supported, buffer addresses **must** not be queried on a logical device created with more than one physical device.

• **vulkanMemoryModel** indicates whether the Vulkan Memory Model is supported, as defined in **Vulkan Memory Model**. This also indicates whether shader modules **can** declare the **VulkanMemoryModel** capability.

• **vulkanMemoryModelDeviceScope** indicates whether the Vulkan Memory Model can use **Device scope** synchronization. This also indicates whether shader modules **can** declare the **VulkanMemoryModelDeviceScope** capability.

• **vulkanMemoryModelAvailabilityVisibilityChains** indicates whether the Vulkan Memory Model can use **availability and visibility chains** with more than one element.

• **shaderOutputViewportIndex** indicates whether the implementation supports the **ShaderViewportIndex** SPIR-V capability enabling variables decorated with the **ViewportIndex** built-in to be exported from vertex or tessellation evaluation shaders. If this feature is not enabled, the **ViewportIndex** built-in decoration **must** not be used on outputs in vertex or tessellation evaluation shaders.

• **shaderOutputLayer** indicates whether the implementation supports the **ShaderLayer** SPIR-V capability enabling variables decorated with the **Layer** built-in to be exported from vertex or tessellation evaluation shaders. If this feature is not enabled, the **Layer** built-in decoration **must** not be used on outputs in vertex or tessellation evaluation shaders.

• If **subgroupBroadcastDynamicId** is **VK_TRUE**, the “Id” operand of **OpGroupNonUniformBroadcast** **can** be dynamically uniform within a subgroup, and the “Index” operand of **OpGroupNonUniformQuadBroadcast** **can** be dynamically uniform within the derivative group. If it is **VK_FALSE**, these operands **must** be constants.

If the **VkPhysicalDeviceVulkan12Features** structure is included in the **pNext** chain of the
VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceVulkan12Features can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceVulkan12Features-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_2_FEATURES

The VkPhysicalDeviceVulkan13Features structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPhysicalDeviceVulkan13Features {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           robustImageAccess;
    VkBool32           inlineUniformBlock;
    VkBool32           descriptorBindingInlineUniformBlockUpdateAfterBind;
    VkBool32           pipelineCreationCacheControl;
    VkBool32           privateData;
    VkBool32           shaderDemoteToHelperInvocation;
    VkBool32           shaderTerminateInvocation;
    VkBool32           subgroupSizeControl;
    VkBool32           computeFullSubgroups;
    VkBool32           synchronization2;
    VkBool32           textureCompressionASTC_HDR;
    VkBool32           shaderZeroInitializeWorkgroupMemory;
    VkBool32           dynamicRendering;
    VkBool32           shaderIntegerDotProduct;
    VkBool32           maintenance4;
} VkPhysicalDeviceVulkan13Features;
```

This structure describes the following features:

- **sType** is the type of this structure.

- **pNext** is NULL or a pointer to a structure extending this structure.

- **robustImageAccess** indicates whether image accesses are tightly bounds-checked against the dimensions of the image view. Invalid texels resulting from out of bounds image loads will be replaced as described in Texel Replacement, with either (0,0,1) or (0,0,0) values inserted for missing G, B, or A components based on the format.

- **inlineUniformBlock** indicates whether the implementation supports inline uniform block descriptors. If this feature is not enabled, VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK must not be used.

- **descriptorBindingInlineUniformBlockUpdateAfterBind** indicates whether the implementation supports updating inline uniform block descriptors after a set is bound. If this feature is not
enabled, \texttt{VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT} \textbf{must} not be used with \texttt{VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK}.

- \texttt{pipelineCreationCacheControl} indicates that the implementation supports:
  - The following \textbf{can} be used in \texttt{Vk*PipelineCreateInfo::flags}:
    - \texttt{VK.PIPELINE_CREATE_FAIL_ON_PIPELINE_COMPILE_REQUIRED_BIT}
    - \texttt{VK.PIPELINE_CREATE_EARLY_RETURN_ON_FAILURE_BIT}
  - The following \textbf{can} be used in \texttt{VkPipeplineCacheCreateInfo::flags}:
    - \texttt{VK.PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT}

- \texttt{privateData} indicates whether the implementation supports private data. See \texttt{Private Data}.

- \texttt{shaderDemoteToHelperInvocation} indicates whether the implementation supports the SPIR-V DemoteToHelperInvocationEXT capability.

- \texttt{shaderTerminateInvocation} specifies whether the implementation supports SPIR-V modules that use the SPV_KHR_terminate_invocation extension.

- \texttt{subgroupSizeControl} indicates whether the implementation supports controlling shader subgroup sizes via the \texttt{VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT} flag and the \texttt{VkPipelineShaderStageRequiredSubgroupSizeCreateInfo} structure.

- \texttt{computeFullSubgroups} indicates whether the implementation supports requiring full subgroups in compute shaders via the \texttt{VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT} flag.

- \texttt{synchronization2} indicates whether the implementation supports the new set of synchronization commands introduced in \texttt{VK_KHR_synchronization2}.

- \texttt{textureCompressionASTC_HDR} indicates whether all of the ASTC HDR compressed texture formats are supported. If this feature is enabled, then the \texttt{VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT}, \texttt{VK_FORMAT_FEATURE_BLIT_SRC_BIT} and \texttt{VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT} features \textbf{must} be supported in \texttt{optimalTilingFeatures} for the following formats:
  - \texttt{VK\_FORMAT\_ASTC\_4x4\_SFLOAT\_BLOCK}
  - \texttt{VK\_FORMAT\_ASTC\_5x4\_SFLOAT\_BLOCK}
  - \texttt{VK\_FORMAT\_ASTC\_5x5\_SFLOAT\_BLOCK}
  - \texttt{VK\_FORMAT\_ASTC\_6x5\_SFLOAT\_BLOCK}
  - \texttt{VK\_FORMAT\_ASTC\_6x6\_SFLOAT\_BLOCK}
  - \texttt{VK\_FORMAT\_ASTC\_8x5\_SFLOAT\_BLOCK}
  - \texttt{VK\_FORMAT\_ASTC\_8x6\_SFLOAT\_BLOCK}
  - \texttt{VK\_FORMAT\_ASTC\_8x8\_SFLOAT\_BLOCK}
  - \texttt{VK\_FORMAT\_ASTC\_10x5\_SFLOAT\_BLOCK}
  - \texttt{VK\_FORMAT\_ASTC\_10x6\_SFLOAT\_BLOCK}
  - \texttt{VK\_FORMAT\_ASTC\_10x8\_SFLOAT\_BLOCK}
  - \texttt{VK\_FORMAT\_ASTC\_10x10\_SFLOAT\_BLOCK}
  - \texttt{VK\_FORMAT\_ASTC\_12x10\_SFLOAT\_BLOCK}
VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK

To query for additional properties, or if the feature is not enabled, `vkGetPhysicalDeviceFormatProperties` and `vkGetPhysicalDeviceImageFormatProperties` can be used to check for supported properties of individual formats as normal.

- **shaderZeroInitializeWorkgroupMemory** specifies whether the implementation supports initializing a variable in Workgroup storage class.
- **dynamicRendering** specifies that the implementation supports dynamic render pass instances using the `vkCmdBeginRendering` command.
- **shaderIntegerDotProduct** specifies whether shader modules can declare the `DotProductInputAllKHR`, `DotProductInput4x8BitKHR`, `DotProductInput4x8BitPackedKHR` and `DotProductKHR` capabilities.
- **maintenance4** indicates that the implementation supports the following:
  - The application **may** destroy a `VkPipelineLayout` object immediately after using it to create another object.
  - `LocalSizeId` **can** be used as an alternative to `LocalSize` to specify the local workgroup size with specialization constants.
  - Images created with identical creation parameters will always have the same alignment requirements.
  - The size memory requirement of a buffer or image is never greater than that of another buffer or image created with a greater or equal size.
  - Push constants do not have to be initialized before they are dynamically accessed.
  - The interface matching rules allow a larger output vector to match with a smaller input vector, with additional values being discarded.

If the `VkPhysicalDeviceVulkan13Features` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceVulkan13Features` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

## Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceVulkan13Features-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_3_FEATURES`.

The `VkPhysicalDeviceVariablePointersFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceVariablePointersFeatures {
    VkStructureType            sType;
    void*                       pNext;
    VkBool32                    variablePointersStorageBuffer;
    VkBool32                    variablePointers;
} VkPhysicalDeviceVariablePointersFeatures;
```
This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.

- **variablePointersStorageBuffer** specifies whether the implementation supports the SPIR-V VariablePointersStorageBuffer capability. When this feature is not enabled, shader modules **must** not declare the SPV_KHR_variable_pointers extension or the VariablePointersStorageBuffer capability.

- **variablePointers** specifies whether the implementation supports the SPIR-V VariablePointers capability. When this feature is not enabled, shader modules **must** not declare the VariablePointers capability.

If the `VkPhysicalDeviceVariablePointersFeatures` structure is included in the **pNext** chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceVariablePointersFeatures` can also be used in the **pNext** chain of `VkDeviceCreateInfo` to selectively enable these features.

**Valid Usage**

- VUID-VkPhysicalDeviceVariablePointersFeatures-variablePointers-01431
  If **variablePointers** is enabled then **variablePointersStorageBuffer** **must** also be enabled

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceVariablePointersFeatures-sType-sType
  **sType** **must** be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTERS_FEATURES

The `VkPhysicalDeviceMultiviewFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceMultiviewFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 multiview;
    VkBool32 multiviewGeometryShader;
    VkBool32 multiviewTessellationShader;
} VkPhysicalDeviceMultiviewFeatures;
```
This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **multiview** specifies whether the implementation supports multiview rendering within a render pass. If this feature is not enabled, the view mask of each subpass must always be zero.
- **multiviewGeometryShader** specifies whether the implementation supports multiview rendering within a render pass, with geometry shaders. If this feature is not enabled, then a pipeline compiled against a subpass with a non-zero view mask must not include a geometry shader.
- **multiviewTessellationShader** specifies whether the implementation supports multiview rendering within a render pass, with tessellation shaders. If this feature is not enabled, then a pipeline compiled against a subpass with a non-zero view mask must not include any tessellation shaders.

If the **VkPhysicalDeviceMultiviewFeatures** structure is included in the **pNext** chain of the **VkPhysicalDeviceFeatures2** structure passed to **vkGetPhysicalDeviceFeatures2**, it is filled in to indicate whether each corresponding feature is supported. **VkPhysicalDeviceMultiviewFeatures** can also be used in the **pNext** chain of **VkDeviceCreateInfo** to selectively enable these features.

### Valid Usage

- VUID-VkPhysicalDeviceMultiviewFeatures-multiviewGeometryShader-00580
  If **multiviewGeometryShader** is enabled then **multiview** must also be enabled
- VUID-VkPhysicalDeviceMultiviewFeatures-multiviewTessellationShader-00581
  If **multiviewTessellationShader** is enabled then **multiview** must also be enabled

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceMultiviewFeatures-sType-sType
  **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_FEATURES**

The **VkPhysicalDeviceShaderAtomicInt64Features** structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceShaderAtomicInt64Features {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderBufferInt64Atomics;
    VkBool32 shaderSharedInt64Atomics;
} VkPhysicalDeviceShaderAtomicInt64Features;
```
This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **shaderBufferInt64Atomics** indicates whether shaders can perform 64-bit unsigned and signed integer atomic operations on buffers.
- **shaderSharedInt64Atomics** indicates whether shaders can perform 64-bit unsigned and signed integer atomic operations on shared memory.

If the `VkPhysicalDeviceShaderAtomicInt64Features` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceShaderAtomicInt64Features` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

---

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceShaderAtomicInt64Features-sType-sType
  
  **sType** must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_INT64_FEATURES`.

The `VkPhysicalDevice8BitStorageFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDevice8BitStorageFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 storageBuffer8BitAccess;
    VkBool32 uniformAndStorageBuffer8BitAccess;
    VkBool32 storagePushConstant8;
} VkPhysicalDevice8BitStorageFeatures;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **storageBuffer8BitAccess** indicates whether objects in the `StorageBuffer`, or `PhysicalStorageBuffer` storage class with the `Block` decoration can have 8-bit integer members. If this feature is not enabled, 8-bit integer members must not be used in such objects. This also indicates whether shader modules can declare the `StorageBuffer8BitAccess` capability.
- **uniformAndStorageBuffer8BitAccess** indicates whether objects in the `Uniform` storage class with the `Block` decoration can have 8-bit integer members. If this feature is not enabled, 8-bit integer members must not be used in such objects. This also indicates whether shader modules can declare the `UniformAndStorageBuffer8BitAccess` capability.
- **storagePushConstant8** indicates whether objects in the `PushConstant` storage class can have 8-bit
integer members. If this feature is not enabled, 8-bit integer members must not be used in such objects. This also indicates whether shader modules can declare the StoragePushConstant8 capability.

If the VkPhysicalDevice8BitStorageFeatures structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDevice8BitStorageFeatures can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

• VUID-VkPhysicalDevice8BitStorageFeatures-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_8BIT_STORAGE_FEATURES

The VkPhysicalDevice16BitStorageFeatures structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDevice16BitStorageFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 storageBuffer16BitAccess;
    VkBool32 uniformAndStorageBuffer16BitAccess;
    VkBool32 storagePushConstant16;
    VkBool32 storageInputOutput16;
} VkPhysicalDevice16BitStorageFeatures;
```

This structure describes the following features:

• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.

• **storageBuffer16BitAccess** specifies whether objects in the StorageBuffer, or PhysicalStorageBuffer storage class with the Block decoration can have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members must not be used in such objects. This also specifies whether shader modules can declare the StorageBuffer16BitAccess capability.

• **uniformAndStorageBuffer16BitAccess** specifies whether objects in the Uniform storage class with the Block decoration can have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members must not be used in such objects. This also specifies whether shader modules can declare the UniformAndStorageBuffer16BitAccess capability.

• **storagePushConstant16** specifies whether objects in the PushConstant storage class can have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or floating-point members must not be used in such objects. This also specifies whether shader modules can declare the StoragePushConstant16 capability.

• **storageInputOutput16** specifies whether objects in the Input and Output storage classes can have
16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members must not be used in such objects. This also specifies whether shader modules can declare the StorageInputOutput16 capability.

If the VkPhysicalDevice16BitStorageFeatures structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDevice16BitStorageFeatures can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDevice16BitStorageFeatures-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_16BIT_STORAGE_FEATURES

The VkPhysicalDeviceShaderFloat16Int8Features structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceShaderFloat16Int8Features {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderFloat16;
    VkBool32 shaderInt8;
} VkPhysicalDeviceShaderFloat16Int8Features;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **shaderFloat16** indicates whether 16-bit floats (halves) are supported in shader code. This also indicates whether shader modules can declare the Float16 capability. However, this only enables a subset of the storage classes that SPIR-V allows for the Float16 SPIR-V capability: Declaring and using 16-bit floats in the Private, Workgroup, and Function storage classes is enabled, while declaring them in the interface storage classes (e.g., UniformConstant, Uniform, StorageBuffer, Input, Output, and PushConstant) is not enabled.
- **shaderInt8** indicates whether 8-bit integers (signed and unsigned) are supported in shader code. This also indicates whether shader modules can declare the Int8 capability. However, this only enables a subset of the storage classes that SPIR-V allows for the Int8 SPIR-V capability: Declaring and using 8-bit integers in the Private, Workgroup, and Function storage classes is enabled, while declaring them in the interface storage classes (e.g., UniformConstant, Uniform, StorageBuffer, Input, Output, and PushConstant) is not enabled.

If the VkPhysicalDeviceShaderFloat16Int8Features structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceShaderFloat16Int8Features can also be used in the pNext chain of
VkDeviceCreateInfo to selectively enable these features.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceShaderFloat16Int8Features-sType-sType
  
  *sType must be* **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_FLOAT16_INT8_FEATURES**

The *VkPhysicalDeviceSamplerYcbcrConversionFeatures* structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceSamplerYcbcrConversionFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 samplerYcbcrConversion;
} VkPhysicalDeviceSamplerYcbcrConversionFeatures;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **samplerYcbcrConversion** specifies whether the implementation supports sampler Y’C_bC_r conversion. If *samplerYcbcrConversion* is VK_FALSE, sampler Y’C_bC_r conversion is not supported, and samplers using sampler Y’C_bC_r conversion must not be used.

If the *VkPhysicalDeviceSamplerYcbcrConversionFeatures* structure is included in the *pNext* chain of the *VkPhysicalDeviceFeatures2* structure passed to *vkGetPhysicalDeviceFeatures2*, it is filled in to indicate whether each corresponding feature is supported. *VkPhysicalDeviceSamplerYcbcrConversionFeatures* can also be used in the *pNext* chain of *VkDeviceCreateInfo* to selectively enable these features.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceSamplerYcbcrConversionFeatures-sType-sType
  
  *sType must be* **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_YCBCR_CONVERSION_FEATURES**

The *VkPhysicalDeviceProtectedMemoryFeatures* structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceProtectedMemoryFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 protectedMemory;
} VkPhysicalDeviceProtectedMemoryFeatures;
```
This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `protectedMemory` specifies whether protected memory is supported.

If the `VkPhysicalDeviceProtectedMemoryFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceProtectedMemoryFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceProtectedMemoryFeatures-sType-sType
  - `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_FEATURES`

The `VkPhysicalDeviceShaderDrawParametersFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceShaderDrawParametersFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderDrawParameters;
} VkPhysicalDeviceShaderDrawParametersFeatures;
```

This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `shaderDrawParameters` specifies whether the implementation supports the SPIR-V DrawParameters capability. When this feature is not enabled, shader modules must not declare the `SPV_KHR_shader_draw_parameters` extension or the `DrawParameters` capability.

If the `VkPhysicalDeviceShaderDrawParametersFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceShaderDrawParametersFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceShaderDrawParametersFeatures-sType-sType

  
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DRAW_PARAMETERS_FEATURES

The VkPhysicalDeviceDescriptorIndexingFeatures structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceDescriptorIndexingFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderInputAttachmentArrayDynamicIndexing;
    VkBool32 shaderUniformTexelBufferArrayDynamicIndexing;
    VkBool32 shaderStorageTexelBufferArrayDynamicIndexing;
    VkBool32 shaderUniformBufferArrayNonUniformIndexing;
    VkBool32 shaderSampledImageArrayNonUniformIndexing;
    VkBool32 shaderStorageBufferArrayNonUniformIndexing;
    VkBool32 shaderInputAttachmentArrayNonUniformIndexing;
    VkBool32 shaderUniformTexelBufferArrayNonUniformIndexing;
    VkBool32 shaderStorageTexelBufferArrayNonUniformIndexing;
    VkBool32 descriptorBindingUniformBufferUpdateAfterBind;
    VkBool32 descriptorBindingSampledImageUpdateAfterBind;
    VkBool32 descriptorBindingStorageImageUpdateAfterBind;
    VkBool32 descriptorBindingStorageBufferUpdateAfterBind;
    VkBool32 descriptorBindingUniformTexelBufferUpdateAfterBind;
    VkBool32 descriptorBindingStorageTexelBufferUpdateAfterBind;
    VkBool32 descriptorBindingUpdateUnusedWhilePending;
    VkBool32 descriptorBindingPartiallyBound;
    VkBool32 descriptorBindingVariableDescriptorCount;
    VkBool32 runtimeDescriptorArray;
} VkPhysicalDeviceDescriptorIndexingFeatures;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.

- **shaderInputAttachmentArrayDynamicIndexing** indicates whether arrays of input attachments can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the InputAttachmentArrayDynamicIndexing capability.

- **shaderUniformTexelBufferArrayDynamicIndexing** indicates whether arrays of uniform texel buffers can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of
**VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER** must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `UniformTexelBufferArrayDynamicIndexing` capability.

- **shaderStorageTexelBufferArrayDynamicIndexing** indicates whether arrays of storage texel buffers can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of **VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER** must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `StorageTexelBufferArrayDynamicIndexing` capability.

- **shaderUniformBufferArrayNonUniformIndexing** indicates whether arrays of uniform buffers can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of **VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER** or **VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC** must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `UniformBufferArrayNonUniformIndexing` capability.

- **shaderSampledImageArrayNonUniformIndexing** indicates whether arrays of samplers or sampled images can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of **VK_DESCRIPTOR_TYPE_SAMPLER**, **VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER**, or **VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE** must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `SampledImageArrayNonUniformIndexing` capability.

- **shaderStorageBufferArrayNonUniformIndexing** indicates whether arrays of storage buffers can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of **VK_DESCRIPTOR_TYPE_STORAGE_BUFFER** or **VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC** must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `StorageBufferArrayNonUniformIndexing` capability.

- **shaderStorageImageArrayNonUniformIndexing** indicates whether arrays of storage images can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of **VK_DESCRIPTOR_TYPE_STORAGE_IMAGE** must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `StorageImageArrayNonUniformIndexing` capability.

- **shaderInputAttachmentArrayNonUniformIndexing** indicates whether arrays of input attachments can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of **VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT** must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `InputAttachmentArrayNonUniformIndexing` capability.

- **shaderUniformTexelBufferArrayNonUniformIndexing** indicates whether arrays of uniform texel buffers can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of **VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER** must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the
UniformTexelBufferArrayNonUniformIndexing capability.

• shaderStorageTexelBufferArrayNonUniformIndexing indicates whether arrays of storage texel buffers can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the StorageTexelBufferArrayNonUniformIndexing capability.

• descriptorBindingUniformBufferUpdateAfterBind indicates whether the implementation supports updating uniform buffer descriptors after a set is bound. If this feature is not enabled, VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT must not be used with VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER.

• descriptorBindingSamplerImageUpdateAfterBind indicates whether the implementation supports updating sampled image descriptors after a set is bound. If this feature is not enabled, VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT must not be used with VK_DESCRIPTOR_TYPE_SAMPLER, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, or VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE.

• descriptorBindingStorageImageUpdateAfterBind indicates whether the implementation supports updating storage image descriptors after a set is bound. If this feature is not enabled, VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT must not be used with VK_DESCRIPTOR_TYPE_STORAGE_IMAGE.

• descriptorBindingStorageBufferUpdateAfterBind indicates whether the implementation supports updating storage buffer descriptors after a set is bound. If this feature is not enabled, VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT must not be used with VK_DESCRIPTOR_TYPE_STORAGE_BUFFER.

• descriptorBindingUniformTexelBufferUpdateAfterBind indicates whether the implementation supports updating uniform texel buffer descriptors after a set is bound. If this feature is not enabled, VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT must not be used with VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER.

• descriptorBindingStorageTexelBufferUpdateAfterBind indicates whether the implementation supports updating storage texel buffer descriptors after a set is bound. If this feature is not enabled, VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT must not be used with VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER.

• descriptorBindingUpdateUnusedWhilePending indicates whether the implementation supports updating descriptors while the set is in use. If this feature is not enabled, VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT must not be used.

• descriptorBindingPartiallyBound indicates whether the implementation supports statically using a descriptor set binding in which some descriptors are not valid. If this feature is not enabled, VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT must not be used.

• descriptorBindingVariableDescriptorCount indicates whether the implementation supports descriptor sets with a variable-sized last binding. If this feature is not enabled, VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT must not be used.

• runtimeDescriptorArray indicates whether the implementation supports the SPIR-V RuntimeDescriptorArray capability. If this feature is not enabled, descriptors must not be
declared in runtime arrays.

If the `VkPhysicalDeviceDescriptorIndexingFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceDescriptorIndexingFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceDescriptorIndexingFeatures-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_FEATURES`

The `VkPhysicalDeviceVulkanMemoryModelFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceVulkanMemoryModelFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 VulkanMemoryModel;
    VkBool32 VulkanMemoryModelDeviceScope;
    VkBool32 VulkanMemoryModelAvailabilityVisibilityChains;
} VkPhysicalDeviceVulkanMemoryModelFeatures;
```

This structure describes the following features:

- **sType** is the type of this structure.

- **pNext** is NULL or a pointer to a structure extending this structure.

- **VulkanMemoryModel** indicates whether the Vulkan Memory Model is supported, as defined in [Vulkan Memory Model](#). This also indicates whether shader modules can declare the `VulkanMemoryModel` capability.

- **VulkanMemoryModelDeviceScope** indicates whether the Vulkan Memory Model can use Device scope synchronization. This also indicates whether shader modules can declare the `VulkanMemoryModelDeviceScope` capability.

- **VulkanMemoryModelAvailabilityVisibilityChains** indicates whether the Vulkan Memory Model can use availability and visibility chains with more than one element.

If the `VkPhysicalDeviceVulkanMemoryModelFeaturesKHR` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceVulkanMemoryModelFeaturesKHR` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceVulkanMemoryModelFeatures-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_MEMORY_MODEL_FEATURES`

The `VkPhysicalDeviceInlineUniformBlockFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPhysicalDeviceInlineUniformBlockFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 inlineUniformBlock;
    VkBool32 descriptorBindingInlineUniformBlockUpdateAfterBind;
} VkPhysicalDeviceInlineUniformBlockFeatures;
```

This structure describes the following features:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `inlineUniformBlock` indicates whether the implementation supports inline uniform block descriptors. If this feature is not enabled, `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK` must not be used.
- `descriptorBindingInlineUniformBlockUpdateAfterBind` indicates whether the implementation supports updating inline uniform block descriptors after a set is bound. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` must not be used with `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK`.

If the `VkPhysicalDeviceInlineUniformBlockFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceInlineUniformBlockFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceInlineUniformBlockFeatures-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INLINE_UNIFORM_BLOCK_FEATURES`

The `VkPhysicalDeviceScalarBlockLayoutFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceScalarBlockLayoutFeatures {
    VkStructureType sType;
    void* pNext;
} VkPhysicalDeviceScalarBlockLayoutFeatures;
```
VkBool32 scalarBlockLayout;
} VkPhysicalDeviceScalarBlockLayoutFeatures;

This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `scalarBlockLayout` indicates that the implementation supports the layout of resource blocks in shaders using scalar alignment.

If the `VkPhysicalDeviceScalarBlockLayoutFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported.

VkPhysicalDeviceScalarBlockLayoutFeatures can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceScalarBlockLayoutFeatures-sType-sType` `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SCALAR_BLOCK_LAYOUT_FEATURES`

The `VkPhysicalDeviceUniformBufferStandardLayoutFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceUniformBufferStandardLayoutFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 uniformBufferStandardLayout;
} VkPhysicalDeviceUniformBufferStandardLayoutFeatures;
```

This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `uniformBufferStandardLayout` indicates that the implementation supports the same layouts for uniform buffers as for storage and other kinds of buffers. See Standard Buffer Layout.

If the `VkPhysicalDeviceUniformBufferStandardLayoutFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported.

VkPhysicalDeviceUniformBufferStandardLayoutFeatures can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceUniformBufferStandardLayoutFeatures-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_UNIFORM_BUFFER_STANDARD_LAYOUT_FEATURES

The `VkPhysicalDeviceBufferDeviceAddressFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceBufferDeviceAddressFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 bufferDeviceAddress;
    VkBool32 bufferDeviceAddressCaptureReplay;
    VkBool32 bufferDeviceAddressMultiDevice;
} VkPhysicalDeviceBufferDeviceAddressFeatures;
```

This structure describes the following features:

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `bufferDeviceAddress` indicates that the implementation supports accessing buffer memory in shaders as storage buffers via an address queried from `vkGetBufferDeviceAddress`.
- `bufferDeviceAddressCaptureReplay` indicates that the implementation supports saving and reusing buffer and device addresses, e.g. for trace capture and replay.
- `bufferDeviceAddressMultiDevice` indicates that the implementation supports the `bufferDeviceAddress` feature for logical devices created with multiple physical devices. If this feature is not supported, buffer addresses must not be queried on a logical device created with more than one physical device.

Note: `bufferDeviceAddressMultiDevice` exists to allow certain legacy platforms to be able to support `bufferDeviceAddress` without needing to support shared GPU virtual addresses for multi-device configurations.

See `vkGetBufferDeviceAddress` for more information.

If the `VkPhysicalDeviceBufferDeviceAddressFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceBufferDeviceAddressFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceBufferDeviceAddressFeatures-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES

The VkPhysicalDeviceImagelessFramebufferFeatures structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceImagelessFramebufferFeatures {
    VkStructureType  sType;
    void*            pNext;
    VkBool32         imagelessFramebuffer;
} VkPhysicalDeviceImagelessFramebufferFeatures;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **imagelessFramebuffer** indicates that the implementation supports specifying the image view for attachments at render pass begin time via `VkRenderPassAttachmentBeginInfo`.

If the VkPhysicalDeviceImagelessFramebufferFeatures structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceImagelessFramebufferFeatures` can also be used in the pNext chain of `VkDeviceCreateInfo` to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceImagelessFramebufferFeatures-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGELESS_FRAMEBUFFER_FEATURES

The VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures {
    VkStructureType  sType;
    void*            pNext;
    VkBool32         shaderSubgroupExtendedTypes;
} VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures;
```

This structure describes the following feature:

- **sType** is the type of this structure.
• `pNext` is `NULL` or a pointer to a structure extending this structure.

• `shaderSubgroupExtendedTypes` is a boolean specifying whether subgroup operations can use 8-bit integer, 16-bit integer, 64-bit integer, 16-bit floating-point, and vectors of these types in group operations with subgroup scope, if the implementation supports the types.

If the `VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

**Valid Usage (Implicit)**

- `VUID-VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures-sType-sType` `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SUBGROUP_EXTENDED_TYPES_FEATURES`.

The `VkPhysicalDeviceHostQueryResetFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceHostQueryResetFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 hostQueryReset;
} VkPhysicalDeviceHostQueryResetFeatures;
```

This structure describes the following feature:

- `sType` is the type of this structure.

- `pNext` is `NULL` or a pointer to a structure extending this structure.

- `hostQueryReset` indicates that the implementation supports resetting queries from the host with `vkResetQueryPool`.

If the `VkPhysicalDeviceHostQueryResetFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceHostQueryResetFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

**Valid Usage (Implicit)**

- `VUID-VkPhysicalDeviceHostQueryResetFeatures-sType-sType` `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_HOST_QUERY_RESET_FEATURES`.

The `VkPhysicalDeviceTimelineSemaphoreFeatures` structure is defined as:
```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceTimelineSemaphoreFeatures {
    VkStructureType       sType;
    void*                 pNext;
    VkBool32              timelineSemaphore;
} VkPhysicalDeviceTimelineSemaphoreFeatures;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **timelineSemaphore** indicates whether semaphores created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_TIMELINE are supported.

If the `VkPhysicalDeviceTimelineSemaphoreFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceTimelineSemaphoreFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceTimelineSemaphoreFeatures-sType-sType
  - **sType** must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_FEATURES

The `VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures {
    VkStructureType       sType;
    void*                 pNext;
    VkBool32              separateDepthStencilLayouts;
} VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **separateDepthStencilLayouts** indicates whether the implementation supports a `VkImageMemoryBarrier` for a depth/stencil image with only one of `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT` set, and whether `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL`, or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL` can be used.
If the `VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures-sType-sType **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SEPARATE_DEPTH_STENCIL_LAYOUTS_FEATURES`

The `VkPhysicalDeviceShaderDemoteToHelperInvocationFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPhysicalDeviceShaderDemoteToHelperInvocationFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderDemoteToHelperInvocation;
} VkPhysicalDeviceShaderDemoteToHelperInvocationFeatures;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **shaderDemoteToHelperInvocation** indicates whether the implementation supports the SPIR-V `DemoteToHelperInvocationEXT` capability.

If the `VkPhysicalDeviceShaderDemoteToHelperInvocationFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceShaderDemoteToHelperInvocationFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceShaderDemoteToHelperInvocationFeatures-sType-sType **must** be `VK_STRUCTURE_TYPE_PHYSICAL DEVICE_SHADER_DEMOTE_TO_HELPER_INVOCATION_FEATURES`

The `VkPhysicalDeviceTextureCompressionASTCHDRFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPhysicalDeviceTextureCompressionASTCHDRFeatures {
```

---

1121
This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **textureCompressionASTC_HDR** indicates whether all of the ASTC HDR compressed texture formats are supported. If this feature is enabled, then the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `VK_FORMAT_FEATURE_BLIT_SRC_BIT` and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` features **must** be supported in `optimalTilingFeatures` for the following formats:
  - `VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK`
  - `VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK`
  - `VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK`
  - `VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK`
  - `VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK`
  - `VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK`
  - `VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK`
  - `VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK`
  - `VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK`
  - `VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK`
  - `VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK`
  - `VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK`
  - `VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK`
  - `VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK`

To query for additional properties, or if the feature is not enabled, `vkGetPhysicalDeviceFormatProperties` and `vkGetPhysicalDeviceImageFormatProperties` can be used to check for supported properties of individual formats as normal.

If the `VkPhysicalDeviceTextureCompressionASTCHDRFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceTextureCompressionASTCHDRFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceTextureCompressionASTCHDRFeatures-sType-sType**
  - `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXTURE_COMPRESSION_ASTC_HDR_FEATURES`
The `VkPhysicalDeviceSubgroupSizeControlFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPhysicalDeviceSubgroupSizeControlFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 subgroupSizeControl;
    VkBool32 computeFullSubgroups;
} VkPhysicalDeviceSubgroupSizeControlFeatures;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **subgroupSizeControl** indicates whether the implementation supports controlling shader subgroup sizes via the `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT` flag and the `VkPipelineShaderStageRequiredSubgroupSizeCreateInfo` structure.
- **computeFullSubgroups** indicates whether the implementation supports requiring full subgroups in compute shaders via the `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT` flag.

If the `VkPhysicalDeviceSubgroupSizeControlFeatures` structure is included in the **pNext** chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceSubgroupSizeControlFeatures` can also be used in the **pNext** chain of `VkDeviceCreateInfo` to selectively enable these features.

**Note**

The `VkPhysicalDeviceSubgroupSizeControlFeaturesEXT` structure was added in version 2 of the `VK_EXT_subgroup_size_control` extension. Version 1 implementations of this extension will not fill out the features structure but applications may assume that both **subgroupSizeControl** and **computeFullSubgroups** are supported if the extension is supported. (See also the Feature Requirements section.) Applications are advised to add a `VkPhysicalDeviceSubgroupSizeControlFeaturesEXT` structure to the **pNext** chain of `VkDeviceCreateInfo` to enable the features regardless of the version of the extension supported by the implementation. If the implementation only supports version 1, it will safely ignore the `VkPhysicalDeviceSubgroupSizeControlFeaturesEXT` structure.

Vulkan 1.3 implementations always support the features structure.

**Valid Usage (Implicit)**

- **VUID-VkPhysicalDeviceSubgroupSizeControlFeatures-sType-sType**
  *sType* must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_FEATURES`
The `VkPhysicalDevicePipelineCreationCacheControlFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPhysicalDevicePipelineCreationCacheControlFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 pipelineCreationCacheControl;
} VkPhysicalDevicePipelineCreationCacheControlFeatures;
```

This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `pipelineCreationCacheControl` indicates that the implementation supports:
  - The following can be used in `Vk*PipelineCreateInfo::flags`:
    - `VK_PIPELINE_CREATE_FAIL_ON_PIPELINE_COMPILE_REQUIRED_BIT`
    - `VK_PIPELINE_CREATE_EARLY_RETURN_ON_FAILURE_BIT`
  - The following can be used in `VkPipelineCacheCreateInfo::flags`:
    - `VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT`

If the `VkPhysicalDevicePipelineCreationCacheControlFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDevicePipelineCreationCacheControlFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

Valid Usage (Implicit)

- `VUID-VkPhysicalDevicePipelineCreationCacheControlFeatures-sType-sType` must be `VK_STRUCTURE_TYPE_PHYSICAL DEVICE_PIPELINE CREATION CACHE CONTROL FEATURES`

The `VkPhysicalDeviceZeroInitializeWorkgroupMemoryFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPhysicalDeviceZeroInitializeWorkgroupMemoryFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderZeroInitializeWorkgroupMemory;
} VkPhysicalDeviceZeroInitializeWorkgroupMemoryFeatures;
```

This structure describes the following feature:
• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **shaderZeroInitializeWorkgroupMemory** specifies whether the implementation supports initializing a variable in Workgroup storage class.

If the **VkPhysicalDeviceZeroInitializeWorkgroupMemoryFeatures** structure is included in the **pNext** chain of the **VkPhysicalDeviceFeatures2** structure passed to **vkGetPhysicalDeviceFeatures2**, it is filled in to indicate whether each corresponding feature is supported. **VkPhysicalDeviceZeroInitializeWorkgroupMemoryFeatures can** also be used in the **pNext** chain of **VkDeviceCreateInfo** to selectively enable these features.

---

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceZeroInitializeWorkgroupMemoryFeatures-sType-sType**
  - **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ZERO_INITIALIZE_WORKGROUP_MEMORY_FEATURES**

---

The **VkPhysicalDevicePrivateDataFeatures** structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPhysicalDevicePrivateDataFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 privateData;
} VkPhysicalDevicePrivateDataFeatures;
```

This structure describes the following feature:

• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **privateData** indicates whether the implementation supports private data. See **Private Data**.

If the **VkPhysicalDevicePrivateDataFeatures** structure is included in the **pNext** chain of the **VkPhysicalDeviceFeatures2** structure passed to **vkGetPhysicalDeviceFeatures2**, it is filled in to indicate whether each corresponding feature is supported. **VkPhysicalDevicePrivateDataFeatures can** also be used in the **pNext** chain of **VkDeviceCreateInfo** to selectively enable these features.

---

### Valid Usage (Implicit)

- **VUID-VkPhysicalDevicePrivateDataFeatures-sType-sType**
  - **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PRIVATE_DATA_FEATURES**

nullDescriptor support requires the **VK_EXT_robustness2** extension.
The `VkPhysicalDeviceImageRobustnessFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPhysicalDeviceImageRobustnessFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 robustImageAccess;
} VkPhysicalDeviceImageRobustnessFeatures;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **robustImageAccess** indicates whether image accesses are tightly bounds-checked against the dimensions of the image view. Invalid texels resulting from out of bounds image loads will be replaced as described in Texel Replacement, with either (0,0,1) or (0,0,0) values inserted for missing G, B, or A components based on the format.

If the `VkPhysicalDeviceImageRobustnessFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceImageRobustnessFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceImageRobustnessFeatures-sType-sType
  
  **sType must be** VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_ROBUSTNESS_FEATURES

The `VkPhysicalDeviceShaderTerminateInvocationFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPhysicalDeviceShaderTerminateInvocationFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderTerminateInvocation;
} VkPhysicalDeviceShaderTerminateInvocationFeatures;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **shaderTerminateInvocation** specifies whether the implementation supports SPIR-V modules that use the SPV_KHR_terminate_invocation extension.
If the `VkPhysicalDeviceShaderTerminateInvocationFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceShaderTerminateInvocationFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceShaderTerminateInvocationFeatures-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_TERMINATE_INVOCATION_FEATURES`

The `VkPhysicalDeviceSynchronization2Features` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPhysicalDeviceSynchronization2Features {  
  VkStructureType sType;
  void* pNext;
  VkBool32 synchronization2;
} VkPhysicalDeviceSynchronization2Features;
```

This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `synchronization2` indicates whether the implementation supports the new set of synchronization commands introduced in `VK_KHR_synchronization2`.

If the `VkPhysicalDeviceSynchronization2Features` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceSynchronization2Features` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceSynchronization2Features-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SYNCHRONIZATION_2_FEATURES`

The `VkPhysicalDeviceShaderIntegerDotProductFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPhysicalDeviceShaderIntegerDotProductFeatures {  
  VkStructureType sType;
  void* pNext;
  VkBool32 shaderIntegerDotProduct;
} VkPhysicalDeviceShaderIntegerDotProductFeatures;
```
This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.

- **shaderIntegerDotProduct** specifies whether shader modules can declare the DotProductInputAllKHR, DotProductInput4x8BitKHR, DotProductInput4x8BitPackedKHR and DotProductKHR capabilities.

If the `VkPhysicalDeviceShaderIntegerDotProductFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceShaderIntegerDotProductFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceShaderIntegerDotProductFeatures-sType-sType
  
  **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_INTEGER_DOT_PRODUCT_FEATURES**

The `VkPhysicalDeviceMaintenance4Features` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPhysicalDeviceMaintenance4Features {
    VkStructureType      sType;
    void*                pNext;
    VkBool32             maintenance4;
} VkPhysicalDeviceMaintenance4Features;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.

- **maintenance4** indicates that the implementation supports the following:
  
  - The application may destroy a **VkPipelineLayout** object immediately after using it to create another object.
  
  - **LocalSizeId** can be used as an alternative to **LocalSize** to specify the local workgroup size with specialization constants.
  
  - Images created with identical creation parameters will always have the same alignment requirements.
  
  - The size memory requirement of a buffer or image is never greater than that of another buffer or image created with a greater or equal size.
Push constants do not have to be initialized before they are dynamically accessed.

The interface matching rules allow a larger output vector to match with a smaller input vector, with additional values being discarded.

If the `VkPhysicalDeviceMaintenance4Features` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceMaintenance4Features` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceMaintenance4Features-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_4_FEATURES`

The `VkPhysicalDeviceDynamicRenderingFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPhysicalDeviceDynamicRenderingFeatures {
    VkStructureType   sType;
    void*             pNext;
    VkBool32          dynamicRendering;
} VkPhysicalDeviceDynamicRenderingFeatures;
```

This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `dynamicRendering` specifies that the implementation supports dynamic render pass instances using the `vkCmdBeginRendering` command.

If the `VkPhysicalDeviceDynamicRenderingFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceDynamicRenderingFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceDynamicRenderingFeatures-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DYNAMIC_RENDERING_FEATURES`

### 32.1. Feature Requirements

All Vulkan graphics implementations must support the following features:
• robustBufferAccess
• multiview, if Vulkan 1.1 is supported.
• uniformBufferStandardLayout, if Vulkan 1.2 or the VK_KHR_uniform_buffer_standard_layout extension is supported.
• storageBuffer8BitAccess, if uniformAndStorageBuffer8BitAccess is enabled.
• If the descriptorIndexing feature is supported, or if the VK_EXT_descriptor_indexing extension is supported:
  ◦ shaderSampledImageArrayDynamicIndexing
  ◦ shaderStorageBufferArrayDynamicIndexing
  ◦ shaderUniformTexelBufferArrayDynamicIndexing
  ◦ shaderStorageTexelBufferArrayDynamicIndexing
  ◦ shaderSampledImageArrayNonUniformIndexing
  ◦ shaderStorageBufferArrayNonUniformIndexing
  ◦ shaderUniformTexelBufferArrayNonUniformIndexing
  ◦ descriptorBindingSampledImageUpdateAfterBind
  ◦ descriptorBindingStorageImageUpdateAfterBind
  ◦ descriptorBindingStorageBufferUpdateAfterBind (see also robustBufferAccessUpdateAfterBind)
  ◦ descriptorBindingUniformTexelBufferUpdateAfterBind (see also robustBufferAccessUpdateAfterBind)
  ◦ descriptorBindingStorageTexelBufferUpdateAfterBind (see also robustBufferAccessUpdateAfterBind)
  ◦ descriptorBindingUpdateUnusedWhilePending
  ◦ descriptorBindingPartiallyBound
  ◦ runtimeDescriptorArray
• If Vulkan 1.3 is supported:
  ◦ vulkanMemoryModel
  ◦ vulkanMemoryModelDeviceScope
• inlineUniformBlock, if Vulkan 1.3 or the VK_EXT_inline_uniform_block extension is supported.
• descriptorBindingInlineUniformBlockUpdateAfterBind, if Vulkan 1.3 or the VK_EXT_inline_uniform_block extension is supported; and if the descriptorIndexing feature is supported, or the VK_EXT_descriptor_indexing extension is supported.
• subgroupBroadcastDynamicId, if Vulkan 1.2 is supported.
• subgroupSizeControl, if Vulkan 1.3 or the VK_EXT_subgroup_size_control extension is supported.
• computeFullSubgroups, if Vulkan 1.3 or the VK_EXT_subgroup_size_control extension is supported.
• imagelessFramebuffer, if Vulkan 1.2 or the VK_KHR_imageless_framebuffer extension is supported.
• separateDepthStencilLayouts, if Vulkan 1.2 or the VK_KHR_separate_depthStencil_layouts extension is supported.
• hostQueryReset, if Vulkan 1.2 or the VK_EXT_host_query_reset extension is supported.
• timelineSemaphore, if Vulkan 1.2 or the VK_KHR_timeline_semaphore extension is supported.
• pipelineCreationCacheControl, if Vulkan 1.3 or the VK_EXT_pipeline_creation_cache_control extension is supported.
• shaderSubgroupExtendedTypes, if Vulkan 1.2 or the VK_KHR_shader_subgroup_extended_types extension is supported.
• textureCompressionASTC_HDR, if the VK_EXT_texture_compression_astc_hdr extension is supported.
• shaderDemoteToHelperInvocation, if Vulkan 1.3 or the VK_EXT_shader_demote_to_helper_invocation extension is supported.
• texelBufferAlignment, if Vulkan 1.3 or the VK_EXT_texel_buffer_alignment extension is supported.
• bufferDeviceAddress, if Vulkan 1.3 or the VK_KHR_buffer_device_address extension is supported.
• shaderInt64, if the shaderSharedInt64Atomics or shaderBufferInt64Atomics features are supported.
• storageBuffer16BitAccess, if uniformAndStorageBuffer16BitAccess is enabled.
• robustImageAccess, if Vulkan 1.3 or the VK_EXT_image_robustness extension is supported.
• shaderTerminateInvocation if Vulkan 1.3 or the VK_KHR_shader_terminate_invocation extension is supported.
• shaderZeroInitializeWorkgroupMemory, if Vulkan 1.3 or the VK_KHR_zero_initialize_workgroup_memory extension is supported.
• synchronization2 if Vulkan 1.3 or the VK_KHR_synchronization2 extension is supported.
• shaderIntegerDotProduct if Vulkan 1.3 or the VK_KHR_shader_integer_dot_product extension is supported.
• maintenance4, if Vulkan 1.3 or the VK_KHR_maintenance4 extension is supported.
• privateData, if Vulkan 1.3 or the VK_EXT_private_data extension is supported.
• dynamicRendering, if Vulkan 1.3 or the VK_KHR_dynamic_rendering extension is supported.

All other features defined in the Specification are optional.

32.2. Profile Features

32.2.1. Roadmap 2022

Implementations that claim support for the Roadmap 2022 profile must support the following features:

• fullDrawIndexUint32
• imageCubeArray
• independentBlend
• sampleRateShading
• drawIndirectFirstInstance
• depthClamp
• depthBiasClamp
• samplerAnisotropy
• occlusionQueryPrecise
• fragmentStoresAndAtomics
• shaderStorageImageExtendedFormats
• shaderUniformBufferArrayDynamicIndexing
• shaderSampledImageArrayDynamicIndexing
• shaderStorageBufferArrayDynamicIndexing
• shaderStorageImageArrayDynamicIndexing
• samplerYcbcrConversion
• samplerMirrorClampToEdge
• descriptorIndexing
• shaderUniformTexelBufferArrayDynamicIndexing
• shaderStorageTexelBufferArrayDynamicIndexing
• shaderUniformBufferArrayNonUniformIndexing
• shaderSampledImageArrayNonUniformIndexing
• shaderStorageBufferArrayNonUniformIndexing
• shaderStorageImageArrayNonUniformIndexing
• shaderUniformTexelBufferArrayNonUniformIndexing
• shaderStorageTexelBufferArrayNonUniformIndexing
• descriptorBindingSampledImageUpdateAfterBind
• descriptorBindingStorageImageUpdateAfterBind
• descriptorBindingStorageBufferUpdateAfterBind
• descriptorBindingUniformTexelBufferUpdateAfterBind
• descriptorBindingStorageTexelBufferUpdateAfterBind
• descriptorBindingUpdateUnusedWhilePending
• descriptorBindingPartiallyBound
• descriptorBindingVariableDescriptorCount
• runtimeDescriptorArray
• scalarBlockLayout
Chapter 33. Limits

Limits are implementation-dependent minimums, maximums, and other device characteristics that an application may need to be aware of.

**Note**

Limits are reported via the basic `VkPhysicalDeviceLimits` structure as well as the extensible structure `VkPhysicalDeviceProperties2`, which was added in `VK_KHR_get_physical_device_properties2` and included in Vulkan 1.1. When limits are added in future Vulkan versions or extensions, each extension should introduce one new limit structure, if needed. This structure can be added to the `pNext` chain of the `VkPhysicalDeviceProperties2` structure.

The `VkPhysicalDeviceLimits` structure is defined as:

```c
// Provided by VK_VERSION_1_0
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;
The VkPhysicalDeviceLimits are properties of the physical device. These are available in the limits member of the VkPhysicalDeviceProperties structure which is returned from vkGetPhysicalDeviceProperties.

- **maxImageDimension1D** is the largest dimension (width) that is guaranteed to be supported for all images created with an imageType of VK_IMAGE_TYPE_1D. Some combinations of image parameters (format, usage, etc.) may allow support for larger dimensions, which can be queried using vkGetPhysicalDeviceImageFormatProperties.

- **maxImageDimension2D** is the largest dimension (width or height) that is guaranteed to be supported for all images created with an imageType of VK_IMAGE_TYPE_2D and without VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT set in flags. Some combinations of image parameters (format, usage, etc.) may allow support for larger dimensions, which can be queried using vkGetPhysicalDeviceImageFormatProperties.

- **maxImageDimension3D** is the largest dimension (width, height, or depth) that is guaranteed to be supported for all images created with an imageType of VK_IMAGE_TYPE_3D. Some combinations of image parameters (format, usage, etc.) may allow support for larger dimensions, which can be queried using vkGetPhysicalDeviceImageFormatProperties.

- **maxImageDimensionCube** is the largest dimension (width or height) that is guaranteed to be supported for all images created with an imageType of VK_IMAGE_TYPE_2D and with VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT set in flags. Some combinations of image parameters (format, usage, etc.) may allow support for larger dimensions, which can be queried using vkGetPhysicalDeviceImageFormatProperties.
• **maxImageArrayLayers** is the maximum number of layers (arrayLayers) for an image.

• **maxTexelBufferElements** is the maximum number of addressable texels for a buffer view created on a buffer which was created with the `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT` or `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT` set in the `usage` member of the `VkBufferCreateInfo` structure.

• **maxUniformBufferRange** is the maximum value that can be specified in the `range` member of a `VkDescriptorBufferInfo` structure passed to `vkUpdateDescriptorSets` for descriptors of type `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`.

• **maxStorageBufferRange** is the maximum value that can be specified in the `range` member of a `VkDescriptorBufferInfo` structure passed to `vkUpdateDescriptorSets` for descriptors of type `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`.

• **maxPushConstantsSize** is the maximum size, in bytes, of the pool of push constant memory. For each of the push constant ranges indicated by the `pPushConstantRanges` member of the `VkPipelineLayoutCreateInfo` structure, \((\text{offset} + \text{size})\) must be less than or equal to this limit.

• **maxMemoryAllocationCount** is the maximum number of device memory allocations, as created by `vkAllocateMemory`, which can simultaneously exist.

• **maxSamplerAllocationCount** is the maximum number of sampler objects, as created by `vkCreateSampler`, which can simultaneously exist on a device.

• **bufferImageGranularity** is the granularity, in bytes, at which buffer or linear image resources, and optimal image resources can be bound to adjacent offsets in the same `VkDeviceMemory` object without aliasing. See Buffer-Image Granularity for more details.

• **sparseAddressSpaceSize** is the total amount of address space available, in bytes, for sparse memory resources. This is an upper bound on the sum of the sizes of all sparse resources, regardless of whether any memory is bound to them.

• **maxBoundDescriptorSets** is the maximum number of descriptor sets that can be simultaneously used by a pipeline. All `DescriptorSet` decorations in shader modules must have a value less than `maxBoundDescriptorSets`. See Descriptor Sets.

• **maxPerStageDescriptorSamplers** is the maximum number of samplers that can be accessible to a single shader stage in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_SAMPLER` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set count against this limit. A descriptor is accessible to a shader stage when the `stageFlags` member of the `VkDescriptorSetLayoutBinding` structure has the bit for that shader stage set. See Sampler and Combined Image Sampler.

• **maxPerStageDescriptorUniformBuffers** is the maximum number of uniform buffers that can be accessible to a single shader stage in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set count against this limit. A descriptor is accessible to a shader stage when the `stageFlags` member of the `VkDescriptorSetLayoutBinding` structure has the bit for that shader stage set. See Uniform Buffer and Dynamic Uniform Buffer.
• **maxPerStageDescriptorStorageBuffers** is the maximum number of storage buffers that can be accessible to a single shader stage in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set count against this limit. A descriptor is accessible to a pipeline shader stage when the `stageFlags` member of the `VkDescriptorSetLayoutBinding` structure has the bit for that shader stage set. See Storage Buffer and Dynamic Storage Buffer.

• **maxPerStageDescriptorSampledImages** is the maximum number of sampled images that can be accessible to a single shader stage in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, or `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set count against this limit. A descriptor is accessible to a pipeline shader stage when the `stageFlags` member of the `VkDescriptorSetLayoutBinding` structure has the bit for that shader stage set. See Combined Image Sampler, Sampled Image, and Uniform Texel Buffer.

• **maxPerStageDescriptorStorageImages** is the maximum number of storage images that can be accessible to a single shader stage in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, or `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set count against this limit. A descriptor is accessible to a pipeline shader stage when the `stageFlags` member of the `VkDescriptorSetLayoutBinding` structure has the bit for that shader stage set. See Storage Image, and Storage Texel Buffer.

• **maxPerStageDescriptorInputAttachments** is the maximum number of input attachments that can be accessible to a single shader stage in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set count against this limit. A descriptor is accessible to a pipeline shader stage when the `stageFlags` member of the `VkDescriptorSetLayoutBinding` structure has the bit for that shader stage set. These are only supported for the fragment stage. See Input Attachment.

• **maxPerStageResources** is the maximum number of resources that can be accessible to a single shader stage in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER`, `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER`, `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`, `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER`, `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`, or `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set count against this limit. For the fragment shader stage the framebuffer color attachments also count against this limit.

• **maxDescriptorSetSamplers** is the maximum number of samplers that can be included in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_SAMPLER` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set count against this limit. See
Sampler and Combined Image Sampler.

- **maxDescriptorSetUniformBuffers** is the maximum number of uniform buffers that can be included in a pipeline layout. Descriptors with a type of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC count against this limit. Only descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit. See Uniform Buffer and Dynamic Uniform Buffer.

- **maxDescriptorSetUniformBuffersDynamic** is the maximum number of dynamic uniform buffers that can be included in a pipeline layout. Descriptors with a type of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC count against this limit. Only descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit. See Dynamic Uniform Buffer.

- **maxDescriptorSetStorageBuffers** is the maximum number of storage buffers that can be included in a pipeline layout. Descriptors with a type of VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC count against this limit. Only descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit. See Storage Buffer and Dynamic Storage Buffer.

- **maxDescriptorSetStorageBuffersDynamic** is the maximum number of dynamic storage buffers that can be included in a pipeline layout. Descriptors with a type of VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC count against this limit. Only descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit. See Dynamic Storage Buffer.

- **maxDescriptorSetSampledImages** is the maximum number of sampled images that can be included in a pipeline layout. Descriptors with a type of VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, or VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER count against this limit. Only descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit. See Combined Image Sampler, Sampled Image, and Uniform Texel Buffer.

- **maxDescriptorSetStorageImages** is the maximum number of storage images that can be included in a pipeline layout. Descriptors with a type of VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, or VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER count against this limit. Only descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit. See Storage Image, and Storage Texel Buffer.

- **maxDescriptorSetInputAttachments** is the maximum number of input attachments that can be included in a pipeline layout. Descriptors with a type of VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT count against this limit. Only descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit. See Input Attachment.

- **maxVertexInputAttributes** is the maximum number of vertex input attributes that can be specified for a graphics pipeline. These are described in the array of VkVertexInputAttributeDescription structures that are provided at graphics pipeline creation time via the pVertexAttributeDescriptions member of the VkPipelineVertexInputStateCreateInfo.
structure. See Vertex Attributes and Vertex Input Description.

- **maxVertexInputBindings** is the maximum number of vertex buffers that can be specified for providing vertex attributes to a graphics pipeline. These are described in the array of VkVertexInputBindingDescription structures that are provided at graphics pipeline creation time via the pVertexBindingDescriptions member of the VkPipelineVertexInputStateCreateInfo structure. The binding member of VkVertexInputBindingDescription must be less than this limit. See Vertex Input Description.

- **maxVertexInputAttributeOffset** is the maximum vertex input attribute offset that can be added to the vertex input binding stride. The offset member of the VkVertexInputAttributeDescription structure must be less than or equal to this limit. See Vertex Input Description.

- **maxVertexInputBindingStride** is the maximum vertex input binding stride that can be specified in a vertex input binding. The stride member of the VkVertexInputBindingDescription structure must be less than or equal to this limit. See Vertex Input Description.

- **maxVertexOutputComponents** is the maximum number of components of output variables which can be output by a vertex shader. See Vertex Shaders.

- **maxTessellationGenerationLevel** is the maximum tessellation generation level supported by the fixed-function tessellation primitive generator. See Tessellation.

- **maxTessellationPatchSize** is the maximum patch size, in vertices, of patches that can be processed by the tessellation control shader and tessellation primitive generator. The patchControlPoints member of the VkPipelineTessellationStateCreateInfo structure specified at pipeline creation time and the value provided in the OutputVertices execution mode of shader modules must be less than or equal to this limit. See Tessellation.

- **maxTessellationControlPerVertexInputComponents** is the maximum number of components of input variables which can be provided as per-vertex inputs to the tessellation control shader stage.

- **maxTessellationControlPerVertexOutputComponents** is the maximum number of components of per-vertex output variables which can be output from the tessellation control shader stage.

- **maxTessellationControlPerPatchOutputComponents** is the maximum number of components of per-patch output variables which can be output from the tessellation control shader stage.

- **maxTessellationControlTotalOutputComponents** is the maximum total number of components of per-vertex and per-patch output variables which can be output from the tessellation control shader stage.

- **maxTessellationEvaluationInputComponents** is the maximum number of components of input variables which can be provided as per-vertex inputs to the tessellation evaluation shader stage.

- **maxTessellationEvaluationOutputComponents** is the maximum number of components of per-vertex output variables which can be output from the tessellation evaluation shader stage.

- **maxGeometryShaderInvocations** is the maximum invocation count supported for instanced geometry shaders. The value provided in the Invocations execution mode of shader modules must be less than or equal to this limit. See Geometry Shading.

- **maxGeometryInputComponents** is the maximum number of components of input variables which can be provided as inputs to the geometry shader stage.
• **maxGeometryOutputComponents** is the maximum number of components of output variables which can be output from the geometry shader stage.

• **maxGeometryOutputVertices** is the maximum number of vertices which can be emitted by any geometry shader.

• **maxGeometryTotalOutputComponents** is the maximum total number of components of output variables, across all emitted vertices, which can be output from the geometry shader stage.

• **maxFragmentInputComponents** is the maximum number of components of input variables which can be provided as inputs to the fragment shader stage.

• **maxFragmentOutputAttachments** is the maximum number of output attachments which can be written to by the fragment shader stage.

• **maxFragmentDualSrcAttachments** is the maximum number of output attachments which can be written to by the fragment shader stage when blending is enabled and one of the dual source blend modes is in use. See Dual-Source Blending and dualSrcBlend.

• **maxFragmentCombinedOutputResources** is the total number of storage buffers, storage images, and output Location decorated color attachments (described in Fragment Output Interface) which can be used in the fragment shader stage.

• **maxComputeSharedMemorySize** is the maximum total storage size, in bytes, available for variables declared with the Workgroup storage class in shader modules (or with the shared storage qualifier in GLSL) in the compute shader stage. The amount of storage consumed by the variables declared with the Workgroup storage class is implementation-dependent. However, the amount of storage consumed may not exceed the largest block size that would be obtained if all active variables declared with Workgroup storage class were assigned offsets in an arbitrary order by successively taking the smallest valid offset according to the Standard Storage Buffer Layout rules, and with Boolean values considered as 32-bit integer values for the purpose of this calculation. (This is equivalent to using the GLSL std430 layout rules.)

• **maxComputeWorkGroupCount[3]** is the maximum number of local workgroups that can be dispatched by a single dispatching command. These three values represent the maximum number of local workgroups for the X, Y, and Z dimensions, respectively. The workgroup count parameters to the dispatching commands must be less than or equal to the corresponding limit. See Dispatching Commands.

• **maxComputeWorkGroupInvocations** is the maximum total number of compute shader invocations in a single local workgroup. The product of the X, Y, and Z sizes, as specified by the LocalSize or LocalSizeId execution mode in shader modules or by the object decorated by the WorkgroupSize decoration, must be less than or equal to this limit.

• **maxComputeWorkGroupSize[3]** is the maximum size of a local compute workgroup, per dimension. These three values represent the maximum local workgroup size in the X, Y, and Z dimensions, respectively. The x, y, and z sizes, as specified by the LocalSize or LocalSizeId execution mode or by the object decorated by the WorkgroupSize decoration in shader modules, must be less than or equal to the corresponding limit.

• **subPixelPrecisionBits** is the number of bits of subpixel precision in framebuffer coordinates \( x_f \) and \( y_f \). See Rasterization.

• **subTexelPrecisionBits** is the number of bits of precision in the division along an axis of an image used for minification and magnification filters. \( 2^{\text{subTexelPrecisionBits}} \) is the actual number of
divisions along each axis of the image represented. Sub-texel values calculated during image sampling will snap to these locations when generating the filtered results.

- **mipmapPrecisionBits** is the number of bits of division that the LOD calculation for mipmap fetching get snapped to when determining the contribution from each mip level to the mip filtered results. $2^{\text{mipmapPrecisionBits}}$ is the actual number of divisions.

- **maxDrawIndexedIndexValue** is the maximum index value that can be used for indexed draw calls when using 32-bit indices. This excludes the primitive restart index value of 0xFFFFFFFF. See [fullDrawIndexUint32].

- **maxDrawIndirectCount** is the maximum draw count that is supported for indirect drawing calls. See [multiDrawIndirect].

- **maxSamplerLodBias** is the maximum absolute sampler LOD bias. The sum of the mipLodBias member of the VkSamplerCreateInfo structure and the Bias operand of image sampling operations in shader modules (or 0 if no Bias operand is provided to an image sampling operation) are clamped to the range $[-\text{maxSamplerLodBias}, +\text{maxSamplerLodBias}]$. See [samplers-mipLodBias].

- **maxSamplerAnisotropy** is the maximum degree of sampler anisotropy. The maximum degree of anisotropic filtering used for an image sampling operation is the minimum of the maxAnisotropy member of the VkSamplerCreateInfo structure and this limit. See [samplers-maxAnisotropy].

- **maxViewports** is the maximum number of active viewports. The viewportCount member of the VkPipelineViewportStateCreateInfo structure that is provided at pipeline creation must be less than or equal to this limit.

- **maxViewportDimensions[2]** are the maximum viewport dimensions in the X (width) and Y (height) dimensions, respectively. The maximum viewport dimensions must be greater than or equal to the largest image which can be created and used as a framebuffer attachment. See Controlling the Viewport.

- **viewportBoundsRange[2]** is the $[\text{minimum, maximum}]$ range that the corners of a viewport must be contained in. This range must be at least $[-2 \times \text{size}, 2 \times \text{size} - 1]$, where $\text{size} = \max(\text{maxViewportDimensions}[0], \text{maxViewportDimensions}[1])$. See Controlling the Viewport.

  **Note**
  The intent of the viewportBoundsRange limit is to allow a maximum sized viewport to be arbitrarily shifted relative to the output target as long as at least some portion intersects. This would give a bounds limit of $[-\text{size} + 1, 2 \times \text{size} - 1]$ which would allow all possible non-empty-set intersections of the output target and the viewport. Since these numbers are typically powers of two, picking the signed number range using the smallest possible number of bits ends up with the specified range.

- **viewportSubPixelBits** is the number of bits of subpixel precision for viewport bounds. The subpixel precision that floating-point viewport bounds are interpreted at is given by this limit.

- **minMemoryMapAlignment** is the minimum required alignment, in bytes, of host visible memory allocations within the host address space. When mapping a memory allocation with vkMapMemory, subtracting offset bytes from the returned pointer will always produce an integer multiple of this limit. See Host Access to Device Memory Objects. The value must be a
power of two.

- **minTexelBufferOffsetAlignment** is the minimum **required** alignment, in bytes, for the **offset** member of the `VkBufferViewCreateInfo` structure for texel buffers. The value **must** be a power of two. If `texelBufferAlignment` is enabled, this limit is equivalent to the maximum of the `uniformTexelBufferOffsetAlignmentBytes` and `storageTexelBufferOffsetAlignmentBytes` members of `VkPhysicalDeviceTexelBufferAlignmentProperties`, but smaller alignment is **optionally** allowed by `storageTexelBufferOffsetSingleTexelAlignment` and `uniformTexelBufferOffsetSingleTexelAlignment`. If `texelBufferAlignment` is not enabled, `VkBufferViewCreateInfo::offset` **must** be a multiple of this value.

- **minUniformBufferOffsetAlignment** is the minimum **required** alignment, in bytes, for the **offset** member of the `VkDescriptorBufferInfo` structure for uniform buffers. When a descriptor of type `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` is updated, the **offset** **must** be an integer multiple of this limit. Similarly, dynamic offsets for uniform buffers **must** be multiples of this limit. The value **must** be a power of two.

- **minStorageBufferOffsetAlignment** is the minimum **required** alignment, in bytes, for the **offset** member of the `VkDescriptorBufferInfo` structure for storage buffers. When a descriptor of type `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` is updated, the **offset** **must** be an integer multiple of this limit. Similarly, dynamic offsets for storage buffers **must** be multiples of this limit. The value **must** be a power of two.

- **minTexelOffset** is the minimum offset value for the `ConstOffset` image operand of any of the `OpImageSample*` or `OpImageFetch*` image instructions.

- **maxTexelOffset** is the maximum offset value for the `ConstOffset` image operand of any of the `OpImageSample*` or `OpImageFetch*` image instructions.

- **minTexelGatherOffset** is the minimum offset value for the `Offset`, `ConstOffset`, or `ConstOffsets` image operands of any of the `OpImage*Gather` image instructions.

- **maxTexelGatherOffset** is the maximum offset value for the `Offset`, `ConstOffset`, or `ConstOffsets` image operands of any of the `OpImage*Gather` image instructions.

- **minInterpolationOffset** is the base minimum (inclusive) negative offset value for the **offset** operand of the `InterpolateAtOffset` extended instruction.

- **maxInterpolationOffset** is the base maximum (inclusive) positive offset value for the **offset** operand of the `InterpolateAtOffset` extended instruction.

- **subPixelInterpolationOffsetBits** is the number of fractional bits that the *x* and *y* offsets to the `InterpolateAtOffset` extended instruction **may** be rounded to as fixed-point values.

- **maxFramebufferWidth** is the maximum width for a framebuffer. The **width** member of the `VkFramebufferCreateInfo` structure **must** be less than or equal to this limit.

- **maxFramebufferHeight** is the maximum height for a framebuffer. The **height** member of the `VkFramebufferCreateInfo` structure **must** be less than or equal to this limit.

- **maxFramebufferLayers** is the maximum layer count for a layered framebuffer. The **layers** member of the `VkFramebufferCreateInfo` structure **must** be less than or equal to this limit.

- **framebufferColorSampleCounts** is a bitmask\(^1\) of `VkSampleCountFlagBits` indicating the color sample counts that are supported for all framebuffer color attachments with floating- or fixed-point formats. For color attachments with integer formats, see
framebufferIntegerColorSampleCounts.

• framebufferDepthSampleCounts is a bitmask of VkSampleCountFlagBits indicating the supported depth sample counts for all framebuffer depth/stencil attachments, when the format includes a depth component.

• framebufferStencilSampleCounts is a bitmask of VkSampleCountFlagBits indicating the supported stencil sample counts for all framebuffer depth/stencil attachments, when the format includes a stencil component.

• framebufferNoAttachmentsSampleCounts is a bitmask of VkSampleCountFlagBits indicating the supported sample counts for a subpass which uses no attachments.

• maxColorAttachments is the maximum number of color attachments that can be used by a subpass in a render pass. The colorAttachmentCount member of the VkSubpassDescription or VkSubpassDescription2 structure must be less than or equal to this limit.

• sampledImageColorSampleCounts is a bitmask of VkSampleCountFlagBits indicating the sample counts supported for all 2D images created with VK_IMAGE_TILING_OPTIMAL, usage containing VK_IMAGE_USAGE_SAMPLED_BIT, and a non-integer color format.

• sampledImageIntegerSampleCounts is a bitmask of VkSampleCountFlagBits indicating the sample counts supported for all 2D images created with VK_IMAGE_TILING_OPTIMAL, usage containing VK_IMAGE_USAGE_SAMPLED_BIT, and an integer color format.

• sampledImageDepthSampleCounts is a bitmask of VkSampleCountFlagBits indicating the sample counts supported for all 2D images created with VK_IMAGE_TILING_OPTIMAL, usage containing VK_IMAGE_USAGE_SAMPLED_BIT, and a depth format.

• sampledImageStencilSampleCounts is a bitmask of VkSampleCountFlagBits indicating the sample counts supported for all 2D images created with VK_IMAGE_TILING_OPTIMAL, usage containing VK_IMAGE_USAGE_SAMPLED_BIT, and a stencil format.

• storageImageSampleCounts is a bitmask of VkSampleCountFlagBits indicating the sample counts supported for all 2D images created with VK_IMAGE_TILING_OPTIMAL, and usage containing VK_IMAGE_USAGE_STORAGE_BIT.

• maxSampleMaskWords is the maximum number of array elements of a variable decorated with the SampleMask built-in decoration.

• timestampComputeAndGraphics specifies support for timestamps on all graphics and compute queues. If this limit is set to VK_TRUE, all queues that advertise the VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT in the VkQueueFamilyProperties::queueFlags support VkQueueFamilyProperties::timestampValidBits of at least 36. See Timestamp Queries.

• timestampPeriod is the number of nanoseconds required for a timestamp query to be incremented by 1. See Timestamp Queries.

• maxClipDistances is the maximum number of clip distances that can be used in a single shader stage. The size of any array declared with the ClipDistance built-in decoration in a shader module must be less than or equal to this limit.

• maxCullDistances is the maximum number of cull distances that can be used in a single shader stage. The size of any array declared with the CullDistance built-in decoration in a shader module must be less than or equal to this limit.

• maxCombinedClipAndCullDistances is the maximum combined number of clip and cull distancess.
that can be used in a single shader stage. The sum of the sizes of any pair of arrays declared with the `ClipDistance` and `CullDistance` built-in decoration used by a single shader stage in a shader module must be less than or equal to this limit.

- **discreteQueuePriorities** is the number of discrete priorities that can be assigned to a queue based on the value of each member of `VkDeviceQueueCreateInfo::pQueuePriorities`. This must be at least 2, and levels must be spread evenly over the range, with at least one level at 1.0, and another at 0.0. See Queue Priority.

- **pointSizeRange[2]** is the range `[minimum, maximum]` of supported sizes for points. Values written to variables decorated with the `PointSize` built-in decoration are clamped to this range.

- **lineWidthRange[2]** is the range `[minimum, maximum]` of supported widths for lines. Values specified by the `lineWidth` member of the `VkPipelineRasterizationStateCreateInfo` or the `lineWidth` parameter to `vkCmdSetLineWidth` are clamped to this range.

- **pointSizeGranularity** is the granularity of supported point sizes. Not all point sizes in the range defined by `pointSizeRange` are supported. This limit specifies the granularity (or increment) between successive supported point sizes.

- **lineWidthGranularity** is the granularity of supported line widths. Not all line widths in the range defined by `lineWidthRange` are supported. This limit specifies the granularity (or increment) between successive supported line widths.

- **strictLines** specifies whether lines are rasterized according to the preferred method of rasterization. If set to `VK_FALSE`, lines may be rasterized under a relaxed set of rules. If set to `VK_TRUE`, lines are rasterized as per the strict definition. See Basic Line Segment Rasterization.

- **standardSampleLocations** specifies whether rasterization uses the standard sample locations as documented in Multisampling. If set to `VK_TRUE`, the implementation uses the documented sample locations. If set to `VK_FALSE`, the implementation may use different sample locations.

- **optimalBufferCopyOffsetAlignment** is the optimal buffer offset alignment in bytes for `vkCmdCopyBufferToImage2`, `vkCmdCopyBufferToImage`, `vkCmdCopyImageToBuffer2`, and `vkCmdCopyImageToBuffer`. The per texel alignment requirements are enforced, but applications should use the optimal alignment for optimal performance and power use. The value must be a power of two.

- **optimalBufferCopyRowPitchAlignment** is the optimal buffer row pitch alignment in bytes for `vkCmdCopyBufferToImage2`, `vkCmdCopyBufferToImage`, `vkCmdCopyImageToBuffer2`, and `vkCmdCopyImageToBuffer`. Row pitch is the number of bytes between texels with the same X coordinate in adjacent rows (Y coordinates differ by one). The per texel alignment requirements are enforced, but applications should use the optimal alignment for optimal performance and power use. The value must be a power of two.

- **nonCoherentAtomSize** is the size and alignment in bytes that bounds concurrent access to host-mapped device memory. The value must be a power of two.

1 For all bitmasks of `VkSampleCountFlagBits`, the sample count limits defined above represent the minimum supported sample counts for each image type. Individual images may support additional sample counts, which are queried using `vkGetPhysicalDeviceImageFormatProperties` as described in Supported Sample Counts.
Bits which may be set in the sample count limits returned by `VkPhysicalDeviceLimits`, as well as in other queries and structures representing image sample counts, are:

```c
// Provided by VK_VERSION_1_0
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;
```

- `VK_SAMPLE_COUNT_1_BIT` specifies an image with one sample per pixel.
- `VK_SAMPLE_COUNT_2_BIT` specifies an image with 2 samples per pixel.
- `VK_SAMPLE_COUNT_4_BIT` specifies an image with 4 samples per pixel.
- `VK_SAMPLE_COUNT_8_BIT` specifies an image with 8 samples per pixel.
- `VK_SAMPLE_COUNT_16_BIT` specifies an image with 16 samples per pixel.
- `VK_SAMPLE_COUNT_32_BIT` specifies an image with 32 samples per pixel.
- `VK_SAMPLE_COUNT_64_BIT` specifies an image with 64 samples per pixel.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkSampleCountFlags;
```

`VkSampleCountFlags` is a bitmask type for setting a mask of zero or more `VkSampleCountFlagBits`.

The `VkPhysicalDeviceMultiviewProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceMultiviewProperties {
    VkStructureType sType;
    void* pNext;
    uint32_t maxMultiviewViewCount;
    uint32_t maxMultiviewInstanceIndex;
} VkPhysicalDeviceMultiviewProperties;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `maxMultiviewViewCount` is one greater than the maximum view index that can be used in a subpass.
- `maxMultiviewInstanceIndex` is the maximum valid value of instance index allowed to be generated by a drawing command recorded within a subpass of a multiview render pass.
If the `VkPhysicalDeviceMultiviewProperties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceMultiviewProperties-sType-sType
  
sType must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PROPERTIES`

The `VkPhysicalDeviceFloatControlsProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceFloatControlsProperties {
    VkStructureType sType;
    void* pNext;
    VkShaderFloatControlsIndependence denormBehaviorIndependence;
    VkShaderFloatControlsIndependence roundingModeIndependence;
    VkBool32 shaderSignedZeroInfNanPreserveFloat16;
    VkBool32 shaderSignedZeroInfNanPreserveFloat32;
    VkBool32 shaderSignedZeroInfNanPreserveFloat64;
    VkBool32 shaderDenormPreserveFloat16;
    VkBool32 shaderDenormPreserveFloat32;
    VkBool32 shaderDenormPreserveFloat64;
    VkBool32 shaderDenormFlushToZeroFloat16;
    VkBool32 shaderDenormFlushToZeroFloat32;
    VkBool32 shaderDenormFlushToZeroFloat64;
    VkBool32 shaderRoundingModeRTEFloat16;
    VkBool32 shaderRoundingModeRTEFloat32;
    VkBool32 shaderRoundingModeRTEFloat64;
    VkBool32 shaderRoundingModeRTZFloat16;
    VkBool32 shaderRoundingModeRTZFloat32;
    VkBool32 shaderRoundingModeRTZFloat64;
} VkPhysicalDeviceFloatControlsProperties;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **denormBehaviorIndependence** is a `VkShaderFloatControlsIndependence` value indicating whether, and how, denorm behavior can be set independently for different bit widths.
- **roundingModeIndependence** is a `VkShaderFloatControlsIndependence` value indicating whether, and how, rounding modes can be set independently for different bit widths.
- **shaderSignedZeroInfNanPreserveFloat16** is a boolean value indicating whether sign of a zero, Nans and ±∞ can be preserved in 16-bit floating-point computations. It also indicates whether the `SignedZeroInfNanPreserve` execution mode can be used for 16-bit floating-point types.
• `shaderSignedZeroInfNanPreserveFloat32` is a boolean value indicating whether sign of a zero, Nans and ±∞ can be preserved in 32-bit floating-point computations. It also indicates whether the `SignedZeroInfNanPreserve` execution mode can be used for 32-bit floating-point types.

• `shaderSignedZeroInfNanPreserveFloat64` is a boolean value indicating whether sign of a zero, Nans and ±∞ can be preserved in 64-bit floating-point computations. It also indicates whether the `SignedZeroInfNanPreserve` execution mode can be used for 64-bit floating-point types.

• `shaderDenormPreserveFloat16` is a boolean value indicating whether denormals can be preserved in 16-bit floating-point computations. It also indicates whether the `DenormPreserve` execution mode can be used for 16-bit floating-point types.

• `shaderDenormPreserveFloat32` is a boolean value indicating whether denormals can be preserved in 32-bit floating-point computations. It also indicates whether the `DenormPreserve` execution mode can be used for 32-bit floating-point types.

• `shaderDenormPreserveFloat64` is a boolean value indicating whether denormals can be preserved in 64-bit floating-point computations. It also indicates whether the `DenormPreserve` execution mode can be used for 64-bit floating-point types.

• `shaderDenormFlushToZeroFloat16` is a boolean value indicating whether denormals can be flushed to zero in 16-bit floating-point computations. It also indicates whether the `DenormFlushToZero` execution mode can be used for 16-bit floating-point types.

• `shaderDenormFlushToZeroFloat32` is a boolean value indicating whether denormals can be flushed to zero in 32-bit floating-point computations. It also indicates whether the `DenormFlushToZero` execution mode can be used for 32-bit floating-point types.

• `shaderDenormFlushToZeroFloat64` is a boolean value indicating whether denormals can be flushed to zero in 64-bit floating-point computations. It also indicates whether the `DenormFlushToZero` execution mode can be used for 64-bit floating-point types.

• `shaderRoundingModeRTEFloat16` is a boolean value indicating whether an implementation supports the round-to-nearest-even rounding mode for 16-bit floating-point arithmetic and conversion instructions. It also indicates whether the `RoundingModeRTE` execution mode can be used for 16-bit floating-point types.

• `shaderRoundingModeRTEFloat32` is a boolean value indicating whether an implementation supports the round-to-nearest-even rounding mode for 32-bit floating-point arithmetic and conversion instructions. It also indicates whether the `RoundingModeRTE` execution mode can be used for 32-bit floating-point types.

• `shaderRoundingModeRTEFloat64` is a boolean value indicating whether an implementation supports the round-to-nearest-even rounding mode for 64-bit floating-point arithmetic and conversion instructions. It also indicates whether the `RoundingModeRTE` execution mode can be used for 64-bit floating-point types.

• `shaderRoundingModeRTZFloat16` is a boolean value indicating whether an implementation supports the round-towards-zero rounding mode for 16-bit floating-point arithmetic and conversion instructions. It also indicates whether the `RoundingModeRTZ` execution mode can be used for 16-bit floating-point types.

• `shaderRoundingModeRTZFloat32` is a boolean value indicating whether an implementation supports the round-towards-zero rounding mode for 32-bit floating-point arithmetic and conversion instructions. It also indicates whether the `RoundingModeRTZ` execution mode can be used for 32-bit floating-point types.

• `shaderRoundingModeRTZFloat64` is a boolean value indicating whether an implementation supports the round-towards-zero rounding mode for 64-bit floating-point arithmetic and conversion instructions. It also indicates whether the `RoundingModeRTZ` execution mode can be used for 64-bit floating-point types.
used for 32-bit floating-point types.

- `shaderRoundingModeRTZFloat64` is a boolean value indicating whether an implementation supports the round-towards-zero rounding mode for 64-bit floating-point arithmetic and conversion instructions. It also indicates whether the `RoundingModeRTZ` execution mode can be used for 64-bit floating-point types.

If the `VkPhysicalDeviceFloatControlsProperties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceFloatControlsProperties-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FLOAT_CONTROLS_PROPERTIES`.

Values which may be returned in the `denormBehaviorIndependence` and `roundingModeIndependence` fields of `VkPhysicalDeviceFloatControlsProperties` are:

```c
// Provided by VK_VERSION_1_2
typedef enum VkShaderFloatControlsIndependence {
    VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_32_BIT_ONLY = 0,
    VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_ALL = 1,
    VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_NONE = 2,
} VkShaderFloatControlsIndependence;
```

- `VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_32_BIT_ONLY` specifies that shader float controls for 32-bit floating point can be set independently; other bit widths must be set identically to each other.
- `VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_ALL` specifies that shader float controls for all bit widths can be set independently.
- `VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_NONE` specifies that shader float controls for all bit widths must be set identically.

The `VkPhysicalDevicePointClippingProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDevicePointClippingProperties {
    VkStructureType sType;
    void* pNext;
    VkPointClippingBehavior pointClippingBehavior;
} VkPhysicalDevicePointClippingProperties;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `pointClippingBehavior` is a `VkPointClippingBehavior` value specifying the point clipping
behavior supported by the implementation.

If the `VkPhysicalDevicePointClippingProperties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

Valid Usage (Implicit)

- VUID-VkPhysicalDevicePointClippingProperties-sType-sType
  sType must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_POINT_CLIPPING_PROPERTIES`

The `VkPhysicalDeviceSubgroupProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceSubgroupProperties {
    VkStructureType sType;
    void* pNext;
    uint32_t subgroupSize;
    VkShaderStageFlags supportedStages;
    VkSubgroupFeatureFlags supportedOperations;
    VkBool32 quadOperationsInAllStages;
} VkPhysicalDeviceSubgroupProperties;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `subgroupSize` is the default number of invocations in each subgroup. `subgroupSize` is at least 1 if any of the physical device's queues support `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT`. `subgroupSize` is a power-of-two.
- `supportedStages` is a bitfield of `VkShaderStageFlagBits` describing the shader stages that group operations with subgroup scope are supported in. `supportedStages` will have the `VK_SHADER_STAGE_COMPUTE_BIT` bit set if any of the physical device's queues support `VK_QUEUE_COMPUTE_BIT`.
- `supportedOperations` is a bitmask of `VkSubgroupFeatureFlagBits` specifying the sets of group operations with subgroup scope supported on this device. `supportedOperations` will have the `VK_SUBGROUP_FEATURE_BASIC_BIT` bit set if any of the physical device's queues support `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT`.
- `quadOperationsInAllStages` is a boolean specifying whether quad group operations are available in all stages, or are restricted to fragment and compute stages.

If the `VkPhysicalDeviceSubgroupProperties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

If `supportedOperations` includes `VK_SUBGROUP_FEATURE_QUAD_BIT`, `subgroupSize` must be greater than or equal to 4.
Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceSubgroupProperties-sType-sType**
  
  **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_PROPERTIES**

Bits which **can** be set in **VkPhysicalDeviceSubgroupProperties::supportedOperations** and **VkPhysicalDeviceVulkan11Properties::subgroupSupportedOperations** to specify supported **group operations** with **subgroup scope** are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkSubgroupFeatureFlagBits {
    VK_SUBGROUP_FEATURE_BASIC_BIT = 0x00000001,
    VK_SUBGROUP_FEATURE_VOTE_BIT = 0x00000002,
    VK_SUBGROUP_FEATURE_ARITHMETIC_BIT = 0x00000004,
    VK_SUBGROUP_FEATURE_BALLOT_BIT = 0x00000008,
    VK_SUBGROUP_FEATURE_SHUFFLE_BIT = 0x00000010,
    VK_SUBGROUP_FEATURE_SHUFFLE_RELATIVE_BIT = 0x00000020,
    VK_SUBGROUP_FEATURE_CLUSTERED_BIT = 0x00000040,
    VK_SUBGROUP_FEATURE_QUAD_BIT = 0x00000080,
} VkSubgroupFeatureFlagBits;
```

- **VK_SUBGROUP_FEATURE_BASIC_BIT** specifies the device will accept SPIR-V shader modules containing the **GroupNonUniform** capability.
- **VK_SUBGROUP_FEATURE_VOTE_BIT** specifies the device will accept SPIR-V shader modules containing the **GroupNonUniformVote** capability.
- **VK_SUBGROUP_FEATURE_ARITHMETIC_BIT** specifies the device will accept SPIR-V shader modules containing the **GroupNonUniformArithmetic** capability.
- **VK_SUBGROUP_FEATURE_BALLOT_BIT** specifies the device will accept SPIR-V shader modules containing the **GroupNonUniformBallot** capability.
- **VK_SUBGROUP_FEATURE_SHUFFLE_BIT** specifies the device will accept SPIR-V shader modules containing the **GroupNonUniformShuffle** capability.
- **VK_SUBGROUP_FEATURE_SHUFFLE_RELATIVE_BIT** specifies the device will accept SPIR-V shader modules containing the **GroupNonUniformShuffleRelative** capability.
- **VK_SUBGROUP_FEATURE_CLUSTERED_BIT** specifies the device will accept SPIR-V shader modules containing the **GroupNonUniformClustered** capability.
- **VK_SUBGROUP_FEATURE_QUAD_BIT** specifies the device will accept SPIR-V shader modules containing the **GroupNonUniformQuad** capability.

```c
// Provided by VK_VERSION_1_1
typedef VkFlags VkSubgroupFeatureFlags;
```

**VkSubgroupFeatureFlags** is a bitmask type for setting a mask of zero or more **VkSubgroupFeatureFlagBits**.
The `VkPhysicalDeviceSubgroupSizeControlProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPhysicalDeviceSubgroupSizeControlProperties {
    VkStructureType sType;
    void* pNext;
    uint32_t minSubgroupSize;
    uint32_t maxSubgroupSize;
    uint32_t maxComputeWorkgroupSubgroups;
    VkShaderStageFlags requiredSubgroupSizeStages;
} VkPhysicalDeviceSubgroupSizeControlProperties;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **minSubgroupSize** is the minimum subgroup size supported by this device. `minSubgroupSize` is at least one if any of the physical device’s queues support `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT`. `minSubgroupSize` is a power-of-two. `minSubgroupSize` is less than or equal to `maxSubgroupSize`. `minSubgroupSize` is less than or equal to `subgroupSize`.
- **maxSubgroupSize** is the maximum subgroup size supported by this device. `maxSubgroupSize` is at least one if any of the physical device’s queues support `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT`. `maxSubgroupSize` is a power-of-two. `maxSubgroupSize` is greater than or equal to `minSubgroupSize`. `maxSubgroupSize` is greater than or equal to `subgroupSize`.
- **maxComputeWorkgroupSubgroups** is the maximum number of subgroups supported by the implementation within a workgroup.
- **requiredSubgroupSizeStages** is a bitfield of what shader stages support having a required subgroup size specified.

If the `VkPhysicalDeviceSubgroupSizeControlProperties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

If `VkPhysicalDeviceSubgroupProperties::supportedOperations` includes `VK_SUBGROUP_FEATURE_QUAD_BIT`, `minSubgroupSize` must be greater than or equal to 4.

**Valid Usage (Implicit)**

- `VUID-VkPhysicalDeviceSubgroupSizeControlProperties-sType-sType`
  - **sType** must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_PROPERTIES`

The `VkPhysicalDeviceSamplerFilterMinmaxProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceSamplerFilterMinmaxProperties {
    VkStructureType sType;
```
void* pNext;
VkBool32 filterMinmaxSingleComponentFormats;
VkBool32 filterMinmaxImageComponentMapping;
} VkPhysicalDeviceSamplerFilterMinmaxProperties;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.

- **filterMinmaxSingleComponentFormats** is a boolean value indicating whether a minimum set of required formats support min/max filtering.
- **filterMinmaxImageComponentMapping** is a boolean value indicating whether the implementation supports non-identity component mapping of the image when doing min/max filtering.

If the **VkPhysicalDeviceSamplerFilterMinmaxProperties** structure is included in the **pNext** chain of the **VkPhysicalDeviceProperties2** structure passed to **vkGetPhysicalDeviceProperties2**, it is filled in with each corresponding implementation-dependent property.

If **filterMinmaxSingleComponentFormats** is **VK_TRUE**, the following formats must support the **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT** feature with **VK_IMAGE_TILING_OPTIMAL**, if they support **VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT**:

- **VK_FORMAT_R8_UNORM**
- **VK_FORMAT_R8_SNORM**
- **VK_FORMAT_R16_UNORM**
- **VK_FORMAT_R16_SNORM**
- **VK_FORMAT_R16_SFLOAT**
- **VK_FORMAT_R32_SFLOAT**
- **VK_FORMAT_D16_UNORM**
- **VK_FORMAT_X8_D24_UNORM_PACK32**
- **VK_FORMAT_D32_SFLOAT**
- **VK_FORMAT_D16_UNORM_S8_UINT**
- **VK_FORMAT_D24_UNORM_S8_UINT**
- **VK_FORMAT_D32_SFLOAT_S8_UINT**

If the format is a depth/stencil format, this bit only specifies that the depth aspect (not the stencil aspect) of an image of this format supports min/max filtering, and that min/max filtering of the depth aspect is supported when depth compare is disabled in the sampler.

If **filterMinmaxImageComponentMapping** is **VK_FALSE** the component mapping of the image view used with min/max filtering must have been created with the **r** component set to the identity swizzle. Only the **r** component of the sampled image value is defined and the other component values are undefined. If **filterMinmaxImageComponentMapping** is **VK_TRUE** this restriction does not apply and image component mapping works as normal.
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceSamplerFilterMinmaxProperties-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_FILTER_MINMAX_PROPERTIES

The VkPhysicalDeviceProtectedMemoryProperties structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceProtectedMemoryProperties {
    VkStructureType sType;
    void* pNext;
    VkBool32 protectedNoFault;
} VkPhysicalDeviceProtectedMemoryProperties;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `protectedNoFault` specifies how an implementation behaves when an application attempts to write to unprotected memory in a protected queue operation, read from protected memory in an unprotected queue operation, or perform a query in a protected queue operation. If this limit is VK_TRUE, such writes will be discarded or have undefined values written, reads and queries will return undefined values. If this limit is VK_FALSE, applications must not perform these operations. See Protected Memory Access Rules for more information.

If the VkPhysicalDeviceProtectedMemoryProperties structure is included in the `pNext` chain of the VkPhysicalDeviceProperties2 structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceProtectedMemoryProperties-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_PROPERTIES

The VkPhysicalDeviceMaintenance3Properties structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceMaintenance3Properties {
    VkStructureType sType;
    void* pNext;
    uint32_t maxPerSetDescriptors;
    VkDeviceSize maxMemoryAllocationSize;
} VkPhysicalDeviceMaintenance3Properties;
```

- `sType` is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.

• maxPerSetDescriptors is a maximum number of descriptors (summed over all descriptor types) in a single descriptor set that is guaranteed to satisfy any implementation-dependent constraints on the size of a descriptor set itself. Applications can query whether a descriptor set that goes beyond this limit is supported using vkGetDescriptorSetLayoutSupport.

• maxMemoryAllocationSize is the maximum size of a memory allocation that can be created, even if there is more space available in the heap.

If the VkPhysicalDeviceMaintenance3Properties structure is included in the pNext chain of the VkPhysicalDeviceProperties2 structure passed to vkGetPhysicalDeviceProperties2, it is filled in with each corresponding implementation-dependent property.

Valid Usage (Implicit)

• VUID-VkPhysicalDeviceMaintenance3Properties-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_3_PROPERTIES

The VkPhysicalDeviceMaintenance4Properties structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPhysicalDeviceMaintenance4Properties {
    VkStructureType sType;
    void* pNext;
    VkDeviceSize maxBufferSize;
} VkPhysicalDeviceMaintenance4Properties;
```

• sType is the type of this structure.

• pNext is NULL or a pointer to a structure extending this structure.

• maxBufferSize is the maximum size VkBuffer that can be created.

If the VkPhysicalDeviceMaintenance4Properties structure is included in the pNext chain of the VkPhysicalDeviceProperties2 structure passed to vkGetPhysicalDeviceProperties2, it is filled in with each corresponding implementation-dependent property.

Valid Usage (Implicit)

• VUID-VkPhysicalDeviceMaintenance4Properties-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_4_PROPERTIES

The VkPhysicalDeviceDescriptorIndexingProperties structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceDescriptorIndexingProperties {
```

1154
**VkStructureType**  
`sType;`  

**void**  
`pNext;`  

**uint32_t**  
`maxUpdateAfterBindDescriptorsInAllPools;`  

**VkBool32**  
`shaderUniformBufferArrayNonUniformIndexingNative;`  

**VkBool32**  
`shaderSampledImageArrayNonUniformIndexingNative;`  

**VkBool32**  
`shaderStorageBufferArrayNonUniformIndexingNative;`  

**VkBool32**  
`shaderInputAttachmentArrayNonUniformIndexingNative;`  

**VkBool32**  
`robustBufferAccessUpdateAfterBind;`  

**VkBool32**  
`quadDivergentImplicitLod;`  

**uint32_t**  
`maxPerStageDescriptorUpdateAfterBindSamplers;`  

**uint32_t**  
`maxPerStageDescriptorUpdateAfterBindUniformBuffers;`  

**uint32_t**  
`maxPerStageDescriptorUpdateAfterBindStorageBuffers;`  

**uint32_t**  
`maxPerStageDescriptorUpdateAfterBindSampledImages;`  

**uint32_t**  
`maxPerStageDescriptorUpdateAfterBindStorageImages;`  

**uint32_t**  
`maxPerStageDescriptorUpdateAfterBindInputAttachments;`  

**uint32_t**  
`maxPerStageUpdateAfterBindResources;`  

**uint32_t**  
`maxDescriptorSetUpdateAfterBindSamplers;`  

**uint32_t**  
`maxDescriptorSetUpdateAfterBindUniformBuffers;`  

**uint32_t**  
`maxDescriptorSetUpdateAfterBindUniformBuffersDynamic;`  

**uint32_t**  
`maxDescriptorSetUpdateAfterBindStorageBuffers;`  

**uint32_t**  
`maxDescriptorSetUpdateAfterBindStorageBuffersDynamic;`  

**uint32_t**  
`maxDescriptorSetUpdateAfterBindSampledImages;`  

**uint32_t**  
`maxDescriptorSetUpdateAfterBindStorageImages;`  

**uint32_t**  
`maxDescriptorSetUpdateAfterBindInputAttachments;`  

```cpp
} VkPhysicalDeviceDescriptorIndexingProperties;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.

**maxUpdateAfterBindDescriptorsInAllPools** is the maximum number of descriptors (summed over all descriptor types) that **can** be created across all pools that are created with the **VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT** bit set. Pool creation **may** fail when this limit is exceeded, or when the space this limit represents is unable to satisfy a pool creation due to fragmentation.

- **shaderUniformBufferArrayNonUniformIndexingNative** is a boolean value indicating whether uniform buffer descriptors natively support nonuniform indexing. If this is **VK_FALSE**, then a single dynamic instance of an instruction that nonuniformly indexes an array of uniform buffers **may** execute multiple times in order to access all the descriptors.

- **shaderSampledImageArrayNonUniformIndexingNative** is a boolean value indicating whether sampler and image descriptors natively support nonuniform indexing. If this is **VK_FALSE**, then a single dynamic instance of an instruction that nonuniformly indexes an array of samplers or images **may** execute multiple times in order to access all the descriptors.

- **shaderStorageBufferArrayNonUniformIndexingNative** is a boolean value indicating whether storage buffer descriptors natively support nonuniform indexing. If this is **VK_FALSE**, then a single dynamic instance of an instruction that nonuniformly indexes an array of storage buffers **may** execute multiple times in order to access all the descriptors.
• **shaderStorageImageArrayNonUniformIndexingNative** is a boolean value indicating whether storage image descriptors natively support nonuniform indexing. If this is **VK_FALSE**, then a single dynamic instance of an instruction that nonuniformly indexes an array of storage images **may** execute multiple times in order to access all the descriptors.

• **shaderInputAttachmentArrayNonUniformIndexingNative** is a boolean value indicating whether input attachment descriptors natively support nonuniform indexing. If this is **VK_FALSE**, then a single dynamic instance of an instruction that nonuniformly indexes an array of input attachments **may** execute multiple times in order to access all the descriptors.

• **robustBufferAccessUpdateAfterBind** is a boolean value indicating whether **robustBufferAccess** can be enabled on a device simultaneously with **descriptorBindingUniformBufferUpdateAfterBind**, **descriptorBindingStorageBufferUpdateAfterBind**, **descriptorBindingUniformTexelBufferUpdateAfterBind**, and/or **descriptorBindingStorageTexelBufferUpdateAfterBind**. If this is **VK_FALSE**, then either **robustBufferAccess** must be disabled or all of these update-after-bind features **must** be disabled.

• **quadDivergentImplicitLod** is a boolean value indicating whether implicit level of detail calculations for image operations have well-defined results when the image and/or sampler objects used for the instruction are not uniform within a quad. See Derivative Image Operations.

• **maxPerStageDescriptorUpdateAfterBindSamplers** is similar to **maxPerStageDescriptorSamplers** but counts descriptors from descriptor sets created with or without the **VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT** bit set.

• **maxPerStageDescriptorUpdateAfterBindUniformBuffers** is similar to **maxPerStageDescriptorUniformBuffers** but counts descriptors from descriptor sets created with or without the **VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT** bit set.

• **maxPerStageDescriptorUpdateAfterBindStorageBuffers** is similar to **maxPerStageDescriptorStorageBuffers** but counts descriptors from descriptor sets created with or without the **VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT** bit set.

• **maxPerStageDescriptorUpdateAfterBindSampledImages** is similar to **maxPerStageDescriptorSampledImages** but counts descriptors from descriptor sets created with or without the **VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT** bit set.

• **maxPerStageDescriptorUpdateAfterBindStorageImages** is similar to **maxPerStageDescriptorStorageImages** but counts descriptors from descriptor sets created with or without the **VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT** bit set.

• **maxPerStageDescriptorUpdateAfterBindInputAttachments** is similar to **maxPerStageDescriptorInputAttachments** but counts descriptors from descriptor sets created with or without the **VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT** bit set.

• **maxPerStageUpdateAfterBindResources** is similar to **maxPerStageResources** but counts descriptors from descriptor sets created with or without the **VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT** bit set.

• **maxDescriptorSetUpdateAfterBindSamplers** is similar to **maxDescriptorSetSamplers** but counts descriptors from descriptor sets created with or without the **VK_DESCRIPTOR_SET_LAYOUT>Create_UPDATE_AFTER_BIND_POOL_BIT** bit set.

• **maxDescriptorSetUpdateAfterBindUniformBuffers** is similar to **maxDescriptorSetUniformBuffers** but
counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set.

- `maxDescriptorSetUpdateAfterBindUniformBuffersDynamic` is similar to `maxDescriptorSetUniformBuffersDynamic` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set. While an application can allocate dynamic uniform buffer descriptors from a pool created with the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT`, bindings for these descriptors must not be present in any descriptor set layout that includes bindings created with `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT`.

- `maxDescriptorSetUpdateAfterBindStorageBuffers` is similar to `maxDescriptorSetStorageBuffers` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set.

- `maxDescriptorSetUpdateAfterBindStorageBuffersDynamic` is similar to `maxDescriptorSetStorageBuffersDynamic` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set. While an application can allocate dynamic storage buffer descriptors from a pool created with the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT`, bindings for these descriptors must not be present in any descriptor set layout that includes bindings created with `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT`.

- `maxDescriptorSetUpdateAfterBindSampledImages` is similar to `maxDescriptorSetSampledImages` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set.

- `maxDescriptorSetUpdateAfterBindStorageImages` is similar to `maxDescriptorSetStorageImages` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set.

- `maxDescriptorSetUpdateAfterBindInputAttachments` is similar to `maxDescriptorSetInputAttachments` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set.

If the `VkPhysicalDeviceDescriptorIndexingProperties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceDescriptorIndexingProperties-sType-sType`  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_PROPERTIES`

The `VkPhysicalDeviceInlineUniformBlockProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPhysicalDeviceInlineUniformBlockProperties {
    VkStructureType sType;
    void* pNext;
    uint32_t maxInlineUniformBlockSize;
} VkPhysicalDeviceInlineUniformBlockProperties;
```
uint32_t maxPerStageDescriptorInlineUniformBlocks;
uint32_t maxPerStageDescriptorUpdateAfterBindInlineUniformBlocks;
uint32_t maxDescriptorSetInlineUniformBlocks;
uint32_t maxDescriptorSetUpdateAfterBindInlineUniformBlocks;
}

VkPhysicalDeviceInlineUniformBlockProperties

• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.

• **maxInlineUniformBlockSize** is the maximum size in bytes of an inline uniform block binding.

• **maxPerStageDescriptorInlineUniformBlock** is the maximum number of inline uniform block bindings that can be accessible to a single shader stage in a pipeline layout. Descriptor bindings with a descriptor type of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK count against this limit. Only descriptor bindings in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit.

• **maxPerStageDescriptorUpdateAfterBindInlineUniformBlocks** is similar to maxPerStageDescriptorInlineUniformBlock but counts descriptor bindings from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

• **maxDescriptorSetInlineUniformBlocks** is the maximum number of inline uniform block bindings that can be included in descriptor bindings in a pipeline layout across all pipeline shader stages and descriptor set numbers. Descriptor bindings with a descriptor type of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK count against this limit. Only descriptor bindings in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit.

• **maxDescriptorSetUpdateAfterBindInlineUniformBlocks** is similar to maxDescriptorSetInlineUniformBlocks but counts descriptor bindings from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

If the VkPhysicalDeviceInlineUniformBlockProperties structure is included in the pNext chain of the VkPhysicalDeviceProperties2 structure passed to vkGetPhysicalDeviceProperties2, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceInlineUniformBlockProperties-sType-sType**
  - **sType** must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INLINE_UNIFORM_BLOCK_PROPERTIES

The VkPhysicalDeviceDepthStencilResolveProperties structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceDepthStencilResolveProperties {
    VkStructureType sType;
    void* pNext;
    VkResolveModeFlags supportedDepthResolveModes;
    VkResolveModeFlags supportedStencilResolveModes;
} VkPhysicalDeviceDepthStencilResolveProperties;
```
VkBool32 independentResolveNone;
VkBool32 independentResolve;
} VkPhysicalDeviceDepthStencilResolveProperties;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.

- **supportedDepthResolveModes** is a bitmask of VkResolveModeFlagBits indicating the set of supported depth resolve modes. VK_RESOLVE_MODE_SAMPLE_ZERO_BIT must be included in the set but implementations may support additional modes.

- **supportedStencilResolveModes** is a bitmask of VkResolveModeFlagBits indicating the set of supported stencil resolve modes. VK_RESOLVE_MODE_SAMPLE_ZERO_BIT must be included in the set but implementations may support additional modes. VK_RESOLVE_MODE_AVERAGE_BIT must not be included in the set.

- **independentResolveNone** is VK_TRUE if the implementation supports setting the depth and stencil resolve modes to different values when one of those modes is VK_RESOLVE_MODE_NONE. Otherwise the implementation only supports setting both modes to the same value.

- **independentResolve** is VK_TRUE if the implementation supports all combinations of the supported depth and stencil resolve modes, including setting either depth or stencil resolve mode to VK_RESOLVE_MODE_NONE. An implementation that supports independentResolve must also support independentResolveNone.

If the VkPhysicalDeviceDepthStencilResolveProperties structure is included in the pNext chain of the VkPhysicalDeviceProperties2 structure passed to vkGetPhysicalDeviceProperties2, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceDepthStencilResolveProperties-sType-sType
  - **sType** must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEPTH_STENCIL_RESOLVE_PROPERTIES

The VkPhysicalDeviceTexelBufferAlignmentProperties structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkPhysicalDeviceTexelBufferAlignmentProperties {
    VkStructureType sType;
    void* pNext;
    VkDeviceSize storageTexelBufferOffsetAlignmentBytes;
    VkBool32 storageTexelBufferOffsetSingleTexelAlignment;
    VkDeviceSize uniformTexelBufferOffsetAlignmentBytes;
    uniformTexelBufferOffsetSingleTexelAlignment;
} VkPhysicalDeviceTexelBufferAlignmentProperties;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
• `storageTexelBufferOffsetAlignmentBytes` is a byte alignment that is sufficient for a storage texel buffer of any format. The value **must** be a power of two.

• `storageTexelBufferOffsetSingleTexelAlignment` indicates whether single texel alignment is sufficient for a storage texel buffer of any format.

• `uniformTexelBufferOffsetAlignmentBytes` is a byte alignment that is sufficient for a uniform texel buffer of any format. The value **must** be a power of two.

• `uniformTexelBufferOffsetSingleTexelAlignment` indicates whether single texel alignment is sufficient for a uniform texel buffer of any format.

If the `VkPhysicalDeviceTexelBufferAlignmentProperties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

If the single texel alignment property is `VK_FALSE`, then the buffer view’s offset **must** be aligned to the corresponding byte alignment value. If the single texel alignment property is `VK_TRUE`, then the buffer view’s offset **must** be aligned to the lesser of the corresponding byte alignment value or the size of a single texel, based on `VkBufferViewCreateInfo::format`. If the size of a single texel is a multiple of three bytes, then the size of a single component of the format is used instead.

These limits **must** not advertise a larger alignment than the required maximum minimum value of `VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment`, for any format that supports use as a texel buffer.

---

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceTexelBufferAlignmentProperties-sType-sType  
  `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_PROPERTIES`

The `VkPhysicalDeviceTimelineSemaphoreProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceTimelineSemaphoreProperties {
    VkStructureType sType;
    void* pNext;
    uint64_t maxTimelineSemaphoreValueDifference;
} VkPhysicalDeviceTimelineSemaphoreProperties;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `maxTimelineSemaphoreValueDifference` indicates the maximum difference allowed by the implementation between the current value of a timeline semaphore and any pending signal or wait operations.

If the `VkPhysicalDeviceTimelineSemaphoreProperties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with
each corresponding implementation-dependent property.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceTimelineSemaphoreProperties-sType-sType
  
  `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_PROPERTIES`

### 33.1. Limit Requirements

The following table specifies the **required** minimum/maximum for all Vulkan graphics implementations. Where a limit corresponds to a fine-grained device feature which is **optional**, the feature name is listed with two **required** limits, one when the feature is supported and one when it is not supported. If an implementation supports a feature, the limits reported are the same whether or not the feature is enabled.

**Table 32. Required Limit Types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Limit</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxImageDimension1D</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxImageDimension2D</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxImageDimension3D</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxImageDimensionCube</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxImageArrayLayers</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxTexelBufferElements</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxUniformBufferRange</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxStorageBufferRange</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPushConstantsSize</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxMemoryAllocationCount</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxSamplerAllocationCount</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkDeviceSize</code></td>
<td><code>bufferImageGranularity</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkDeviceSize</code></td>
<td><code>sparseAddressSpaceSize</code></td>
<td><code>sparseBinding</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxBoundDescriptorSets</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageDescriptorSamplers</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageDescriptorUniformBuffers</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageDescriptorStorageBuffers</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageDescriptorSampledImages</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageDescriptorStorageImages</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageDescriptorInputAttachments</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageResources</code></td>
<td>-</td>
</tr>
<tr>
<td>Type</td>
<td>Limit</td>
<td>Feature</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetSamplers</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetUniformBuffers</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetUniformBuffersDynamic</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetStorageBuffers</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetStorageBuffersDynamic</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetSampledImages</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetStorageImages</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetInputAttachments</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxVertexInputAttributes</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxVertexInputBindings</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxVertexInputAttributeOffset</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxVertexInputBindingStride</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxVertexInputComponents</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTessellationGenerationLevel</td>
<td>tessellationShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTessellationPatchSize</td>
<td>tessellationShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTessellationControlPerVertexInputComponents</td>
<td>tessellationShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTessellationControlPerVertexOutputComponents</td>
<td>tessellationShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTessellationControlPerPatchOutputComponents</td>
<td>tessellationShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTessellationControlTotalOutputComponents</td>
<td>tessellationShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTessellationEvaluationInputComponents</td>
<td>tessellationShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTessellationEvaluationOutputComponents</td>
<td>tessellationShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxGeometryShaderInvocations</td>
<td>geometryShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxGeometryInputComponents</td>
<td>geometryShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxGeometryOutputComponents</td>
<td>geometryShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxGeometryOutputVertices</td>
<td>geometryShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxGeometryTotalOutputComponents</td>
<td>geometryShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxFragmentInputComponents</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxFragmentOutputAttachments</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxFragmentDualSrcAttachments</td>
<td>dualSrcBlend</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxFragmentCombinedOutputResources</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxComputeSharedMemorySize</td>
<td>-</td>
</tr>
<tr>
<td>3 × uint32_t</td>
<td>maxComputeWorkGroupCount</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxComputeWorkGroupInvocations</td>
<td>-</td>
</tr>
<tr>
<td>3 × uint32_t</td>
<td>maxComputeWorkGroupSize</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>subPixelPrecisionBits</td>
<td>-</td>
</tr>
<tr>
<td>Type</td>
<td>Limit</td>
<td>Feature</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>uint32_t</td>
<td>subTexelPrecisionBits</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>mipmapPrecisionBits</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDrawIndexedIndexValue</td>
<td>fullDrawIndexUint32</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDrawIndirectCount</td>
<td>multiDrawIndirect</td>
</tr>
<tr>
<td>float</td>
<td>maxSamplerLodBias</td>
<td>-</td>
</tr>
<tr>
<td>float</td>
<td>maxSamplerAnisotropy</td>
<td>samplerAnisotropy</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxViewports</td>
<td>multiViewport</td>
</tr>
<tr>
<td>2 × uint32_t</td>
<td>maxViewportDimensions</td>
<td>-</td>
</tr>
<tr>
<td>2 × float</td>
<td>viewportBoundsRange</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>viewportSubPixelBits</td>
<td>-</td>
</tr>
<tr>
<td>size_t</td>
<td>minMemoryMapAlignment</td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>minTexelBufferOffsetAlignment</td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>minUniformBufferOffsetAlignment</td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>minStorageBufferOffsetAlignment</td>
<td>-</td>
</tr>
<tr>
<td>int32_t</td>
<td>minTexelOffset</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTexelOffset</td>
<td>-</td>
</tr>
<tr>
<td>int32_t</td>
<td>minTexelGatherOffset</td>
<td>shaderImageGatherExtended</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTexelGatherOffset</td>
<td>shaderImageGatherExtended</td>
</tr>
<tr>
<td>float</td>
<td>minInterpolationOffset</td>
<td>sampleRateShading</td>
</tr>
<tr>
<td>float</td>
<td>maxInterpolationOffset</td>
<td>sampleRateShading</td>
</tr>
<tr>
<td>uint32_t</td>
<td>subPixelInterpolationOffsetBits</td>
<td>sampleRateShading</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxFramebufferWidth</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxFramebufferHeight</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxFramebufferLayers</td>
<td>-</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>framebufferColorSampleCounts</td>
<td>-</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>framebufferIntegerColorSampleCounts</td>
<td>-</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>framebufferDepthSampleCounts</td>
<td>-</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>framebufferStencilSampleCounts</td>
<td>-</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>framebufferNoAttachmentsSampleCounts</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxColorAttachments</td>
<td>-</td>
</tr>
<tr>
<td>Type</td>
<td>Limit</td>
<td>Feature</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>sampledImageColorSampleCounts</td>
<td>-</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>sampledImageIntegerSampleCounts</td>
<td>-</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>sampledImageDepthSampleCounts</td>
<td>-</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>sampledImageStencilSampleCounts</td>
<td>-</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>storageImageSampleCounts</td>
<td>shaderStorageImageMultisample</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxSampleMaskWords</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>timestampComputeAndGraphics</td>
<td>-</td>
</tr>
<tr>
<td>float</td>
<td>timestampPeriod</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxClipDistances</td>
<td>shaderClipDistance</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxCullDistances</td>
<td>shaderCullDistance</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxCombinedClipAndCullDistances</td>
<td>shaderCullDistance</td>
</tr>
<tr>
<td>uint32_t</td>
<td>discreteQueuePriorities</td>
<td>-</td>
</tr>
<tr>
<td>2 × float</td>
<td>pointSizeRange</td>
<td>largePoints</td>
</tr>
<tr>
<td>2 × float</td>
<td>lineWidthRange</td>
<td>wideLines</td>
</tr>
<tr>
<td>float</td>
<td>pointSizeGranularity</td>
<td>largePoints</td>
</tr>
<tr>
<td>float</td>
<td>lineWidthGranularity</td>
<td>wideLines</td>
</tr>
<tr>
<td>VkBool32</td>
<td>strictLines</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>standardSampleLocations</td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>optimalBufferCopyOffsetAlignment</td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>optimalBufferCopyRowPitchAlignment</td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>nonCoherentAtomSize</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>filterMinMaxSingleComponentFormats</td>
<td>samplerFilterMinMax</td>
</tr>
<tr>
<td>VkBool32</td>
<td>filterMinMaxImageComponentMapping</td>
<td>samplerFilterMinMax</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>maxBufferSize</td>
<td>maintenance4</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxUpdateAfterBindDescriptorsInAllPools</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>VkBool32</td>
<td>shaderUniformBufferArrayNonUniformIndexingNative</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>shaderSampledImageArrayNonUniformIndexingNative</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>shaderStorageBufferArrayNonUniformIndexingNative</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>shaderStorageImageArrayNonUniformIndexingNative</td>
<td>-</td>
</tr>
<tr>
<td>Type</td>
<td>Limit</td>
<td>Feature</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>VkBool32</td>
<td>shaderInputAttachmentArrayNonUniformIndexingNative</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorUpdateAfterBindSamplers</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorUpdateAfterBindUniformBuffers</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorUpdateAfterBindStorageBuffers</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorUpdateAfterBindSampledImages</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorUpdateAfterBindStorageImages</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorUpdateAfterBindInputAttachments</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageUpdateAfterBindResources</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetUpdateAfterBindSamplers</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetUpdateAfterBindUniformBuffers</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetUpdateAfterBindUniformBuffersDynamic</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetUpdateAfterBindStorageBuffersDynamic</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetUpdateAfterBindSampledImages</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetUpdateAfterBindStorageImages</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorInlineUniformBlocks</td>
<td>inlineUniformBlock</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxInlineUniformBlockSize</td>
<td>inlineUniformBlock</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorInlineUniformBlocks</td>
<td>inlineUniformBlock</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetInlineUniformBlocks</td>
<td>inlineUniformBlock</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetUpdateAfterBindInlineUniformBlocks</td>
<td>inlineUniformBlock</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxInlineUniformTotalSize</td>
<td>inlineUniformBlock</td>
</tr>
<tr>
<td>uint64_t</td>
<td>maxTimelineSemaphoreValueDifference</td>
<td>timelineSemaphore</td>
</tr>
</tbody>
</table>

Table 33. Required Limits

<table>
<thead>
<tr>
<th>Limit</th>
<th>Unsupported Limit</th>
<th>Supported Limit</th>
<th>Limit Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxImageDimension1D</td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td>maxImageDimension2D</td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td>maxImageDimension3D</td>
<td>-</td>
<td>256</td>
<td>min</td>
</tr>
<tr>
<td>maxImageDimensionCube</td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td>Limit</td>
<td>Unsupported Limit</td>
<td>Supported Limit</td>
<td>Limit Type(^1)</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------</td>
<td>-----------------</td>
<td>------------------</td>
</tr>
<tr>
<td>maxImageArrayLayers</td>
<td>-</td>
<td>256</td>
<td>min</td>
</tr>
<tr>
<td>maxTexelBufferElements</td>
<td>-</td>
<td>65536</td>
<td>min</td>
</tr>
<tr>
<td>maxUniformBufferRange</td>
<td>-</td>
<td>16384</td>
<td>min</td>
</tr>
<tr>
<td>maxStorageBufferRange</td>
<td>-</td>
<td>(2^{27})</td>
<td>min</td>
</tr>
<tr>
<td>maxPushConstantsSize</td>
<td>-</td>
<td>128</td>
<td>min</td>
</tr>
<tr>
<td>maxMemoryAllocationCount</td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td>maxSamplerAllocationCount</td>
<td>-</td>
<td>4000</td>
<td>min</td>
</tr>
<tr>
<td>bufferImageGranularity</td>
<td>-</td>
<td>131072</td>
<td>max</td>
</tr>
<tr>
<td>sparseAddressSpaceSize</td>
<td>0</td>
<td>(2^{31})</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSets</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorSamplers</td>
<td>-</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUniformBuffers</td>
<td>-</td>
<td>12</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorStorageBuffers</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorSampledImages</td>
<td>-</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorStorageImages</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorInputAttachments</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageResources</td>
<td>-</td>
<td>(128^2)</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetSamplers</td>
<td>-</td>
<td>96</td>
<td>(\times) PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetUniformBuffers</td>
<td>-</td>
<td>72 (^8)</td>
<td>(\times) PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetUniformBuffersDynamic</td>
<td>-</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetStorageBuffers</td>
<td>-</td>
<td>24 (^8)</td>
<td>(\times) PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetStorageBuffersDynamic</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetSampledImages</td>
<td>-</td>
<td>96 (^8)</td>
<td>(\times) PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetStorageImages</td>
<td>-</td>
<td>24 (^8)</td>
<td>(\times) PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetInputAttachments</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxVertexInputAttributes</td>
<td>-</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxVertexInputBindings</td>
<td>-</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxVertexInputAttributeOffset</td>
<td>-</td>
<td>2047</td>
<td>min</td>
</tr>
<tr>
<td>maxVertexInputBindingStride</td>
<td>-</td>
<td>2048</td>
<td>min</td>
</tr>
</tbody>
</table>

\(^1\) Limit Type: min (minimum), max (maximum)
<table>
<thead>
<tr>
<th>Limit</th>
<th>Unsupported Limit</th>
<th>Supported Limit</th>
<th>Limit Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxVertexOutputComponents</td>
<td>-</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxTessellationGenerationLevel</td>
<td>0</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxTessellationPatchSize</td>
<td>0</td>
<td>32</td>
<td>min</td>
</tr>
<tr>
<td>maxTessellationControlPerVertexInputComponents</td>
<td>0</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxTessellationControlPerVertexOutputComponents</td>
<td>0</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxTessellationControlPerPatchOutputComponents</td>
<td>0</td>
<td>120</td>
<td>min</td>
</tr>
<tr>
<td>maxTessellationControlTotalOutputComponents</td>
<td>0</td>
<td>2048</td>
<td>min</td>
</tr>
<tr>
<td>maxTessellationEvaluationInputComponents</td>
<td>0</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxTessellationEvaluationOutputComponents</td>
<td>0</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxGeometryShaderInvocations</td>
<td>0</td>
<td>32</td>
<td>min</td>
</tr>
<tr>
<td>maxGeometryInputComponents</td>
<td>0</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxGeometryOutputComponents</td>
<td>0</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxGeometryOutputVertices</td>
<td>0</td>
<td>256</td>
<td>min</td>
</tr>
<tr>
<td>maxGeometryTotalOutputComponents</td>
<td>0</td>
<td>1024</td>
<td>min</td>
</tr>
<tr>
<td>maxFragmentInputComponents</td>
<td>-</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxFragmentOutputAttachments</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxFragmentDualSrcAttachments</td>
<td>0</td>
<td>1</td>
<td>min</td>
</tr>
<tr>
<td>maxFragmentCombinedOutputResources</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxComputeSharedMemorySize</td>
<td>-</td>
<td>16384</td>
<td>min</td>
</tr>
<tr>
<td>maxComputeWorkGroupCount</td>
<td>-</td>
<td>(65535,65535,65535)</td>
<td>min</td>
</tr>
<tr>
<td>maxComputeWorkGroupInvocations</td>
<td>-</td>
<td>128</td>
<td>min</td>
</tr>
<tr>
<td>maxComputeWorkGroupSize</td>
<td>-</td>
<td>(128,128,64)</td>
<td>min</td>
</tr>
<tr>
<td>subPixelPrecisionBits</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>subTexelPrecisionBits</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>mipmapPrecisionBits</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxDrawIndexedIndexValue</td>
<td>$2^{24} - 1$</td>
<td>$2^{32} - 1$</td>
<td>min</td>
</tr>
<tr>
<td>maxDrawIndirectCount</td>
<td>1</td>
<td>$2^{16} - 1$</td>
<td>min</td>
</tr>
<tr>
<td>maxSamplerLodBias</td>
<td>-</td>
<td>2</td>
<td>min</td>
</tr>
<tr>
<td>maxSamplerAnisotropy</td>
<td>1</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxViewports</td>
<td>1</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxViewportDimensions</td>
<td>-</td>
<td>(4096,4096)</td>
<td>min</td>
</tr>
<tr>
<td>viewportBoundsRange</td>
<td>-</td>
<td>(-8192,8191)</td>
<td>(max,min)</td>
</tr>
<tr>
<td>Limit</td>
<td>Unsupported Limit</td>
<td>Supported Limit</td>
<td>Limit Type</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-------------------</td>
<td>-----------------</td>
<td>------------</td>
</tr>
<tr>
<td>viewportSubPixelBits</td>
<td>-</td>
<td>0</td>
<td>min</td>
</tr>
<tr>
<td>minMemoryMapAlignment</td>
<td>-</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>minTexelBufferOffsetAlignment</td>
<td>-</td>
<td>256</td>
<td>max</td>
</tr>
<tr>
<td>minUniformBufferOffsetAlignment</td>
<td>-</td>
<td>256</td>
<td>max</td>
</tr>
<tr>
<td>minStorageBufferOffsetAlignment</td>
<td>-</td>
<td>256</td>
<td>max</td>
</tr>
<tr>
<td>minTexelOffset</td>
<td>-</td>
<td>-8</td>
<td>max</td>
</tr>
<tr>
<td>maxTexelOffset</td>
<td>-</td>
<td>7</td>
<td>min</td>
</tr>
<tr>
<td>minTexelGatherOffset</td>
<td>0</td>
<td>-8</td>
<td>max</td>
</tr>
<tr>
<td>maxTexelGatherOffset</td>
<td>0</td>
<td>7</td>
<td>min</td>
</tr>
<tr>
<td>minInterpolationOffset</td>
<td>0.0</td>
<td>-0.5 (^5)</td>
<td>max</td>
</tr>
<tr>
<td>maxInterpolationOffset</td>
<td>0.0</td>
<td>0.5 - (1 ULP)(^5)</td>
<td>min</td>
</tr>
<tr>
<td>subPixelInterpolationOffsetBits</td>
<td>0</td>
<td>4 (^5)</td>
<td>min</td>
</tr>
<tr>
<td>maxFramebufferWidth</td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td>maxFramebufferHeight</td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td>maxFramebufferLayers</td>
<td>-</td>
<td>256</td>
<td>min</td>
</tr>
<tr>
<td>framebufferColorSampleCounts</td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td>framebufferIntegerColorSampleCounts</td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT)</td>
<td>min</td>
</tr>
<tr>
<td>framebufferDepthSampleCounts</td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td>framebufferStencilSampleCounts</td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td>framebufferNoAttachmentsSampleCounts</td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td>maxColorAttachments</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>sampledImageColorSampleCounts</td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td>Limit</td>
<td>Unsupported Limit</td>
<td>Supported Limit</td>
<td>Limit Type</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-------------------</td>
<td>-----------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td><code>sampledImageIntegerSampleCounts</code></td>
<td>-</td>
<td>VK_SAMPLE_COUNT_1_BIT</td>
<td>min</td>
</tr>
<tr>
<td><code>sampledImageDepthSampleCounts</code></td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td><code>sampledImageStencilSampleCounts</code></td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td><code>storageImageSampleCounts</code></td>
<td>VK_SAMPLE_COUNT_1_BIT</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td><code>maxSampleMaskWords</code></td>
<td>-</td>
<td>1</td>
<td>min</td>
</tr>
<tr>
<td><code>timestampComputeAndGraphics</code></td>
<td>-</td>
<td>-</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td><code>timestampPeriod</code></td>
<td>-</td>
<td>-</td>
<td>duration</td>
</tr>
<tr>
<td><code>maxClipDistances</code></td>
<td>0</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td><code>maxCullDistances</code></td>
<td>0</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td><code>maxCombinedClipAndCullDistances</code></td>
<td>0</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td><code>discreteQueuePriorities</code></td>
<td>-</td>
<td>2</td>
<td>min</td>
</tr>
<tr>
<td><code>pointSizeRange</code></td>
<td>(1.0,1.0)</td>
<td>(1.0,64.0 - ULP)⁶</td>
<td>(max,min)</td>
</tr>
<tr>
<td><code>lineWidthRange</code></td>
<td>(1.0,1.0)</td>
<td>(1.0,8.0 - ULP)⁷</td>
<td>(max,min)</td>
</tr>
<tr>
<td><code>pointSizeGranularity</code></td>
<td>0.0</td>
<td>1.0⁶</td>
<td>max, fixed point increment</td>
</tr>
<tr>
<td><code>lineWidthGranularity</code></td>
<td>0.0</td>
<td>1.0⁷</td>
<td>max, fixed point increment</td>
</tr>
<tr>
<td><code>strictLines</code></td>
<td>-</td>
<td>-</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td><code>standardSampleLocations</code></td>
<td>-</td>
<td>-</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td><code>optimalBufferCopyOffsetAlignment</code></td>
<td>-</td>
<td>-</td>
<td>recommendation</td>
</tr>
<tr>
<td><code>optimalBufferCopyRowPitchAlignment</code></td>
<td>-</td>
<td>-</td>
<td>recommendation</td>
</tr>
<tr>
<td>Limit</td>
<td>Unsupported Limit</td>
<td>Supported Limit</td>
<td>Limit Type¹</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-------------------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>nonCoherentAtomSize</td>
<td>-</td>
<td>256 max</td>
<td></td>
</tr>
<tr>
<td>maxMultiviewViewCount</td>
<td>-</td>
<td>6 min</td>
<td></td>
</tr>
<tr>
<td>maxMultiviewInstanceIndex</td>
<td>-</td>
<td>$2^{27}$-1 min</td>
<td></td>
</tr>
<tr>
<td>filterMinmaxSingleComponentFormats</td>
<td>-</td>
<td>- implementation-dependent</td>
<td></td>
</tr>
<tr>
<td>filterMinmaxImageComponentMapping</td>
<td>-</td>
<td>- implementation-dependent</td>
<td></td>
</tr>
<tr>
<td>maxPerSetDescriptors</td>
<td>-</td>
<td>1024 min</td>
<td></td>
</tr>
<tr>
<td>maxMemoryAllocationSize</td>
<td>-</td>
<td>$2^{30}$ min</td>
<td></td>
</tr>
<tr>
<td>maxBufferSize</td>
<td>-</td>
<td>$2^{30}$ min</td>
<td></td>
</tr>
<tr>
<td>maxUpdateAfterBindDescriptorsInAllPools</td>
<td>0</td>
<td>500000 min</td>
<td>min</td>
</tr>
<tr>
<td>shaderUniformBufferArrayNonUniformIndexingNative</td>
<td>-</td>
<td>false implementation-dependent</td>
<td></td>
</tr>
<tr>
<td>shaderSampledImageArrayNonUniformIndexingNative</td>
<td>-</td>
<td>false implementation-dependent</td>
<td></td>
</tr>
<tr>
<td>shaderStorageBufferArrayNonUniformIndexingNative</td>
<td>-</td>
<td>false implementation-dependent</td>
<td></td>
</tr>
<tr>
<td>shaderStorageImageArrayNonUniformIndexingNative</td>
<td>-</td>
<td>false implementation-dependent</td>
<td></td>
</tr>
<tr>
<td>shaderInputAttachmentArrayNonUniformIndexingNative</td>
<td>-</td>
<td>false implementation-dependent</td>
<td></td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindSamplers</td>
<td>$0^9$</td>
<td>$500000^9$ min</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindUniformBuffers</td>
<td>$0^9$</td>
<td>$12^9$ min</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindStorageBuffers</td>
<td>$0^9$</td>
<td>$500000^9$ min</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindSampledImages</td>
<td>$0^9$</td>
<td>$500000^9$ min</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindStorageImages</td>
<td>$0^9$</td>
<td>$500000^9$ min</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindInputAttachments</td>
<td>$0^9$</td>
<td>$4^9$ min</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageUpdateAfterBindResources</td>
<td>$0^9$</td>
<td>$500000^9$ min</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindSamplers</td>
<td>$0^9$</td>
<td>$500000^9$ min</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindUniformBuffers</td>
<td>$0^9$</td>
<td>$72^8^9$ min, $n \times$ PerStage</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindUniformBuffersDynamic</td>
<td>$0^9$</td>
<td>$8^9$ min</td>
<td>min</td>
</tr>
<tr>
<td>Limit</td>
<td>Unsupported Limit</td>
<td>Supported Limit</td>
<td>Limit Type</td>
</tr>
<tr>
<td>------------------------------------------------------------</td>
<td>-------------------</td>
<td>-----------------</td>
<td>------------</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindStorageBuffers</td>
<td>0^9</td>
<td>500000^9</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindStorageBuffersDyamic</td>
<td>0^9</td>
<td>4^9</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindSampledImages</td>
<td>0^9</td>
<td>500000^9</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindStorageImages</td>
<td>0^9</td>
<td>500000^9</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindInputAttachments</td>
<td>0^9</td>
<td>4^9</td>
<td>min</td>
</tr>
<tr>
<td>maxInlineUniformBlockSize</td>
<td>-</td>
<td>256</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorInlineUniformBlocks</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindInlineUniformBlocks</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetInlineUniformBlocks</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindInlineUniformBlocks</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxInlineUniformTotalSize</td>
<td>-</td>
<td>256</td>
<td>min</td>
</tr>
<tr>
<td>maxTimelineSemaphoreValueDifference</td>
<td>-</td>
<td>2^{31}-1</td>
<td>min</td>
</tr>
</tbody>
</table>

1

The **Limit Type** column specifies the limit is either the minimum limit all implementations **must** support, the maximum limit all implementations **must** support, or the exact value all implementations **must** support. For bitmasks a minimum limit is the least bits all implementations **must** set, but they **may** have additional bits set beyond this minimum.

2

The **maxPerStageResources** **must** be at least the smallest of the following:

- the sum of the `maxPerStageDescriptorUniformBuffers`, `maxPerStageDescriptorStorageBuffers`, `maxPerStageDescriptorSampledImages`, `maxPerStageDescriptorStorageImages`, `maxPerStageDescriptorInputAttachments`, `maxColorAttachments` limits, or
- 128.

It **may** not be possible to reach this limit in every stage.

3

See **maxViewportDimensions** for the **required** relationship to other limits.

4

See **viewportBoundsRange** for the **required** relationship to other limits.

5

The values **minInterpolationOffset** and **maxInterpolationOffset** describe the closed interval of supported interpolation offsets: `[minInterpolationOffset, maxInterpolationOffset]`. The ULP is determined by `subPixelInterpolationOffsetBits`. If `subPixelInterpolationOffsetBits` is 4, this

1171
provides increments of \((1/2^4) = 0.0625\), and thus the range of supported interpolation offsets would be \([-0.5, 0.4375]\).

6

The point size ULP is determined by \texttt{pointSizeGranularity}. If the \texttt{pointSizeGranularity} is 0.125, the range of supported point sizes \textbf{must} be at least \([1.0, 63.875]\).

7

The line width ULP is determined by \texttt{lineWidthGranularity}. If the \texttt{lineWidthGranularity} is 0.0625, the range of supported line widths \textbf{must} be at least \([1.0, 7.9375]\).

8

The minimum \texttt{maxDescriptorSet\*} limit is \(n\) times the corresponding \textit{specification} minimum \texttt{maxPerStageDescriptor\*} limit, where \(n\) is the number of shader stages supported by the \texttt{VkPhysicalDevice}. If all shader stages are supported, \(n = 6\) (vertex, tessellation control, tessellation evaluation, geometry, fragment, compute).

9

The \texttt{UpdateAfterBind} descriptor limits \textbf{must} each be greater than or equal to the corresponding \texttt{non-UpdateAfterBind} limit.

### 33.2. Profile Limits

#### 33.2.1. Roadmap 2022

Implementations that claim support for the Roadmap 2022 profile \textbf{must} satisfy the following additional limit requirements:

<table>
<thead>
<tr>
<th>Limit</th>
<th>Supported Limit</th>
<th>Limit Type¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{maxImageDimension1D}</td>
<td>8192</td>
<td>min</td>
</tr>
<tr>
<td>\texttt{maxImageDimension2D}</td>
<td>8192</td>
<td>min</td>
</tr>
<tr>
<td>\texttt{maxImageDimensionCube}</td>
<td>8192</td>
<td>min</td>
</tr>
<tr>
<td>\texttt{maxImageArrayLayers}</td>
<td>2048</td>
<td>min</td>
</tr>
<tr>
<td>\texttt{maxUniformBufferRange}</td>
<td>65536</td>
<td>min</td>
</tr>
<tr>
<td>\texttt{bufferImageGranularity}</td>
<td>4096</td>
<td>max</td>
</tr>
<tr>
<td>\texttt{maxPerStageDescriptorSamplers}</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>\texttt{maxPerStageDescriptorUniformBuffers}</td>
<td>15</td>
<td>min</td>
</tr>
<tr>
<td>\texttt{maxPerStageDescriptorStorageBuffers}</td>
<td>30</td>
<td>min</td>
</tr>
<tr>
<td>\texttt{maxPerStageDescriptorSampledImages}</td>
<td>200</td>
<td>min</td>
</tr>
<tr>
<td>\texttt{maxPerStageDescriptorStorageImages}</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>\texttt{maxPerStageResources}</td>
<td>200</td>
<td>min</td>
</tr>
<tr>
<td>\texttt{maxDescriptorSetSamplers}</td>
<td>576</td>
<td>min</td>
</tr>
<tr>
<td>Limit</td>
<td>Supported Limit</td>
<td>Limit Type</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------------</td>
<td>------------</td>
</tr>
<tr>
<td>maxDescriptorSetUniformBuffers</td>
<td>90</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetStorageBuffers</td>
<td>96</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetSampledImages</td>
<td>1800</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetStorageImages</td>
<td>144</td>
<td>min</td>
</tr>
<tr>
<td>maxFragmentCombinedOutputResources</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxComputeWorkGroupInvocations</td>
<td>256</td>
<td>min</td>
</tr>
<tr>
<td>maxComputeWorkGroupSize</td>
<td>(256,256,64)</td>
<td>min</td>
</tr>
<tr>
<td>subTexelPrecisionBits</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td>mipmapPrecisionBits</td>
<td>6</td>
<td>min</td>
</tr>
<tr>
<td>maxSamplerLodBias</td>
<td>14</td>
<td>min</td>
</tr>
<tr>
<td>pointSizeGranularity</td>
<td>0.125</td>
<td>max</td>
</tr>
<tr>
<td>lineWidthGranularity</td>
<td>0.5</td>
<td>max</td>
</tr>
<tr>
<td>standardSampleLocations</td>
<td>VK_TRUE</td>
<td>Boolean</td>
</tr>
<tr>
<td>maxColorAttachments</td>
<td>7</td>
<td>min</td>
</tr>
<tr>
<td>subgroupSize</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>subgroupSupportedStages</td>
<td>VK_SHADER_STAGE_COMPUTE_BIT, VK_SHADER_STAGE_FRAGMENT_BIT</td>
<td>bitfield</td>
</tr>
<tr>
<td>subgroupSupportedOperations</td>
<td>VK_SUBGROUP_FEATURE_BASIC_BIT, VK_SUBGROUP_FEATURE_VOTE_BIT, VK_SUBGROUP_FEATURE_ARITHMETIC_BIT, VK_SUBGROUP_FEATURE_BALLOT_BIT, VK_SUBGROUP_FEATURE_HUFFLE_BIT, VK_SUBGROUP_FEATURE_HUFFLE_RELATIVE_BIT, VK_SUBGROUP_FEATURE_QUAD_BIT</td>
<td>bitfield</td>
</tr>
<tr>
<td>shaderSignedZeroInfNanPreserveFloat16</td>
<td>VK_TRUE</td>
<td>Boolean</td>
</tr>
<tr>
<td>shaderSignedZeroInfNanPreserveFloat32</td>
<td>VK_TRUE</td>
<td>Boolean</td>
</tr>
<tr>
<td>maxSubgroupSize</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindInputAttachments</td>
<td>7</td>
<td>min</td>
</tr>
</tbody>
</table>
Chapter 34. Formats

Supported buffer and image formats may vary across implementations. A minimum set of format features are guaranteed, but others must be explicitly queried before use to ensure they are supported by the implementation.

The features for the set of formats (VkFormat) supported by the implementation are queried individually using the vkGetPhysicalDeviceFormatProperties command.

34.1. Format Definition

The following image formats can be passed to, and may be returned from Vulkan commands. The memory required to store each format is discussed with that format, and also summarized in the Representation and Texel Block Size section and the Compatible formats table.

```c
// Provided by VK_VERSION_1_0
typedef enum VkFormat {
  VK_FORMAT_UNDEFINED = 0,
  VK_FORMAT_R4G4_UNORM_PACK8 = 1,
  VK_FORMAT_R4G4B4A4_UNORM_PACK16 = 2,
  VK_FORMAT_B4G4R4A4_UNORM_PACK16 = 3,
  VK_FORMAT_R5G6B5_UNORM_PACK16 = 4,
  VK_FORMAT_B5G6R5_UNORM_PACK16 = 5,
  VK_FORMAT_R5G5B5A1_UNORM_PACK16 = 6,
  VK_FORMAT_B5G5R5A1_UNORM_PACK16 = 7,
  VK_FORMAT_A1R5G5B5_UNORM_PACK16 = 8,
  VK_FORMAT_R8_UNORM = 9,
  VK_FORMAT_R8_SNORM = 10,
  VK_FORMAT_R8_USCALED = 11,
  VK_FORMAT_R8_SSCALED = 12,
  VK_FORMAT_R8_UINT = 13,
  VK_FORMAT_R8_SINT = 14,
  VK_FORMAT_R8_SRGB = 15,
  VK_FORMAT_R8G8_UNORM = 16,
  VK_FORMAT_R8G8_SNORM = 17,
  VK_FORMAT_R8G8_USCALED = 18,
  VK_FORMAT_R8G8_SSCALED = 19,
  VK_FORMAT_R8G8_UINT = 20,
  VK_FORMAT_R8G8_SINT = 21,
  VK_FORMAT_R8G8_SRGB = 22,
  VK_FORMAT_R8G8B8_UNORM = 23,
  VK_FORMAT_R8G8B8_SNORM = 24,
  VK_FORMAT_R8G8B8_USCALED = 25,
  VK_FORMAT_R8G8B8_SSCALED = 26,
  VK_FORMAT_R8G8B8_UINT = 27,
  VK_FORMAT_R8G8B8_SINT = 28,
  VK_FORMAT_R8G8B8_SRGB = 29,
  VK_FORMAT_B8G8R8_UNORM = 30,
  VK_FORMAT_B8G8R8_SNORM = 31,
};
```
VK_FORMAT_B8G8R8_USCALED = 32,
VK_FORMAT_B8G8R8_SSCALED = 33,
VK_FORMAT_B8G8R8_UINT = 34,
VK_FORMAT_B8G8R8_SINT = 35,
VK_FORMAT_B8G8R8_SRGB = 36,
VK_FORMAT_R8G8B8A8_UNORM = 37,
VK_FORMAT_R8G8B8A8_SNORM = 38,
VK_FORMAT_R8G8B8A8_USCALED = 39,
VK_FORMAT_R8G8B8A8_SSCALED = 40,
VK_FORMAT_R8G8B8A8_UINT = 41,
VK_FORMAT_R8G8B8A8_SINT = 42,
VK_FORMAT_R8G8B8A8_SRGB = 43,
VK_FORMAT_B8G8R8A8_USCALED = 44,
VK_FORMAT_B8G8R8A8_SSCALED = 45,
VK_FORMAT_B8G8R8A8_UINT = 46,
VK_FORMAT_B8G8R8A8_SINT = 47,
VK_FORMAT_B8G8R8A8_SRGB = 48,
VK_FORMAT_A8B8G8R8_UNORM_PACK32 = 49,
VK_FORMAT_A8B8G8R8_SNORM_PACK32 = 50,
VK_FORMAT_A8B8G8R8_USCALED_PACK32 = 51,
VK_FORMAT_A8B8G8R8_SSCALED_PACK32 = 52,
VK_FORMAT_A8B8G8R8_UINT_PACK32 = 53,
VK_FORMAT_A8B8G8R8_SINT_PACK32 = 54,
VK_FORMAT_A8B8G8R8_SRGB_PACK32 = 55,
VK_FORMAT_A2R10G10B10_UNORM_PACK32 = 56,
VK_FORMAT_A2R10G10B10_SNORM_PACK32 = 57,
VK_FORMAT_A2R10G10B10_USCALED_PACK32 = 58,
VK_FORMAT_A2R10G10B10_SSCALED_PACK32 = 59,
VK_FORMAT_A2R10G10B10_UINT_PACK32 = 60,
VK_FORMAT_A2R10G10B10_SINT_PACK32 = 61,
VK_FORMAT_A2B10G10R10_UNORM_PACK32 = 62,
VK_FORMAT_A2B10G10R10_SNORM_PACK32 = 63,
VK_FORMAT_A2B10G10R10_USCALED_PACK32 = 64,
VK_FORMAT_A2B10G10R10_SSCALED_PACK32 = 65,
VK_FORMAT_A2B10G10R10_UINT_PACK32 = 66,
VK_FORMAT_A2B10G10R10_SINT_PACK32 = 67,
VK_FORMAT_R16_UNORM = 68,
VK_FORMAT_R16_SNORM = 69,
VK_FORMAT_R16_USCALED = 70,
VK_FORMAT_R16_SSCALED = 71,
VK_FORMAT_R16_UINT = 72,
VK_FORMAT_R16_SINT = 73,
VK_FORMAT_R16_SFLOAT = 74,
VK_FORMAT_R16G16_UNORM_PACK32 = 75,
VK_FORMAT_R16G16_SNORM_PACK32 = 76,
VK_FORMAT_R16G16_USCALED_PACK32 = 77,
VK_FORMAT_R16G16_SSCALED_PACK32 = 78,
VK_FORMAT_R16G16_UINT_PACK32 = 79,
VK_FORMAT_R16G16_SINT_PACK32 = 80,
VK_FORMAT_BC1_RGBA_SRGB_BLOCK = 134,
VK_FORMAT_BC2_UNORM_BLOCK = 135,
VK_FORMAT_BC2_SRGB_BLOCK = 136,
VK_FORMAT_BC3_UNORM_BLOCK = 137,
VK_FORMAT_BC3_SRGB_BLOCK = 138,
VK_FORMAT_BC4_UNORM_BLOCK = 139,
VK_FORMAT_BC4_SNORM_BLOCK = 140,
VK_FORMAT_BC5_UNORM_BLOCK = 141,
VK_FORMAT_BC5_SNORM_BLOCK = 142,
VK_FORMAT_BCGH_UFLOAT_BLOCK = 143,
VK_FORMAT_BCGH_SFLOAT_BLOCK = 144,
VK_FORMAT_BC7_UNORM_BLOCK = 145,
VK_FORMAT_BC7_SRGB_BLOCK = 146,
VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK = 147,
VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK = 148,
VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK = 149,
VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK = 150,
VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK = 151,
VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK = 152,
VK_FORMAT_EAC_R11_UNORM_BLOCK = 153,
VK_FORMAT_EAC_R11_SNORM_BLOCK = 154,
VK_FORMAT_EAC_R11G11_UNORM_BLOCK = 155,
VK_FORMAT_EAC_R11G11_SNORM_BLOCK = 156,
VK_FORMAT_ASTC_4x4_UNORM_BLOCK = 157,
VK_FORMAT_ASTC_4x4_SRGB_BLOCK = 158,
VK_FORMAT_ASTC_5x4_UNORM_BLOCK = 159,
VK_FORMAT_ASTC_5x4_SRGB_BLOCK = 160,
VK_FORMAT_ASTC_5x5_UNORM_BLOCK = 161,
VK_FORMAT_ASTC_5x5_SRGB_BLOCK = 162,
VK_FORMAT_ASTC_6x5_UNORM_BLOCK = 163,
VK_FORMAT_ASTC_6x5_SRGB_BLOCK = 164,
VK_FORMAT_ASTC_6x6_UNORM_BLOCK = 165,
VK_FORMAT_ASTC_6x6_SRGB_BLOCK = 166,
VK_FORMAT_ASTC_8x5_UNORM_BLOCK = 167,
VK_FORMAT_ASTC_8x5_SRGB_BLOCK = 168,
VK_FORMAT_ASTC_8x6_UNORM_BLOCK = 169,
VK_FORMAT_ASTC_8x6_SRGB_BLOCK = 170,
VK_FORMAT_ASTC_8x8_UNORM_BLOCK = 171,
VK_FORMAT_ASTC_8x8_SRGB_BLOCK = 172,
VK_FORMAT_ASTC_10x5_UNORM_BLOCK = 173,
VK_FORMAT_ASTC_10x5_SRGB_BLOCK = 174,
VK_FORMAT_ASTC_10x6_UNORM_BLOCK = 175,
VK_FORMAT_ASTC_10x6_SRGB_BLOCK = 176,
VK_FORMAT_ASTC_10x8_UNORM_BLOCK = 177,
VK_FORMAT_ASTC_10x8_SRGB_BLOCK = 178,
VK_FORMAT_ASTC_10x10_UNORM_BLOCK = 179,
VK_FORMAT_ASTC_10x10_SRGB_BLOCK = 180,
VK_FORMAT_ASTC_12x10_UNORM_BLOCK = 181,
VK_FORMAT_ASTC_12x10_SRGB_BLOCK = 182,
VK_FORMAT_ASTC_12x12_UNORM_BLOCK = 183,
VK_FORMAT_ASTC_12x12_SRGB_BLOCK = 184,
VK_FORMAT_G8B8R8_422_UNORM = 1000156000,
VK_FORMAT_B8G8R8G8_422_UNORM = 1000156001,
VK_FORMAT_G8B8_R8_3PLANE_420_UNORM = 1000156002,
VK_FORMAT_G8B8R8_2PLANE_420_UNORM = 1000156003,
VK_FORMAT_G8B8_R8_3PLANE_422_UNORM = 1000156004,
VK_FORMAT_G8B8R8_2PLANE_422_UNORM = 1000156005,
VK_FORMAT_G8B8_R8_3PLANE_444_UNORM = 1000156006,
VK_FORMAT_R10X6_UNORM_PACK16 = 1000156007,
VK_FORMAT_R10X6G10X6_UNORM_2PACK16 = 1000156008,
VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16 = 1000156009,
VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16 = 1000156010,
VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16 = 1000156011,
VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16 = 1000156012,
VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16 = 1000156013,
VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16 = 1000156014,
VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16 = 1000156015,
VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16 = 1000156016,
VK_FORMAT_R12X4_UNORM_PACK16 = 1000156017,
VK_FORMAT_R12X4G12X4_UNORM_2PACK16 = 1000156018,
VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16 = 1000156025,
// Provided by VK_VERSION_1_1
VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16 = 1000156026,
// Provided by VK_VERSION_1_1
VK_FORMAT_G16B16G16R16_422_UNORM = 1000156027,
// Provided by VK_VERSION_1_1
VK_FORMAT_B16G16R16_2PLANE_420_UNORM = 1000156028,
// Provided by VK_VERSION_1_1
VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM = 1000156029,
// Provided by VK_VERSION_1_1
VK_FORMAT_G16_B16R16_2PLANE_420_UNORM = 1000156030,
// Provided by VK_VERSION_1_1
VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM = 1000156031,
// Provided by VK_VERSION_1_1
VK_FORMAT_G16_B16R16_2PLANE_422_UNORM = 1000156032,
// Provided by VK_VERSION_1_1
VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM = 1000156033,
// Provided by VK_VERSION_1_1
VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16 = 1000330001,
// Provided by VK_VERSION_1_3
VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16 = 1000330002,
// Provided by VK_VERSION_1_3
VK_FORMAT_G16_B16R16_2PLANE_444_UNORM = 1000330003,
// Provided by VK_VERSION_1_3
VK_FORMAT_A4R4G4B4_UNORM_PACK16 = 1000340000,
// Provided by VK_VERSION_1_3
VK_FORMAT_A4B4G4R4_UNORM_PACK16 = 1000340001,
// Provided by VK_VERSION_1_3
VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK = 1000066000,
// Provided by VK_VERSION_1_3
VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK = 1000066001,
// Provided by VK_VERSION_1_3
VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK = 1000066002,
// Provided by VK_VERSION_1_3
VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK = 1000066003,
// Provided by VK_VERSION_1_3
VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK = 1000066004,
// Provided by VK_VERSION_1_3
VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK = 1000066005,
// Provided by VK_VERSION_1_3
VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK = 1000066006,
// Provided by VK_VERSION_1_3
VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK = 1000066007,
// Provided by VK_VERSION_1_3
VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK = 1000066008,
// Provided by VK_VERSION_1_3
VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK = 1000066009,
// Provided by VK_VERSION_1_3
VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK = 1000066010,
• **VK_FORMAT_UNDEFINED** specifies that the format is not specified.

• **VK_FORMAT_R4G4_UNORM_PACK8** specifies a two-component, 8-bit packed unsigned normalized format that has a 4-bit R component in bits 4..7, and a 4-bit G component in bits 0..3.

• **VK_FORMAT_R4G4B4A4_UNORM_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 4-bit R component in bits 12..15, a 4-bit G component in bits 8..11, a 4-bit B component in bits 4..7, and a 4-bit A component in bits 0..3.

• **VK_FORMAT_B4G4R4A4_UNORM_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 4-bit B component in bits 12..15, a 4-bit G component in bits 8..11, a 4-bit R component in bits 4..7, and a 4-bit A component in bits 0..3.

• **VK_FORMAT_A4R4G4B4_UNORM_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 4-bit A component in bits 12..15, a 4-bit R component in bits 8..11, a 4-bit G component in bits 4..7, and a 4-bit B component in bits 0..3.

• **VK_FORMAT_A4B4G4R4_UNORM_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 4-bit A component in bits 12..15, a 4-bit B component in bits 8..11, a 4-bit G component in bits 4..7, and a 4-bit R component in bits 0..3.

• **VK_FORMAT_R5G6B5_UNORM_PACK16** specifies a three-component, 16-bit packed unsigned normalized format that has a 5-bit R component in bits 11..15, a 6-bit G component in bits 5..10, and a 5-bit B component in bits 0..4.

• **VK_FORMAT_B5G6R5_UNORM_PACK16** specifies a three-component, 16-bit packed unsigned normalized format that has a 5-bit B component in bits 11..15, a 6-bit G component in bits 5..10, and a 5-bit R component in bits 0..4.

• **VK_FORMAT_R5G5B5A1_UNORM_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 5-bit R component in bits 11..15, a 5-bit G component in bits 6..10, a 5-bit B component in bits 1..5, and a 1-bit A component in bit 0.

• **VK_FORMAT_B5G5R5A1_UNORM_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 5-bit B component in bits 11..15, a 5-bit G component in bits 6..10, a 5-bit R component in bits 1..5, and a 1-bit A component in bit 0.

• **VK_FORMAT_A1R5G5B5_UNORM_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 1-bit A component in bit 15, a 5-bit R component in bits 10..14, a 5-bit G component in bits 5..9, and a 5-bit B component in bits 0..4.

• **VK_FORMAT_R8_UNORM** specifies a one-component, 8-bit unsigned normalized format that has a single 8-bit R component.

• **VK_FORMAT_R8_SNORM** specifies a one-component, 8-bit signed normalized format that has a single 8-bit R component.

• **VK_FORMAT_R8_USCALED** specifies a one-component, 8-bit unsigned scaled integer format that has a
single 8-bit R component.

- **VK_FORMAT_R8_SSCALED** specifies a one-component, 8-bit signed scaled integer format that has a single 8-bit R component.
- **VK_FORMAT_R8_UINT** specifies a one-component, 8-bit unsigned integer format that has a single 8-bit R component.
- **VK_FORMAT_R8_SINT** specifies a one-component, 8-bit signed integer format that has a single 8-bit R component.
- **VK_FORMAT_R8_SRGB** specifies a one-component, 8-bit unsigned normalized format that has a single 8-bit R component stored with sRGB nonlinear encoding.
- **VK_FORMAT_R8G8_UNORM** specifies a two-component, 16-bit unsigned normalized format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.
- **VK_FORMAT_R8G8_SNORM** specifies a two-component, 16-bit signed normalized format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.
- **VK_FORMAT_R8G8_USCALED** specifies a two-component, 16-bit unsigned scaled integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.
- **VK_FORMAT_R8G8_SSCALED** specifies a two-component, 16-bit signed scaled integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.
- **VK_FORMAT_R8G8_UINT** specifies a two-component, 16-bit unsigned integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.
- **VK_FORMAT_R8G8_SINT** specifies a two-component, 16-bit signed integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.
- **VK_FORMAT_R8G8_SRGB** specifies a two-component, 16-bit unsigned normalized format that has an 8-bit R component stored with sRGB nonlinear encoding in byte 0, and an 8-bit G component stored with sRGB nonlinear encoding in byte 1.
- **VK_FORMAT_R8G8B8_UNORM** specifies a three-component, 24-bit unsigned normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
- **VK_FORMAT_R8G8B8_SNORM** specifies a three-component, 24-bit signed normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
- **VK_FORMAT_R8G8B8_USCALED** specifies a three-component, 24-bit unsigned scaled format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
- **VK_FORMAT_R8G8B8_SSCALED** specifies a three-component, 24-bit signed scaled integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
- **VK_FORMAT_R8G8B8_UINT** specifies a three-component, 24-bit unsigned integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
- **VK_FORMAT_R8G8B8_SINT** specifies a three-component, 24-bit signed integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
- **VK_FORMAT_R8G8B8_SRGB** specifies a three-component, 24-bit unsigned normalized format that has an 8-bit R component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, and an 8-bit B component stored with sRGB nonlinear encoding in byte 2.
nonlinear encoding in byte 2.

- **VK_FORMAT_B8G8R8_UNORM** specifies a three-component, 24-bit unsigned normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

- **VK_FORMAT_B8G8R8_SNORM** specifies a three-component, 24-bit signed normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

- **VK_FORMAT_B8G8R8_USCALED** specifies a three-component, 24-bit unsigned scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

- **VK_FORMAT_B8G8R8_SSCALED** specifies a three-component, 24-bit signed scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

- **VK_FORMAT_B8G8R8_UINT** specifies a three-component, 24-bit unsigned integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

- **VK_FORMAT_B8G8R8_SINT** specifies a three-component, 24-bit signed integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

- **VK_FORMAT_B8G8R8_SRGB** specifies a three-component, 24-bit unsigned normalized format that has an 8-bit B component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, and an 8-bit R component stored with sRGB nonlinear encoding in byte 2.

- **VK_FORMAT_R8G8B8A8_UNORM** specifies a four-component, 32-bit unsigned normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_R8G8B8A8_SNORM** specifies a four-component, 32-bit signed normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_R8G8B8A8_USCALED** specifies a four-component, 32-bit unsigned scaled format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_R8G8B8A8_SSCALED** specifies a four-component, 32-bit signed scaled format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_R8G8B8A8_UINT** specifies a four-component, 32-bit unsigned integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_R8G8B8A8_SINT** specifies a four-component, 32-bit signed integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_R8G8B8A8_SRGB** specifies a four-component, 32-bit unsigned normalized format that has an 8-bit R component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, an 8-bit B component stored with sRGB nonlinear encoding in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_B8G8R8A8_UNORM** specifies a four-component, 32-bit unsigned normalized format that
has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_B8G8R8A8_SNORM** specifies a four-component, 32-bit signed normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_B8G8R8A8_USCALED** specifies a four-component, 32-bit unsigned scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_B8G8R8A8_SSCALED** specifies a four-component, 32-bit signed scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_B8G8R8A8_UINT** specifies a four-component, 32-bit unsigned integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_B8G8R8A8_SINT** specifies a four-component, 32-bit signed integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_B8G8R8A8_SRGB** specifies a four-component, 32-bit unsigned normalized format that has an 8-bit B component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, an 8-bit R component stored with sRGB nonlinear encoding in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_A8B8G8R8_UNORM_PACK32** specifies a four-component, 32-bit packed unsigned normalized format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

- **VK_FORMAT_A8B8G8R8_SNORM_PACK32** specifies a four-component, 32-bit packed signed normalized format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

- **VK_FORMAT_A8B8G8R8_USCALED_PACK32** specifies a four-component, 32-bit packed unsigned scaled integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

- **VK_FORMAT_A8B8G8R8_SSCALED_PACK32** specifies a four-component, 32-bit packed signed scaled integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

- **VK_FORMAT_A8B8G8R8_UINT_PACK32** specifies a four-component, 32-bit packed unsigned integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

- **VK_FORMAT_A8B8G8R8_SINT_PACK32** specifies a four-component, 32-bit packed signed integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

- **VK_FORMAT_A8B8G8R8_SRGB_PACK32** specifies a four-component, 32-bit packed unsigned normalized format that has an 8-bit A component in bits 24..31, an 8-bit B component stored with sRGB nonlinear encoding in bits 16..23, an 8-bit G component stored with sRGB nonlinear encoding in bits 8..15, and an 8-bit R component stored with sRGB nonlinear encoding in bits 0..7.
- **VK_FORMAT_A2R10G10B10_UNORM_PACK32** specifies a four-component, 32-bit packed unsigned normalized format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

- **VK_FORMAT_A2R10G10B10_SNORM_PACK32** specifies a four-component, 32-bit packed signed normalized format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

- **VK_FORMAT_A2R10G10B10_USCALED_PACK32** specifies a four-component, 32-bit packed unsigned scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

- **VK_FORMAT_A2R10G10B10_SSCALED_PACK32** specifies a four-component, 32-bit packed signed scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

- **VK_FORMAT_A2R10G10B10_UINT_PACK32** specifies a four-component, 32-bit packed unsigned integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

- **VK_FORMAT_A2R10G10B10_SINT_PACK32** specifies a four-component, 32-bit packed signed integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

- **VK_FORMAT_A2B10G10R10_UNORM_PACK32** specifies a four-component, 32-bit packed unsigned normalized format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

- **VK_FORMAT_A2B10G10R10_SNORM_PACK32** specifies a four-component, 32-bit packed signed normalized format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

- **VK_FORMAT_A2B10G10R10_USCALED_PACK32** specifies a four-component, 32-bit packed unsigned scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

- **VK_FORMAT_A2B10G10R10_SSCALED_PACK32** specifies a four-component, 32-bit packed signed scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

- **VK_FORMAT_A2B10G10R10_UINT_PACK32** specifies a four-component, 32-bit packed unsigned integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

- **VK_FORMAT_A2B10G10R10_SINT_PACK32** specifies a four-component, 32-bit packed signed integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

- **VK_FORMAT_R16_UNORM** specifies a one-component, 16-bit unsigned normalized format that has a single 16-bit R component.

- **VK_FORMAT_R16_SNORM** specifies a one-component, 16-bit signed normalized format that has a single 16-bit R component.

- **VK_FORMAT_R16_USCALED** specifies a one-component, 16-bit unsigned scaled integer format that has a single 16-bit R component.
• **VK_FORMAT_R16_SSCALED** specifies a one-component, 16-bit signed scaled integer format that has a single 16-bit R component.

• **VK_FORMAT_R16_UINT** specifies a one-component, 16-bit unsigned integer format that has a single 16-bit R component.

• **VK_FORMAT_R16_SINT** specifies a one-component, 16-bit signed integer format that has a single 16-bit R component.

• **VK_FORMAT_R16_SFLOAT** specifies a one-component, 16-bit signed floating-point format that has a single 16-bit R component.

• **VK_FORMAT_R16G16_UNORM** specifies a two-component, 32-bit unsigned normalized format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

• **VK_FORMAT_R16G16_SNORM** specifies a two-component, 32-bit signed normalized format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

• **VK_FORMAT_R16G16_USCALED** specifies a two-component, 32-bit unsigned scaled integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

• **VK_FORMAT_R16G16_SSCALED** specifies a two-component, 32-bit signed scaled integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

• **VK_FORMAT_R16G16_UINT** specifies a two-component, 32-bit unsigned integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

• **VK_FORMAT_R16G16_SINT** specifies a two-component, 32-bit signed integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

• **VK_FORMAT_R16G16_SFLOAT** specifies a two-component, 32-bit signed floating-point format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

• **VK_FORMAT_R16G16B16_UNORM** specifies a three-component, 48-bit unsigned normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

• **VK_FORMAT_R16G16B16_SNORM** specifies a three-component, 48-bit signed normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

• **VK_FORMAT_R16G16B16_USCALED** specifies a three-component, 48-bit unsigned scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

• **VK_FORMAT_R16G16B16_SSCALED** specifies a three-component, 48-bit signed scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

• **VK_FORMAT_R16G16B16_UINT** specifies a three-component, 48-bit unsigned integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

• **VK_FORMAT_R16G16B16_SINT** specifies a three-component, 48-bit signed integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

• **VK_FORMAT_R16G16B16_SFLOAT** specifies a three-component, 48-bit signed floating-point format that
has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

- **VK_FORMAT_R16G16B16A16_UNORM** specifies a four-component, 64-bit unsigned normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

- **VK_FORMAT_R16G16B16A16_SNORM** specifies a four-component, 64-bit signed normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

- **VK_FORMAT_R16G16B16A16_USCALED** specifies a four-component, 64-bit unsigned scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

- **VK_FORMAT_R16G16B16A16_SSCALED** specifies a four-component, 64-bit signed scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

- **VK_FORMAT_R16G16B16A16_UINT** specifies a four-component, 64-bit unsigned integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

- **VK_FORMAT_R16G16B16A16_SINT** specifies a four-component, 64-bit signed integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

- **VK_FORMAT_R16G16B16A16_SFLOAT** specifies a four-component, 64-bit signed floating-point format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

- **VK_FORMAT_R32_UINT** specifies a one-component, 32-bit unsigned integer format that has a single 32-bit R component.

- **VK_FORMAT_R32_SINT** specifies a one-component, 32-bit signed integer format that has a single 32-bit R component.

- **VK_FORMAT_R32_SFLOAT** specifies a one-component, 32-bit signed floating-point format that has a single 32-bit R component.

- **VK_FORMAT_R32G32_UINT** specifies a two-component, 64-bit unsigned integer format that has a 32-bit R component in bytes 0..3, and a 32-bit G component in bytes 4..7.

- **VK_FORMAT_R32G32_SINT** specifies a two-component, 64-bit signed integer format that has a 32-bit R component in bytes 0..3, and a 32-bit G component in bytes 4..7.

- **VK_FORMAT_R32G32_SFLOAT** specifies a two-component, 64-bit signed floating-point format that has a 32-bit R component in bytes 0..3, and a 32-bit G component in bytes 4..7.

- **VK_FORMAT_R32G32B32_UINT** specifies a three-component, 96-bit unsigned integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, and a 32-bit B component in bytes 8..11.

- **VK_FORMAT_R32G32B32_SINT** specifies a three-component, 96-bit signed integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, and a 32-bit B component in bytes 8..11.
• **VK_FORMAT_R32G32B32_SFLOAT** specifies a three-component, 96-bit signed floating-point format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, and a 32-bit B component in bytes 8..11.

• **VK_FORMAT_R32G32B32A32_UINT** specifies a four-component, 128-bit unsigned integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, a 32-bit B component in bytes 8..11, and a 32-bit A component in bytes 12..15.

• **VK_FORMAT_R32G32B32A32_SINT** specifies a four-component, 128-bit signed integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, a 32-bit B component in bytes 8..11, and a 32-bit A component in bytes 12..15.

• **VK_FORMAT_R32G32B32A32_SFLOAT** specifies a four-component, 128-bit signed floating-point format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, a 32-bit B component in bytes 8..11, and a 32-bit A component in bytes 12..15.

• **VK_FORMAT_R64_UINT** specifies a one-component, 64-bit unsigned integer format that has a single 64-bit R component.

• **VK_FORMAT_R64_SINT** specifies a one-component, 64-bit signed integer format that has a single 64-bit R component.

• **VK_FORMAT_R64_SFLOAT** specifies a one-component, 64-bit signed floating-point format that has a single 64-bit R component.

• **VK_FORMAT_R64G64_UINT** specifies a two-component, 128-bit unsigned integer format that has a 64-bit R component in bytes 0..7, and a 64-bit G component in bytes 8..15.

• **VK_FORMAT_R64G64_SINT** specifies a two-component, 128-bit signed integer format that has a 64-bit R component in bytes 0..7, and a 64-bit G component in bytes 8..15.

• **VK_FORMAT_R64G64_SFLOAT** specifies a two-component, 128-bit signed floating-point format that has a 64-bit R component in bytes 0..7, and a 64-bit G component in bytes 8..15.

• **VK_FORMAT_R64G64B64_UINT** specifies a three-component, 192-bit unsigned integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, and a 64-bit B component in bytes 16..23.

• **VK_FORMAT_R64G64B64_SINT** specifies a three-component, 192-bit signed integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, and a 64-bit B component in bytes 16..23.

• **VK_FORMAT_R64G64B64_SFLOAT** specifies a three-component, 192-bit signed floating-point format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, and a 64-bit B component in bytes 16..23.

• **VK_FORMAT_R64G64B64A64_UINT** specifies a four-component, 256-bit unsigned integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, a 64-bit B component in bytes 16..23, and a 64-bit A component in bytes 24..31.

• **VK_FORMAT_R64G64B64A64_SINT** specifies a four-component, 256-bit signed integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, a 64-bit B component in bytes 16..23, and a 64-bit A component in bytes 24..31.

• **VK_FORMAT_R64G64B64A64_SFLOAT** specifies a four-component, 256-bit signed floating-point format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, a 64-bit B component in bytes 16..23, and a 64-bit A component in bytes 24..31.
• **VK_FORMAT_B10G11R11_UFLOAT_PACK32** specifies a three-component, 32-bit packed unsigned floating-point format that has a 10-bit B component in bits 22..31, an 11-bit G component in bits 11..21, an 11-bit R component in bits 0..10. See [Unsigned 10-Bit Floating-Point Numbers](#) and [Unsigned 11-Bit Floating-Point Numbers](#).

• **VK_FORMAT_E5B9G9R9_UFLOAT_PACK32** specifies a three-component, 32-bit packed unsigned floating-point format that has a 5-bit shared exponent in bits 27..31, a 9-bit B component mantissa in bits 18..26, a 9-bit G component mantissa in bits 9..17, and a 9-bit R component mantissa in bits 0..8.

• **VK_FORMAT_D16_UNORM** specifies a one-component, 16-bit unsigned normalized format that has a single 16-bit depth component.

• **VK_FORMAT_X8_D24_UNORM_PACK32** specifies a two-component, 32-bit format that has 24 unsigned normalized bits in the depth component and, optionally, 8 bits that are unused.

• **VK_FORMAT_D32_SFLOAT** specifies a one-component, 32-bit signed floating-point format that has 32 bits in the depth component.

• **VK_FORMAT_S8_UINT** specifies a one-component, 8-bit unsigned integer format that has 8 bits in the stencil component.

• **VK_FORMAT_D16_UNORM_S8_UINT** specifies a two-component, 24-bit format that has 16 unsigned normalized bits in the depth component and 8 unsigned integer bits in the stencil component.

• **VK_FORMAT_D24_UNORM_S8_UINT** specifies a two-component, 32-bit packed format that has 8 unsigned integer bits in the stencil component, and 24 unsigned normalized bits in the depth component.

• **VK_FORMAT_D32_SFLOAT_S8_UINT** specifies a two-component format that has 32 signed float bits in the depth component and 8 unsigned integer bits in the stencil component. There are optionally 24 bits that are unused.

• **VK_FORMAT_BC1_RGB_UNORM_BLOCK** specifies a three-component, block-compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data. This format has no alpha and is considered opaque.

• **VK_FORMAT_BC1_RGB_SRGB_BLOCK** specifies a three-component, block-compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data with sRGB nonlinear encoding. This format has no alpha and is considered opaque.

• **VK_FORMAT_BC1_RGBA_UNORM_BLOCK** specifies a four-component, block-compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data, and provides 1 bit of alpha.

• **VK_FORMAT_BC1_RGBA_SRGB_BLOCK** specifies a four-component, block-compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data with sRGB nonlinear encoding, and provides 1 bit of alpha.

• **VK_FORMAT_BC2_UNORM_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values.

• **VK_FORMAT_BC2_SRGB_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values with sRGB nonlinear encoding.
• **VK_FORMAT_BC3_UNORM_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values.

• **VK_FORMAT_BC3_SRGB_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values with sRGB nonlinear encoding.

• **VK_FORMAT_BC4_UNORM_BLOCK** specifies a one-component, block-compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized red texel data.

• **VK_FORMAT_BC4_SNORM_BLOCK** specifies a one-component, block-compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of signed normalized red texel data.

• **VK_FORMAT_BC5_UNORM_BLOCK** specifies a two-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RG texel data with the first 64 bits encoding red values followed by 64 bits encoding green values.

• **VK_FORMAT_BC5_SNORM_BLOCK** specifies a two-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of signed normalized RG texel data with the first 64 bits encoding red values followed by 64 bits encoding green values.

• **VK_FORMAT_BC6H_UFLOAT_BLOCK** specifies a three-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned floating-point RGB texel data.

• **VK_FORMAT_BC6H_SFLOAT_BLOCK** specifies a three-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of signed floating-point RGB texel data.

• **VK_FORMAT_BC7_UNORM_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data.

• **VK_FORMAT_BC7_SRGB_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

• **VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK** specifies a three-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data. This format has no alpha and is considered opaque.

• **VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK** specifies a three-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data with sRGB nonlinear encoding. This format has no alpha and is considered opaque.

• **VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK** specifies a four-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data, and provides 1 bit of alpha.

• **VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK** specifies a four-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data with sRGB nonlinear encoding, and provides 1 bit of alpha.

• **VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK** specifies a four-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBtexel data.
RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values.

- **VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK** specifies a four-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values with sRGB nonlinear encoding applied.

- **VK_FORMAT_EAC_R11_UNORM_BLOCK** specifies a one-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized red texel data.

- **VK_FORMAT_EAC_R11_SNORM_BLOCK** specifies a one-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of signed normalized red texel data.

- **VK_FORMAT_EAC_R11G11_UNORM_BLOCK** specifies a two-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RG texel data with the first 64 bits encoding red values followed by 64 bits encoding green values.

- **VK_FORMAT_EAC_R11G11_SNORM_BLOCK** specifies a two-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of signed normalized RG texel data with the first 64 bits encoding red values followed by 64 bits encoding green values.

- **VK_FORMAT_ASTC_4x4_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_4x4_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_5x4_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5×4 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_5x4_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5×4 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5×4 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_5x5_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5×5 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_5x5_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5×5 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5×5 rectangle of signed floating-point RGBA texel data.
• **VK_FORMAT_ASTC_6x5_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6×5 rectangle of unsigned normalized RGBA texel data.

• **VK_FORMAT_ASTC_6x5_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6×5 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

• **VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6×5 rectangle of signed floating-point RGBA texel data.

• **VK_FORMAT_ASTC_6x6_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6×6 rectangle of unsigned normalized RGBA texel data.

• **VK_FORMAT_ASTC_6x6_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6×6 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

• **VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6×6 rectangle of signed floating-point RGBA texel data.

• **VK_FORMAT_ASTC_8x5_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×5 rectangle of unsigned normalized RGBA texel data.

• **VK_FORMAT_ASTC_8x5_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×5 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

• **VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×5 rectangle of signed floating-point RGBA texel data.

• **VK_FORMAT_ASTC_8x6_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×6 rectangle of unsigned normalized RGBA texel data.

• **VK_FORMAT_ASTC_8x6_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×6 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

• **VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×6 rectangle of signed floating-point RGBA texel data.

• **VK_FORMAT_ASTC_8x8_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×8 rectangle of unsigned normalized RGBA texel data.

• **VK_FORMAT_ASTC_8x8_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×8 rectangle of unsigned normalized RGBA texel data.
texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 8×8 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_10x5_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×5 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_10x5_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×5 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×5 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_10x6_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×6 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_10x6_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×6 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×6 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_10x8_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×8 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_10x8_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×8 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×8 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_10x10_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×10 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_10x10_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×10 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×10 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_12x10_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12×10 rectangle of unsigned normalized RGBA texel data.
texel data.

- **VK_FORMAT_ASTC_12x10_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12×10 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12×10 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_12x12_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12×12 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_12x12_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12×12 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12×12 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_G8B8G8R8_422_UNORM** specifies a four-component, 32-bit format containing a pair of G components, an R component, and a B component, collectively encoding a 2×1 rectangle of unsigned normalized RGB texel data. One G value is present at each \( i \) coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has an 8-bit G component for the even \( i \) coordinate in byte 0, an 8-bit B component in byte 1, an 8-bit G component for the odd \( i \) coordinate in byte 2, and an 8-bit R component in byte 3. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a 2×1 compressed texel block.

- **VK_FORMAT_B8G8R8G8_422_UNORM** specifies a four-component, 32-bit format containing a pair of G components, an R component, and a B component, collectively encoding a 2×1 rectangle of unsigned normalized RGB texel data. One G value is present at each \( i \) coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has an 8-bit B component in byte 0, an 8-bit G component for the even \( i \) coordinate in byte 1, an 8-bit R component in byte 2, and an 8-bit G component for the odd \( i \) coordinate in byte 3. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a 2×1 compressed texel block.

- **VK_FORMAT_G8B8G8R8_422_UNORM** specifies an unsigned normalized multi-planar format that has an 8-bit G component in plane 0, an 8-bit B component in plane 1, and an 8-bit R component in plane 2. The horizontal and vertical dimensions of the R and B planes are halved relative to the image dimensions, and each R and B component is shared with the G components for which \( \lfloor i_G \times 0.5 \rfloor = i_B = i_R \) and \( \lfloor j_G \times 0.5 \rfloor = j_B = j_R \). The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane. This format only supports images with a width and height that is a multiple of two.

- **VK_FORMAT_G8_B8R8_2PLANE_420_UNORM** specifies an unsigned normalized multi-planar format that
has an 8-bit G component in plane 0, and a two-component, 16-bit BR plane 1 consisting of an 8-bit B component in byte 0 and an 8-bit R component in byte 1. The horizontal and vertical dimensions of the BR plane are halved relative to the image dimensions, and each R and B value is shared with the G components for which \(|i_G \times 0.5| = i_B = i_R\) and \(|j_G \times 0.5| = j_B = j_R\). The location of each plane when this image is in linear layout can be determined via vkGetImageSubresourceLayout, using VK_IMAGE_ASPECT_PLANE_0_BIT for the G plane, and VK_IMAGE_ASPECT_PLANE_1_BIT for the BR plane. This format only supports images with a width and height that is a multiple of two.

- **VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM** specifies an unsigned normalized multi-planar format that has an 8-bit G component in plane 0, an 8-bit B component in plane 1, and an 8-bit R component in plane 2. The horizontal dimension of the R and B plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which \(|i_G \times 0.5| = i_B = i_R\). The location of each plane when this image is in linear layout can be determined via vkGetImageSubresourceLayout, using VK_IMAGE_ASPECT_PLANE_0_BIT for the G plane, VK_IMAGE_ASPECT_PLANE_1_BIT for the B plane, and VK_IMAGE_ASPECT_PLANE_2_BIT for the R plane. This format only supports images with a width that is a multiple of two.

- **VK_FORMAT_G8_B8R8_2PLANE_422_UNORM** specifies an unsigned normalized multi-planar format that has an 8-bit G component in plane 0, and a two-component, 16-bit BR plane 1 consisting of an 8-bit B component in byte 0 and an 8-bit R component in byte 1. The horizontal dimension of the BR plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which \(|i_G \times 0.5| = i_B = i_R\). The location of each plane when this image is in linear layout can be determined via vkGetImageSubresourceLayout, using VK_IMAGE_ASPECT_PLANE_0_BIT for the G plane, and VK_IMAGE_ASPECT_PLANE_1_BIT for the BR plane.

- **VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM** specifies an unsigned normalized multi-planar format that has an 8-bit G component in plane 0, an 8-bit B component in plane 1, and an 8-bit R component in plane 2. Each plane has the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via vkGetImageSubresourceLayout, using VK_IMAGE_ASPECT_PLANE_0_BIT for the G plane, VK_IMAGE_ASPECT_PLANE_1_BIT for the B plane, and VK_IMAGE_ASPECT_PLANE_2_BIT for the R plane.

- **VK_FORMAT_R10X6_UNORM_PACK16** specifies a one-component, 16-bit unsigned normalized format that has a single 10-bit R component in the top 10 bits of a 16-bit word, with the bottom 6 bits unused.

- **VK_FORMAT_R10X6G10X6_UNORM_2PACK16** specifies a two-component, 32-bit unsigned normalized format that has a 10-bit R component in the top 10 bits of the word in bytes 0..1, and a 10-bit G component in the top 10 bits of the word in bytes 2..3, with the bottom 6 bits of each word unused.

- **VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16** specifies a four-component, 64-bit unsigned normalized format that has a 10-bit R component in the top 10 bits of the word in bytes 0..1, a 10-bit G component in the top 10 bits of the word in bytes 2..3, a 10-bit B component in the top 10 bits of the word in bytes 4..5, and a 10-bit A component in the top 10 bits of the word in bytes 6..7, with the bottom 6 bits of each word unused.

- **VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16** specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a 2×1 rectangle of unsigned normalized RGB texel data. One G value is present at each i
coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 10-bit G component for the even $i$ coordinate in the top 10 bits of the word in bytes 0..1, a 10-bit B component in the top 10 bits of the word in bytes 2..3, a 10-bit G component for the odd $i$ coordinate in the top 10 bits of the word in bytes 4..5, and a 10-bit R component in the top 10 bits of the word in bytes 6..7, with the bottom 6 bits of each word unused. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a $2\times1$ compressed texel block.

- **VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16** specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a $2\times1$ rectangle of unsigned normalized RGB texel data. One G value is present at each $i$ coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 10-bit B component in the top 10 bits of the word in bytes 0..1, a 10-bit G component for the even $i$ coordinate in the top 10 bits of the word in bytes 2..3, a 10-bit R component in the top 10 bits of the word in bytes 4..5, and a 10-bit G component for the odd $i$ coordinate in the top 10 bits of the word in bytes 6..7, with the bottom 6 bits of each word unused. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a $2\times1$ compressed texel block.

- **VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16** specifies an unsigned normalized multi-planar format that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, a 10-bit B component in the top 10 bits of each 16-bit word of plane 1, and a 10-bit R component in the top 10 bits of each 16-bit word of plane 2, with the bottom 6 bits of each word unused. The horizontal and vertical dimensions of the R and B planes are halved relative to the image dimensions, and each R and B component is shared with the G components for which $\lfloor i_G \times 0.5 \rfloor = i_B = i_R$ and $\lfloor j_G \times 0.5 \rfloor = j_R = j_B$. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane. This format only supports images with a width and height that is a multiple of two.

- **VK_FORMAT_G10X6_B10X6_R10X6_2PLANE_420_UNORM_3PACK16** specifies an unsigned normalized multi-planar format that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 10-bit B component in the top 10 bits of the word in bytes 0..1, and a 10-bit R component in the top 10 bits of the word in bytes 2..3, with the bottom 6 bits of each word unused. The horizontal and vertical dimensions of the BR plane are halved relative to the image dimensions, and each R and B value is shared with the G components for which $\lfloor i_G \times 0.5 \rfloor = i_B = i_R$ and $\lfloor j_G \times 0.5 \rfloor = j_R = j_B$. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane. This format only supports images with a width and height that is a multiple of two.

- **VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16** specifies an unsigned normalized multi-planar format that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, a 10-bit B component in the top 10 bits of each 16-bit word of plane 1, and a 10-bit R component in the top 10 bits of each 16-bit word of plane 2, with the bottom 6 bits of each word unused. The horizontal dimension of the R and B plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which $\lfloor i_G \times 0.5 \rfloor = i_R = i_B$. The location of each
plane when this image is in linear layout can be determined via \( \text{vkGetImageSubresourceLayout} \), using \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) for the G plane, \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \) for the B plane, and \( \text{VK_IMAGE_ASPECT_PLANE_2_BIT} \) for the R plane. This format only supports images with a width that is a multiple of two.

- **VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16** specifies an unsigned normalized multi-planar format that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 10-bit B component in the top 10 bits of the word in bytes 0..1, and a 10-bit R component in the top 10 bits of the word in bytes 2..3, with the bottom 6 bits of each word unused. The horizontal dimension of the BR plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which \( 1_G \times 0.5 \cdot \overline{i} = i_B = i_R \). The location of each plane when this image is in linear layout can be determined via \( \text{vkGetImageSubresourceLayout} \), using \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) for the G plane, and \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \) for the BR plane. This format only supports images with a width that is a multiple of two.

- **VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16** specifies an unsigned normalized multi-planar format that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, a 10-bit B component in the top 10 bits of each 16-bit word of plane 1, and a 10-bit R component in the top 10 bits of each 16-bit word of plane 2, with the bottom 6 bits of each word unused. Each plane has the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via \( \text{vkGetImageSubresourceLayout} \), using \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) for the G plane, \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \) for the B plane, and \( \text{VK_IMAGE_ASPECT_PLANE_2_BIT} \) for the R plane.

- **VK_FORMAT_R12X4_UNORM_PACK16** specifies a one-component, 16-bit unsigned normalized format that has a single 12-bit R component in the top 12 bits of a 16-bit word, with the bottom 4 bits unused.

- **VK_FORMAT_R12X4G12X4_UNORM_2PACK16** specifies a two-component, 32-bit unsigned normalized format that has a 12-bit R component in the top 12 bits of the word in bytes 0..1, and a 12-bit G component in the top 12 bits of the word in bytes 2..3, with the bottom 4 bits of each word unused.

- **VK_FORMAT_R12X4G12X4B12X4A12X4_UNORM_4PACK16** specifies a four-component, 64-bit unsigned normalized format that has a 12-bit R component in the top 12 bits of the word in bytes 0..1, a 12-bit G component in the top 12 bits of the word in bytes 2..3, a 12-bit B component in the top 12 bits of the word in bytes 4..5, and a 12-bit A component in the top 12 bits of the word in bytes 6..7, with the bottom 4 bits of each word unused.

- **VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16** specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a 2×1 rectangle of unsigned normalized RGB texel data. One G value is present at each \( i \) coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 12-bit G component for the even \( i \) coordinate in the top 12 bits of the word in bytes 0..1, a 12-bit B component in the top 12 bits of the word in bytes 2..3, a 12-bit G component for the odd \( i \) coordinate in the top 12 bits of the word in bytes 4..5, and a 12-bit R component in the top 12 bits of the word in bytes 6..7, with the bottom 4 bits of each word unused. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a 2×1 compressed texel block.
• **VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16** specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a 2×1 rectangle of unsigned normalized RGB texel data. One G value is present at each i coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 12-bit B component in the top 12 bits of the word in bytes 0..1, a 12-bit G component for the even i coordinate in the top 12 bits of the word in bytes 2..3, a 12-bit R component in the top 12 bits of the word in bytes 4..5, and a 12-bit G component for the odd i coordinate in the top 12 bits of the word in bytes 6..7, with the bottom 4 bits of each word unused. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a 2×1 compressed texel block.

• **VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16** specifies an unsigned normalized multi-planar format that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, a 12-bit B component in the top 12 bits of each 16-bit word of plane 1, and a 12-bit R component in the top 12 bits of each 16-bit word of plane 2, with the bottom 4 bits of each word unused. The horizontal and vertical dimensions of the R and B planes are halved relative to the image dimensions, and each R and B component is shared with the G components for which \(|i_G \times 0.5| = i_B = i_R \) and \(|j_G \times 0.5| = j_B = j_R\). The location of each plane when this image is in linear layout can be determined via \texttt{vkGetImageSubresourceLayout}, using \texttt{VK_IMAGE_ASPECT_PLANE_0_BIT} for the G plane, \texttt{VK_IMAGE_ASPECT_PLANE_1_BIT} for the B plane, and \texttt{VK_IMAGE_ASPECT_PLANE_2_BIT} for the R plane. This format only supports images with a width and height that is a multiple of two.

• **VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16** specifies an unsigned normalized multi-planar format that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 12-bit B component in the top 12 bits of the word in bytes 0..1, and a 12-bit R component in the top 12 bits of the word in bytes 2..3, with the bottom 4 bits of each word unused. The horizontal and vertical dimensions of the BR plane are halved relative to the image dimensions, and each R and B value is shared with the G components for which \(|i_G \times 0.5| = i_B = i_R \) and \(|j_G \times 0.5| = j_B = j_R\). The location of each plane when this image is in linear layout can be determined via \texttt{vkGetImageSubresourceLayout}, using \texttt{VK_IMAGE_ASPECT_PLANE_0_BIT} for the G plane, and \texttt{VK_IMAGE_ASPECT_PLANE_1_BIT} for the BR plane. This format only supports images with a width and height that is a multiple of two.

• **VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16** specifies an unsigned normalized multi-planar format that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, a 12-bit B component in the top 12 bits of each 16-bit word of plane 1, and a 12-bit R component in the top 12 bits of each 16-bit word of plane 2, with the bottom 4 bits of each word unused. The horizontal dimension of the R and B plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which \(|i_G \times 0.5| = i_B = i_R \). The location of each plane when this image is in linear layout can be determined via \texttt{vkGetImageSubresourceLayout}, using \texttt{VK_IMAGE_ASPECT_PLANE_0_BIT} for the G plane, \texttt{VK_IMAGE_ASPECT_PLANE_1_BIT} for the B plane, and \texttt{VK_IMAGE_ASPECT_PLANE_2_BIT} for the R plane. This format only supports images with a width that is a multiple of two.

• **VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16** specifies an unsigned normalized multi-planar format that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 12-bit B component in the top 12 bits of the word in bytes 0..1, and a 12-bit R component in the top 12 bits of the word in bytes 2..3, with
the bottom 4 bits of each word unused. The horizontal dimension of the BR plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which \( i_G \times 0.5 = i_B = i_R \). The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane. This format only supports images with a width that is a multiple of two.

- **VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16** specifies an unsigned normalized multi-planar format that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, a 12-bit B component in the top 12 bits of each 16-bit word of plane 1, and a 12-bit R component in the top 12 bits of each 16-bit word of plane 2, with the bottom 4 bits of each word unused. Each plane has the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane.

- **VK_FORMAT_G16B16G16R16_422_UNORM** specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a 2×1 rectangle of unsigned normalized RGB texel data. One G value is present at each \( i \) coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 16-bit G component for the even \( i \) coordinate in the word in bytes 0..1, a 16-bit B component in the word in bytes 2..3, a 16-bit G component for the odd \( i \) coordinate in the word in bytes 4..5, and a 16-bit R component in the word in bytes 6..7. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a 2×1 compressed texel block.

- **VK_FORMAT_B16G16R16G16_422_UNORM** specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a 2×1 rectangle of unsigned normalized RGB texel data. One G value is present at each \( i \) coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 16-bit B component in the word in bytes 0..1, a 16-bit G component for the even \( i \) coordinate in the word in bytes 2..3, a 16-bit R component in the word in bytes 4..5, and a 16-bit G component for the odd \( i \) coordinate in the word in bytes 6..7. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a 2×1 compressed texel block.

- **VK_FORMAT_G16_B16R16_2PLANE_420_UNORM** specifies an unsigned normalized multi-planar format that has a 16-bit G component in each 16-bit word of plane 0, a 16-bit B component in each 16-bit word of plane 1, and a 16-bit R component in each 16-bit word of plane 2. The horizontal and vertical dimensions of the R and B planes are halved relative to the image dimensions, and each R and B component is shared with the G components for which \( i_G \times 0.5 = i_B = i_R \) and \( i_G \times 0.5 = j_B = j_R \). The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane. This format only supports images with a width and height that is a multiple of two.

- **VK_FORMAT_G16_B16R16_2PLANE_420_UNORM** specifies an unsigned normalized multi-planar format that has a 16-bit G component in each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 16-bit B component in the word in bytes 0..1, and a 16-bit R component in the word in bytes 2..3. The horizontal and vertical dimensions of the BR plane are halved relative to the image dimensions, and each R and B value is shared with the G components for
which $i_G \times 0.5 = i_R$ and $i_G \times 0.5 = j_B = j_R$. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane. This format only supports images with a width and height that is a multiple of two.

- **VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM** specifies an unsigned normalized multi-planar format that has a 16-bit G component in each 16-bit word of plane 0, a 16-bit B component in each 16-bit word of plane 1, and a 16-bit R component in each 16-bit word of plane 2. The horizontal dimension of the R and B plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which $i_G \times 0.5 = i_R$. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane. This format only supports images with a width that is a multiple of two.

- **VK_FORMAT_G16_B16R16_2PLANE_422_UNORM** specifies an unsigned normalized multi-planar format that has a 16-bit G component in each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 16-bit B component in the word in bytes 0..1, and a 16-bit R component in the word in bytes 2..3. The horizontal dimension of the BR plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which $i_G \times 0.5 = i_R$. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane. This format only supports images with a width that is a multiple of two.

- **VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM** specifies an unsigned normalized multi-planar format that has a 16-bit G component in each 16-bit word of plane 0, a 16-bit B component in each 16-bit word of plane 1, and a 16-bit R component in each 16-bit word of plane 2. Each plane has the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane.

- **VK_FORMAT_G16_B16R16_2PLANE_444_UNORM** specifies an unsigned normalized multi-planar format that has a 16-bit G component in each 16-bit word of plane 0, a 16-bit B component in each 16-bit word of plane 1, and a 16-bit R component in each 16-bit word of plane 2. Each plane has the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane.

- **VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16** specifies an unsigned normalized multi-planar format that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, and an 8-bit R component in byte 1 and a 10-bit R component in byte 2. Both planes have the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane.

- **VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16** specifies an unsigned normalized multi-planar format that has a 12-bit G component in the top 10 bits of each 16-bit word of plane 0, and a 16-bit R component in byte 1 and a 10-bit R component in byte 2. Both planes have the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane.
planar format that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 12-bit B component in the top 12 bits of the word in bytes 0..1, and a 12-bit R component in the top 12 bits of the word in bytes 2..3, the bottom 4 bits of each word unused. Both planes have the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane.

- `VK_FORMAT_G16_B16R16_2PLANE_444_UNORM` specifies an unsigned normalized multi-planar format that has a 16-bit G component in each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 16-bit B component in the word in bytes 0..1, and a 16-bit R component in the word in bytes 2..3. Both planes have the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane.

### 34.1.1. Compatible formats of planes of multi-planar formats

Individual planes of multi-planar formats are size-compatible with single-plane color formats if they occupy the same number of bits per texel block, and are compatible with those formats if they have the same block extent.

In the following table, individual planes of a multi-planar format are compatible with the format listed against the relevant plane index for that multi-planar format, and any format compatible with the listed single-plane format according to Format Compatibility Classes. These planes are also size-compatible with any format that is size-compatible with the listed single-plane format.

<table>
<thead>
<tr>
<th>Plane</th>
<th>Compatible format for plane</th>
<th>Width relative to the width ( w ) of the plane with the largest dimensions</th>
<th>Height relative to the height ( h ) of the plane with the largest dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td><code>VK_FORMAT_R8_UNORM</code></td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>1</td>
<td><code>VK_FORMAT_R8_UNORM</code></td>
<td>( w/2 )</td>
<td>( h/2 )</td>
</tr>
<tr>
<td>2</td>
<td><code>VK_FORMAT_R8_UNORM</code></td>
<td>( w/2 )</td>
<td>( h/2 )</td>
</tr>
<tr>
<td><code>VK_FORMAT_G8_B8R8_2PLANE_420_UNORM</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td><code>VK_FORMAT_R8_G8B8_UNORM</code></td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>1</td>
<td><code>VK_FORMAT_R8G8B8_UNORM</code></td>
<td>( w/2 )</td>
<td>( h/2 )</td>
</tr>
<tr>
<td><code>VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td><code>VK_FORMAT_R8_UNORM</code></td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>1</td>
<td><code>VK_FORMAT_R8_UNORM</code></td>
<td>( w/2 )</td>
<td>( h )</td>
</tr>
<tr>
<td>2</td>
<td><code>VK_FORMAT_R8_UNORM</code></td>
<td>( w/2 )</td>
<td>( h )</td>
</tr>
<tr>
<td><code>VK_FORMAT_G8_B8R8_2PLANE_422_UNORM</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plane</td>
<td>Compatible format for plane</td>
<td>Width relative to the width $w$ of the plane with the largest dimensions</td>
<td>Height relative to the height $h$ of the plane with the largest dimensions</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------------</td>
<td>-------------------------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R8_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R8G8_UNORM</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R8_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R8_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R8_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R10X6G10X6_UNORM_2PACK16</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G10X6_B10X6R10X6_3PLANE_422_UNORM_3PACK16</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R10X6G10X6_UNORM_2PACK16</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R12X4G12X4_UNORM_2PACK16</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plane</td>
<td>Compatible format for plane</td>
<td>Width relative to the width $w$ of the plane with the largest dimensions</td>
<td>Height relative to the height $h$ of the plane with the largest dimensions</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------------</td>
<td>-------------------------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R12X4G12X4_UNORM_2PACK16</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G16_B16R16_2PLANE_420_UNORM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R16G16_UNORM</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G16_B16R16_2PLANE_422_UNORM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R16G16_UNORM</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G8_B8R8_2PLANE_444_UNORM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R8_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R8G8_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 34.1.2. Packed Formats

For the purposes of address alignment when accessing buffer memory containing vertex attribute or texel data, the following formats are considered *packed* - components of the texels or attributes are stored in bitfields packed into one or more 8-, 16-, or 32-bit fundamental data type.

- **Packed into 8-bit data types:**
  - `VK_FORMAT_R4G4_UNORM_PACK8`

- **Packed into 16-bit data types:**
  - `VK_FORMAT_R4G4B4A4_UNORM_PACK16`
  - `VK_FORMAT_B4G4R4A4_UNORM_PACK16`
  - `VK_FORMAT_R5G5B5B5A1_UNORM_PACK16`
  - `VK_FORMAT_B5G5B5B5A1_UNORM_PACK16`

<table>
<thead>
<tr>
<th>Plane</th>
<th>Compatible format for plane</th>
<th>Width relative to the width ( w ) of the plane with the largest dimensions</th>
<th>Height relative to the height ( h ) of the plane with the largest dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>VK_FORMAT_R10X6_UNORM_PACK16</code></td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>1</td>
<td><code>VK_FORMAT_R10X6G10X6_UNORM_2PACK16</code></td>
<td>( w )</td>
<td>( h )</td>
</tr>
</tbody>
</table>

**`VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16`**

<table>
<thead>
<tr>
<th>Plane</th>
<th>Compatible format for plane</th>
<th>Width relative to the width ( w ) of the plane with the largest dimensions</th>
<th>Height relative to the height ( h ) of the plane with the largest dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>VK_FORMAT_R12X4_UNORM_PACK16</code></td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>1</td>
<td><code>VK_FORMAT_R12X4G12X4_UNORM_2PACK16</code></td>
<td>( w )</td>
<td>( h )</td>
</tr>
</tbody>
</table>

**`VK_FORMAT_G16_B16R16_2PLANE_444_UNORM`**

<table>
<thead>
<tr>
<th>Plane</th>
<th>Compatible format for plane</th>
<th>Width relative to the width ( w ) of the plane with the largest dimensions</th>
<th>Height relative to the height ( h ) of the plane with the largest dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>VK_FORMAT_R16_UNORM</code></td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>1</td>
<td><code>VK_FORMAT_R16G16_UNORM</code></td>
<td>( w )</td>
<td>( h )</td>
</tr>
</tbody>
</table>
VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16
VK_FORMAT_R12X4_UNORM_PACK16
VK_FORMAT_R12X4G12X4_UNORM_2PACK16
VK_FORMAT_R12X4G12X4B12X4A12X4_UNORM_4PACK16
VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16
VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16
VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16
VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16
VK_FORMAT_G12X4_B12X4R12X4_3PLANE_422_UNORM_3PACK16
VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16
VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16
VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16
VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16
VK_FORMAT_A4R4G4B4_UNORM_PACK16
VK_FORMAT_A4B4G4R4_UNORM_PACK16

• Packed into 32-bit data types:
  VK_FORMAT_A8B8G8R8_UNORM_PACK32
  VK_FORMAT_A8B8G8R8_SNORM_PACK32
  VK_FORMAT_A8B8G8R8_USCALED_PACK32
  VK_FORMAT_A8B8G8R8_SSCALED_PACK32
  VK_FORMAT_A8B8G8R8_UINT_PACK32
  VK_FORMAT_A8B8G8R8_SINT_PACK32
  VK_FORMAT_A8B8G8R8_SRGB_PACK32
  VK_FORMAT_A2R10G10B10_UNORM_PACK32
  VK_FORMAT_A2R10G10B10_SNORM_PACK32
  VK_FORMAT_A2R10G10B10_USCALED_PACK32
  VK_FORMAT_A2R10G10B10_SSCALED_PACK32
  VK_FORMAT_A2R10G10B10_UINT_PACK32
  VK_FORMAT_A2R10G10B10_SINT_PACK32
  VK_FORMAT_A2B10G10R10_UNORM_PACK32
  VK_FORMAT_A2B10G10R10_SNORM_PACK32
  VK_FORMAT_A2B10G10R10_USCALED_PACK32
  VK_FORMAT_A2B10G10R10_SSCALED_PACK32
  VK_FORMAT_A2B10G10R10_UINT_PACK32
  VK_FORMAT_A2B10G10R10_SINT_PACK32
34.1.3. Identification of Formats

A “format” is represented by a single enum value. The name of a format is usually built up by using the following pattern:

\[
\text{VK_FORMAT}_{\text{component-format|compression-scheme}}_{\text{numeric-format}}
\]

The component-format indicates either the size of the R, G, B, and A components (if they are present) in the case of a color format, or the size of the depth (D) and stencil (S) components (if they are present) in the case of a depth/stencil format (see below). An X indicates a component that is unused, but may be present for padding.
### Table 35. Interpretation of Numeric Format

<table>
<thead>
<tr>
<th>Numeric format</th>
<th>SPIR-V Sampled Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNORM</td>
<td>OpTypeFloat</td>
<td>The components are unsigned normalized values in the range [0,1]</td>
</tr>
<tr>
<td>SNORM</td>
<td>OpTypeFloat</td>
<td>The components are signed normalized values in the range [-1,1]</td>
</tr>
<tr>
<td>USCALED</td>
<td>OpTypeFloat</td>
<td>The components are unsigned integer values that get converted to floating-point in the range [0,2^{n-1}]</td>
</tr>
<tr>
<td>SSCALED</td>
<td>OpTypeFloat</td>
<td>The components are signed integer values that get converted to floating-point in the range [-2^{n-1},2^{n-1}-1]</td>
</tr>
<tr>
<td>UINT</td>
<td>OpTypeInt</td>
<td>The components are unsigned integer values in the range [0,2^n-1]</td>
</tr>
<tr>
<td>SINT</td>
<td>OpTypeInt</td>
<td>The components are signed integer values in the range [-2^{n-1},2^{n-1}-1]</td>
</tr>
<tr>
<td>UFLOAT</td>
<td>OpTypeFloat</td>
<td>The components are unsigned floating-point numbers (used by packed, shared exponent, and some compressed formats)</td>
</tr>
<tr>
<td>SFLOAT</td>
<td>OpTypeFloat</td>
<td>The components are signed floating-point numbers</td>
</tr>
<tr>
<td>SRGB</td>
<td>OpTypeFloat</td>
<td>The R, G, and B components are unsigned normalized values that represent values using sRGB nonlinear encoding, while the A component (if one exists) is a regular unsigned normalized value</td>
</tr>
</tbody>
</table>

n is the number of bits in the component.

The suffix \_PACKnn indicates that the format is packed into an underlying type with nn bits. The suffix \_mPACKnn is a short-hand that indicates that the format has m groups of components (which may or may not be stored in separate planes) that are each packed into an underlying type with nn bits.

The suffix \_BLOCK indicates that the format is a block-compressed format, with the representation of multiple pixels encoded interdependently within a region.

### Table 36. Interpretation of Compression Scheme

<table>
<thead>
<tr>
<th>Compression scheme</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC</td>
<td>Block Compression. See Block-Compressed Image Formats.</td>
</tr>
<tr>
<td>ETC2</td>
<td>Ericsson Texture Compression. See ETC Compressed Image Formats.</td>
</tr>
<tr>
<td>EAC</td>
<td>ETC2 Alpha Compression. See ETC Compressed Image Formats.</td>
</tr>
<tr>
<td>ASTC</td>
<td>Adaptive Scalable Texture Compression (LDR Profile). See ASTC Compressed Image Formats.</td>
</tr>
</tbody>
</table>

For multi-planar images, the components in separate planes are separated by underscores, and the

1206
number of planes is indicated by the addition of a _2PLANE or _3PLANE suffix. Similarly, the separate aspects of depth-stencil formats are separated by underscores, although these are not considered separate planes. Formats are suffixed by _422 to indicate that planes other than the first are reduced in size by a factor of two horizontally or that the R and B values appear at half the horizontal frequency of the G values, _420 to indicate that planes other than the first are reduced in size by a factor of two both horizontally and vertically, and _444 for consistency to indicate that all three planes of a three-planar image are the same size.

Note
No common format has a single plane containing both R and B components but does not store these components at reduced horizontal resolution.

34.1.4. Representation and Texel Block Size

Color formats must be represented in memory in exactly the form indicated by the format’s name. This means that promoting one format to another with more bits per component and/or additional components must not occur for color formats. Depth/stencil formats have more relaxed requirements as discussed below.

Each format has a texel block size, the number of bytes used to store one texel block (a single addressable element of an uncompressed image, or a single compressed block of a compressed image). The texel block size for each format is shown in the Compatible formats table.

The representation of non-packed formats is that the first component specified in the name of the format is in the lowest memory addresses and the last component specified is in the highest memory addresses. See Byte mappings for non-packed/compressed color formats. The in-memory ordering of bytes within a component is determined by the host endianness.

Table 37. Byte mappings for non-packed/compressed color formats

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_R8_*</td>
</tr>
<tr>
<td>R G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_R8G8_*</td>
</tr>
<tr>
<td>R G B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_R8G8B8_*</td>
</tr>
<tr>
<td>B G R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_B8G8R8_*</td>
</tr>
<tr>
<td>R G B A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_R8G8B8A8_*</td>
</tr>
<tr>
<td>B G R A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_B8G8R8A8_*</td>
</tr>
<tr>
<td>G₀ B G₁ R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_B8G8R8G8_422_UNORM</td>
</tr>
<tr>
<td>B G₀ R G₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_B8G8R8G8_422_UNORM</td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_R16_*</td>
</tr>
<tr>
<td>R G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_R16G16_*</td>
</tr>
<tr>
<td>R G B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_R16G16B16_*</td>
</tr>
<tr>
<td>R G B A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_R16G16B16A16_*</td>
</tr>
</tbody>
</table>
Packed formats store multiple components within one underlying type. The bit representation is that the first component specified in the name of the format is in the most-significant bits and the last component specified is in the least-significant bits of the underlying type. The in-memory ordering of bytes comprising the underlying type is determined by the host endianness.

Table 38. Bit mappings for packed 8-bit formats

<table>
<thead>
<tr>
<th>Bit</th>
<th>R</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 39. Bit mappings for packed 16-bit formats

<table>
<thead>
<tr>
<th>Bit</th>
<th>R</th>
<th>G</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

VK_FORMAT_R32G32B32A32_4 Pack 16

VK_FORMAT_R32G32B32A32 as VK_FORMAT_R64G64B64A64 but with A in bytes 24-31

VK_FORMAT_R64G64B64A64 as VK_FORMAT_R64G64B64A64 but with B in bytes 16-23
**Table 40.** Bit mappings for packed 32-bit formats

<table>
<thead>
<tr>
<th>Bit</th>
<th>A</th>
<th>B</th>
<th>G</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>R</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**VK_FORMAT_R5G6B5_UNORM_PACK16**

<table>
<thead>
<tr>
<th>R</th>
<th>G</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

**VK_FORMAT_B5G6R5_UNORM_PACK16**

<table>
<thead>
<tr>
<th>B</th>
<th>G</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

**VK_FORMAT_R5G5B5A1_UNORM_PACK16**

<table>
<thead>
<tr>
<th>R</th>
<th>G</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**VK_FORMAT_B5G5R5A1_UNORM_PACK16**

<table>
<thead>
<tr>
<th>B</th>
<th>G</th>
<th>R</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**VK_FORMAT_A1R5G5B5_UNORM_PACK16**

<table>
<thead>
<tr>
<th>A</th>
<th>R</th>
<th>G</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

**VK_FORMAT_R10X6_UNORM_PACK16**

<table>
<thead>
<tr>
<th>R</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
</tbody>
</table>

**VK_FORMAT_R12X4_UNORM_PACK16**

<table>
<thead>
<tr>
<th>R</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

**VK_FORMAT_A8B8G8R8_*_PACK32**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>G</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

**VK_FORMAT_A2R10G10B10_*_PACK32**

<table>
<thead>
<tr>
<th>A</th>
<th>R</th>
<th>G</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>9</td>
<td>8</td>
</tr>
</tbody>
</table>

**VK_FORMAT_A2B10G10R10_*_PACK32**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>G</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>9</td>
<td>8</td>
</tr>
</tbody>
</table>

**VK_FORMAT_B10G11R11_UFLOAT_PACK32**

<table>
<thead>
<tr>
<th>B</th>
<th>G</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>
34.1.5. Depth/Stencil Formats

Depth/stencil formats are considered opaque and need not be stored in the exact number of bits per texel or component ordering indicated by the format enum. However, implementations must not substitute a different depth or stencil precision than is described in the format (e.g. D16 must not be implemented as D24 or D32).

34.1.6. Format Compatibility Classes

Uncompressed color formats are compatible with each other if they occupy the same number of bits per texel block. Compressed color formats are compatible with each other if the only difference between them is the numerical type of the uncompressed pixels (e.g. signed vs. unsigned, or SRGB vs. UNORM encoding). Each depth/stencil format is only compatible with itself. In the following table, all the formats in the same row are compatible. Each format has a defined texel block extent specifying how many texels each texel block represents in each dimension.

Table 41. Compatible Formats

<table>
<thead>
<tr>
<th>Class, Texel Block Size, Texel Block Extent, # Texels/Block</th>
<th>Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit Block size 1 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_R4G4_UNORM_PACK8, VK_FORMAT_R8_UNORM, VK_FORMAT_R8_SNORM, VK_FORMAT_R8_USCALED, VK_FORMAT_R8_SSCALED, VK_FORMAT_R8_UINT, VK_FORMAT_R8_SINT, VK_FORMAT_R8_SRGB</td>
</tr>
<tr>
<td>Class, Texel Block Size, Texel Block Extent, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>16-bit Block size 2 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16, VK_FORMAT_R12X4_UNORM_PACK16, VK_FORMAT_A4R4G4B4_UNORM_PACK16, VK_FORMAT_A4B4G4R4_UNORM_PACK16, VK_FORMAT_R4G4B4A4_UNORM_PACK16, VK_FORMAT_B4G4R4A4_UNORM_PACK16, VK_FORMAT_R5G6B5_UNORM_PACK16, VK_FORMAT_B5G6R5_UNORM_PACK16, VK_FORMAT_R5G5B5A1_UNORM_PACK16, VK_FORMAT_B5G5R5A1_UNORM_PACK16, VK_FORMAT_A1R5G5B5_UNORM_PACK16, VK_FORMAT_R8G8_UNORM, VK_FORMAT_R8G8_SNORM, VK_FORMAT_R8G8_USCALED, VK_FORMAT_R8G8_SSCALED, VK_FORMAT_R8G8_UINT, VK_FORMAT_R8G8_SINT, VK_FORMAT_R8G8_SRGB, VK_FORMAT_R16_UNORM, VK_FORMAT_R16_SNORM, VK_FORMAT_R16_USCALED, VK_FORMAT_R16_SSCALED, VK_FORMAT_R16_UINT, VK_FORMAT_R16_SINT, VK_FORMAT_R16_SFLOAT</td>
</tr>
<tr>
<td>24-bit Block size 3 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_R8G8B8_UNORM, VK_FORMAT_R8G8B8_SNORM, VK_FORMAT_R8G8B8_USCALED, VK_FORMAT_R8G8B8_SSCALED, VK_FORMAT_R8G8B8_UINT, VK_FORMAT_R8G8B8_SINT, VK_FORMAT_R8G8B8_SRGB, VK_FORMAT_B8G8R8_UNORM, VK_FORMAT_B8G8R8_SNORM, VK_FORMAT_B8G8R8_USCALED, VK_FORMAT_B8G8R8_SSCALED, VK_FORMAT_B8G8R8_UINT, VK_FORMAT_B8G8R8_SINT, VK_FORMAT_B8G8R8_SRGB</td>
</tr>
<tr>
<td>Class, Texel Block Size, Texel Block Extent, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>-----------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>32-bit&lt;br&gt;Block size 4 byte&lt;br&gt;1x1x1 block extent&lt;br&gt;1 texel/block</td>
<td>VK_FORMAT_R10X6G10X6_UNORM_2PACK16,&lt;br&gt;VK_FORMAT_R12X4G12X4_UNORM_2PACK16,&lt;br&gt;VK_FORMAT_R8G8B8A8_UNORM,&lt;br&gt;VK_FORMAT_R8G8B8A8_SNORM,&lt;br&gt;VK_FORMAT_R8G8B8A8_USCALED,&lt;br&gt;VK_FORMAT_R8G8B8A8_SSCALED,&lt;br&gt;VK_FORMAT_R8G8B8A8_UINT,&lt;br&gt;VK_FORMAT_R8G8B8A8_SINT,&lt;br&gt;VK_FORMAT_R8G8B8A8_SRGB,&lt;br&gt;VK_FORMAT_B8G8R8A8_UNORM,&lt;br&gt;VK_FORMAT_B8G8R8A8_SNORM,&lt;br&gt;VK_FORMAT_B8G8R8A8_USCALED,&lt;br&gt;VK_FORMAT_B8G8R8A8_SSCALED,&lt;br&gt;VK_FORMAT_B8G8R8A8_UINT,&lt;br&gt;VK_FORMAT_B8G8R8A8_SINT,&lt;br&gt;VK_FORMAT_B8G8R8A8_SRGB,&lt;br&gt;VK_FORMAT_A8B8G8R8_UNORM_PACK32,&lt;br&gt;VK_FORMAT_A8B8G8R8_SNORM_PACK32,&lt;br&gt;VK_FORMAT_A8B8G8R8_USCALED_PACK32,&lt;br&gt;VK_FORMAT_A8B8G8R8_SSCALED_PACK32,&lt;br&gt;VK_FORMAT_A8B8G8R8_UINT_PACK32,&lt;br&gt;VK_FORMAT_A8B8G8R8_SINT_PACK32,&lt;br&gt;VK_FORMAT_A8B8G8R8_SRGB_PACK32,&lt;br&gt;VK_FORMAT_A2R10G10B10_UNORM_PACK32,&lt;br&gt;VK_FORMAT_A2R10G10B10_SNORM_PACK32,&lt;br&gt;VK_FORMAT_A2R10G10B10_USCALED_PACK32,&lt;br&gt;VK_FORMAT_A2R10G10B10_SSCALED_PACK32,&lt;br&gt;VK_FORMAT_A2R10G10B10_UINT_PACK32,&lt;br&gt;VK_FORMAT_A2R10G10B10_SINT_PACK32,&lt;br&gt;VK_FORMAT_A2R10G10B10_SRGB_PACK32,&lt;br&gt;VK_FORMAT_R16G16_UNORM,&lt;br&gt;VK_FORMAT_R16G16_SNORM,&lt;br&gt;VK_FORMAT_R16G16_USCALED,&lt;br&gt;VK_FORMAT_R16G16_SSCALED,&lt;br&gt;VK_FORMAT_R16G16_UINT,&lt;br&gt;VK_FORMAT_R16G16_SINT,&lt;br&gt;VK_FORMAT_R16G16_SFLOAT,&lt;br&gt;VK_FORMAT_R32_UINT,&lt;br&gt;VK_FORMAT_R32_SINT,&lt;br&gt;VK_FORMAT_R32_SFLOAT,&lt;br&gt;VK_FORMAT_B10G11R10_UFLOAT_PACK32,&lt;br&gt;VK_FORMAT_E5B9G9R9_UFLOAT_PACK32</td>
</tr>
<tr>
<td>Class, Texel Block Size, Texel Block Extent, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>-------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>48-bit Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_R16G16B16_UNORM, VK_FORMAT_R16G16B16_SNORM, VK_FORMAT_R16G16B16_USCALED, VK_FORMAT_R16G16B16_SSCALED, VK_FORMAT_R16G16B16_UINT, VK_FORMAT_R16G16B16_SINT, VK_FORMAT_R16G16B16_SFLOAT</td>
</tr>
<tr>
<td>96-bit Block size 12 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_R32G32B32_UINT, VK_FORMAT_R32G32B32_SINT, VK_FORMAT_R32G32B32_SFLOAT</td>
</tr>
<tr>
<td>128-bit Block size 16 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_R32G32B32A32_UINT, VK_FORMAT_R32G32B32A32_SINT, VK_FORMAT_R32G32B32A32_SFLOAT, VK_FORMAT_R64G64_UINT, VK_FORMAT_R64G64_SINT, VK_FORMAT_R64G64_SFLOAT</td>
</tr>
<tr>
<td>192-bit Block size 24 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_R64G64B64_UINT, VK_FORMAT_R64G64B64_SINT, VK_FORMAT_R64G64B64_SFLOAT</td>
</tr>
<tr>
<td>256-bit Block size 32 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_R64G64B64A64_UINT, VK_FORMAT_R64G64B64A64_SINT, VK_FORMAT_R64G64B64A64_SFLOAT</td>
</tr>
<tr>
<td>Class, Texel Block Size, Texel Block Extent, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>-------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td><strong>D16</strong>&lt;br&gt;Block size 2 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_D16_UNORM</td>
</tr>
<tr>
<td><strong>D24</strong>&lt;br&gt;Block size 4 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_X8_D24_UNORM_PACK32</td>
</tr>
<tr>
<td><strong>D32</strong>&lt;br&gt;Block size 4 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_D32_SFLOAT</td>
</tr>
<tr>
<td><strong>S8</strong>&lt;br&gt;Block size 1 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_S8_UINT</td>
</tr>
<tr>
<td><strong>D16S8</strong>&lt;br&gt;Block size 3 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_D16_UNORM_S8_UINT</td>
</tr>
<tr>
<td><strong>D24S8</strong>&lt;br&gt;Block size 4 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_D24_UNORM_S8_UINT</td>
</tr>
<tr>
<td><strong>D32S8</strong>&lt;br&gt;Block size 5 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_D32_SFLOAT_S8_UINT</td>
</tr>
<tr>
<td><strong>BC1_RGB</strong>&lt;br&gt;Block size 8 byte 4x4x1 block extent 16 texel/block</td>
<td>VK_FORMAT_BC1_RGB_UNORM_BLOCK, VK_FORMAT_BC1_RGB_SRGB_BLOCK</td>
</tr>
<tr>
<td><strong>BC1_RGBA</strong>&lt;br&gt;Block size 8 byte 4x4x1 block extent 16 texel/block</td>
<td>VK_FORMAT_BC1_RGBA_UNORM_BLOCK, VK_FORMAT_BC1_RGBA_SRGB_BLOCK</td>
</tr>
<tr>
<td><strong>BC2</strong>&lt;br&gt;Block size 16 byte 4x4x1 block extent 16 texel/block</td>
<td>VK_FORMAT_BC2_UNORM_BLOCK, VK_FORMAT_BC2_SRGB_BLOCK</td>
</tr>
<tr>
<td>Class, Texel Block Size, Texel Block Extent, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>-----------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td><strong>BC3</strong>&lt;br&gt;Block size 16 byte 4x4x1 block extent 16 texel/block</td>
<td>VK_FORMAT_BC3_UNORM_BLOCK, VK_FORMAT_BC3_SRGB_BLOCK</td>
</tr>
<tr>
<td><strong>BC4</strong>&lt;br&gt;Block size 8 byte 4x4x1 block extent 16 texel/block</td>
<td>VK_FORMAT_BC4_UNORM_BLOCK, VK_FORMAT_BC4_SNORM_BLOCK</td>
</tr>
<tr>
<td><strong>BC5</strong>&lt;br&gt;Block size 16 byte 4x4x1 block extent 16 texel/block</td>
<td>VK_FORMAT_BC5_UNORM_BLOCK, VK_FORMAT_BC5_SNORM_BLOCK</td>
</tr>
<tr>
<td><strong>BC6H</strong>&lt;br&gt;Block size 16 byte 4x4x1 block extent 16 texel/block</td>
<td>VK_FORMAT_BC6H_UFLOAT_BLOCK, VK_FORMAT_BC6H_SFLOAT_BLOCK</td>
</tr>
<tr>
<td><strong>BC7</strong>&lt;br&gt;Block size 16 byte 4x4x1 block extent 16 texel/block</td>
<td>VK_FORMAT_BC7_UNORM_BLOCK, VK_FORMAT_BC7_SRGB_BLOCK</td>
</tr>
<tr>
<td><strong>ETC2_RGB</strong>&lt;br&gt;Block size 8 byte 4x4x1 block extent 16 texel/block</td>
<td>VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK</td>
</tr>
<tr>
<td><strong>ETC2_RGBA</strong>&lt;br&gt;Block size 8 byte 4x4x1 block extent 16 texel/block</td>
<td>VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK</td>
</tr>
<tr>
<td><strong>ETC2_EAC_RGBA</strong>&lt;br&gt;Block size 16 byte 4x4x1 block extent 16 texel/block</td>
<td>VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK</td>
</tr>
<tr>
<td><strong>EAC_R</strong>&lt;br&gt;Block size 8 byte 4x4x1 block extent 16 texel/block</td>
<td>VK_FORMAT_EAC_R11_UNORM_BLOCK, VK_FORMAT_EAC_R11_SNORM_BLOCK</td>
</tr>
<tr>
<td><strong>EAC_RG</strong>&lt;br&gt;Block size 16 byte 4x4x1 block extent 16 texel/block</td>
<td>VK_FORMAT_EAC_R11G11_UNORM_BLOCK, VK_FORMAT_EAC_R11G11_SNORM_BLOCK</td>
</tr>
<tr>
<td>Class, Texel Block Size, Texel Block Extent, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>-------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td><strong>ASTC_4x4</strong>&lt;br&gt;Block size 16 byte&lt;br&gt;4x4x1 block extent&lt;br&gt;16 texel/block</td>
<td>VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK, VK_FORMAT_ASTC_4x4_UNORM_BLOCK, VK_FORMAT_ASTC_4x4_SRGB_BLOCK</td>
</tr>
<tr>
<td><strong>ASTC_5x4</strong>&lt;br&gt;Block size 16 byte&lt;br&gt;5x4x1 block extent&lt;br&gt;20 texel/block</td>
<td>VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK, VK_FORMAT_ASTC_5x4_UNORM_BLOCK, VK_FORMAT_ASTC_5x4_SRGB_BLOCK</td>
</tr>
<tr>
<td><strong>ASTC_5x5</strong>&lt;br&gt;Block size 16 byte&lt;br&gt;5x5x1 block extent&lt;br&gt;25 texel/block</td>
<td>VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK, VK_FORMAT_ASTC_5x5_UNORM_BLOCK, VK_FORMAT_ASTC_5x5_SRGB_BLOCK</td>
</tr>
<tr>
<td><strong>ASTC_6x5</strong>&lt;br&gt;Block size 16 byte&lt;br&gt;6x5x1 block extent&lt;br&gt;30 texel/block</td>
<td>VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK, VK_FORMAT_ASTC_6x5_UNORM_BLOCK, VK_FORMAT_ASTC_6x5_SRGB_BLOCK</td>
</tr>
<tr>
<td><strong>ASTC_6x6</strong>&lt;br&gt;Block size 16 byte&lt;br&gt;6x6x1 block extent&lt;br&gt;36 texel/block</td>
<td>VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK, VK_FORMAT_ASTC_6x6_UNORM_BLOCK, VK_FORMAT_ASTC_6x6_SRGB_BLOCK</td>
</tr>
<tr>
<td><strong>ASTC_8x5</strong>&lt;br&gt;Block size 16 byte&lt;br&gt;8x5x1 block extent&lt;br&gt;40 texel/block</td>
<td>VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK, VK_FORMAT_ASTC_8x5_UNORM_BLOCK, VK_FORMAT_ASTC_8x5_SRGB_BLOCK</td>
</tr>
<tr>
<td><strong>ASTC_8x6</strong>&lt;br&gt;Block size 16 byte&lt;br&gt;8x6x1 block extent&lt;br&gt;48 texel/block</td>
<td>VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK, VK_FORMAT_ASTC_8x6_UNORM_BLOCK, VK_FORMAT_ASTC_8x6_SRGB_BLOCK</td>
</tr>
<tr>
<td><strong>ASTC_8x8</strong>&lt;br&gt;Block size 16 byte&lt;br&gt;8x8x1 block extent&lt;br&gt;64 texel/block</td>
<td>VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK, VK_FORMAT_ASTC_8x8_UNORM_BLOCK, VK_FORMAT_ASTC_8x8_SRGB_BLOCK</td>
</tr>
<tr>
<td><strong>ASTC_10x5</strong>&lt;br&gt;Block size 16 byte&lt;br&gt;10x5x1 block extent&lt;br&gt;50 texel/block</td>
<td>VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK, VK_FORMAT_ASTC_10x5_UNORM_BLOCK, VK_FORMAT_ASTC_10x5_SRGB_BLOCK</td>
</tr>
<tr>
<td><strong>ASTC_10x6</strong>&lt;br&gt;Block size 16 byte&lt;br&gt;10x6x1 block extent&lt;br&gt;60 texel/block</td>
<td>VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK, VK_FORMAT_ASTC_10x6_UNORM_BLOCK, VK_FORMAT_ASTC_10x6_SRGB_BLOCK</td>
</tr>
<tr>
<td>Class, Texel Block Size, Texel Block Extent, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>ASTC_10x8 Block size 16 byte 10x8x1 block extent 80 texel/block</td>
<td>VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK, VK_FORMAT_ASTC_10x8_UNORM_BLOCK, VK_FORMAT_ASTC_10x8_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_10x10 Block size 16 byte 10x10x1 block extent 100 texel/block</td>
<td>VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK, VK_FORMAT_ASTC_10x10_UNORM_BLOCK, VK_FORMAT_ASTC_10x10_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_12x10 Block size 16 byte 12x10x1 block extent 120 texel/block</td>
<td>VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK, VK_FORMAT_ASTC_12x10_UNORM_BLOCK, VK_FORMAT_ASTC_12x10_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_12x12 Block size 16 byte 12x12x1 block extent 144 texel/block</td>
<td>VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK, VK_FORMAT_ASTC_12x12_UNORM_BLOCK, VK_FORMAT_ASTC_12x12_SRGB_BLOCK</td>
</tr>
<tr>
<td>32-bit G8B8G8R8 Block size 4 byte 2x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G8B8G8R8_422_UNORM</td>
</tr>
<tr>
<td>32-bit B8G8R8G8 Block size 4 byte 2x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_B8G8R8G8_422_UNORM</td>
</tr>
<tr>
<td>8-bit 3-plane 420 Block size 3 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM</td>
</tr>
<tr>
<td>8-bit 2-plane 420 Block size 3 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G8_B8R8_2PLANE_420_UNORM</td>
</tr>
<tr>
<td>8-bit 3-plane 422 Block size 3 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM</td>
</tr>
<tr>
<td>8-bit 2-plane 422 Block size 3 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G8_B8R8_2PLANE_422_UNORM</td>
</tr>
<tr>
<td>Class, Texel Block Size, Texel Block Extent, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>8-bit 3-plane 444 Block size 3 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM</td>
</tr>
<tr>
<td>64-bit R10G10B10A10 Block size 8 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16</td>
</tr>
<tr>
<td>64-bit G10B10G10R10 Block size 8 byte 2x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16</td>
</tr>
<tr>
<td>64-bit B10G10R10G10 Block size 8 byte 2x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16</td>
</tr>
<tr>
<td>10-bit 3-plane 420 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16</td>
</tr>
<tr>
<td>10-bit 2-plane 420 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16</td>
</tr>
<tr>
<td>10-bit 3-plane 422 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16</td>
</tr>
<tr>
<td>10-bit 2-plane 422 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16</td>
</tr>
<tr>
<td>10-bit 3-plane 444 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16</td>
</tr>
<tr>
<td>64-bit R12G12B12A12 Block size 8 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_R12X4G12X4B12X4A12X4_UNORM_4PACK16</td>
</tr>
<tr>
<td>Class, Texel Block Size, Texel Block Extent, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>-------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>64-bit G12B12G12R12 Block size 8 byte 2x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16</td>
</tr>
<tr>
<td>64-bit B12G12R12G12 Block size 8 byte 2x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16</td>
</tr>
<tr>
<td>12-bit 3-plane 420 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16</td>
</tr>
<tr>
<td>12-bit 2-plane 420 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16</td>
</tr>
<tr>
<td>12-bit 3-plane 422 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16</td>
</tr>
<tr>
<td>12-bit 2-plane 422 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16</td>
</tr>
<tr>
<td>12-bit 3-plane 444 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16</td>
</tr>
<tr>
<td>64-bit G16B16G16R16 Block size 8 byte 2x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G16B16G16R16_422_UNORM</td>
</tr>
<tr>
<td>64-bit B16G16R16G16 Block size 8 byte 2x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_B16G16R16G16_422_UNORM</td>
</tr>
<tr>
<td>16-bit 3-plane 420 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM</td>
</tr>
<tr>
<td>Class, Texel Block Size, Texel Block Extent, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>16-bit 2-plane 420 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G16_B16R16_2PLANE_420_UNORM</td>
</tr>
<tr>
<td>16-bit 3-plane 422 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM</td>
</tr>
<tr>
<td>16-bit 2-plane 422 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G16_B16R16_2PLANE_422_UNORM</td>
</tr>
<tr>
<td>16-bit 3-plane 444 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM</td>
</tr>
<tr>
<td>8-bit 2-plane 444 Block size 3 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G8_B8R8_2PLANE_444_UNORM</td>
</tr>
<tr>
<td>10-bit 2-plane 444 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16</td>
</tr>
<tr>
<td>12-bit 2-plane 444 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16</td>
</tr>
<tr>
<td>16-bit 2-plane 444 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G16_B16R16_2PLANE_444_UNORM</td>
</tr>
</tbody>
</table>

**Size Compatibility**

Color formats with the same texel block size are considered *size-compatible*. If two size-compatible formats have different block extents (i.e. for compressed formats), then an image with size $A \times B \times C$ in one format with a block extent of $a \times b \times c$ can be represented as an image with size $X \times Y \times Z$ in the other format with block extent $x \times y \times z$ at the ratio between the block extents for each format, where
\[ A/a = X/x \]
\[ B/b = Y/y \]
\[ C/c = Z/z \]

**Note**

For example, a 7x3 image in the `VK_FORMAT_ASTC_8x5_UNORM_BLOCK` format can be represented as a 1x1 `VK_FORMAT_R64G64_UINT` image.

Images created with the `VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT` flag can have size-compatible views created from them to enable access via different size-compatible formats. Image views created in this way will be sized to match the expectations of the block extents noted above.

Copy operations are able to copy between size-compatible formats in different resources to enable manipulation of data in different formats. The extent used in these copy operations always matches the source image, and is resized to the expectations of the block extents noted above for the destination image.

### 34.2. Format Properties

To query supported format features which are properties of the physical device, call:

```c
// Provided by VK_VERSION_1_0
void vkGetPhysicalDeviceFormatProperties(
    VkPhysicalDevice physicalDevice,
    VkFormat format,
    VkFormatProperties* pFormatProperties);
```

- `physicalDevice` is the physical device from which to query the format properties.
- `format` is the format whose properties are queried.
- `pFormatProperties` is a pointer to a `VkFormatProperties` structure in which physical device properties for `format` are returned.

**Valid Usage (Implicit)**

- VUID-vkGetPhysicalDeviceFormatProperties-physicalDevice-parameter `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceFormatProperties-format-parameter `format` must be a valid `VkFormat` value
- VUID-vkGetPhysicalDeviceFormatProperties-pFormatProperties-parameter `pFormatProperties` must be a valid pointer to a `VkFormatProperties` structure
The `VkFormatProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkFormatProperties {
    VkFormatFeatureFlags linearTilingFeatures;
    VkFormatFeatureFlags optimalTilingFeatures;
    VkFormatFeatureFlags bufferFeatures;
} VkFormatProperties;
```

- `linearTilingFeatures` is a bitmask of `VkFormatFeatureFlagBits` specifying features supported by images created with a `tiling` parameter of `VK_IMAGE_TILING_LINEAR`.
- `optimalTilingFeatures` is a bitmask of `VkFormatFeatureFlagBits` specifying features supported by images created with a `tiling` parameter of `VK_IMAGE_TILING_OPTIMAL`.
- `bufferFeatures` is a bitmask of `VkFormatFeatureFlagBits` specifying features supported by buffers.

**Note**

If no format feature flags are supported, the format itself is not supported, and images of that format cannot be created.

If `format` is a block-compressed format, then `bufferFeatures` **must** not support any features for the format.

If `format` is not a multi-plane format then `linearTilingFeatures` and `optimalTilingFeatures` **must** not contain `VK_FORMAT_FEATURE_DISJOINT_BIT`.

Bits which **can** be set in the `VkFormatProperties` features `linearTilingFeatures`, `optimalTilingFeatures`, and `bufferFeatures` are:

```c
// Provided by VK_VERSION_1_0
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,
    // Provided by VK_VERSION_1_1
    VK_FORMAT_FEATURE_TRANSFER_SRC_BIT = 0x00004000,
    // Provided by VK_VERSION_1_1
    // Provided by VK_VERSION_1_1
    VK_FORMAT_FEATURE_TRANSFER_SRC_BIT = 0x00004000,
} VkFormatFeatureFlagBits;
```
VK_FORMAT_FEATURE_TRANSFER_DST_BIT = 0x00008000,
// Provided by VK_VERSION_1_1
VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT = 0x00020000,
// Provided by VK_VERSION_1_1
VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT = 0x00040000,
// Provided by VK_VERSION_1_1
VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT = 0x00080000,
// Provided by VK_VERSION_1_1
VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT = 0x00100000,
// Provided by VK_VERSION_1_1
VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT = 0x00200000,
// Provided by VK_VERSION_1_1
VK_FORMAT_FEATURE_DISJOINT_BIT = 0x00400000,
// Provided by VK_VERSION_1_1
VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT = 0x00800000,
// Provided by VK_VERSION_1_2
VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT = 0x00010000,
} VkFormatFeatureFlagBits;

These values all have the same meaning as the equivalently named values for
VkFormatFeatureFlags2 and may be set in linearTilingFeatures and optimalTilingFeatures,
specifying that the features are supported by images or image views or sampler Y’C_bC_r conversion
objects created with the queried vkGetPhysicalDeviceFormatProperties::format:

- **VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT** specifies that an image view can be sampled from.
- **VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT** specifies that an image view can be used as a storage image.
- **VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT** specifies that an image view can be used as storage image that supports atomic operations.
- **VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT** specifies that an image view can be used as a framebuffer color attachment and as an input attachment.
- **VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT** specifies that an image view can be used as a framebuffer color attachment that supports blending and as an input attachment.
- **VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT** specifies that an image view can be used as a framebuffer depth/stencil attachment and as an input attachment.
- **VK_FORMAT_FEATURE_BLIT_SRC_BIT** specifies that an image can be used as srcImage for the vkCmdBlitImage2 and vkCmdBlitImage commands.
- **VK_FORMAT_FEATURE_BLIT_DST_BIT** specifies that an image can be used as dstImage for the vkCmdBlitImage2 and vkCmdBlitImage commands.
- **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT** specifies that if
VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT is also set, an image view can be used with a sampler that has either of magFilter or minFilter set to VK_FILTER_LINEAR, or mipmapMode set to VK_SAMPLER_MIPMAP_MODE_LINEAR. If VK_FORMAT_FEATURE_BLIT_SRC_BIT is also set, an image can be used as the srcImage to vkCmdBlitImage2 and vkCmdBlitImage with a filter of VK_FILTER_LINEAR. This bit must only be exposed for formats that also support the VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT or VK_FORMAT_FEATURE_BLIT_SRC_BIT.

If the format being queried is a depth/stencil format, this bit only specifies that the depth aspect (not the stencil aspect) of an image of this format supports linear filtering, and that linear filtering of the depth aspect is supported whether depth compare is enabled in the sampler or not. Where depth comparison is supported it may be linear filtered whether this bit is present or not, but where this bit is not present the filtered value may be computed in an implementation-dependent manner which differs from the normal rules of linear filtering. The resulting value must be in the range [0,1] and should be proportional to, or a weighted average of, the number of comparison passes or failures.

- **VK_FORMAT_FEATURE_TRANSFER_SRC_BIT** specifies that an image can be used as a source image for copy commands. If the application apiVersion is Vulkan 1.0 and VK_KHR_maintenance1 is not supported, VK_FORMAT_FEATURE_TRANSFER_SRC_BIT is implied to be set when the format feature flag is not 0.

- **VK_FORMAT_FEATURE_TRANSFER_DST_BIT** specifies that an image can be used as a destination image for copy commands and clear commands. If the application apiVersion is Vulkan 1.0 and VK_KHR_maintenance1 is not supported, VK_FORMAT_FEATURE_TRANSFER_DST_BIT is implied to be set when the format feature flag is not 0.

- **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT** specifies that an image can be used as a sampled image with a min or max VkSamplerReductionMode. This bit must only be exposed for formats that also support the VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT.

- **VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT** specifies that an application can define a sampler Y′C_{b,c} conversion using this format as a source, and that an image of this format can be used with a VkSamplerYcbcrConversionCreateInfo xChromaOffset and/or yChromaOffset of VK_CHROMA_LOCATION_MIDPOINT. Otherwise both xChromaOffset and yChromaOffset must be VK_CHROMA_LOCATION_COSITED_EVEN. If a format does not incorporate chroma downsampling (it is not a “422” or “420” format) but the implementation supports sampler Y′C_{b,c} conversion for this format, the implementation must set VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT.

- **VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT** specifies that an application can define a sampler Y′C_{b,c} conversion using this format as a source, and that an image of this format can be used with a VkSamplerYcbcrConversionCreateInfo xChromaOffset and/or yChromaOffset of VK_CHROMA_LOCATION_COSITED_EVEN. Otherwise both xChromaOffset and yChromaOffset must be VK_CHROMA_LOCATION_MIDPOINT. If neither VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT nor VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT is set, the application must not define a sampler Y′C_{b,c} conversion using this format as a source.

- **VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT** specifies that an application can define a sampler Y′C_{b,c} conversion using this format as a source with chromaFilter set to VK_FILTER_LINEAR.

- **VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT** specifies that the format can have different chroma, min, and mag filters.
• **VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT** specifies that reconstruction is explicit, as described in Chroma Reconstruction. If this bit is not present, reconstruction is implicit by default.

• **VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT** specifies that reconstruction can be forcibly made explicit by setting VkSamplerYcbcrConversionCreateInfo::forceExplicitReconstruction to VK_TRUE. If the format being queried supports VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT it must also support VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT.

• **VK_FORMAT_FEATURE_DISJOINT_BIT** specifies that a multi-planar image can have the VK_IMAGE_CREATE_DISJOINT_BIT set during image creation. An implementation must not set VK_FORMAT_FEATURE_DISJOINT_BIT for single-plane formats.

The following bits may be set in bufferFeatures, specifying that the features are supported by buffers or buffer views created with the queried vkGetPhysicalDeviceFormatProperties::format:

• **VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT** specifies that the format can be used to create a buffer view that can be bound to a VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER descriptor.

• **VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT** specifies that the format can be used to create a buffer view that can be bound to a VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER descriptor.

• **VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT** specifies that atomic operations are supported on VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER with this format.

• **VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT** specifies that the format can be used as a vertex attribute format (VkVertexInputAttributeDescription::format).

// Provided by VK_VERSION_1_0
typedef VkFlags VkFormatFeatureFlags;

VkFormatFeatureFlags is a bitmask type for setting a mask of zero or more VkFormatFeatureFlagBits.

To query supported format features which are properties of the physical device, call:

// Provided by VK_VERSION_1_1
void vkGetPhysicalDeviceFormatProperties2(
    VkPhysicalDevice physicalDevice,  
    VkFormat format, 
    VkFormatProperties2* pFormatProperties);

• **physicalDevice** is the physical device from which to query the format properties.

• **format** is the format whose properties are queried.

• **pFormatProperties** is a pointer to a VkFormatProperties2 structure in which physical device properties for format are returned.
vkGetPhysicalDeviceFormatProperties2 behaves similarly to vkGetPhysicalDeviceFormatProperties, with the ability to return extended information in a pNext chain of output structures.

**Valid Usage (Implicit)**

- VUID-vkGetPhysicalDeviceFormatProperties2-physicalDevice-parameter
  
  physicalDevice must be a valid VkPhysicalDevice handle

- VUID-vkGetPhysicalDeviceFormatProperties2-format-parameter
  
  format must be a valid VkFormat value

- VUID-vkGetPhysicalDeviceFormatProperties2-pFormatProperties-parameter
  
  pFormatProperties must be a valid pointer to a VkFormatProperties2 structure

The VkFormatProperties2 structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkFormatProperties2 {
    VkStructureType sType;
    void* pNext;
    VkFormatProperties formatProperties;
} VkFormatProperties2;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- formatProperties is a VkFormatProperties structure describing features supported by the requested format.

**Valid Usage (Implicit)**

- VUID-VkFormatProperties2-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_2

- VUID-VkFormatProperties2-pNext-pNext
  
  pNext must be NULL or a pointer to a valid instance of VkFormatProperties3

- VUID-VkFormatProperties2-sType-unique
  
  The sType value of each struct in the pNext chain must be unique

To query supported format extended features which are properties of the physical device, add VkFormatProperties3 structure to the pNext chain of VkFormatProperties2.

The VkFormatProperties3 structure is defined as:

```c
// Provided by VK_VERSION_1_3
typedef struct VkFormatProperties3 {
    VkStructureType sType;
    ...
} VkFormatProperties3;
```
void* pNext;
VkFormatFeatureFlags2 linearTilingFeatures;
VkFormatFeatureFlags2 optimalTilingFeatures;
VkFormatFeatureFlags2 bufferFeatures;
} VkFormatProperties3;

- **linearTilingFeatures** is a bitmask of `VkFormatFeatureFlagBits2` specifying features supported by images created with a tiling parameter of `VK_IMAGE_TILING_LINEAR`.

- **optimalTilingFeatures** is a bitmask of `VkFormatFeatureFlagBits2` specifying features supported by images created with a tiling parameter of `VK_IMAGE_TILING_OPTIMAL`.

- **bufferFeatures** is a bitmask of `VkFormatFeatureFlagBits2` specifying features supported by buffers.

The bits reported in **linearTilingFeatures**, **optimalTilingFeatures** and **bufferFeatures** must include the bits reported in the corresponding fields of `VkFormatProperties2::formatProperties`.

### Valid Usage (Implicit)

- VUID-VkFormatProperties3-sType-sType
  
sType must be `VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_3`

Bits which can be set in the `VkFormatProperties3` features **linearTilingFeatures**, **optimalTilingFeatures**, and **bufferFeatures** are:

```c
// Provided by VK_VERSION_1_3
// Flag bits for VkFormatFeatureFlagBits2
typedef VkFlags64 VkFormatFeatureFlagBits2;
static const VkFormatFeatureFlagBits2 VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_BIT = 0x00000001ULL;
static const VkFormatFeatureFlagBits2 VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_BIT_KHR = 0x00000001ULL;
static const VkFormatFeatureFlagBits2 VK_FORMAT_FEATURE_2_STORAGE_IMAGE_BIT = 0x00000002ULL;
static const VkFormatFeatureFlagBits2 VK_FORMAT_FEATURE_2_STORAGE_IMAGE_BIT_KHR = 0x00000002ULL;
static const VkFormatFeatureFlagBits2 VK_FORMAT_FEATURE_2_STORAGE_IMAGE_ATOMIC_BIT = 0x00000004ULL;
static const VkFormatFeatureFlagBits2 VK_FORMAT_FEATURE_2_STORAGE_IMAGE_ATOMIC_BIT_KHR = 0x00000004ULL;
static const VkFormatFeatureFlagBits2 VK_FORMAT_FEATURE_2_UNIFORM_TEXEL_BUFFER_BIT = 0x00000008ULL;
static const VkFormatFeatureFlagBits2 VK_FORMAT_FEATURE_2_UNIFORM_TEXEL_BUFFER_BIT_KHR = 0x00000008ULL;
```
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_STORAGE_TEXEL_BUFFER_ATOMIC_BIT = 0x00000020ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_STORAGE_TEXEL_BUFFER_ATOMIC_BIT_KHR = 0x00000020ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_VERTEX_BUFFER_BIT = 0x00000040ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_VERTEX_BUFFER_BIT_KHR = 0x00000040ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_COLOR_ATTACHMENT_BIT = 0x00000080ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_COLOR_ATTACHMENT_BIT_KHR = 0x00000080ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_COLOR_ATTACHMENT_BLEND_BIT = 0x00000100ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_COLOR_ATTACHMENT_BLEND_BIT_KHR = 0x00000100ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_DEPTH_STENCIL_ATTACHMENT_BIT = 0x00000200ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_DEPTH_STENCIL_ATTACHMENT_BIT_KHR = 0x00000200ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_BLIT_SRC_BIT = 0x00000400ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_BLIT_SRC_BIT_KHR = 0x00000400ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_BLIT_DST_BIT = 0x00000800ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_BLIT_DST_BIT_KHR = 0x00000800ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_FILTER_LINEAR_BIT = 0x00001000ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_FILTER_LINEAR_BIT_KHR = 0x00001000ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_FILTER_CUBIC_BIT = 0x00002000ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT = 0x00002000ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_TRANSFER_SRC_BIT = 0x00004000ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_TRANSFER_SRC_BIT_KHR = 0x00004000ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_TRANSFER_DST_BIT = 0x00008000ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_TRANSFER_DST_BIT_KHR = 0x00008000ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_FILTER_MINMAX_BIT = 0x00010000ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_FILTER_MINMAX_BIT_KHR = 0x00010000ULL;
static const VkFormatFeatureFlagBits2
  VK_FORMAT_FEATURE_2_MIDPOINT_CHROMA_SAMPLES_BIT = 0x00020000ULL;
The following bits may be set in `linearTilingFeatures` and `optimalTilingFeatures`, specifying that the features are supported by images or image views or sampler Y'CbCr conversion objects created with the queried `vkGetPhysicalDeviceFormatProperties2::format`:

- `VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_BIT` specifies that an image view can be sampled from.
• **VK_FORMAT_FEATURE_2_STORAGE_IMAGE_BIT** specifies that an image view can be used as a storage image.

• **VK_FORMAT_FEATURE_2_STORAGE_IMAGE_ATOMIC_BIT** specifies that an image view can be used as storage image that supports atomic operations.

• **VK_FORMAT_FEATURE_2_COLOR_ATTACHMENT_BIT** specifies that an image view can be used as a framebuffer color attachment and as an input attachment.

• **VK_FORMAT_FEATURE_2_COLOR_ATTACHMENT_BLEND_BIT** specifies that an image view can be used as a framebuffer color attachment that supports blending and as an input attachment.

• **VK_FORMAT_FEATURE_2_DEPTH_STENCIL_ATTACHMENT_BIT** specifies that an image view can be used as a framebuffer depth/stencil attachment and as an input attachment.

• **VK_FORMAT_FEATURE_2_BLIT_SRC_BIT** specifies that an image can be used as the srcImage for `vkCmdBlitImage2` and `vkCmdBlitImage`.

• **VK_FORMAT_FEATURE_2_BLIT_DST_BIT** specifies that an image can be used as the dstImage for `vkCmdBlitImage2` and `vkCmdBlitImage`.

• **VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_FILTER_LINEAR_BIT** specifies that if **VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_BIT** is also set, an image view can be used with a sampler that has either of `magFilter` or `minFilter` set to `VK_FILTER_LINEAR`, or `mipmapMode` set to `VK_SAMPLER_MIPMAP_MODE_LINEAR`. If **VK_FORMAT_FEATURE_2_BLIT_SRC_BIT** is also set, an image can be used as the srcImage for `vkCmdBlitImage2` and `vkCmdBlitImage` with a filter of `VK_FILTER_LINEAR`. This bit must only be exposed for formats that also support the **VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_BIT** or **VK_FORMAT_FEATURE_2_BLIT_SRC_BIT**.

If the format being queried is a depth/stencil format, this bit only specifies that the depth aspect (not the stencil aspect) of an image of this format supports linear filtering. Where depth comparison is supported it may be linear filtered whether this bit is present or not, but where this bit is not present the filtered value may be computed in an implementation-dependent manner which differs from the normal rules of linear filtering. The resulting value must be in the range [0,1] and should be proportional to, or a weighted average of, the number of comparison passes or failures.

• **VK_FORMAT_FEATURE_2_TRANSFER_SRC_BIT** specifies that an image can be used as a source image for copy commands.

• **VK_FORMAT_FEATURE_2_TRANSFER_DST_BIT** specifies that an image can be used as a destination image for copy commands and clear commands.

• **VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_FILTER_MINMAX_BIT** specifies `VkImage` can be used as a sampled image with a min or max `VkSamplerReductionMode`. This bit must only be exposed for formats that also support the **VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_BIT**.

• **VK_FORMAT_FEATURE_2_MIDPOINT_CHROMA_SAMPLES_BIT** specifies that an application can define a sampler Y’CbCr conversion using this format as a source, and that an image of this format can be used with a `VkSamplerYcbcrConversionCreateInfo` `xChromaOffset` and/or `yChromaOffset` of `VK_CHROMA_LOCATION_MIDPOINT`. Otherwise both `xChromaOffset` and `yChromaOffset` must be `VK_CHROMA_LOCATION_COSITED_EVEN`. If a format does not incorporate chroma downsampling (it is not a “422” or “420” format) but the implementation supports sampler Y’CbCr conversion for this format, the implementation must set **VK_FORMAT_FEATURE_2_MIDPOINT_CHROMA_SAMPLES_BIT**.
• **VK_FORMAT_FEATURE_2_COSITED_CHROMA_SAMPLES_BIT** specifies that an application can define a sampler Y′C_bC_R conversion using this format as a source, and that an image of this format can be used with a VkSamplerYcbcrConversionCreateInfo xChromaOffset and/or yChromaOffset of VK_CHROMA_LOCATION_COSITED_EVEN. Otherwise both xChromaOffset and yChromaOffset must be VK_CHROMA_LOCATION_MIDPOINT. If neither VK_FORMAT_FEATURE_2_COSITED_CHROMA_SAMPLES_BIT nor VK_FORMAT_FEATURE_2_MIDPOINT_CHROMA_SAMPLES_BIT is set, the application must not define a sampler Y′C_bC_R conversion using this format as a source.

• **VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT** specifies that an application can define a sampler Y′C_bC_R conversion using this format as a source with chromaFilter set to VK_FILTER_LINEAR.

• **VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT** specifies that the format can have different chroma, min, and mag filters.

• **VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT** specifies that reconstruction is explicit, as described in Chroma Reconstruction. If this bit is not present, reconstruction is implicit by default.

• **VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT** specifies that reconstruction can be forcibly made explicit by setting VkSamplerYcbcrConversionCreateInfo::forceExplicitReconstruction to VK_TRUE. If the format being queried supports VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT it must also support VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT.

• **VK_FORMAT_FEATURE_2_DISJOINT_BIT** specifies that a multi-planar image can have the VK_IMAGE_CREATE_DISJOINT_BIT set during image creation. An implementation must not set VK_FORMAT_FEATURE_2_DISJOINT_BIT for single-plane formats.

• **VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT** specifies that image views or buffer views created with this format can be used as storage images for read operations without specifying a format.

• **VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT** specifies that image views or buffer views created with this format can be used as storage images for write operations without specifying a format.

• **VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_DEPTH_COMPARISON_BIT** specifies that image views created with this format can be used for depth comparison performed by OpImage*Dref* instructions.

The following bits may be set in bufferFeatures, specifying that the features are supported by buffers or buffer views created with the queried VkGetPhysicalDeviceFormatProperties2::format:

• **VK_FORMAT_FEATURE_2_UNIFORM_TEXEL_BUFFER_BIT** specifies that the format can be used to create a buffer view that can be bound to a VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER descriptor.

• **VK_FORMAT_FEATURE_2_STORAGE_TEXEL_BUFFER_BIT** specifies that the format can be used to create a buffer view that can be bound to a VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER descriptor.

• **VK_FORMAT_FEATURE_2_STORAGE_TEXEL_BUFFER_ATOMIC_BIT** specifies that atomic operations are supported on VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER with this format.
• \texttt{VK_FORMAT_FEATURE_2_VERTEX_BUFFER_BIT} specifies that the format \texttt{can} be used as a vertex attribute format (\texttt{VkVertexInputAttributeDescription::format}).

• \texttt{VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT} specifies that buffer views created with this format \texttt{can} be used as \texttt{storage texel buffers} for read operations without specifying a format.

• \texttt{VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT} specifies that buffer views created with this format \texttt{can} be used as \texttt{storage texel buffers} for write operations without specifying a format.

```c
// Provided by VK_VERSION_1_3
typedef VkFlags64 VkFormatFeatureFlags2;
```

\textit{VkFormatFeatureFlags2} is a bitmask type for setting a mask of zero or more \textit{VkFormatFeatureFlagBits2}.

### 34.2.1. Potential Format Features

Some valid usage conditions depend on the format features supported by a \textit{VkImage} whose \textit{VkImageTiling} is unknown. In such cases the exact \textit{VkFormatFeatureFlagBits} supported by the \textit{VkImage} cannot be determined, so the valid usage conditions are expressed in terms of the \textit{potential format features} of the \textit{VkImage} format.

The \textit{potential format features} of a \textit{VkFormat} are defined as follows:

• The union of \textit{VkFormatFeatureFlagBits} and \textit{VkFormatFeatureFlagBits2}, supported when the \textit{VkImageTiling} is \texttt{VK_IMAGE_TILING_OPTIMAL} or \texttt{VK_IMAGE_TILING_LINEAR}

### 34.3. Required Format Support

Implementations \textbf{must} support at least the following set of features on the listed formats. For images, these features \textbf{must} be supported for every \textit{VkImageType} (including arrayed and cube variants) unless otherwise noted. These features are supported on existing formats without needing to advertise an extension or needing to explicitly enable them. Support for additional functionality beyond the requirements listed here is queried using the \texttt{vkGetPhysicalDeviceFormatProperties} command.

\textit{Note}

Unless otherwise excluded below, the required formats are supported for all \textit{VkImageCreateFlags} values as long as those flag values are otherwise allowed.

The following tables show which feature bits \textbf{must} be supported for each format. Formats that are required to support \texttt{VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT} \textbf{must} also support \texttt{VK_FORMAT_FEATURE_TRANSFER_SRC_BIT} and \texttt{VK_FORMAT_FEATURE_TRANSFER_DST_BIT}.

\textit{Table 42. Key for format feature tables}

| Key | This feature \textbf{must} be supported on the named format |
This feature **must** be supported on at least some of the named formats, with more information in the table where the symbol appears

This feature **must** be supported with some caveats or preconditions, with more information in the table where the symbol appears

<table>
<thead>
<tr>
<th><strong>Table 43. Feature bits in</strong> <code>optimalTilingFeatures</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VK_FORMAT_FEATURE_TRANSFER_SRC_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_FEATURE_TRANSFER_DST_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_FEATURE_BLIT_SRC_BIT</code></td>
</tr>
<tr>
<td>` VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</td>
</tr>
<tr>
<td><code>VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_FEATURE_BLIT_DST_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</code></td>
</tr>
<tr>
<td>` VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Table 44. Feature bits in</strong> <code>bufferFeatures</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</code></td>
</tr>
<tr>
<td><code> VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</code></td>
</tr>
</tbody>
</table>
### Table 45. Mandatory format support: sub-byte components

<table>
<thead>
<tr>
<th>Format Features</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_R4G4_UNORM_PACK8</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</td>
<td>VK_FORMAT_R4G4B4A4_UNORM_PACK16</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</td>
<td>VK_FORMAT_B4G4R4A4_UNORM_PACK16</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT</td>
<td>VK_FORMAT_R5G6B5_UNORM_PACK16</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
<td>VK_FORMAT_B5G5R5A1_UNORM_PACK16</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</td>
<td>VK_FORMAT_A1R5G5B5_UNORM_PACK16</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_DST_BIT</td>
<td>VK_FORMAT_A4R4G4B4_UNORM_PACK16</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</td>
<td>VK_FORMAT_A4B4G4R4_UNORM_PACK16</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</td>
<td>Format features marked † must be supported for optimalTilingFeatures if the VkPhysicalDevice supports the VkPhysicalDevice4444FormatsFeaturesEXT::formatA4R4G4B4 feature.</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT</td>
<td>Format features marked ‡ must be supported for optimalTilingFeatures if the VkPhysicalDevice supports the VkPhysicalDevice4444FormatsFeaturesEXT::formatA4B4G4R4 feature.</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_SRC_BIT</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT</td>
<td></td>
</tr>
</tbody>
</table>
Table 46. Mandatory format support: 1-3 byte-sized components

<table>
<thead>
<tr>
<th>Format</th>
<th>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</th>
<th>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</th>
<th>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</th>
<th>VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT</th>
<th>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</th>
<th>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</th>
<th>VK_FORMAT_FEATURE_BLIT_DST_BIT</th>
<th>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</th>
<th>VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</th>
<th>VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT</th>
<th>VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</th>
<th>VK_FORMAT_FEATURE_BLIT_SRC_BIT</th>
<th>VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_R8_UNORM</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8_SNORM</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8_USCALED</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8_SSCALED</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8_UINT</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8_SINT</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8_SRGB</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8_UNORM</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8_SNORM</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8_USCALED</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8_SSCALED</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8_UINT</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8_SINT</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8_SRGB</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8_UNORM</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8_SNORM</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8_USCALED</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8_SSCALED</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8_UINT</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8_SINT</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8_SRGB</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8B8_UNORM</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8B8_SNORM</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8B8_USCALED</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8B8_SSCALED</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8B8_UINT</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8B8_SINT</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8B8_SRGB</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8_UNORM</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>
Format features marked with ‡ must be supported for optimalTilingFeatures if the 
VkPhysicalDevice supports the shaderStorageImageExtendedFormats feature.
Table 47. Mandatory format support: 4 byte-sized components

<table>
<thead>
<tr>
<th>Format</th>
<th>VK_FORMAT_R8G8B8A8_UNORM</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VK_FORMAT_R8G8B8A8_SNORM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_R8G8B8A8_USCALED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_R8G8B8A8_SSCALED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_R8G8B8A8_UINT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_R8G8B8A8_SINT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_R8G8B8A8_SRGB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_B8G8R8A8_UNORM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_B8G8R8A8_SNORM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_B8G8R8A8_USCALED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_B8G8R8A8_SSCALED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_B8G8R8A8_UINT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_B8G8R8A8_SINT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_B8G8R8A8_SRGB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_A8B8G8R8_UNORM_PACK32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_A8B8G8R8_SNORM_PACK32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_A8B8G8R8_USCALED_PACK32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_A8B8G8R8_SSCALED_PACK32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_A8B8G8R8_UINT_PACK32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_A8B8G8R8_SINT_PACK32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_A8B8G8R8_SRGB_PACK32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>VK_FORMAT_A2R10G10B10_UNORM_PACK32</td>
<td>VK_FORMAT_A2R10G10B10_SNORM_PACK32</td>
<td>VK_FORMAT_A2R10G10B10_USCALED_PACK32</td>
<td>VK_FORMAT_A2R10G10B10_SSCALED_PACK32</td>
<td>VK_FORMAT_A2R10G10B10_UINT_PACK32</td>
<td>VK_FORMAT_A2R10G10B10_SINT_PACK32</td>
<td>VK_FORMAT_A2B10G10R10_UNORM_PACK32</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------</td>
<td>---------------------------------</td>
<td>---------------------------------</td>
<td>---------------------------------</td>
<td>---------------------------------</td>
<td>---------------------------------</td>
<td>---------------------------------</td>
</tr>
</tbody>
</table>
| Format features marked with ‡ must be supported for optimalTilingFeatures if the VkPhysicalDevice supports the shaderStorageImageExtendedFormats feature.
Table 49. Mandatory format support: 16-bit components

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
</tr>
<tr>
<td>Format String</td>
<td>VS</td>
<td>GS</td>
<td>RS</td>
<td>SS</td>
<td>MS</td>
<td>GS</td>
<td>RS</td>
<td>SS</td>
<td>MS</td>
<td>GS</td>
<td>RS</td>
<td>SS</td>
<td>MS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16A16_USCALED</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16A16_SSCALED</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16A16_UINT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16A16_SINT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16A16_SFLOAT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format features marked with ‡ must be supported for optimalTilingFeatures if the VkPhysicalDevice supports the shaderStorageImageExtendedFormats feature.
Table 50. Mandatory format support: 32-bit components

<table>
<thead>
<tr>
<th>Feature</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_R32_UINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_DST_BIT</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_SRC_BIT</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT</td>
<td>✓</td>
</tr>
</tbody>
</table>
Table 51. Mandatory format support: 64-bit/uneven components

<table>
<thead>
<tr>
<th>Format features</th>
<th>VK_FORMAT_R64_UINT</th>
<th>VK_FORMAT_R64_SINT</th>
<th>VK_FORMAT_R64_SFLOAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_DST_BIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_SRC_BIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format features marked with ‡ must be supported for optimalTilingFeatures if the VkPhysicalDevice supports the shaderStorageImageExtendedFormats feature.
Table 52. Mandatory format support: depth/stencil with \( \text{VkImageType VK_IMAGE_TYPE_2D} \)

<table>
<thead>
<tr>
<th>Feature</th>
<th>VK_FORMAT_D16_UNORM</th>
<th>VK_FORMAT_X8_D24_UNORM_PACK32</th>
<th>VK_FORMAT_D32_SFLOAT</th>
<th>VK_FORMAT_S8_UINT</th>
<th>VK_FORMAT_D16_UNORM_S8_UINT</th>
<th>VK_FORMAT_D24_UNORM_S8_UINT</th>
<th>VK_FORMAT_D32_SFLOAT_S8_UINT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_DST_BIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_SRC_BIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_SRC_BIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE.Bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Feature must be supported for at least one of**

- VK_FORMAT_X8_D24_UNORM_PACK32 and VK_FORMAT_D32_SFLOAT,
- and **must be supported for at least one of**
  - VK_FORMAT_D24_UNORM_S8_UINT and VK_FORMAT_D32_SFLOAT_S8_UINT.

**bufferFeatures must not support any features for these formats**
Table 53. Mandatory format support: BC compressed formats with VkImageType VK_IMAGE_TYPE_2D and VK_IMAGE_TYPE_3D

<table>
<thead>
<tr>
<th>Format</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_BC1_RGB_UNORM_BLOCK</td>
<td>†††</td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGB_SRGB_BLOCK</td>
<td>†††</td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGBA_UNORM_BLOCK</td>
<td>†††</td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGBA_SRGB_BLOCK</td>
<td>†††</td>
</tr>
<tr>
<td>VK_FORMAT_BC2_UNORM_BLOCK</td>
<td>†††</td>
</tr>
<tr>
<td>VK_FORMAT_BC2_SRGB_BLOCK</td>
<td>†††</td>
</tr>
<tr>
<td>VK_FORMAT_BC3_UNORM_BLOCK</td>
<td>†††</td>
</tr>
<tr>
<td>VK_FORMAT_BC3_SRGB_BLOCK</td>
<td>†††</td>
</tr>
<tr>
<td>VK_FORMAT_BC4_UNORM_BLOCK</td>
<td>†††</td>
</tr>
<tr>
<td>VK_FORMAT_BC4_SNORM_BLOCK</td>
<td>†††</td>
</tr>
<tr>
<td>VK_FORMAT_BC5_UNORM_BLOCK</td>
<td>†††</td>
</tr>
<tr>
<td>VK_FORMAT_BC5_SNORM_BLOCK</td>
<td>†††</td>
</tr>
<tr>
<td>VK_FORMAT_BC6H_UFLOAT_BLOCK</td>
<td>†††</td>
</tr>
<tr>
<td>VK_FORMAT_BC6H_SFLOAT_BLOCK</td>
<td>†††</td>
</tr>
<tr>
<td>VK_FORMAT_BC7_UNORM_BLOCK</td>
<td>†††</td>
</tr>
<tr>
<td>VK_FORMAT_BC7_SRGB_BLOCK</td>
<td>†††</td>
</tr>
</tbody>
</table>

The VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT, VK_FORMAT_FEATURE_BLIT_SRC_BIT and VK_FORMATFEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT features must be supported in optimalTilingFeatures for all the formats in at least one of: this table, Mandatory format support: ETC2 and EAC compressed formats with VkImageType VK_IMAGE_TYPE_2D, or Mandatory format support: ASTC LDR compressed formats with VkImageType VK_IMAGE_TYPE_2D.
Table 54. Mandatory format support: ETC2 and EAC compressed formats with <code>VkImageType VK_IMAGE_TYPE_2D</code>

<table>
<thead>
<tr>
<th>Format</th>
<th>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</th>
<th>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</th>
<th>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</th>
<th>VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT</th>
<th>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</th>
<th>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</th>
<th>VK_FORMAT_FEATURE_BLIT_DST_BIT</th>
<th>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</th>
<th>VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</th>
<th>VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT</th>
<th>VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</th>
<th>VK_FORMAT_FEATURE_BLIT_SRC_BIT</th>
<th>VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>†</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>†</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>†</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>†</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>†</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>†</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_EAC_R11_UNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>†</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_EAC_R11_SNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>†</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_EAC_R11G11_UNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>†</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_EAC_R11G11_SNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>†</td>
<td></td>
</tr>
</tbody>
</table>

The <code>VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT</code>, <code>VK_FORMAT_FEATURE_BLIT_SRC_BIT</code> and <code>VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</code> features must be supported in <code>optimalTilingFeatures</code> for all the formats in at least one of: this table, Mandatory format support: BC compressed formats with <code>VkImageType VK_IMAGE_TYPE_2D</code> and <code>VK_IMAGE_TYPE_3D</code>, or Mandatory format support: ASTC LDR compressed formats with <code>VkImageType VK_IMAGE_TYPE_2D</code>. 

---

1245
Table 55. Mandatory format support: ASTC LDR compressed formats with \texttt{VkImageType VK\_IMAGE\_TYPE\_2D}

<table>
<thead>
<tr>
<th>Format</th>
<th>Feature Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{VK_FORMAT_ASTC_4x4_UNORM_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_4x4_SRGB_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_5x4_UNORM_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_5x4_SRGB_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_5x5_UNORM_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_5x5_SRGB_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_6x5_UNORM_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_6x5_SRGB_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_6x6_UNORM_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_6x6_SRGB_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_8x5_UNORM_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_8x5_SRGB_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_8x6_UNORM_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_8x6_SRGB_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_8x8_UNORM_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_8x8_SRGB_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_10x5_UNORM_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_10x5_SRGB_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_10x6_UNORM_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_10x6_SRGB_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ASTC_10x8_UNORM_BLOCK}</td>
<td>† † †</td>
</tr>
</tbody>
</table>
The `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `VK_FORMAT_FEATURE_BLIT_SRC_BIT` and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` features must be supported in `optimalTilingFeatures` for all the formats in at least one of: this table, Mandatory format support: BC compressed formats with `VkImageType` `VK_IMAGE_TYPE_2D` and `VK_IMAGE_TYPE_3D`, or Mandatory format support: ETC2 and EAC compressed formats with `VkImageType` `VK_IMAGE_TYPE_2D`.

To be used with `VkImageView` with `subresourceRange.aspectMask` equal to `VK_IMAGE_ASPECT_COLOR_BIT`, sampler `Y’C_bC_r` conversion must be enabled for the following formats:

**Table 56. Formats requiring sampler `Y’C_bC_r` conversion for `VK_IMAGE_ASPECT_COLOR_BIT` image views**

<table>
<thead>
<tr>
<th>Format</th>
<th>Planes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VK_FORMAT_G8B8G8R8_422_UNORM</code></td>
<td>1</td>
</tr>
<tr>
<td><code>VK_FORMAT_B8G8R8G8_422_UNORM</code></td>
<td>1</td>
</tr>
<tr>
<td><code>VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM</code></td>
<td>3</td>
</tr>
<tr>
<td><code>VK_FORMAT_G8_B8R8_2PLANE_420_UNORM</code></td>
<td>2</td>
</tr>
<tr>
<td><code>VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM</code></td>
<td>3</td>
</tr>
<tr>
<td><code>VK_FORMAT_G8_B8R8_2PLANE_422_UNORM</code></td>
<td>2</td>
</tr>
<tr>
<td><code>VK_FORMAT_G8_B8R8_3PLANE_444_UNORM</code></td>
<td>3</td>
</tr>
<tr>
<td><code>VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16</code></td>
<td>1</td>
</tr>
<tr>
<td><code>VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16</code></td>
<td>1</td>
</tr>
<tr>
<td><code>VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16</code></td>
<td>1</td>
</tr>
</tbody>
</table>
Format features marked † must be supported for optimalTilingFeatures with VkImageType VK_IMAGE_TYPE_2D if the VkPhysicalDevice supports the VkPhysicalDeviceSamplerYcbcrConversionFeatures feature.

Implementations are not required to support the VK_IMAGE_CREATE_SPARSE_BINDING_BIT, VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT, or VK_IMAGE_CREATE_SPARSE_ALIASED_BIT VkImageCreateFlags for the above formats that require sampler Y’CbCr conversion. To determine whether the implementation supports sparse image creation flags with these formats use vkGetPhysicalDeviceImageFormatProperties or vkGetPhysicalDeviceImageFormatProperties2.

### 34.3.1. Formats without shader storage format

The device-level features for using a storage image with an image format of Unknown, shaderStorageImageReadWithoutFormat and shaderStorageImageWriteWithoutFormat, only apply to the following formats:
- VK_FORMAT_R8G8B8A8_UNORM
- VK_FORMAT_R8G8B8A8_SNORM
- VK_FORMAT_R8G8B8A8_UINT
- VK_FORMAT_R8G8B8A8_SINT
- VK_FORMAT_R32_UINT
- VK_FORMAT_R32_SINT
- VK_FORMAT_R32_SFLOAT
- VK_FORMAT_R32G32_UINT
- VK_FORMAT_R32G32_SINT
- VK_FORMAT_R32G32_SFLOAT
- VK_FORMAT_R32G32B32A32_UINT
- VK_FORMAT_R32G32B32A32_SINT
- VK_FORMAT_R32G32B32A32_SFLOAT
- VK_FORMAT_R16G16B16A16_UINT
- VK_FORMAT_R16G16B16A16_SINT
- VK_FORMAT_R16G16B16A16_SFLOAT
- VK_FORMAT_R16G16_SFLOAT
- VK_FORMAT_B10G11R11_UFLOAT_PACK32
- VK_FORMAT_R16_SFLOAT
- VK_FORMAT_R16G16B16A16_UNORM
- VK_FORMAT_A2B10G10R10_UNORM_PACK32
- VK_FORMAT_R16G16_UNORM
- VK_FORMAT_R8G8_UNORM
- VK_FORMAT_R16_UNORM
- VK_FORMAT_R8_UNORM
- VK_FORMAT_R16G16B16A16_SNORM
- VK_FORMAT_R16G16_SNORM
- VK_FORMAT_R8G8_SNORM
- VK_FORMAT_R16_SNORM
- VK_FORMAT_R8_SNORM
- VK_FORMAT_R16G16_SINT
- VK_FORMAT_R8G8_SINT
- VK_FORMAT_R16_SINT
- VK_FORMAT_R8_SINT
- VK_FORMAT_A2B10G10R10_UINT_PACK32
• VK_FORMAT_R16G16_UINT
• VK_FORMAT_R8G8_UINT
• VK_FORMAT_R16_UINT
• VK_FORMAT_R8_UINT

Note
This list of formats is the union of required storage formats from Required Format Support section and formats listed in shaderStorageImageExtendedFormats.

An implementation that supports VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT for any format from the given list of formats and supports shaderStorageImageReadWithoutFormat must support VK_FORMAT_FEATURE_2_STORAGE_READ_WITHOUT_FORMAT_BIT for that same format if Vulkan 1.3 or the VK_KHR_format_feature_flags2 extension is supported.

An implementation that supports VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT for any format from the given list of formats and supports shaderStorageImageWriteWithoutFormat must support VK_FORMAT_FEATURE_2_STORAGE_WRITE_WITHOUT_FORMAT_BIT for that same format if Vulkan 1.3 or the VK_KHR_format_feature_flags2 extension is supported.

34.3.2. Depth comparison format support

If Vulkan 1.3 or the VK_KHR_format_feature_flags2 extension is supported, a depth/stencil format with a depth component supporting VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT must support VK_FORMAT_FEATURE_2_SAMPLED_IMAGE_DEPTH_COMPARISON_BIT.
Chapter 35. Additional Capabilities

This chapter describes additional capabilities beyond the minimum capabilities described in the Limits and Formats chapters, including:

- Additional Image Capabilities
- Additional Buffer Capabilities
- Optional Semaphore Capabilities
- Optional Fence Capabilities

35.1. Additional Image Capabilities

Additional image capabilities, such as larger dimensions or additional sample counts for certain image types, or additional capabilities for linear tiling format images, are described in this section.

To query additional capabilities specific to image types, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkGetPhysicalDeviceImageFormatProperties(
    VkPhysicalDevice physicalDevice,
    VkFormat format,
    VkImageType type,
    VkImageTiling tiling,
    VkImageUsageFlags usage,
    VkImageCreateFlags flags,
    VkImageFormatProperties* pImageFormatProperties);
```

- `physicalDevice` is the physical device from which to query the image capabilities.
- `format` is a `VkFormat` value specifying the image format, corresponding to `VkImageCreateInfo::format`.
- `type` is a `VkImageType` value specifying the image type, corresponding to `VkImageCreateInfo::imageType`.
- `tiling` is a `VkImageTiling` value specifying the image tiling, corresponding to `VkImageCreateInfo::tiling`.
- `usage` is a bitmask of `VkImageUsageFlagBits` specifying the intended usage of the image, corresponding to `VkImageCreateInfo::usage`.
- `flags` is a bitmask of `VkImageCreateFlagBits` specifying additional parameters of the image, corresponding to `VkImageCreateInfo::flags`.
- `pImageFormatProperties` is a pointer to a `VkImageFormatProperties` structure in which capabilities are returned.

The `format`, `type`, `tiling`, `usage`, and `flags` parameters correspond to parameters that would be consumed by `vkCreateImage` (as members of `VkImageCreateInfo`).
If \(\text{format}\) is not a supported image format, or if the combination of \(\text{format, type, tiling, usage}\), and \(\text{flags}\) is not supported for images, then \(\text{vkGetPhysicalDeviceImageFormatProperties}\) returns \(\text{VK_ERROR_FORMAT_NOT_SUPPORTED}\).

The limitations on an image format that are reported by \(\text{vkGetPhysicalDeviceImageFormatProperties}\) have the following property: if \(\text{usage}_1\) and \(\text{usage}_2\) of type \(\text{VkImageUsageFlags}\) are such that the bits set in \(\text{usage}_1\) are a subset of the bits set in \(\text{usage}_2\), and \(\text{flags}_1\) and \(\text{flags}_2\) of type \(\text{VkImageCreateFlags}\) are such that the bits set in \(\text{flags}_1\) are a subset of the bits set in \(\text{flags}_2\), then the limitations for \(\text{usage}_1\) and \(\text{flags}_1\) must be no more strict than the limitations for \(\text{usage}_2\) and \(\text{flags}_2\), for all values of \(\text{format, type, and tiling}\).

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceImageFormatProperties-physicalDevice-parameter \(\text{physicalDevice}\) must be a valid \(\text{VkPhysicalDevice}\) handle
- VUID-vkGetPhysicalDeviceImageFormatProperties-format-parameter \(\text{format}\) must be a valid \(\text{VkFormat}\) value
- VUID-vkGetPhysicalDeviceImageFormatProperties-type-parameter \(\text{type}\) must be a valid \(\text{VkImageType}\) value
- VUID-vkGetPhysicalDeviceImageFormatProperties-tiling-parameter \(\text{tiling}\) must be a valid \(\text{VkImageTiling}\) value
- VUID-vkGetPhysicalDeviceImageFormatProperties-usage-parameter \(\text{usage}\) must be a valid combination of \(\text{VkImageUsageFlagBits}\) values
- VUID-vkGetPhysicalDeviceImageFormatProperties-usage-requiredbitmask \(\text{usage}\) must not be 0
- VUID-vkGetPhysicalDeviceImageFormatProperties-flags-parameter \(\text{flags}\) must be a valid combination of \(\text{VkImageCreateFlagBits}\) values
- VUID-vkGetPhysicalDeviceImageFormatProperties-pImageFormatProperties-parameter \(\text{pImageFormatProperties}\) must be a valid pointer to a \(\text{VkImageFormatProperties}\) structure

### Return Codes

**Success**
- \(\text{VK_SUCCESS}\)

**Failure**
- \(\text{VK_ERROR_OUT_OF_HOST_MEMORY}\)
- \(\text{VK_ERROR_OUT_OF_DEVICE_MEMORY}\)
- \(\text{VK_ERROR_FORMAT_NOT_SUPPORTED}\)

The \(\text{VkImageFormatProperties}\) structure is defined as:
typedef struct VkImageFormatProperties {
    VkExtent3D maxExtent;
    uint32_t maxMipLevels;
    uint32_t maxArrayLayers;
    VkSampleCountFlags sampleCounts;
    VkDeviceSize maxResourceSize;
} VkImageFormatProperties;

- `maxExtent` are the maximum image dimensions. See the Allowed Extent Values section below for how these values are constrained by `type`.

- `maxMipLevels` is the maximum number of mipmap levels. `maxMipLevels` **must** be equal to the number of levels in the complete mipmap chain based on the `maxExtent.width`, `maxExtent.height`, and `maxExtent.depth`, except when one of the following conditions is true, in which case it **may** instead be 1:
  - `vkGetPhysicalDeviceImageFormatProperties::tiling` was `VK_IMAGE_TILING_LINEAR`
  - the `VkPhysicalDeviceImageFormatInfo2::pNext` chain included a `VkPhysicalDeviceExternalImageFormatInfo` structure with a handle type included in the `handleTypes` member for which mipmap image support is not required
  - image format is one of the formats that require a sampler Y’CbCr conversion

- `maxArrayLayers` is the maximum number of array layers. `maxArrayLayers` **must** be no less than `VkPhysicalDeviceLimits::maxImageArrayLayers`, except when one of the following conditions is true, in which case it **may** instead be 1:
  - `tiling` is `VK_IMAGE_TILING_LINEAR`
  - `tiling` is `VK_IMAGE_TILING_OPTIMAL` and `type` is `VK_IMAGE_TYPE_3D`
  - `format` is one of the formats that require a sampler Y’CbCr conversion

- `sampleCounts` is a bitmask of `VkSampleCountFlagBits` specifying all the supported sample counts for this image as described **below**.

- `maxResourceSize` is an upper bound on the total image size in bytes, inclusive of all image subresources. Implementations **may** have an address space limit on total size of a resource, which is advertised by this property. `maxResourceSize` **must** be at least $2^{31}$.

---

**Note**

There is no mechanism to query the size of an image before creating it, to compare that size against `maxResourceSize`. If an application attempts to create an image that exceeds this limit, the creation will fail and `vkCreateImage` will return `VK_ERROR_OUT_OF_DEVICE_MEMORY`. While the advertised limit **must** be at least $2^{31}$, it **may** not be possible to create an image that approaches that size, particularly for `VK_IMAGE_TYPE_1D`.

If the combination of parameters to `vkGetPhysicalDeviceImageFormatProperties` is not supported by the implementation for use in `vkCreateImage`, then all members of `VkImageFormatProperties` will be filled with zero.
Note

Filling `VkImageFormatProperties` with zero for unsupported formats is an exception to the usual rule that output structures have undefined contents on error. This exception was unintentional, but is preserved for backwards compatibility.

To query additional capabilities specific to image types, call:

```
// Provided by VK_VERSION_1_1
VkResult vkGetPhysicalDeviceImageFormatProperties2(
    VkPhysicalDevice physicalDevice,
    const VkPhysicalDeviceImageFormatInfo2* pImageFormatInfo,
    VkImageFormatProperties2* pImageFormatProperties);
```

- `physicalDevice` is the physical device from which to query the image capabilities.
- `pImageFormatInfo` is a pointer to a `VkPhysicalDeviceImageFormatInfo2` structure describing the parameters that would be consumed by `vkCreateImage`.
- `pImageFormatProperties` is a pointer to a `VkImageFormatProperties2` structure in which capabilities are returned.

`vkGetPhysicalDeviceImageFormatProperties2` behaves similarly to `vkGetPhysicalDeviceImageFormatProperties`, with the ability to return extended information in a `pNext` chain of output structures.

**Valid Usage (Implicit)**

- VUID-vkGetPhysicalDeviceImageFormatProperties2-physicalDevice-parameter `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceImageFormatProperties2-pImageFormatInfo-parameter `pImageFormatInfo` must be a valid pointer to a valid `VkPhysicalDeviceImageFormatInfo2` structure
- VUID-vkGetPhysicalDeviceImageFormatProperties2-pImageFormatProperties-parameter `pImageFormatProperties` must be a valid pointer to a `VkImageFormatProperties2` structure

**Return Codes**

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_FORMAT_NOT_SUPPORTED
The `VkPhysicalDeviceImageFormatInfo2` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceImageFormatInfo2 {
    VkStructureType sType;
    const void* pNext;
    VkFormat format;
    VkImageType type;
    VkImageTiling tiling;
    VkImageUsageFlags usage;
    VkImageCreateFlags flags;
} VkPhysicalDeviceImageFormatInfo2;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure. The `pNext` chain of `VkPhysicalDeviceImageFormatInfo2` is used to provide additional image parameters to `vkGetPhysicalDeviceImageFormatProperties2`.
- `format` is a `VkFormat` value indicating the image format, corresponding to `VkImageCreateInfo::format`.
- `type` is a `VkImageType` value indicating the image type, corresponding to `VkImageCreateInfo::imageType`.
- `tiling` is a `VkImageTiling` value indicating the image tiling, corresponding to `VkImageCreateInfo::tiling`.
- `usage` is a bitmask of `VkImageUsageFlagBits` indicating the intended usage of the image, corresponding to `VkImageCreateInfo::usage`.
- `flags` is a bitmask of `VkImageCreateFlagBits` indicating additional parameters of the image, corresponding to `VkImageCreateInfo::flags`.

The members of `VkPhysicalDeviceImageFormatInfo2` correspond to the arguments to `vkGetPhysicalDeviceImageFormatProperties`, with `sType` and `pNext` added for extensibility.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceImageFormatInfo2-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICALDEVICE_IMAGEFORMATINFO_2`

- **VUID-VkPhysicalDeviceImageFormatInfo2-pNext-pNext**
  Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkImageFormatListCreateInfo`, `VkImageStencilUsageCreateInfo`, or `VkPhysicalDeviceExternalImageFormatInfo`.

- **VUID-VkPhysicalDeviceImageFormatInfo2-sType-unique**
  The `sType` value of each struct in the `pNext` chain must be unique.

- **VUID-VkPhysicalDeviceImageFormatInfo2-format-parameter**
  `format` must be a valid `VkFormat` value
• VUID-VkPhysicalDeviceImageFormatInfo2-type-parameter
type must be a valid VkImageType value

• VUID-VkPhysicalDeviceImageFormatInfo2-tiling-parameter
tiling must be a valid VkImageTiling value

• VUID-VkPhysicalDeviceImageFormatInfo2-usage-parameter
usage must be a valid combination of VkImageUsageFlagBits values

• VUID-VkPhysicalDeviceImageFormatInfo2-usage-requiredbitmask
usage must not be 0

• VUID-VkPhysicalDeviceImageFormatInfo2-flags-parameter
flags must be a valid combination of VkImageCreateFlagBits values

The VkImageFormatProperties2 structure is defined as:

```
// Provided by VK_VERSION_1_1
typedef struct VkImageFormatProperties2 {
    VkStructureType          sType;
    void*                    pNext;
    VkImageFormatProperties   imageFormatProperties;
} VkImageFormatProperties2;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure. The **pNext** chain of VkImageFormatProperties2 is used to allow the specification of additional capabilities to be returned from vkGetPhysicalDeviceImageFormatProperties2.
- **imageFormatProperties** is a VkImageFormatProperties structure in which capabilities are returned.

If the combination of parameters to vkGetPhysicalDeviceImageFormatProperties2 is not supported by the implementation for use in vkCreateImage, then all members of imageFormatProperties will be filled with zero.

---

**Note**

Filling imageFormatProperties with zero for unsupported formats is an exception to the usual rule that output structures have undefined contents on error. This exception was unintentional, but is preserved for backwards compatibility. This exception only applies to imageFormatProperties, not sType, pNext, or any structures chained from pNext.

---

**Valid Usage (Implicit)**

- VUID-VkImageFormatProperties2-sType-sType
  sType must be VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2

- VUID-VkImageFormatProperties2-pNext-pNext
Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkExternalImageFormatProperties` or `VkSamplerYcbcrConversionImageFormatProperties`

- VUID-VkImageFormatProperties2-sType-unique
  The `sType` value of each struct in the `pNext` chain must be unique

To determine the image capabilities compatible with an external memory handle type, add a `VkPhysicalDeviceExternalImageFormatInfo` structure to the `pNext` chain of the `VkPhysicalDeviceImageFormatInfo2` structure and a `VkExternalImageFormatProperties` structure to the `pNext` chain of the `VkImageFormatProperties2` structure.

The `VkPhysicalDeviceExternalImageFormatInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceExternalImageFormatInfo {
    VkStructureType         sType;
    const void*             pNext;
    VkExternalMemoryHandleTypeFlagBits handleType;
} VkPhysicalDeviceExternalImageFormatInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `handleType` is a `VkExternalMemoryHandleTypeFlagBits` value specifying the memory handle type that will be used with the memory associated with the image.

If `handleType` is 0, `vkGetPhysicalDeviceImageFormatProperties2` will behave as if `VkPhysicalDeviceExternalImageFormatInfo` was not present, and `VkExternalImageFormatProperties` will be ignored.

If `handleType` is not compatible with the `format`, `type`, `tiling`, `usage`, and `flags` specified in `VkPhysicalDeviceImageFormatInfo2`, then `vkGetPhysicalDeviceImageFormatProperties2` returns `VK_ERROR_FORMAT_NOT_SUPPORTED`.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceExternalImageFormatInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_IMAGE_FORMAT_INFO`

- VUID-VkPhysicalDeviceExternalImageFormatInfo-handleType-parameter
  If `handleType` is not 0, `handleType` must be a valid `VkExternalMemoryHandleTypeFlagBits` value

Possible values of `VkPhysicalDeviceExternalImageFormatInfo::handleType`, specifying an external memory handle type, are:

```c
// Provided by VK_VERSION_1_1
```
typedef enum VkExternalMemoryHandleTypeFlagBits {
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT = 0x00000001,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT = 0x00000002,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT = 0x00000004,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT = 0x00000008,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT = 0x00000010,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT = 0x00000020,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT = 0x00000040,
} VkExternalMemoryHandleTypeFlagBits;

• **VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT** specifies a POSIX file descriptor handle that has only limited valid usage outside of Vulkan and other compatible APIs. It **must** be compatible with the POSIX system calls **dup**, **dup2**, **close**, and the non-standard system call **dup3**. Additionally, it **must** be transportable over a socket using an **SCM_RIGHTS** control message. It owns a reference to the underlying memory resource represented by its Vulkan memory object.

• **VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT** specifies an NT handle that has only limited valid usage outside of Vulkan and other compatible APIs. It **must** be compatible with the functions **DuplicateHandle**, **CloseHandle**, **CompareObjectHandles**, **GetHandleInformation**, and **SetHandleInformation**. It owns a reference to the underlying memory resource represented by its Vulkan memory object.

• **VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT** specifies a global share handle that has only limited valid usage outside of Vulkan and other compatible APIs. It is not compatible with any native APIs. It does not own a reference to the underlying memory resource represented by its Vulkan memory object, and will therefore become invalid when all Vulkan memory objects associated with it are destroyed.

• **VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT** specifies an NT handle returned by **IDXGIResource1::CreateSharedHandle** referring to a Direct3D 10 or 11 texture resource. It owns a reference to the memory used by the Direct3D resource.

• **VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT** specifies a global share handle returned by **IDXGIResource::GetSharedHandle** referring to a Direct3D 10 or 11 texture resource. It does not own a reference to the underlying Direct3D resource, and will therefore become invalid when all Vulkan memory objects and Direct3D resources associated with it are destroyed.

• **VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT** specifies an NT handle returned by **ID3D12Device::CreateSharedHandle** referring to a Direct3D 12 heap resource. It owns a reference to the resources used by the Direct3D heap.

• **VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT** specifies an NT handle returned by **ID3D12Device::CreateSharedHandle** referring to a Direct3D 12 committed resource. It owns a reference to the memory used by the Direct3D resource.
Some external memory handle types can only be shared within the same underlying physical device and/or the same driver version, as defined in the following table:

**Table 57. External memory handle types compatibility**

<table>
<thead>
<tr>
<th>Handle type</th>
<th>VkPhysicalDeviceIDProperties::driverUUID</th>
<th>VkPhysicalDeviceIDProperties::deviceUUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
</tbody>
</table>

```c
// Provided by VK_VERSION_1_1
typedef VkFlags VkExternalMemoryHandleTypeFlags;
```

**VkExternalMemoryHandleTypeFlags** is a bitmask type for setting a mask of zero or more **VkExternalMemoryHandleTypeFlagBits**.

The **VkExternalImageFormatProperties** structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkExternalImageFormatProperties {
    VkStructureType sType;
    void* pNext;
    VkExternalMemoryProperties externalMemoryProperties;
} VkExternalImageFormatProperties;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **externalMemoryProperties** is a **VkExternalMemoryProperties** structure specifying various capabilities of the external handle type when used with the specified image creation parameters.

**Valid Usage (Implicit)**

- VUID-VkExternalImageFormatProperties-sType-sType
The `VkExternalMemoryProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkExternalMemoryProperties {
    VkExternalMemoryFeatureFlags externalMemoryFeatures;
    VkExternalMemoryHandleTypeFlags exportFromImportedHandleTypes;
    VkExternalMemoryHandleTypeFlags compatibleHandleTypes;
} VkExternalMemoryProperties;
```

- **externalMemoryFeatures** is a bitmask of `VkExternalMemoryFeatureFlagBits` specifying the features of `handleType`.
- **exportFromImportedHandleTypes** is a bitmask of `VkExternalMemoryHandleTypeFlagBits` specifying which types of imported handle `handleType` can be exported from.
- **compatibleHandleTypes** is a bitmask of `VkExternalMemoryHandleTypeFlagBits` specifying handle types which can be specified at the same time as `handleType` when creating an image compatible with external memory.

`compatibleHandleTypes` **must** include at least `handleType`. Inclusion of a handle type in `compatibleHandleTypes` does not imply the values returned in `VkImageFormatProperties2` will be the same when `VkPhysicalDeviceExternalImageFormatInfo::handleType` is set to that type. The application is responsible for querying the capabilities of all handle types intended for concurrent use in a single image and intersecting them to obtain the compatible set of capabilities.

Bits which **may** be set in `VkExternalMemoryProperties::externalMemoryFeatures`, specifying features of an external memory handle type, are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkExternalMemoryFeatureFlagBits {
    VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT = 0x00000001,
    VK_EXTERNAL_MEMORY_FEATURE_EXPORTABLE_BIT = 0x00000002,
    VK_EXTERNAL_MEMORY_FEATURE_IMPORTABLE_BIT = 0x00000004,
} VkExternalMemoryFeatureFlagBits;
```

- **VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT** specifies that images or buffers created with the specified parameters and handle type **must** use the mechanisms defined by `VkMemoryDedicatedRequirements` and `VkMemoryDedicatedAllocateInfo` to create (or import) a dedicated allocation for the image or buffer.
- **VK_EXTERNAL_MEMORY_FEATURE_EXPORTABLE_BIT** specifies that handles of this type **can** be exported from Vulkan memory objects.
- **VK_EXTERNAL_MEMORY_FEATURE_IMPORTABLE_BIT** specifies that handles of this type **can** be imported as Vulkan memory objects.

Because their semantics in external APIs roughly align with that of an image or buffer with a
dedicated allocation in Vulkan, implementations are required to report 
VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT for the following external handle types:

- VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT
- VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT
- VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT

// Provided by VK_VERSION_1_1
typedef VkFlags VkExternalMemoryFeatureFlags;

VkExternalMemoryFeatureFlags is a bitmask type for setting a mask of zero or more 
VkExternalMemoryFeatureFlagBits.

To determine the number of combined image samplers required to support a multi-planar format, 
add VkSamplerYcbcrConversionImageFormatProperties to the pNext chain of the 

The VkSamplerYcbcrConversionImageFormatProperties structure is defined as:

// Provided by VK_VERSION_1_1
typedef struct VkSamplerYcbcrConversionImageFormatProperties {
    VkStructureType sType;
    void* pNext;
    uint32_t combinedImageSamplerDescriptorCount;
} VkSamplerYcbcrConversionImageFormatProperties;

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- combinedImageSamplerDescriptorCount is the number of combined image sampler descriptors that 
  the implementation uses to access the format.

Valid Usage (Implicit)

- VUID-VkSamplerYcbcrConversionImageFormatProperties-sType-sType
  sType must be VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_IMAGE_FORMAT_PROPERTIES

combinedImageSamplerDescriptorCount is a number between 1 and the number of planes in the 
format. A descriptor set layout binding with immutable YCbCr conversion samplers will have a maximum 
combinedImageSamplerDescriptorCount which is the maximum across all formats supported by its samplers of the 
combinedImageSamplerDescriptorCount for each format. Descriptor sets with that layout will internally use that maximum combinedImageSamplerDescriptorCount 
descriptors for each descriptor in the binding. This expanded number of descriptors will be 
consumed from the descriptor pool when a descriptor set is allocated, and counts towards the 
maxDescriptorSetSamplers, maxDescriptorSetSampledImages, maxPerStageDescriptorSamplers, and 
maxPerStageDescriptorSampledImages limits.
Note
All descriptors in a binding use the same maximum `combinedImageSamplerDescriptorCount` descriptors to allow implementations to use a uniform stride for dynamic indexing of the descriptors in the binding.

For example, consider a descriptor set layout binding with two descriptors and immutable samplers for multi-planar formats that have `VkSamplerYcbcrConversionImageFormatProperties::combinedImageSamplerDescriptorCount` values of 2 and 3 respectively. There are two descriptors in the binding and the maximum `combinedImageSamplerDescriptorCount` is 3, so descriptor sets with this layout consume 6 descriptors from the descriptor pool. To create a descriptor pool that allows allocating four descriptor sets with this layout, `descriptorCount` must be at least 24.

35.1.1. Supported Sample Counts

`vkGetPhysicalDeviceImageFormatProperties` returns a bitmask of `VkSampleCountFlagBits` in `sampleCounts` specifying the supported sample counts for the image parameters.

`sampleCounts` will be set to `VK_SAMPLE_COUNT_1_BIT` if at least one of the following conditions is true:

- `tiling` is `VK_IMAGE_TILING_LINEAR`
- `type` is not `VK_IMAGE_TYPE_2D`
- `flags` contains `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`
- Neither the `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT` flag nor the `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT` flag in `VkFormatProperties::optimalTilingFeatures` returned by `vkGetPhysicalDeviceFormatProperties` is set
- `VkPhysicalDeviceExternalImageFormatInfo::handleType` is an external handle type for which multisampled image support is not required.
- `format` is one of the formats that require a sampler Y’CbCr conversion

Otherwise, the bits set in `sampleCounts` will be the sample counts supported for the specified values of `usage` and `format`. For each bit set in `usage`, the supported sample counts relate to the limits in `VkPhysicalDeviceLimits` as follows:

- If `usage` includes `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` and `format` is a floating- or fixed-point color format, a superset of `VkPhysicalDeviceLimits::framebufferColorSampleCounts`
- If `usage` includes `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` and `format` is an integer format, a superset of `VkPhysicalDeviceVulkan12Properties::framebufferIntegerColorSampleCounts`
- If `usage` includes `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, and `format` includes a depth aspect, a superset of `VkPhysicalDeviceLimits::framebufferDepthSampleCounts`
- If `usage` includes `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, and `format` includes a stencil aspect, a superset of `VkPhysicalDeviceLimits::framebufferStencilSampleCounts`
- If `usage` includes `VK_IMAGE_USAGE_SAMPLED_BIT`, and `format` includes a color aspect, a superset of `VkPhysicalDeviceLimits::sampledImageColorSampleCounts`
If usage includes `VK_IMAGE_USAGE_SAMPLED_BIT`, and format includes a depth aspect, a superset of `VkPhysicalDeviceLimits::sampledImageDepthSampleCounts`

If usage includes `VK_IMAGE_USAGE_SAMPLED_BIT`, and format is an integer format, a superset of `VkPhysicalDeviceLimits::sampledImageIntegerSampleCounts`

If usage includes `VK_IMAGE_USAGE_STORAGE_BIT`, a superset of `VkPhysicalDeviceLimits::storageImageSampleCounts`

If multiple bits are set in usage, `sampleCounts` will be the intersection of the per-usage values described above.

If none of the bits described above are set in usage, then there is no corresponding limit in `VkPhysicalDeviceLimits`. In this case, `sampleCounts` must include at least `VK_SAMPLE_COUNT_1_BIT`.

### 35.1.2. Allowed Extent Values Based On Image Type

Implementations may support extent values larger than the required minimum/maximum values for certain types of images. `VkImageFormatProperties::maxExtent` for each type is subject to the constraints below.

**Note**

Implementations must support images with dimensions up to the required minimum/maximum values for all types of images. It follows that the query for additional capabilities must return extent values that are at least as large as the required values.

For `VK_IMAGE_TYPE_1D`:

- `maxExtent.width \geq VkPhysicalDeviceLimits::maxImageDimension1D`
- `maxExtent.height = 1`
- `maxExtent.depth = 1`

For `VK_IMAGE_TYPE_2D` when flags does not contain `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`:

- `maxExtent.width \geq VkPhysicalDeviceLimits::maxImageDimension2D`
- `maxExtent.height \geq VkPhysicalDeviceLimits::maxImageDimension2D`
- `maxExtent.depth = 1`

For `VK_IMAGE_TYPE_2D` when flags contains `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`:

- `maxExtent.width \geq VkPhysicalDeviceLimits::maxImageDimensionCube`
- `maxExtent.height \geq VkPhysicalDeviceLimits::maxImageDimensionCube`
- `maxExtent.depth = 1`

For `VK_IMAGE_TYPE_3D`:

- `maxExtent.width \geq VkPhysicalDeviceLimits::maxImageDimension3D`
35.2. Additional Buffer Capabilities

To query the external handle types supported by buffers, call:

```c
// Provided by VK_VERSION_1_1
void vkGetPhysicalDeviceExternalBufferProperties(
    VkPhysicalDevice physicalDevice,
    const VkPhysicalDeviceExternalBufferInfo* pExternalBufferInfo,
    VkExternalBufferProperties* pExternalBufferProperties);
```

- `physicalDevice` is the physical device from which to query the buffer capabilities.
- `pExternalBufferInfo` is a pointer to a `VkPhysicalDeviceExternalBufferInfo` structure describing the parameters that would be consumed by `vkCreateBuffer`.
- `pExternalBufferProperties` is a pointer to a `VkExternalBufferProperties` structure in which capabilities are returned.

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceExternalBufferProperties-physicalDevice-parameter
  - `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceExternalBufferProperties-pExternalBufferInfo-parameter
  - `pExternalBufferInfo` must be a valid pointer to a valid `VkPhysicalDeviceExternalBufferInfo` structure
- VUID-vkGetPhysicalDeviceExternalBufferProperties-pExternalBufferProperties-parameter
  - `pExternalBufferProperties` must be a valid pointer to a `VkExternalBufferProperties` structure

The `VkPhysicalDeviceExternalBufferInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceExternalBufferInfo {  
    VkStructureType sType;
    const void*pNext;
    VkBufferCreateFlags flags;
    VkBufferUsageFlags usage;
    VkExternalMemoryHandleTypeFlagBits handleType;
} VkPhysicalDeviceExternalBufferInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is a bitmask of `VkBufferCreateFlagBits` describing additional parameters of the buffer, corresponding to `VkBufferCreateInfo::flags`.
- `usage` is a bitmask of `VkBufferUsageFlagBits` describing the intended usage of the buffer, corresponding to `VkBufferCreateInfo::usage`.
- `handleType` is a `VkExternalMemoryHandleTypeFlagBits` value specifying the memory handle type that will be used with the memory associated with the buffer.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_BUFFER_INFO`
- `pNext` must be `NULL`
- `flags` must be a valid combination of `VkBufferCreateFlagBits` values
- `usage` must be a valid combination of `VkBufferUsageFlagBits` values
- `usage` must not be `0`
- `handleType` must be a valid `VkExternalMemoryHandleTypeFlagBits` value

The `VkExternalBufferProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkExternalBufferProperties {
    VkStructureType sType;
    void* pNext;
    VkExternalMemoryProperties externalMemoryProperties;
} VkExternalBufferProperties;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `externalMemoryProperties` is a `VkExternalMemoryProperties` structure specifying various capabilities of the external handle type when used with the specified buffer creation parameters.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_EXTERNAL_BUFFER_PROPERTIES`
35.3. Optional Semaphore Capabilities

Semaphores may support import and export of their payload to external handles. To query the external handle types supported by semaphores, call:

```c
// Provided by VK_VERSION_1_1
void vkGetPhysicalDeviceExternalSemaphoreProperties(
    VkPhysicalDevice physicalDevice,
    const VkPhysicalDeviceExternalSemaphoreInfo* pExternalSemaphoreInfo,
    VkExternalSemaphoreProperties* pExternalSemaphoreProperties);
```

- `physicalDevice` is the physical device from which to query the semaphore capabilities.
- `pExternalSemaphoreInfo` is a pointer to a `VkPhysicalDeviceExternalSemaphoreInfo` structure describing the parameters that would be consumed by `vkCreateSemaphore`.
- `pExternalSemaphoreProperties` is a pointer to a `VkExternalSemaphoreProperties` structure in which capabilities are returned.

Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceExternalSemaphoreProperties-physicalDevice-parameter `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceExternalSemaphoreProperties-pExternalSemaphoreInfo-parameter `pExternalSemaphoreInfo` must be a valid pointer to a valid `VkPhysicalDeviceExternalSemaphoreInfo` structure
- VUID-vkGetPhysicalDeviceExternalSemaphoreProperties-pExternalSemaphoreProperties-parameter `pExternalSemaphoreProperties` must be a valid pointer to a `VkExternalSemaphoreProperties` structure

The `VkPhysicalDeviceExternalSemaphoreInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceExternalSemaphoreInfo {
    VkStructureType sType;
    const void* pNext;
    VkExternalSemaphoreHandleTypeFlagBits handleType;
} VkPhysicalDeviceExternalSemaphoreInfo;
```

- `sType` is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.

• handleType is a VkExternalSemaphoreHandleTypeFlagBits value specifying the external semaphore handle type for which capabilities will be returned.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceExternalSemaphoreInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SEMAPHORE_INFO

- VUID-VkPhysicalDeviceExternalSemaphoreInfo-pNext-pNext
  pNext must be NULL or a pointer to a valid instance of VkSemaphoreTypeCreateInfo

- VUID-VkPhysicalDeviceExternalSemaphoreInfo-sType-unique
  The sType value of each struct in the pNext chain must be unique

- VUID-VkPhysicalDeviceExternalSemaphoreInfo-handleType-parameter
  handleType must be a valid VkExternalSemaphoreHandleTypeFlagBits value

Bits which may be set in VkPhysicalDeviceExternalSemaphoreInfo::handleType, specifying an external semaphore handle type, are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkExternalSemaphoreHandleTypeFlagBits {
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT = 0x00000001,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT = 0x00000002,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT = 0x00000004,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT = 0x00000008,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT = 0x00000010,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_FENCE_BIT
} VkExternalSemaphoreHandleTypeFlagBits;
```

- VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT specifies a POSIX file descriptor handle that has only limited valid usage outside of Vulkan and other compatible APIs. It must be compatible with the POSIX system calls dup, dup2, close, and the non-standard system call dup3. Additionally, it must be transportable over a socket using an SCM_RIGHTS control message. It owns a reference to the underlying synchronization primitive represented by its Vulkan semaphore object.

- VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT specifies an NT handle that has only limited valid usage outside of Vulkan and other compatible APIs. It must be compatible with the functions DuplicateHandle, CloseHandle, CompareObjectHandles, GetHandleInformation, and SetHandleInformation. It owns a reference to the underlying synchronization primitive represented by its Vulkan semaphore object.

- VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT specifies a global share handle that has only limited valid usage outside of Vulkan and other compatible APIs. It is not compatible with any native APIs. It does not own a reference to the underlying synchronization primitive represented by its Vulkan semaphore object, and will therefore become invalid when all Vulkan semaphore objects associated with it are destroyed.
• `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT` specifies an NT handle returned by `ID3D12Device::CreateSharedHandle` referring to a Direct3D 12 fence, or `ID3D11Device5::CreateFence` referring to a Direct3D 11 fence. It owns a reference to the underlying synchronization primitive associated with the Direct3D fence.

• `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_FENCE_BIT` is an alias of `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT` with the same meaning. It is provided for convenience and code clarity when interacting with D3D11 fences.

• `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT` specifies a POSIX file descriptor handle to a Linux Sync File or Android Fence object. It can be used with any native API accepting a valid sync file or fence as input. It owns a reference to the underlying synchronization primitive associated with the file descriptor. Implementations which support importing this handle type must accept any type of sync or fence FD supported by the native system they are running on.

**Note**

Handles of type `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT` generated by the implementation may represent either Linux Sync Files or Android Fences at the implementation's discretion. Applications should only use operations defined for both types of file descriptors, unless they know via means external to Vulkan the type of the file descriptor, or are prepared to deal with the system-defined operation failures resulting from using the wrong type.
Some external semaphore handle types can only be shared within the same underlying physical device and/or the same driver version, as defined in the following table:

<table>
<thead>
<tr>
<th>Handle type</th>
<th>VkPhysicalDeviceIDProperties::driverUUID</th>
<th>VkPhysicalDeviceIDProperties::deviceUUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_ZIRCON_EVENT_BIT_FUCHSIA</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
</tbody>
</table>

// Provided by VK_VERSION_1_1

typedef VkFlags VkExternalSemaphoreHandleTypeFlags;

VkExternalSemaphoreHandleTypeFlags is a bitmask type for setting a mask of zero or more VkExternalSemaphoreHandleTypeFlagBits.

The VkExternalSemaphoreProperties structure is defined as:

// Provided by VK_VERSION_1_1

typedef struct VkExternalSemaphoreProperties {
    VkStructureType sType;
    void* pNext;
    VkExternalSemaphoreHandleTypeFlags exportFromImportedHandleTypes;
    VkExternalSemaphoreHandleTypeFlags compatibleHandleTypes;
    VkExternalSemaphoreFeatureFlags externalSemaphoreFeatures;
} VkExternalSemaphoreProperties;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **exportFromImportedHandleTypes** is a bitmask of VkExternalSemaphoreHandleTypeFlagBits specifying which types of imported handle handleType can be exported from.
- **compatibleHandleTypes** is a bitmask of VkExternalSemaphoreHandleTypeFlagBits specifying handle types which can be specified at the same time as handleType when creating a semaphore.
- **externalSemaphoreFeatures** is a bitmask of VkExternalSemaphoreFeatureFlagBits describing the features of handleType.
If `handleType` is not supported by the implementation, then `VkExternalSemaphoreProperties::externalSemaphoreFeatures` will be set to zero.

**Valid Usage (Implicit)**

- VUID-VkExternalSemaphoreProperties-sType-sType
  - `sType` must be `VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES`
- VUID-VkExternalSemaphoreProperties-pNext-pNext
  - `pNext` must be `NULL`

Bits which *may* be set in `VkExternalSemaphoreProperties::externalSemaphoreFeatures`, specifying the features of an external semaphore handle type, are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkExternalSemaphoreFeatureFlagBits {
    VK_EXTERNAL_SEMAPHORE_FEATURE_EXPORTABLE_BIT = 0x00000001,
    VK_EXTERNAL_SEMAPHORE_FEATURE_IMPORTABLE_BIT = 0x00000002,
} VkExternalSemaphoreFeatureFlagBits;
```

- `VK_EXTERNAL_SEMAPHORE_FEATURE_EXPORTABLE_BIT` specifies that handles of this type *can* be exported from Vulkan semaphore objects.
- `VK_EXTERNAL_SEMAPHORE_FEATURE_IMPORTABLE_BIT` specifies that handles of this type *can* be imported as Vulkan semaphore objects.

```c
// Provided by VK_VERSION_1_1
typedef VkFlags VkExternalSemaphoreFeatureFlags;
```

`VkExternalSemaphoreFeatureFlags` is a bitmask type for setting a mask of zero or more `VkExternalSemaphoreFeatureFlagBits`.

### 35.4. Optional Fence Capabilities

Fences *may* support import and export of their payload to external handles. To query the external handle types supported by fences, call:

```c
// Provided by VK_VERSION_1_1
void vkGetPhysicalDeviceExternalFenceProperties(
    VkPhysicalDevice physicalDevice,
    const VkPhysicalDeviceExternalFenceInfo* pExternalFenceInfo,
    VkExternalFenceProperties* pExternalFenceProperties);
```

- `physicalDevice` is the physical device from which to query the fence capabilities.
- `pExternalFenceInfo` is a pointer to a `VkPhysicalDeviceExternalFenceInfo` structure describing the
parameters that would be consumed by `vkCreateFence`.

- **pExternalFenceProperties** is a pointer to a `VkExternalFenceProperties` structure in which capabilities are returned.

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceExternalFenceProperties-physicalDevice-parameter
  
  `physicalDevice` **must** be a valid `VkPhysicalDevice` handle

- VUID-vkGetPhysicalDeviceExternalFenceProperties-pExternalFenceInfo-parameter
  
  `pExternalFenceInfo` **must** be a valid pointer to a valid `VkPhysicalDeviceExternalFenceInfo` structure

- VUID-vkGetPhysicalDeviceExternalFenceProperties-pExternalFenceProperties-parameter
  
  `pExternalFenceProperties` **must** be a valid pointer to a `VkExternalFenceProperties` structure

The `VkPhysicalDeviceExternalFenceInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceExternalFenceInfo {
    VkStructureType sType;
    const void* pNext;
    VkExternalFenceHandleTypeFlagBits handleType;
} VkPhysicalDeviceExternalFenceInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `handleType` is a `VkExternalFenceHandleTypeFlagBits` value specifying an external fence handle type for which capabilities will be returned.

**Note**

Handles of type `VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT` generated by the implementation may represent either Linux Sync Files or Android Fences at the implementation's discretion. Applications **should** only use operations defined for both types of file descriptors, unless they know via means external to Vulkan the type of the file descriptor, or are prepared to deal with the system-defined operation failures resulting from using the wrong type.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceExternalFenceInfo-sType-sType
  
  `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_FENCE_INFO`

- VUID-VkPhysicalDeviceExternalFenceInfo-pNext-pNext
  
  `pNext` **must** be `NULL`
**handleType** must be a valid `VkExternalFenceHandleTypeFlagBits` value.

Bits which **may** be set in

- `VkPhysicalDeviceExternalFenceInfo::handleType`
- `VkExternalFenceProperties::exportFromImportedHandleTypes`
- `VkExternalFenceProperties::compatibleHandleTypes`

indicate external fence handle types, and are:

```c
typedef enum VkExternalFenceHandleTypeFlagBits {
    VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_FD_BIT = 0x00000001,
    VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT = 0x00000002,
    VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT = 0x00000004,
    VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT = 0x00000008,
} VkExternalFenceHandleTypeFlagBits;
```

- **VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_FD_BIT** specifies a POSIX file descriptor handle that has only limited valid usage outside of Vulkan and other compatible APIs. It **must** be compatible with the POSIX system calls `dup`, `dup2`, `close`, and the non-standard system call `dup3`. Additionally, it **must** be transportable over a socket using an `SCM_RIGHTS` control message. It owns a reference to the underlying synchronization primitive represented by its Vulkan fence object.

- **VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT** specifies an NT handle that has only limited valid usage outside of Vulkan and other compatible APIs. It **must** be compatible with the functions `DuplicateHandle`, `CloseHandle`, `CompareObjectHandles`, `GetHandleInformation`, and `SetHandleInformation`. It owns a reference to the underlying synchronization primitive represented by its Vulkan fence object.

- **VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT** specifies a global share handle that has only limited valid usage outside of Vulkan and other compatible APIs. It is not compatible with any native APIs. It does not own a reference to the underlying synchronization primitive represented by its Vulkan fence object, and will therefore become invalid when all Vulkan fence objects associated with it are destroyed.

- **VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT** specifies a POSIX file descriptor handle to a Linux Sync File or Android Fence. It can be used with any native API accepting a valid sync file or fence as input. It owns a reference to the underlying synchronization primitive associated with the file descriptor. Implementations which support importing this handle type **must** accept any type of sync or fence FD supported by the native system they are running on.
Some external fence handle types can only be shared within the same underlying physical device and/or the same driver version, as defined in the following table:

Table 59. External fence handle types compatibility

<table>
<thead>
<tr>
<th>Handle type</th>
<th>VkPhysicalDeviceIDProperties::driverUUID</th>
<th>VkPhysicalDeviceIDProperties::deviceUUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_FD_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BITMAP</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
</tbody>
</table>

// Provided by VK_VERSION_1_1
typedef VkFlags VkExternalFenceHandleTypeFlags;

VkExternalFenceHandleTypeFlags is a bitmask type for setting a mask of zero or more VkExternalFenceHandleTypeFlagBits.

The VkExternalFenceProperties structure is defined as:

// Provided by VK_VERSION_1_1
typedef struct VkExternalFenceProperties {
    VkStructureType sType;
    void* pNext;
    VkExternalFenceHandleTypeFlags exportFromImportedHandleTypes;
    VkExternalFenceHandleTypeFlags compatibleHandleTypes;
    VkExternalFenceFeatureFlags externalFenceFeatures;
} VkExternalFenceProperties;

- exportFromImportedHandleTypes is a bitmask of VkExternalFenceHandleTypeFlagBits indicating which types of imported handle handleType can be exported from.

- compatibleHandleTypes is a bitmask of VkExternalFenceHandleTypeFlagBits specifying handle types which can be specified at the same time as handleType when creating a fence.

- externalFenceFeatures is a bitmask of VkExternalFenceFeatureFlagBits indicating the features of handleType.

If handleType is not supported by the implementation, then VkExternalFenceProperties::externalFenceFeatures will be set to zero.

Valid Usage (Implicit)

- VUID-VkExternalFenceProperties-sType-sType
  sType must be VK_STRUCTURE_TYPE_EXTERNAL_FENCE_PROPERTIES
Bits which may be set in `VkExternalFenceProperties::externalFenceFeatures`, indicating features of a fence external handle type, are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkExternalFenceFeatureFlagBits {
    VK_EXTERNAL_FENCE_FEATURE_EXPORTABLE_BIT = 0x00000001,
    VK_EXTERNAL_FENCE_FEATURE_IMPORTABLE_BIT = 0x00000002,
} VkExternalFenceFeatureFlagBits;
```

- `VK_EXTERNAL_FENCE_FEATURE_EXPORTABLE_BIT` specifies handles of this type can be exported from Vulkan fence objects.
- `VK_EXTERNAL_FENCE_FEATURE_IMPORTABLE_BIT` specifies handles of this type can be imported to Vulkan fence objects.

```c
// Provided by VK_VERSION_1_1
typedef VkFlags VkExternalFenceFeatureFlags;
```

`VkExternalFenceFeatureFlags` is a bitmask type for setting a mask of zero or more `VkExternalFenceFeatureFlagBits`. 
Chapter 36. Debugging

To aid developers in tracking down errors in the application’s use of Vulkan, particularly in combination with an external debugger or profiler, debugging extensions may be available.

The VkObjectType enumeration defines values, each of which corresponds to a specific Vulkan handle type. These values can be used to associate debug information with a particular type of object through one or more extensions.

```c
// Provided by VK_VERSION_1_0
typedef enum VkObjectType {
    VK_OBJECT_TYPE_UNKNOWN = 0,
    VK_OBJECT_TYPE_INSTANCE = 1,
    VK_OBJECT_TYPE_PHYSICAL_DEVICE = 2,
    VK_OBJECT_TYPE_DEVICE = 3,
    VK_OBJECT_TYPE_QUEUE = 4,
    VK_OBJECT_TYPE_SEMAPHORE = 5,
    VK_OBJECT_TYPE_COMMAND_BUFFER = 6,
    VK_OBJECT_TYPE_FENCE = 7,
    VK_OBJECT_TYPE_DEVICE_MEMORY = 8,
    VK_OBJECT_TYPE_BUFFER = 9,
    VK_OBJECT_TYPE_IMAGE = 10,
    VK_OBJECT_TYPE_EVENT = 11,
    VK_OBJECT_TYPE_QUERY_POOL = 12,
    VK_OBJECT_TYPE_BUFFER_VIEW = 13,
    VK_OBJECT_TYPE_IMAGE_VIEW = 14,
    VK_OBJECT_TYPE_SHADER_MODULE = 15,
    VK_OBJECT_TYPE_PIPELINE_CACHE = 16,
    VK_OBJECT_TYPE_PIPELINE_LAYOUT = 17,
    VK_OBJECT_TYPE_RENDER_PASS = 18,
    VK_OBJECT_TYPE_PIPELINE = 19,
    VK_OBJECT_TYPE_DESCRIPTOR_SET_LAYOUT = 20,
    VK_OBJECT_TYPE_SAMPLER = 21,
    VK_OBJECT_TYPE_DESCRIPTOR_POOL = 22,
    VK_OBJECT_TYPE_DESCRIPTOR_SET = 23,
    VK_OBJECT_TYPE_FRAMEBUFFER = 24,
    VK_OBJECT_TYPE_COMMAND_POOL = 25,

    // Provided by VK_VERSION_1_1
    VK_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION = 1000156000,

    // Provided by VK_VERSION_1_1
    VK_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE = 1000085000,

    // Provided by VK_VERSION_1_3
    VK_OBJECT_TYPE_PRIVATE_DATA_SLOT = 1000295000,
} VkObjectType;
```

<table>
<thead>
<tr>
<th>VkObjectType</th>
<th>Vulkan Handle Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_OBJECT_TYPE_UNKNOWN</td>
<td>Unknown/Undefined Handle</td>
</tr>
</tbody>
</table>

Table 60. VkObjectType and Vulkan Handle Relationship
<table>
<thead>
<tr>
<th>VkObjectType</th>
<th>Vulkan Handle Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_OBJECT_TYPE_INSTANCE</td>
<td>VkInstance</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_PHYSICAL_DEVICE</td>
<td>VkPhysicalDevice</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DEVICE</td>
<td>VkDevice</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_QUEUE</td>
<td>VkQueue</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_SEMAPHORE</td>
<td>VkSemaphore</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_COMMAND_BUFFER</td>
<td>VkSemaphore</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_fence</td>
<td>VkFence</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DEVICE_MEMORY</td>
<td>VkDeviceMemory</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_BUFFER</td>
<td>VkBuffer</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_IMAGE</td>
<td>VkImage</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_EVENT</td>
<td>VkEvent</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_QUERY_POOL</td>
<td>VkQueryPool</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_BUFFER_VIEW</td>
<td>VkBufferView</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_IMAGE_VIEW</td>
<td>VkImageView</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_SHADER_MODULE</td>
<td>VkShaderModule</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_PIPELINE_CACHE</td>
<td>VkPipelineCache</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_PIPELINE_LAYOUT</td>
<td>VkPipelineLayout</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_RENDER_PASS</td>
<td>VkRenderPass</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_PIPELINE</td>
<td>VkPipeline</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DESCRIPTOR_SET_LAYOUT</td>
<td>VkDescriptorSetLayout</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_SAMPLER</td>
<td>VkSampler</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DESCRIPTOR_POOL</td>
<td>VkDescriptorPool</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DESCRIPTOR_SET</td>
<td>VkDescriptorSet</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_FRAMEBUFFER</td>
<td>VkFramebuffer</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_COMMAND_POOL</td>
<td>VkCommandPool</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION</td>
<td>VkSamplerYcbcrConversion</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE</td>
<td>VkDescriptorUpdateTemplate</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_PRIVATE_DATA_SLOT</td>
<td>VkPrivateDataSlot</td>
</tr>
</tbody>
</table>

If this Specification was generated with any such extensions included, they will be described in the remainder of this chapter.

### 36.1. Active Tooling Information

Information about tools providing debugging, profiling, or similar services, active for a given
physical device, can be obtained by calling:

```c
// Provided by VK_VERSION_1_3
VkResult vkGetPhysicalDeviceToolProperties(
    VkPhysicalDevice physicalDevice,
    uint32_t* pToolCount,
    VkPhysicalDeviceToolProperties* pToolProperties);
```

- `physicalDevice` is the handle to the physical device to query for active tools.
- `pToolCount` is a pointer to an integer describing the number of tools active on `physicalDevice`.
- `pToolProperties` is either `NULL` or a pointer to an array of `VkPhysicalDeviceToolProperties` structures.

If `pToolProperties` is `NULL`, then the number of tools currently active on `physicalDevice` is returned in `pToolCount`. Otherwise, `pToolCount` must point to a variable set by the user to the number of elements in the `pToolProperties` array, and on return the variable is overwritten with the number of structures actually written to `pToolProperties`. If `pToolCount` is less than the number of currently active tools, at most `pToolCount` structures will be written.

The count and properties of active tools may change in response to events outside the scope of the specification. An application should assume these properties might change at any given time.

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceToolProperties-physicalDevice-parameter `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceToolProperties-pToolCount-parameter `pToolCount` must be a valid pointer to a `uint32_t` value
- VUID-vkGetPhysicalDeviceToolProperties-pToolProperties-parameter If the value referenced by `pToolCount` is not 0, and `pToolProperties` is not `NULL`, `pToolProperties` must be a valid pointer to an array of `pToolCount` `VkPhysicalDeviceToolProperties` structures

### Return Codes

#### Success
- `VK_SUCCESS`
- `VK_INCOMPLETE`

#### Failure
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkPhysicalDeviceToolProperties` structure is defined as:
typedef struct VkPhysicalDeviceToolProperties {
    VkStructureType sType;
    void* pNext;
    char* name[VK_MAX_EXTENSION_NAME_SIZE];
    char* version[VK_MAX_EXTENSION_NAME_SIZE];
    VkToolPurposeFlags purposes;
    char* description[VK_MAX_DESCRIPTION_SIZE];
    char* layer[VK_MAX_EXTENSION_NAME_SIZE];
} VkPhysicalDeviceToolProperties;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **name** is a null-terminated UTF-8 string containing the name of the tool.
- **version** is a null-terminated UTF-8 string containing the version of the tool.
- **purposes** is a bitmask of VkToolPurposeFlagBits which is populated with purposes supported by the tool.
- **description** is a null-terminated UTF-8 string containing a description of the tool.
- **layer** is a null-terminated UTF-8 string containing the name of the layer implementing the tool, if the tool is implemented in a layer - otherwise it may be an empty string.

## Valid Usage (Implicit)

- VUID-VkPhysicalDeviceToolProperties-sType-sType
  - sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TOOL_PROPERTIES
- VUID-VkPhysicalDeviceToolProperties-pNext-pNext
  - pNext must be NULL

Bits which can be set in VkPhysicalDeviceToolProperties::purposes, specifying the purposes of an active tool, are:

typedef enum VkToolPurposeFlagBits {
    VK_TOOL_PURPOSE_VALIDATION_BIT = 0x00000001,
    VK_TOOL_PURPOSE_PROFILING_BIT = 0x00000002,
    VK_TOOL_PURPOSE_TRACING_BIT = 0x00000004,
    VK_TOOL_PURPOSE_ADDITIONAL_FEATURES_BIT = 0x00000008,
    VK_TOOL_PURPOSE_MODIFYING_FEATURES_BIT = 0x00000010,
    VK_TOOL_PURPOSE_VALIDATION_BIT_EXT = VK_TOOL_PURPOSE_VALIDATION_BIT,
    VK_TOOL_PURPOSE_PROFILING_BIT_EXT = VK_TOOL_PURPOSE_PROFILING_BIT,
    VK_TOOL_PURPOSE_TRACING_BIT_EXT = VK_TOOL_PURPOSE_TRACING_BIT,
    VK_TOOL_PURPOSE_ADDITIONAL_FEATURES_BIT_EXT = VK_TOOL_PURPOSE_ADDITIONAL_FEATURES_BIT,
    VK_TOOL_PURPOSE_MODIFYING_FEATURES_BIT_EXT =
} VkToolPurposeFlagBits;
VK_TOOL_PURPOSE_MODIFYING_FEATURES_BIT,
} VkToolPurposeFlagBits;

- **VK_TOOL_PURPOSE_VALIDATION_BIT** specifies that the tool provides validation of API usage.
- **VK_TOOL_PURPOSE_PROFILING_BIT** specifies that the tool provides profiling of API usage.
- **VK_TOOL_PURPOSE_TRACING_BIT** specifies that the tool is capturing data about the application’s API usage, including anything from simple logging to capturing data for later replay.
- **VK_TOOL_PURPOSE_ADDITIONAL_FEATURES_BIT** specifies that the tool provides additional API features/extensions on top of the underlying implementation.
- **VK_TOOL_PURPOSE_MODIFYING_FEATURES_BIT** specifies that the tool modifies the API features/limits/extensions presented to the application.

// Provided by VK_VERSION_1_3
typedef VkFlags VkToolPurposeFlags;

**VkToolPurposeFlags** is a bitmask type for setting a mask of zero or more **VkToolPurposeFlagBits**.
Appendix A: Vulkan Environment for SPIR-V

Shaders for Vulkan are defined by the Khronos SPIR-V Specification as well as the Khronos SPIR-V Extended Instructions for GLSL Specification. This appendix defines additional SPIR-V requirements applying to Vulkan shaders.

Versions and Formats

A Vulkan 1.3 implementation must support the 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, and 1.6 versions of SPIR-V and the 1.0 version of the SPIR-V Extended Instructions for GLSL.

A SPIR-V module passed into vkCreateShaderModule is interpreted as a series of 32-bit words in host endianness, with literal strings packed as described in section 2.2 of the SPIR-V Specification. The first few words of the SPIR-V module must be a magic number and a SPIR-V version number, as described in section 2.3 of the SPIR-V Specification.

Capabilities

The table below lists the set of SPIR-V capabilities that may be supported in Vulkan implementations. The application must not use any of these capabilities in SPIR-V passed to vkCreateShaderModule unless one of the following conditions is met for the VkDevice specified in the device parameter of vkCreateShaderModule:

- The corresponding field in the table is blank.
- Any corresponding Vulkan feature is enabled.
- Any corresponding Vulkan extension is enabled.
- Any corresponding Vulkan property is supported.
- The corresponding core version is supported (as returned by VkPhysicalDeviceProperties::apiVersion).

<table>
<thead>
<tr>
<th>SPIR-V OpCapability</th>
<th>Vulkan feature, extension, or core version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix</td>
<td>VK_VERSION_1_0</td>
</tr>
<tr>
<td>Shader</td>
<td>VK_VERSION_1_0</td>
</tr>
<tr>
<td>InputAttachment</td>
<td>VK_VERSION_1_0</td>
</tr>
<tr>
<td>Sampled1D</td>
<td>VK_VERSION_1_0</td>
</tr>
<tr>
<td>Image1D</td>
<td>VK_VERSION_1_0</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SPIR-V OpCapability</td>
<td>Vulkan feature, extension, or core version</td>
</tr>
<tr>
<td>SampledBuffer</td>
<td>VK_VERSION_1_0</td>
</tr>
<tr>
<td>ImageBuffer</td>
<td>VK_VERSION_1_0</td>
</tr>
<tr>
<td>ImageQuery</td>
<td>VK_VERSION_1_0</td>
</tr>
<tr>
<td>DerivativeControl</td>
<td>VK_VERSION_1_0</td>
</tr>
<tr>
<td>Geometry</td>
<td>VkPhysicalDeviceFeatures::geometryShader</td>
</tr>
<tr>
<td>Tessellation</td>
<td>VkPhysicalDeviceFeatures::tessellationShader</td>
</tr>
<tr>
<td>Float64</td>
<td>VkPhysicalDeviceFeatures::shaderFloat64</td>
</tr>
<tr>
<td>Int64</td>
<td>VkPhysicalDeviceFeatures::shaderInt64</td>
</tr>
<tr>
<td>Int64Atomics</td>
<td>VkPhysicalDeviceVulkan12Features::shaderBufferInt64Atomics</td>
</tr>
<tr>
<td>Int64Atomics</td>
<td>VkPhysicalDeviceVulkan12Features::shaderSharedInt64Atomics</td>
</tr>
<tr>
<td>Int16</td>
<td>VkPhysicalDeviceFeatures::shaderInt16</td>
</tr>
<tr>
<td>TessellationPointSize</td>
<td>VkPhysicalDeviceFeatures::shaderTessellationAndGeometryPointSize</td>
</tr>
<tr>
<td>GeometryPointSize</td>
<td>VkPhysicalDeviceFeatures::shaderTessellationAndGeometryPointSize</td>
</tr>
<tr>
<td>ImageGatherExtended</td>
<td>VkPhysicalDeviceFeatures::shaderImageGatherExtended</td>
</tr>
<tr>
<td>StorageImageMultisample</td>
<td>VkPhysicalDeviceFeatures::shaderStorageImageMultisample</td>
</tr>
<tr>
<td>UniformBufferArrayDynamicIndexing</td>
<td>VkPhysicalDeviceFeatures::shaderUniformBufferArrayDynamicIndexing</td>
</tr>
<tr>
<td>SampledImageArrayDynamicIndexing</td>
<td>VkPhysicalDeviceFeatures::shaderSampledImageArrayDynamicIndexing</td>
</tr>
<tr>
<td>StorageBufferArrayDynamicIndexing</td>
<td>VkPhysicalDeviceFeatures::shaderStorageBufferArrayDynamicIndexing</td>
</tr>
<tr>
<td>StorageImageArrayDynamicIndexing</td>
<td>VkPhysicalDeviceFeatures::shaderStorageImageArrayDynamicIndexing</td>
</tr>
</tbody>
</table>

1281
<table>
<thead>
<tr>
<th>SPIR-V OpCapability</th>
<th>Vulkan feature, extension, or core version</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClipDistance</td>
<td>VkPhysicalDeviceFeatures::shaderClipDistance</td>
</tr>
<tr>
<td>CullDistance</td>
<td>VkPhysicalDeviceFeatures::shaderCullDistance</td>
</tr>
<tr>
<td>ImageCubeArray</td>
<td>VkPhysicalDeviceFeatures::imageCubeArray</td>
</tr>
<tr>
<td>SampleRateShading</td>
<td>VkPhysicalDeviceFeatures::sampleRateShading</td>
</tr>
<tr>
<td>SparseResidency</td>
<td>VkPhysicalDeviceFeatures::shaderResourceResidency</td>
</tr>
<tr>
<td>MinLod</td>
<td>VkPhysicalDeviceFeatures::shaderResourceMinLod</td>
</tr>
<tr>
<td>SampledCubeArray</td>
<td>VkPhysicalDeviceFeatures::imageCubeArray</td>
</tr>
<tr>
<td>ImageMSArray</td>
<td>VkPhysicalDeviceFeatures::shaderStorageImageMultisample</td>
</tr>
<tr>
<td>StorageImageExtendedFormats</td>
<td>VK_VERSION_1_0</td>
</tr>
<tr>
<td>InterpolationFunction</td>
<td>VkPhysicalDeviceFeatures::sampleRateShading</td>
</tr>
<tr>
<td>StorageImageReadWithoutFormat</td>
<td>VkPhysicalDeviceFeatures::shaderStorageImageReadWithoutFormat</td>
</tr>
<tr>
<td>StorageImageWriteWithoutFormat</td>
<td>VkPhysicalDeviceFeatures::shaderStorageImageWriteWithoutFormat</td>
</tr>
<tr>
<td>MultiViewport</td>
<td>VkPhysicalDeviceFeatures::multiViewport</td>
</tr>
<tr>
<td>DrawParameters</td>
<td>VkPhysicalDeviceVulkan11Features::shaderDrawParameters</td>
</tr>
<tr>
<td>DrawParameters</td>
<td>VkPhysicalDeviceShaderDrawParametersFeatures::shaderDrawParameters</td>
</tr>
<tr>
<td>MultiView</td>
<td>VkPhysicalDeviceVulkan11Features::multiview</td>
</tr>
<tr>
<td>DeviceGroup</td>
<td>VK_VERSION_1_1</td>
</tr>
<tr>
<td>VariablePointersStorageBuffer</td>
<td>VkPhysicalDeviceVulkan11Features::variablePointersStorageBuffer</td>
</tr>
<tr>
<td>VariablePointers</td>
<td>VkPhysicalDeviceVulkan11Features::variablePointers</td>
</tr>
<tr>
<td>SPIR-V OpCapability</td>
<td>Vulkan feature, extension, or core version</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>ShaderViewportIndex</td>
<td>VkPhysicalDeviceVulkan12Features::shaderOutputViewportIndex</td>
</tr>
<tr>
<td>ShaderLayer</td>
<td>VkPhysicalDeviceVulkan12Features::shaderOutputLayer</td>
</tr>
<tr>
<td>StorageBuffer16BitAccess</td>
<td>VkPhysicalDeviceVulkan11Features::storageBuffer16BitAccess</td>
</tr>
<tr>
<td>UniformAndStorageBuffer16BitAccess</td>
<td>VkPhysicalDeviceVulkan11Features::uniformAndStorageBuffer16BitAccess</td>
</tr>
<tr>
<td>StoragePushConstant16</td>
<td>VkPhysicalDeviceVulkan11Features::storagePushConstant16</td>
</tr>
<tr>
<td>StorageInputOutput16</td>
<td>VkPhysicalDeviceVulkan11Features::storageInputOutput16</td>
</tr>
<tr>
<td>GroupNonUniform</td>
<td>VK_SUBGROUP_FEATURE_BASIC_BIT</td>
</tr>
<tr>
<td>GroupNonUniformVote</td>
<td>VK_SUBGROUP_FEATURE_VOTE_BIT</td>
</tr>
<tr>
<td>GroupNonUniformArithmetic</td>
<td>VK_SUBGROUP_FEATURE_ARITHMETIC_BIT</td>
</tr>
<tr>
<td>GroupNonUniformBallot</td>
<td>VK_SUBGROUP_FEATURE_BALLOT_BIT</td>
</tr>
<tr>
<td>GroupNonUniformShuffle</td>
<td>VK_SUBGROUP_FEATURE_SHUFFLE_BIT</td>
</tr>
<tr>
<td>GroupNonUniformShuffleRelative</td>
<td>VK_SUBGROUP_FEATURE_SHUFFLE_RELATIVE_BIT</td>
</tr>
<tr>
<td>GroupNonUniformClustered</td>
<td>VK_SUBGROUP_FEATURE_CLUSTERED_BIT</td>
</tr>
<tr>
<td>GroupNonUniformQuad</td>
<td>VK_SUBGROUP_FEATURE_QUAD_BIT</td>
</tr>
<tr>
<td>ShaderNonUniform</td>
<td>VK_VERSION_1_2</td>
</tr>
<tr>
<td>RuntimeDescriptorArray</td>
<td>VkPhysicalDeviceVulkan12Features::runtimeDescriptorArray</td>
</tr>
<tr>
<td>InputAttachmentArrayDynamicIndexing</td>
<td>VkPhysicalDeviceVulkan12Features::shaderInputAttachmentArrayDynamicIndexing</td>
</tr>
<tr>
<td>UniformTexelBufferArrayDynamicIndexing</td>
<td>VkPhysicalDeviceVulkan12Features::shaderUniformTexelBufferArrayDynamicIndexing</td>
</tr>
<tr>
<td>StorageTexelBufferArrayDynamicIndexing</td>
<td>VkPhysicalDeviceVulkan12Features::shaderStorageTexelBufferArrayDynamicIndexing</td>
</tr>
<tr>
<td>SPIR-V OpCapability</td>
<td>Vulkan feature, extension, or core version</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>UniformBufferArrayNonUniformIndexing</td>
<td>VkPhysicalDeviceVulkan12Features::shaderUniformBufferArrayNonUniformIndexing</td>
</tr>
<tr>
<td>SampledImageArrayNonUniformIndexing</td>
<td>VkPhysicalDeviceVulkan12Features::shaderSampledImageArrayNonUniformIndexing</td>
</tr>
<tr>
<td>StorageBufferArrayNonUniformIndexing</td>
<td>VkPhysicalDeviceVulkan12Features::shaderStorageBufferArrayNonUniformIndexing</td>
</tr>
<tr>
<td>StorageImageArrayNonUniformIndexing</td>
<td>VkPhysicalDeviceVulkan12Features::shaderStorageImageArrayNonUniformIndexing</td>
</tr>
<tr>
<td>InputAttachmentArrayNonUniformIndexing</td>
<td>VkPhysicalDeviceVulkan12Features::shaderInputAttachmentArrayNonUniformIndexing</td>
</tr>
<tr>
<td>UniformTexelBufferArrayNonUniformIndexing</td>
<td>VkPhysicalDeviceVulkan12Features::shaderUniformTexelBufferArrayNonUniformIndexing</td>
</tr>
<tr>
<td>StorageTexelBufferArrayNonUniformIndexing</td>
<td>VkPhysicalDeviceVulkan12Features::shaderStorageTexelBufferArrayNonUniformIndexing</td>
</tr>
<tr>
<td>Float16</td>
<td>VkPhysicalDeviceVulkan12Features::shaderFloat16</td>
</tr>
<tr>
<td>Int8</td>
<td>VkPhysicalDeviceVulkan12Features::shaderInt8</td>
</tr>
<tr>
<td>StorageBuffer8BitAccess</td>
<td>VkPhysicalDeviceVulkan12Features::storageBuffer8BitAccess</td>
</tr>
<tr>
<td>UniformAndStorageBuffer8BitAccess</td>
<td>VkPhysicalDeviceVulkan12Features::uniformAndStorageBuffer8BitAccess</td>
</tr>
<tr>
<td>StoragePushConstant8</td>
<td>VkPhysicalDeviceVulkan12Features::storagePushConstant8</td>
</tr>
<tr>
<td>VulkanMemoryModel</td>
<td>VkPhysicalDeviceVulkan12Features::vulkanMemoryModel</td>
</tr>
<tr>
<td>VulkanMemoryModelDeviceScope</td>
<td>VkPhysicalDeviceVulkan12Features::vulkanMemoryModelDeviceScope</td>
</tr>
<tr>
<td>DenormPreserve</td>
<td>VkPhysicalDeviceVulkan12Properties::shaderDenormPreserveFloat16</td>
</tr>
<tr>
<td>DenormPreserve</td>
<td>VkPhysicalDeviceVulkan12Properties::shaderDenormPreserveFloat32</td>
</tr>
<tr>
<td>DenormPreserve</td>
<td>VkPhysicalDeviceVulkan12Properties::shaderDenormPreserveFloat64</td>
</tr>
<tr>
<td>DenormFlushToZero</td>
<td>VkPhysicalDeviceVulkan12Properties::shaderDenormFlushToZeroFloat16</td>
</tr>
<tr>
<td>DenormFlushToZero</td>
<td>VkPhysicalDeviceVulkan12Properties::shaderDenormFlushToZeroFloat32</td>
</tr>
<tr>
<td>DenormFlushToZero</td>
<td>VkPhysicalDeviceVulkan12Properties::shaderDenormFlushToZeroFloat64</td>
</tr>
</tbody>
</table>
SPIR-V **OpCapability**

**Vulkan feature, extension, or core version**

<table>
<thead>
<tr>
<th>SignedZeroInfNanPreserve</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkPhysicalDeviceVulkan12Properties::shaderSignedZeroInfNanPreserveFloat16</td>
</tr>
<tr>
<td>VkPhysicalDeviceVulkan12Properties::shaderSignedZeroInfNanPreserveFloat32</td>
</tr>
<tr>
<td>VkPhysicalDeviceVulkan12Properties::shaderSignedZeroInfNanPreserveFloat64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RoundingModeRTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkPhysicalDeviceVulkan12Properties::shaderRoundingModeRTEFloat16</td>
</tr>
<tr>
<td>VkPhysicalDeviceVulkan12Properties::shaderRoundingModeRTEFloat32</td>
</tr>
<tr>
<td>VkPhysicalDeviceVulkan12Properties::shaderRoundingModeRTEFloat64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RoundingModeRTZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkPhysicalDeviceVulkan12Properties::shaderRoundingModeRTZFloat16</td>
</tr>
<tr>
<td>VkPhysicalDeviceVulkan12Properties::shaderRoundingModeRTZFloat32</td>
</tr>
<tr>
<td>VkPhysicalDeviceVulkan12Properties::shaderRoundingModeRTZFloat64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PhysicalStorageBufferAddresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkPhysicalDeviceVulkan12Features::bufferDeviceAddress</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DemoteToHelperInvocationEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkPhysicalDeviceVulkan13Features::shaderDemoteToHelperInvocation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DotProductInputAllKHR</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkPhysicalDeviceVulkan13Features::shaderIntegerDotProduct</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DotProductInput4x8BitKHR</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkPhysicalDeviceVulkan13Features::shaderIntegerDotProduct</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DotProductInput4x8BitPackedKHR</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkPhysicalDeviceVulkan13Features::shaderIntegerDotProduct</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DotProductKHR</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkPhysicalDeviceVulkan13Features::shaderIntegerDotProduct</td>
</tr>
</tbody>
</table>

The application must not pass a SPIR-V module containing any of the following to `vkCreateShaderModule`:

- any **OpCapability** not listed above,
- an unsupported capability, or
- a capability which corresponds to a Vulkan feature or extension which has not been enabled.

**SPIR-V Extensions**

The following table lists SPIR-V extensions that implementations may support. The application must not pass a SPIR-V module to `vkCreateShaderModule` that uses the following SPIR-V extensions unless one of the following conditions is met for the `VkDevice` specified in the `device` parameter of `vkCreateShaderModule`:

- Any corresponding Vulkan extension is enabled.
- The corresponding core version is supported (as returned by `VkPhysicalDeviceProperties`...
Table 62. List of SPIR-V Extensions and corresponding Vulkan extensions or core version

<table>
<thead>
<tr>
<th>SPIR-V OpExtension</th>
<th>Vulkan extension or core version</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPV_KHR_variable_pointers</td>
<td>VK_VERSION_1_1</td>
</tr>
<tr>
<td>SPV_KHR_shader_draw_parameters</td>
<td>VK_VERSION_1_1</td>
</tr>
<tr>
<td>SPV_KHR_8bit_storage</td>
<td>VK_VERSION_1_2</td>
</tr>
<tr>
<td>SPV_KHR_16bit_storage</td>
<td>VK_VERSION_1_1</td>
</tr>
<tr>
<td>SPV_KHR_float_controls</td>
<td>VK_VERSION_1_2</td>
</tr>
<tr>
<td>SPV_KHR_storage_buffer_storage_class</td>
<td>VK_VERSION_1_1</td>
</tr>
<tr>
<td>SPV_EXT_shader_viewport_index_layer</td>
<td>VK_VERSION_1_2</td>
</tr>
<tr>
<td>SPV_EXT_descriptor_indexing</td>
<td>VK_VERSION_1_2</td>
</tr>
<tr>
<td>SPV_KHR_vulkan_memory_model</td>
<td>VK_VERSION_1_2</td>
</tr>
<tr>
<td>SPV_KHR_physical_storage_buffer</td>
<td>VK_VERSION_1_2</td>
</tr>
<tr>
<td>SPV_KHR_multiview</td>
<td>VK_VERSION_1_1</td>
</tr>
</tbody>
</table>

**Validation Rules within a Module**

A SPIR-V module passed to `vkCreateShaderModule` must conform to the following rules:

**Standalone SPIR-V Validation**

The following rules can be validated with only the SPIR-V module itself. They do not depend on knowledge of the implementation and its capabilities or knowledge of runtime information, such as enabled features.

**Valid Usage**

- VUID-StandaloneSpirv-None-04633
  Every entry point must have no return value and accept no arguments
The static function-call graph for an entry point must not contain cycles; that is, static recursion is not allowed.

The Logical or PhysicalStorageBuffer64 addressing model must be selected.

Scope for execution must be limited to Workgroup or Subgroup.

If the Scope for execution is Workgroup, then it must only be used in the task, mesh, tessellation control, or compute execution models.

Scope for execution must be limited to Device, QueueFamily, Workgroup, ShaderCallKHR, Subgroup, or Invocation.

If the ExecutionModel is TessellationControl, and the MemoryModel is GLSL450, the Scope for memory must not be Workgroup.

If the Scope for memory is Workgroup, then it must only be used in the task, mesh, tessellation control, or compute execution models.

If the Scope for memory is ShaderCallKHR, then it must only be used in ray generation, intersection, closest hit, any-hit, miss, and callable execution models.

If the Scope for memory is Invocation, then memory semantics must be None.

Scope for group operations must be limited to Subgroup.

Storage Class must be limited to UniformConstant, Input, Uniform, Output, Workgroup, Private, Function, PushConstant, Image, StorageBuffer, RayPayloadKHR, IncomingRayPayloadKHR, HitAttributeKHR, CallableDataKHR, IncomingCallableDataKHR, ShaderRecordBufferKHR, or PhysicalStorageBuffer.

If the Storage Class is Output, then it must not be used in the GlCompute, RayGenerationKHR, IntersectionKHR, AnyHitKHR, ClosestHitKHR, MissKHR, or CallableKHR execution models.

If the Storage Class is Workgroup, then it must only be used in the task, mesh, or compute execution models.

OpAtomicStore must not use Acquire, AcquireRelease, or SequentiallyConsistent memory semantics.

Scope for memory must be limited to Device, QueueFamily, Workgroup, ShaderCallKHR, Subgroup, or Invocation.

If the ExecutionModel is TessellationControl, and the MemoryModel is GLSL450, the Scope for memory must not be Workgroup.

If the Scope for memory is Workgroup, then it must only be used in the task, mesh, tessellation control, or compute execution models.

If the Scope for memory is ShaderCallKHR, then it must only be used in ray generation, intersection, closest hit, any-hit, miss, and callable execution models.

If the Scope for memory is Invocation, then memory semantics must be None.

Scope for group operations must be limited to Subgroup.

Storage Class must be limited to UniformConstant, Input, Uniform, Output, Workgroup, Private, Function, PushConstant, Image, StorageBuffer, RayPayloadKHR, IncomingRayPayloadKHR, HitAttributeKHR, CallableDataKHR, IncomingCallableDataKHR, ShaderRecordBufferKHR, or PhysicalStorageBuffer.
OpAtomicLoad must not use Release, AcquireRelease, or SequentiallyConsistent memory semantics

- VUID-StandaloneSpirv-OpMemoryBarrier-04732
  OpMemoryBarrier must use one of Acquire, Release, AcquireRelease, or SequentiallyConsistent memory semantics

- VUID-StandaloneSpirv-OpMemoryBarrier-04733
  OpMemoryBarrier must include at least one storage class

- VUID-StandaloneSpirv-OpControlBarrier-04650
  If the semantics for OpControlBarrier includes one of Acquire, Release, AcquireRelease, or SequentiallyConsistent memory semantics, then it must include at least one storage class

- VUID-StandaloneSpirv-OpVariable-04651
  Any OpVariable with an Initializer operand must have Output, Private, Function, or Workgroup as its Storage Class operand

- VUID-StandaloneSpirv-OpVariable-04654
  Any OpVariable with an Initializer operand and Workgroup as its Storage Class operand must use OpConstantNull as the initializer

- VUID-StandaloneSpirv-OpReadClockKHR-04652
  Scope for OpReadClockKHR must be limited to Subgroup or Device

- VUID-StandaloneSpirv-OriginLowerLeft-04653
  The OriginLowerLeft execution mode must not be used; fragment entry points must declare OriginUpperLeft

- VUID-StandaloneSpirv-PixelCenterInteger-04654
  The PixelCenterInteger execution mode must not be used (pixels are always centered at half-integer coordinates)

- VUID-StandaloneSpirv-UniformConstant-04655
  Any variable in the UniformConstant storage class must be typed as either OpTypeImage, OpTypeSampler, OpTypeSampledImage, OpTypeAccelerationStructureKHR, or an array of one of these types

- VUID-StandaloneSpirv-Uniform-06807
  Any variable in the Uniform or StorageBuffer storage class must be typed as OpTypeStruct or an array of this type

- VUID-StandaloneSpirv-PushConstant-06808
  Any variable in the PushConstant storage class must be typed as OpTypeStruct

- VUID-StandaloneSpirv-OpTypeImage-04656
  OpTypeImage must declare a scalar 32-bit float, 64-bit integer, or 32-bit integer type for the “Sampled Type” (RelaxedPrecision can be applied to a sampling instruction and to the variable holding the result of a sampling instruction)

- VUID-StandaloneSpirv-OpTypeImage-04657
  OpTypeImage must have a “Sampled” operand of 1 (sampled image) or 2 (storage image)

- VUID-StandaloneSpirv-OpTypeSampledImage-06671
  OpTypeSampledImage must have a OpTypeImage with a “Sampled” operand of 1 (sampled
The converted bit width, signedness, and numeric type of the Image Format operand of an OpTypeImage must match the Sampled Type, as defined in Image Format and Type Matching.

If an OpImageTexelPointer is used in an atomic operation, the image type of the image parameter to OpImageTexelPointer must have an image format of R64i, R64ui, R32f, R32i, or R32ui.

OpImageQuerySizeLod, OpImageQueryLod, and OpImageQueryLevels must only consume an “Image” operand whose type has its “Sampled” operand set to 1.

An OpTypeImage with a “Dim” operand of SubpassData must have an “Arrayed” operand of 0 (non-arrayed) and a “Sampled” operand of 2 (storage image).

The (u,v) coordinates used for a SubpassData must be the <id> of a constant vector (0,0), or if a layer coordinate is used, must be a vector that was formed with constant 0 for the u and v components.

Objects of types OpTypeImage, OpTypeSampler, OpTypeSampledImage, OpTypeAccelerationStructureKHR, and arrays of these types must not be stored to or modified.

Any variable in the Uniform storage class decorated as Block must not be stored to or modified.

Any image operation must use at most one of the Offset, ConstOffset, and ConstOffsets image operands.

Image operand Offset must only be used with OpImage*Gather instructions.

Any image instruction which uses an Offset, ConstOffset, or ConstOffsets image operand, must only consume a “Sampled Image” operand whose type has its “Sampled” operand set to 1.

The “Component” operand of OpImageGather, and OpImageSparseGather must be the <id> of a constant instruction.

OpImage*Dref* instructions must not consume an image whose Dim is 3D.

Structure types must not contain opaque types.
Any BuiltIn decoration not listed in Built-In Variables must not be used.

- VUID-StandaloneSpirv-Location-06672
  The Location or Component decorations must only be used with the Input, Output, RayPayloadKHR, IncomingRayPayloadKHR, HitAttributeKHR, HitObjectAttributeNV, CallableDataKHR, IncomingCallableDataKHR, or ShaderRecordBufferKHR storage classes.

- VUID-StandaloneSpirv-Location-04915
  The Location or Component decorations must not be used with BuiltIn.

- VUID-StandaloneSpirv-Location-04916
  The Location decorations must be used on user-defined variables.

- VUID-StandaloneSpirv-Location-04917
  The Location decorations must be used on an OpVariable with a structure type that is not a block.

- VUID-StandaloneSpirv-Location-04918
  The Location decorations must not be used on the members of OpVariable with a structure type that is decorated with Location.

- VUID-StandaloneSpirv-Location-04919
  The Location decorations must be used on each member of OpVariable with a structure type that is a block not decorated with Location.

- VUID-StandaloneSpirv-Component-04920
  The Component decoration value must not be greater than 3.

- VUID-StandaloneSpirv-Component-04921
  If the Component decoration is used on an OpVariable that has a OpTypeVector type with a Component Type with a Width that is less than or equal to 32, the sum of its Component Count and the Component decoration value must be less than or equal to 4.

- VUID-StandaloneSpirv-Component-04922
  If the Component decoration is used on an OpVariable that has a OpTypeVector type with a Component Type with a Width that is equal to 64, the sum of two times its Component Count and the Component decoration value must be less than or equal to 4.

- VUID-StandaloneSpirv-Component-04923
  The Component decorations value must not be 1 or 3 for scalar or two-component 64-bit data types.

- VUID-StandaloneSpirv-Component-04924
  The Component decorations must not used with any type that is not a scalar or vector.

- VUID-StandaloneSpirv-Component-07703
  The Component decorations must not be used for a 64-bit vector type with more than two components.

- VUID-StandaloneSpirv-GLSLShared-04669
  The GLSLShared and GLSLOrdered decorations must not be used.

- VUID-StandaloneSpirv-Flat-04670
  The Flat, NoPerspective, Sample, and Centroid decorations must only be used on variables with the Output or Input storage class.

- VUID-StandaloneSpirv-Flat-06201
  The Flat, NoPerspective, Sample, and Centroid decorations must only be used on variables with the Output or Input storage class.
The Flat, NoPerspective, Sample, and Centroid decorations must not be used on variables with the Output storage class in a fragment shader

- VUID-StandaloneSpirv-Flat-06202
  The Flat, NoPerspective, Sample, and Centroid decorations must not be used on variables with the Input storage class in a vertex shader

- VUID-StandaloneSpirv-PerVertexKHR-06777
  The PerVertexKHR decoration must only be used on variables with the Input storage class in a fragment shader

- VUID-StandaloneSpirv-Flat-04744
  Any variable with integer or double-precision floating-point type and with Input storage class in a fragment shader, must be decorated Flat

- VUID-StandaloneSpirv-ViewportRelativeNV-04672
  The ViewportRelativeNV decoration must only be used on a variable decorated with Layer in the vertex, tessellation evaluation, or geometry shader stages

- VUID-StandaloneSpirv-ViewportRelativeNV-04673
  The ViewportRelativeNV decoration must not be used unless a variable decorated with one of ViewportIndex or ViewportMaskNV is also statically used by the same OpEntryPoint

- VUID-StandaloneSpirv-ViewportMaskNV-04674
  The ViewportMaskNV and ViewportIndex decorations must not both be statically used by one or more OpEntryPoint’s that form the pre-rasterization shader stages of a graphics pipeline

- VUID-StandaloneSpirv-FPRoundingMode-04675
  Rounding modes other than round-to-nearest-even and round-towards-zero must not be used for the FPRoundingMode decoration

- VUID-StandaloneSpirv-Invariant-04677
  Variables decorated with Invariant and variables with structure types that have any members decorated with Invariant must be in the Output or Input storage class, Invariant used on an Input storage class variable or structure member has no effect

- VUID-StandaloneSpirv-VulkanMemoryModel-04678
  If the VulkanMemoryModel capability is not declared, the Volatile decoration must be used on any variable declaration that includes one of the SMIDNV, WarpIDNV, SubgroupSize, SubgroupLocalInvocationId, SubgroupEqMask, SubgroupGeMask, SubgroupGtMask, SubgroupLeMask, or SubgroupLtMask BuiltIn decorations when used in the ray generation, closest hit, miss, intersection, or callable shaders, or with the RayTmaxKHR Builtin decoration when used in an intersection shader

- VUID-StandaloneSpirv-VulkanMemoryModel-04679
  If the VulkanMemoryModel capability is declared, the OpLoad instruction must use the Volatile memory semantics when it accesses into any variable that includes one of the SMIDNV, WarpIDNV, SubgroupSize, SubgroupLocalInvocationId, SubgroupEqMask, SubgroupGeMask, SubgroupGtMask, SubgroupLeMask, or SubgroupLtMask BuiltIn decorations when used in the ray generation, closest hit, miss, intersection, or callable shaders, or with the RayTmaxKHR Builtin decoration when used in an intersection shader

- VUID-StandaloneSpirv-OpTypeRuntimeArray-04680
  OpTypeRuntimeArray must only be used for the last member of a Block-decorated
OpTypeStruct in StorageBuffer or PhysicalStorageBuffer storage classes; BufferBlock-decorated OpTypeStruct in Uniform storage class; the outermost dimension of an arrayed variable in the StorageBuffer, Uniform, or UniformConstant storage classes.

- **VUID-StandaloneSpirv-Function-04681**
  A type \( T \) that is an array sized with a specialization constant **must** neither be, nor be contained in, the type \( T_2 \) of a variable \( V \), unless either: a) \( T \) is equal to \( T_2 \), b) \( V \) is declared in the Function, or Private storage classes, c) \( V \) is a non-Block variable in the Workgroup storage class, or d) \( V \) is an interface variable with an additional level of arrayness, **as described in interface matching**, and \( T \) is the member type of the array type \( T_2 \)

- **VUID-StandaloneSpirv-OpControlBarrier-04682**
  If OpControlBarrier is used in ray generation, intersection, any-hit, closest hit, miss, fragment, vertex, tessellation evaluation, or geometry shaders, the execution Scope **must** be Subgroup

- **VUID-StandaloneSpirv-LocalSize-06426**
  For each compute shader entry point, either a LocalSize or LocalSizeId execution mode, or an object decorated with the WorkgroupSize decoration **must** be specified

- **VUID-StandaloneSpirv-DerivativeGroupQuadsNV-04684**
  For compute shaders using the DerivativeGroupQuadsNV execution mode, the first two dimensions of the local workgroup size **must** be a multiple of two

- **VUID-StandaloneSpirv-DerivativeGroupLinearNV-04778**
  For compute shaders using the DerivativeGroupLinearNV execution mode, the product of the dimensions of the local workgroup size **must** be a multiple of four

- **VUID-StandaloneSpirv-OpGroupNonUniformBallotBitCount-04685**
  If OpGroupNonUniformBallotBitCount is used, the group operation **must** be limited to Reduce, InclusiveScan, or ExclusiveScan

- **VUID-StandaloneSpirv-None-04686**
  The Pointer operand of all atomic instructions **must** have a Storage Class limited to Uniform, Workgroup, Image, StorageBuffer, PhysicalStorageBuffer, or TaskPayloadWorkgroupEXT

- **VUID-StandaloneSpirv-Offset-04687**
  Output variables or block members decorated with Offset that have a 64-bit type, or a composite type containing a 64-bit type, **must** specify an Offset value aligned to a 8 byte boundary

- **VUID-StandaloneSpirv-Offset-04689**
  The size of any output block containing any member decorated with Offset that is a 64-bit type **must** be a multiple of 8

- **VUID-StandaloneSpirv-Offset-04690**
  The first member of an output block specifying a Offset decoration **must** specify a Offset value that is aligned to an 8 byte boundary if that block contains any member decorated with Offset and is a 64-bit type

- **VUID-StandaloneSpirv-Offset-04691**
  Output variables or block members decorated with Offset that have a 32-bit type, or a composite type contains a 32-bit type, **must** specify an Offset value aligned to a 4 byte boundary
boundary

• VUID-StandaloneSpirv-Offset-04692
  Output variables, blocks or block members decorated with Offset must only contain base types that have components that are either 32-bit or 64-bit in size

• VUID-StandaloneSpirv-Offset-04716
  Only variables or block members in the output interface decorated with Offset can be captured for transform feedback, and those variables or block members must also be decorated with XfbBuffer and XfbStride, or inherit XfbBuffer and XfbStride decorations from a block containing them

• VUID-StandaloneSpirv-XfbBuffer-04693
  All variables or block members in the output interface of the entry point being compiled decorated with a specific XfbBuffer value must all be decorated with identical XfbStride values

• VUID-StandaloneSpirv-Stream-04694
  If any variables or block members in the output interface of the entry point being compiled are decorated with Stream, then all variables belonging to the same XfbBuffer must specify the same Stream value

• VUID-StandaloneSpirv-XfbBuffer-04696
  For any two variables or block members in the output interface of the entry point being compiled with the same XfbBuffer value, the ranges determined by the Offset decoration and the size of the type must not overlap

• VUID-StandaloneSpirv-XfbBuffer-04697
  All block members in the output interface of the entry point being compiled that are in the same block and have a declared or inherited XfbBuffer decoration must specify the same XfbBuffer value

• VUID-StandaloneSpirv-RayPayloadKHR-04698
  RayPayloadKHR storage class must only be used in ray generation, closest hit or miss shaders

• VUID-StandaloneSpirv-IncomingRayPayloadKHR-04699
  IncomingRayPayloadKHR storage class must only be used in closest hit, any-hit, or miss shaders

• VUID-StandaloneSpirv-IncomingRayPayloadKHR-04700
  There must be at most one variable with the IncomingRayPayloadKHR storage class in the input interface of an entry point

• VUID-StandaloneSpirv-HitAttributeKHR-04701
  HitAttributeKHR storage class must only be used in intersection, any-hit, or closest hit shaders

• VUID-StandaloneSpirv-HitAttributeKHR-04702
  There must be at most one variable with the HitAttributeKHR storage class in the input interface of an entry point

• VUID-StandaloneSpirv-HitAttributeKHR-04703
  A variable with HitAttributeKHR storage class must only be written to in an intersection shader
CallableDataKHR storage class **must** only be used in ray generation, closest hit, miss, and callable shaders

IncomingCallableDataKHR storage class **must** only be used in callable shaders

There **must** be at most one variable with the **IncomingCallableDataKHR** storage class in the input interface of an entry point

ShaderRecordKHR storage class **must** only be used in ray generation, intersection, any-hit, closest hit, callable, or miss shaders

There **must** be at most one variable with the **IncomingCallableDataKHR** storage class in the input interface of an entry point

The **Base** operand of **OpPtrAccessChain** **must** have a storage class of **Workgroup**, **StorageBuffer**, or **PhysicalStorageBuffer**

If the **Base** operand of **OpPtrAccessChain** has a **Workgroup** storage class, then the **VariablePointers** capability **must** be declared

If the **Base** operand of **OpPtrAccessChain** has a **StorageBuffer** storage class, then the **VariablePointers** or **VariablePointersStorageBuffer** capability **must** be declared

If the **PhysicalStorageBuffer64** addressing model is enabled, all instructions that support memory access operands and that use a physical pointer **must** include the **Aligned** operand

If the **PhysicalStorageBuffer64** addressing model is enabled, any access chain instruction that accesses into a **RowMajor** matrix **must** only be used as the **Pointer** operand to **OpLoad** or **OpStore**

If the **PhysicalStorageBuffer64** addressing model is enabled, **OpConvertUToPtr** and **OpConvertPtrToU** **must** use an integer type whose **Width** is 64

**OpTypeForwardPointer** **must** have a storage class of **PhysicalStorageBuffer**

All block members in a variable with a storage class of **PushConstant** declared as an array **must** only be accessed by dynamically uniform indices

There **must** not be more than one **OpVariable** in the **PushConstant** storage class listed in the **Interface** for each **OpEntryPoint**

Each **OpEntryPoint** **must** not statically use more than one **OpVariable** in the **PushConstant** storage class
• VUID-StandaloneSpirv-Result-04780
  The Result Type operand of any OpImageRead or OpImageSparseRead instruction must be a vector of four components

• VUID-StandaloneSpirv-Base-04781
  The Base operand of any OpBitCount, OpBitReverse, OpBitFieldInsert, OpBitFieldSExtract, or OpBitFieldUExtract instruction must be a 32-bit integer scalar or a vector of 32-bit integers

• VUID-StandaloneSpirv-PushConstant-06675
  Any variable in the PushConstant or StorageBuffer storage class must be decorated as Block

• VUID-StandaloneSpirv-Uniform-06676
  Any variable in the Uniform storage class must be decorated as Block or BufferBlock

• VUID-StandaloneSpirv-UniformConstant-06677
  Any variable in the UniformConstant, StorageBuffer, or Uniform storage class must be decorated with DescriptorSet and Binding

• VUID-StandaloneSpirv-InputAttachmentIndex-06678
  Variables decorated with InputAttachmentIndex must be in the UniformConstant storage class

• VUID-StandaloneSpirv-DescriptorSet-06491
  If a variable is decorated by DescriptorSet or Binding, the storage class must correspond to an entry in Shader Resource and Storage Class Correspondence

• VUID-StandaloneSpirv-Input-06778
  Variables with a storage class of Input in a fragment shader stage that are decorated with PerVertexKHR must be declared as arrays

• VUID-StandaloneSpirv-MeshEXT-07102
  The module must not contain both an entry point that uses the TaskEXT or MeshEXT Execution Model and an entry point that uses the TaskNV or MeshNV Execution Model

• VUID-StandaloneSpirv-MeshEXT-07106
  In mesh shaders using the MeshEXT Execution Model OpSetMeshOutputsEXT must be called before any outputs are written

• VUID-StandaloneSpirv-MeshEXT-07107
  In mesh shaders using the MeshEXT Execution Model all variables declared as output must not be read from

• VUID-StandaloneSpirv-MeshEXT-07108
  In mesh shaders using the MeshEXT Execution Model for OpSetMeshOutputsEXT instructions, the “Vertex Count” and “Primitive Count” operands must not depend on ViewIndex

• VUID-StandaloneSpirv-MeshEXT-07109
  In mesh shaders using the MeshEXT Execution Model variables decorated with PrimitivePointIndicesEXT, PrimitiveLineIndicesEXT, or PrimitiveTriangleIndicesEXT declared as an array must not be accessed by indices that depend on ViewIndex

• VUID-StandaloneSpirv-MeshEXT-07110
  In mesh shaders using the MeshEXT Execution Model any values stored in variables decorated with PrimitivePointIndicesEXT, PrimitiveLineIndicesEXT, or PrimitiveTriangleIndicesEXT must not depend on ViewIndex
• VUID-StandalonSpirv-MeshEXT-07111
In mesh shaders using the MeshEXT Execution Model variables in workgroup or private storage class declared as or containing a composite type must not be accessed by indices that depend on ViewIndex

• VUID-StandalonSpirv-MeshEXT-07330
In mesh shaders using the MeshEXT Execution Model the OutputVertices Execution Mode must be greater than 0

• VUID-StandalonSpirv-MeshEXT-07331
In mesh shaders using the MeshEXT Execution Model the OutputPrimitivesEXT Execution Mode must be greater than 0

• VUID-StandalonSpirv-MeshEXT-07728
In mesh shaders using the MeshEXT or MeshNV Execution Model and the OutputPoints Execution Mode, if the number of output points is greater than 0, a PointSize decorated variable must be written to for each output point

• VUID-StandalonSpirv-Input-07290
Variables with a storage class of Input or Output and a type of OpTypeBool must be decorated with the BuiltIn decoration

Runtime SPIR-V Validation

The following rules must be validated at runtime. These rules depend on knowledge of the implementation and its capabilities and knowledge of runtime information, such as enabled features.

Valid Usage

• VUID-RuntimeSpirv-vulkanMemoryModel-06265
If vulkanMemoryModel is enabled and vulkanMemoryModelDeviceScope is not enabled, Device memory scope must not be used

• VUID-RuntimeSpirv-vulkanMemoryModel-06266
If vulkanMemoryModel is not enabled, QueueFamily memory scope must not be used

• VUID-RuntimeSpirv-OpImageWrite-07112
OpImageWrite to any Image whose Image Format is not Unknown must have the Texel operand contain at least as many components as the corresponding VkFormat as given in the SPIR-V Image Format compatibility table

• VUID-RuntimeSpirv-Location-06272
The sum of Location and the number of locations the variable it decorates consumes must be less than or equal to the value for the matching Execution Model defined in Shader Input and Output Locations

• VUID-RuntimeSpirv-Fragment-06427
When blending is enabled and one of the dual source blend modes is in use, the maximum number of output attachments written to in the Fragment Execution Model must be less than or equal to maxFragmentDualSrcAttachments
The maximum number of storage buffers, storage images, and output \texttt{Location} decorated color attachments written to in the \texttt{Fragment Execution Model} \textbf{must} be less than or equal to \texttt{maxFragmentCombinedOutputResources}.

If an instruction loads from or stores to a resource (including atomics and image instructions) and the resource descriptor being accessed is not dynamically uniform, then the operand corresponding to that resource (e.g. the pointer or sampled image operand) \textbf{must} be decorated with \texttt{NonUniform}.

\texttt{shaderSubgroupExtendedTypes} \textbf{must} be enabled for \texttt{group operations} to use 8-bit integer, 16-bit integer, 64-bit integer, 16-bit floating-point, and vectors of these types.

If \texttt{subgroupBroadcastDynamicId} is \texttt{VK_TRUE}, and the shader module version is 1.5 or higher, the “Index” for \texttt{OpGroupNonUniformQuadBroadcast} \textbf{must} be dynamically uniform within the derivative group. Otherwise, “Index” \textbf{must} be a constant.

If \texttt{subgroupBroadcastDynamicId} is \texttt{VK_TRUE}, and the shader module version is 1.5 or higher, the “Id” for \texttt{OpGroupNonUniformBroadcast} \textbf{must} be dynamically uniform within the subgroup. Otherwise, “Id” \textbf{must} be a constant.

If \texttt{denormBehaviorIndependence} is \texttt{VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_32_BIT_ONLY}, then the entry point \textbf{must} use the same denormals execution mode for both 16-bit and 64-bit floating-point types.

If \texttt{denormBehaviorIndependence} is \texttt{VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_NONE}, then the entry point \textbf{must} use the same denormals execution mode for all floating-point types.

If \texttt{roundingModeIndependence} is \texttt{VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_32_BIT_ONLY}, then the entry point \textbf{must} use the same rounding execution mode for both 16-bit and 64-bit floating-point types.

If \texttt{roundingModeIndependence} is \texttt{VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_NONE}, then the entry point \textbf{must} use the same rounding execution mode for all floating-point types.

If \texttt{shaderSignedZeroInfNanPreserveFloat16} is \texttt{VK_FALSE}, then \texttt{SignedZeroInfNanPreserve} for 16-bit floating-point type \textbf{must} not be used.

If \texttt{shaderSignedZeroInfNanPreserveFloat32} is \texttt{VK_FALSE}, then \texttt{SignedZeroInfNanPreserve} for 32-bit floating-point type \textbf{must} not be used.

If \texttt{shaderSignedZeroInfNanPreserveFloat64} is \texttt{VK_FALSE}, then \texttt{SignedZeroInfNanPreserve} for 64-bit floating-point type \textbf{must} not be used.
• VUID-RuntimeSpirv-shaderDenormPreserveFloat16-06296
   If shaderDenormPreserveFloat16 is VK_FALSE, then DenormPreserve for 16-bit floating-point type must not be used

• VUID-RuntimeSpirv-shaderDenormPreserveFloat32-06297
   If shaderDenormPreserveFloat32 is VK_FALSE, then DenormPreserve for 32-bit floating-point type must not be used

• VUID-RuntimeSpirv-shaderDenormPreserveFloat64-06298
   If shaderDenormPreserveFloat64 is VK_FALSE, then DenormPreserve for 64-bit floating-point type must not be used

• VUID-RuntimeSpirv-shaderDenormFlushToZeroFloat16-06299
   If shaderDenormFlushToZeroFloat16 is VK_FALSE, then DenormFlushToZero for 16-bit floating-point type must not be used

• VUID-RuntimeSpirv-shaderDenormFlushToZeroFloat32-06300
   If shaderDenormFlushToZeroFloat32 is VK_FALSE, then DenormFlushToZero for 32-bit floating-point type must not be used

• VUID-RuntimeSpirv-shaderDenormFlushToZeroFloat64-06301
   If shaderDenormFlushToZeroFloat64 is VK_FALSE, then DenormFlushToZero for 64-bit floating-point type must not be used

• VUID-RuntimeSpirv-shaderRoundingModeRTEFloat16-06302
   If shaderRoundingModeRTEFloat16 is VK_FALSE, then RoundingModeRTE for 16-bit floating-point type must not be used

• VUID-RuntimeSpirv-shaderRoundingModeRTEFloat32-06303
   If shaderRoundingModeRTEFloat32 is VK_FALSE, then RoundingModeRTE for 32-bit floating-point type must not be used

• VUID-RuntimeSpirv-shaderRoundingModeRTEFloat64-06304
   If shaderRoundingModeRTEFloat64 is VK_FALSE, then RoundingModeRTE for 64-bit floating-point type must not be used

• VUID-RuntimeSpirv-shaderRoundingModeRTZFloat16-06305
   If shaderRoundingModeRTZFloat16 is VK_FALSE, then RoundingModeRTZ for 16-bit floating-point type must not be used

• VUID-RuntimeSpirv-shaderRoundingModeRTZFloat32-06306
   If shaderRoundingModeRTZFloat32 is VK_FALSE, then RoundingModeRTZ for 32-bit floating-point type must not be used

• VUID-RuntimeSpirv-shaderRoundingModeRTZFloat64-06307
   If shaderRoundingModeRTZFloat64 is VK_FALSE, then RoundingModeRTZ for 64-bit floating-point type must not be used

• VUID-RuntimeSpirv-PhysicalStorageBuffer64-06314
   If the PhysicalStorageBuffer64 addressing model is enabled any load or store through a physical pointer type must be aligned to a multiple of the size of the largest scalar type in the pointed-to type

• VUID-RuntimeSpirv-PhysicalStorageBuffer64-06315
   If the PhysicalStorageBuffer64 addressing model is enabled the pointer value of a memory access instruction must be at least as aligned as specified by the Aligned memory access
• VUID-RuntimeSpirv-DescriptorSet-06323
  DescriptorSet and Binding decorations must obey the constraints on storage class, type, and descriptor type described in DescriptorSet and Binding Assignment

• VUID-RuntimeSpirv-NonWritable-06340
  If fragmentStoresAndAtomics is not enabled, then all storage image, storage texel buffer, and storage buffer variables in the fragment stage must be decorated with the NonWritable decoration

• VUID-RuntimeSpirv-NonWritable-06341
  If vertexPipelineStoresAndAtomics is not enabled, then all storage image, storage texel buffer, and storage buffer variables in the vertex, tessellation, and geometry stages must be decorated with the NonWritable decoration

• VUID-RuntimeSpirv-None-06342
  If subgroupQuadOperationsInAllStages is VK_FALSE, then quad subgroup operations must not be used except for in fragment and compute stages

• VUID-RuntimeSpirv-None-06343
  Group operations with subgroup scope must not be used if the shader stage is not in subgroupSupportedStages

• VUID-RuntimeSpirv-Offset-06344
  The first element of the Offset operand of InterpolateAtOffset must be greater than or equal to:

  \[ \text{fragwidth} \times \minInterpolationOffset \]

  where fragwidth is the width of the current fragment in pixels

• VUID-RuntimeSpirv-Offset-06345
  The first element of the Offset operand of InterpolateAtOffset must be less than or equal to:

  \[ \text{fragwidth} \times (\maxInterpolationOffset + \text{ULP}) - \text{ULP} \]

  where fragwidth is the width of the current fragment in pixels and ULP = 1 / \(2^\text{subPixelInterpolationOffsetBits}\)

• VUID-RuntimeSpirv-Offset-06346
  The second element of the Offset operand of InterpolateAtOffset must be greater than or equal to:

  \[ \text{fragheight} \times \minInterpolationOffset \]

  where fragheight is the height of the current fragment in pixels

• VUID-RuntimeSpirv-Offset-06347
The second element of the Offset operand of InterpolateAtOffset must be less than or equal to:

\[
\text{frag}_\text{height} \times (\text{maxInterpolationOffset} + \text{ULP}) - \text{ULP}
\]

where \(\text{frag}_\text{height}\) is the height of the current fragment in pixels and \(\text{ULP} = 1 / 2^{\text{subPixelInterpolationOffsetBits}}\).

- **VUID-RuntimeSpirv-x-06429**
  The \(x\) size in \text{LocalSize} or \text{LocalSizeId} must be less than or equal to \(\text{VkPhysicalDeviceLimits}::\text{maxComputeWorkGroupSize}[0]\)

- **VUID-RuntimeSpirv-y-06430**
  The \(y\) size in \text{LocalSize} or \text{LocalSizeId} must be less than or equal to \(\text{VkPhysicalDeviceLimits}::\text{maxComputeWorkGroupSize}[1]\)

- **VUID-RuntimeSpirv-z-06431**
  The \(z\) size in \text{LocalSize} or \text{LocalSizeId} must be less than or equal to \(\text{VkPhysicalDeviceLimits}::\text{maxComputeWorkGroupSize}[2]\)

- **VUID-RuntimeSpirv-x-06432**
  The product of \(x\) size, \(y\) size, and \(z\) size in \text{LocalSize} or \text{LocalSizeId} must be less than or equal to \(\text{VkPhysicalDeviceLimits}::\text{maxComputeWorkGroupInvocations}\)

- **VUID-RuntimeSpirv-LocalSizeId-06434**
  If execution mode \text{LocalSizeId} is used, \text{maintenance4} must be enabled

- **VUID-RuntimeSpirv-maintenance4-06817**
  If \text{maintenance4} is not enabled, any \text{OpTypeVector} output interface variables must not have a higher \text{Component Count} than a matching \text{OpTypeVector} input interface variable

- **VUID-RuntimeSpirv-OpEntryPoint-07754**
  Any \text{user-defined variables} between the \text{OpEntryPoint} of two shader stages must have the same type and width for each component

- **VUID-RuntimeSpirv-Workgroup-06530**
  The sum of size in bytes for variables and \text{padding} in the \text{Workgroup} storage class in the \text{GLCompute Execution Model} must be less than or equal to \(\text{maxComputeSharedMemorySize}\)

- **VUID-RuntimeSpirv-shaderZeroInitializeWorkgroupMemory-06372**
  If \text{shaderZeroInitializeWorkgroupMemory} is not enabled, any \text{OpVariable} with \text{Workgroup} as its \text{Storage Class} must not have an \text{Initializer} operand

- **VUID-RuntimeSpirv-OpImage*Gather-06376**
  If an \text{OpImage*Gather} operation has an image operand of Offset, \text{ConstOffset}, or \text{ConstOffsets} the offset value must be greater than or equal to \(\text{minTexelGatherOffset}\)

- **VUID-RuntimeSpirv-OpImage-06377**
  If an \text{OpImage*Gather} operation has an image operand of Offset, \text{ConstOffset}, or \text{ConstOffsets} the offset value must be less than or equal to \(\text{maxTexelGatherOffset}\)

- **VUID-RuntimeSpirv-OpImageSample-06435**
  If an \text{OpImageSample*} or \text{OpImageFetch*} operation has an image operand of \text{ConstOffset} then the offset value must be greater than or equal to \(\text{minTexelOffset}\)
Precision and Operation of SPIR-V Instructions

The following rules apply to half, single, and double-precision floating point instructions:

- Positive and negative infinities and positive and negative zeros are generated as dictated by IEEE 754, but subject to the precisions allowed in the following table.

- Dividing a non-zero by a zero results in the appropriately signed IEEE 754 infinity.

- Signaling NaNs are not required to be generated and exceptions are never raised. Signaling NaN may be converted to quiet NaNs values by any floating point instruction.

- By default, the implementation may perform optimizations on half, single, or double-precision floating-point instructions that ignore sign of a zero, or assume that arguments and results are not NaNs or infinities. If the entry point is declared with the SignedZeroInfNanPreserve execution mode, then NaNs, infinities, and the sign of zero must not be ignored.


- Denormalized values are supported.

  - By default, any half, single, or double-precision denormalized value input into a shader or potentially generated by any instruction (except those listed above) or any extended instructions for GLSL in a shader may be flushed to zero.

  - If the entry point is declared with the DenormFlushToZero execution mode then for the affected instructions the denormalized result must be flushed to zero and the denormalized operands may be flushed to zero. Denormalized values obtained via unpacking an integer into a vector of values with smaller bit width and interpreting those values as floating-point numbers must be flushed to zero.

Cos, Tan, Asin, Acos, Atan, Sin, Cosh, Tanh, Asinh, Acosh, Atanh, Pow, Exp, Log, Exp2, Log2, Sqrt, InverseSqrt, Determinant, MatrixInverse, Modf, ModfStruct, FMin, FMax, FClamp, FMix, Step, SmoothStep, Fma, UnpackHalf2x16, UnpackDouble2x32, Length, Distance, Cross, Normalize, FaceForward, Reflect, Refract, NMin, NMax, NClamp. Other SPIR-V instructions (except those excluded above) may also flush denormalized values.


The precision of double-precision instructions is at least that of single precision.

The precision of individual operations is defined in Precision of Individual Operations. Subject to the constraints below, however, implementations may reorder or combine operations, resulting in expressions exhibiting different precisions than might be expected from the constituent operations.

### Evaluation of Expressions

Implementations may rearrange floating-point operations using any of the mathematical properties governing the expressions in precise arithmetic, even where the floating-point operations do not share these properties. This includes, but is not limited to, associativity and distributivity, and may involve a different number of rounding steps than would occur if the operations were not rearranged. In shaders that use the SignedZeroInfNanPreserve execution mode the values must be preserved if they are generated after any rearrangement but the execution mode does not change which rearrangements are valid. This rearrangement can be prevented for particular operations by using the NoContraction decoration.

#### Note

For example, in the absence of the NoContraction decoration implementations are allowed to implement \( a + b - a \) and \( \frac{a + b}{2} \) as \( b \). The SignedZeroInfNanPreserve does not prevent these transformations, even though they may overflow to infinity or NaN when evaluated in floating-point.

If the NoContraction decoration is applied then operations may not be rearranged, so, for example, \( a + a - a \) must account for possible overflow to infinity. If infinities are not preserved then the expression may be replaced with \( a \), since the replacement is exact when overflow does not occur and infinities may be replaced with undefined values. If both NoContraction and SignedZeroInfNanPreserve are used then the result must be infinity for sufficiently large \( a \).
Precision of Individual Operations

The precision of individual operations is defined either in terms of rounding (correctly rounded), as an error bound in ULP, or as inherited from a formula as follows:

Correctly Rounded

Operations described as “correctly rounded” will return the infinitely precise result, \( x \), rounded so as to be representable in floating-point. The rounding mode is not specified, unless the entry point is declared with the `RoundingModeRTE` or the `RoundingModeRTZ` execution mode. These execution modes affect only correctly rounded SPIR-V instructions. These execution modes do not affect `OpQuantizeToF16`. If the rounding mode is not specified then this rounding is implementation specific, subject to the following rules. If \( x \) is exactly representable then \( x \) will be returned. Otherwise, either the floating-point value closest to and no less than \( x \) or the value closest to and no greater than \( x \) will be returned.

ULP

Where an error bound of \( n \) ULP (units in the last place) is given, for an operation with infinitely precise result \( x \) the value returned must be in the range \([x - n \times \text{ulp}(x), x + n \times \text{ulp}(x)]\). The function \( \text{ulp}(x) \) is defined as follows:

If there exist non-equal floating-point numbers \( a \) and \( b \) such that \( a \leq x \leq b \) then \( \text{ulp}(x) \) is the minimum possible distance between such numbers, \( \text{ulp}(x) = \min_a, b |b - a| \). If such numbers do not exist then \( \text{ulp}(x) \) is defined to be the difference between the two finite floating-point numbers nearest to \( x \).

Where the range of allowed return values includes any value of magnitude larger than that of the largest representable finite floating-point number, operations may, additionally, return either an infinity of the appropriate sign or the finite number with the largest magnitude of the appropriate sign. If the infinitely precise result of the operation is not mathematically defined then the value returned is undefined.

Inherited From ...

Where an operation’s precision is described as being inherited from a formula, the result returned must be at least as accurate as the result of computing an approximation to \( x \) using a formula equivalent to the given formula applied to the supplied inputs. Specifically, the formula given may be transformed using the mathematical associativity, commutativity and distributivity of the operators involved to yield an equivalent formula. The SPIR-V precision rules, when applied to each such formula and the given input values, define a range of permitted values. If NaN is one of the permitted values then the operation may return any result, otherwise let the largest permitted value in any of the ranges be \( F_{\text{max}} \) and the smallest be \( F_{\text{min}} \). The operation must return a value in the range \([x - E, x + E]\) where \( E = \max(|x - F_{\text{min}}|, |x - F_{\text{max}}|) \). If the entry point is declared with the \texttt{DenormFlushToZero} execution mode, then any intermediate denormal value(s) while evaluating the formula may be flushed to zero. Denormal final results must be flushed to zero. If the entry point is declared with the \texttt{DenormPreserve} execution mode, then denormals must be preserved throughout the formula.

For half- (16 bit) and single- (32 bit) precision instructions, precisions are required to be at least as
Table 63. Precision of core SPIR-V Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Single precision, unless decorated with RelaxedPrecision</th>
<th>Half precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpFAdd</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>OpFSub</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>OpDot(x, y)</td>
<td>Inherited from $\sum_{i=0}^{n-1} x_i \times y_i$.</td>
<td></td>
</tr>
<tr>
<td>OpFOrdEqual, OpFUnordEqual</td>
<td>Correct result.</td>
<td></td>
</tr>
<tr>
<td>OpFOrdLessThan, OpFUnordLessThan</td>
<td>Correct result.</td>
<td></td>
</tr>
<tr>
<td>OpFOrdGreaterThan, OpFUnordGreaterThan</td>
<td>Correct result.</td>
<td></td>
</tr>
<tr>
<td>OpFOrdGreaterThanEqual, OpFUnordGreaterThanEqual</td>
<td>Correct result.</td>
<td></td>
</tr>
<tr>
<td>OpFDiv(x, y)</td>
<td>2.5 ULP for $</td>
<td>y</td>
</tr>
<tr>
<td>OpFRem(x, y)</td>
<td>Inherited from $x - y \times \text{trunc}(x/y)$.</td>
<td></td>
</tr>
<tr>
<td>OpFMod(x, y)</td>
<td>Inherited from $x - y \times \text{floor}(x/y)$.</td>
<td></td>
</tr>
<tr>
<td>conversions between types</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
</tbody>
</table>

**Note**

The OpFRem and OpFMod instructions use cheap approximations of remainder, and the error can be large due to the discontinuity in trunc() and floor(). This can produce mathematically unexpected results in some cases, such as FMod(x,x) computing $x$ rather than $0$, and can also cause the result to have a different sign than the infinitely precise result.

Table 64. Precision of GLSL.std.450 Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Single precision, unless decorated with RelaxedPrecision</th>
<th>Half precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>fma()</td>
<td>Inherited from OpFMul followed by OpFAdd.</td>
<td></td>
</tr>
<tr>
<td>exp(x), exp2(x)</td>
<td>3 + 2 ×</td>
<td>x</td>
</tr>
<tr>
<td>Instruction</td>
<td>Single precision, unless decorated with RelaxedPrecision</td>
<td>Half precision</td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>( \log(), \log_2() )</td>
<td>3 ULP outside the range ([0.5, 2.0]). Absolute error (&lt; 2^{-21}) inside the range ([0.5, 2.0]).</td>
<td>3 ULP outside the range ([0.5, 2.0]). Absolute error (&lt; 2^{-7}) inside the range ([0.5, 2.0]).</td>
</tr>
<tr>
<td>( \text{pow}(x, y) )</td>
<td>Inherited from ( \exp_2(y \times \log_2(x)) ).</td>
<td></td>
</tr>
<tr>
<td>( \text{sqrt}() )</td>
<td>Inherited from ( 1.0 / \text{inversesqrt}() ).</td>
<td></td>
</tr>
<tr>
<td>( \text{inversesqrt}() )</td>
<td>2 ULP.</td>
<td></td>
</tr>
<tr>
<td>( \text{radians}(x) )</td>
<td>Inherited from ( x \times \frac{n}{180} ).</td>
<td></td>
</tr>
<tr>
<td>( \text{degrees}(x) )</td>
<td>Inherited from ( x \times \frac{180}{n} ).</td>
<td></td>
</tr>
<tr>
<td>( \sin() )</td>
<td>Absolute error ( \leq 2^{-13} ) inside the range ([-\pi, \pi]).</td>
<td>Absolute error ( \leq 2^{-7} ) inside the range ([-\pi, \pi]).</td>
</tr>
<tr>
<td>( \cos() )</td>
<td>Absolute error ( \leq 2^{-13} ) inside the range ([-\pi, \pi]).</td>
<td>Absolute error ( \leq 2^{-7} ) inside the range ([-\pi, \pi]).</td>
</tr>
<tr>
<td>( \tan() )</td>
<td>Inherited from ( \frac{\sin()}{\cos()} ).</td>
<td></td>
</tr>
<tr>
<td>( \text{asin}(x) )</td>
<td>Inherited from ( \text{atan2}(x, \text{sqrt}(1.0 - x \times x)) ).</td>
<td></td>
</tr>
<tr>
<td>( \text{acos}(x) )</td>
<td>Inherited from ( \text{atan2}(\text{sqrt}(1.0 - x \times x), x) ).</td>
<td></td>
</tr>
<tr>
<td>( \text{atan}, \text{atan2}() )</td>
<td>4096 ULP</td>
<td>5 ULP.</td>
</tr>
<tr>
<td>( \sinh(x) )</td>
<td>Inherited from ( (\exp(x) - \exp(-x)) \times 0.5 ).</td>
<td></td>
</tr>
<tr>
<td>( \cosh(x) )</td>
<td>Inherited from ( (\exp(x) + \exp(-x)) \times 0.5 ).</td>
<td></td>
</tr>
<tr>
<td>( \tanh() )</td>
<td>Inherited from ( \frac{\sinh()}{\cosh()} ).</td>
<td></td>
</tr>
<tr>
<td>( \text{asinh}(x) )</td>
<td>Inherited from ( \log(x + \text{sqrt}(x \times x + 1.0)) ).</td>
<td></td>
</tr>
<tr>
<td>( \text{acosh}(x) )</td>
<td>Inherited from ( \log(x + \text{sqrt}(x \times x - 1.0)) ).</td>
<td></td>
</tr>
<tr>
<td>( \text{atanh}(x) )</td>
<td>Inherited from ( \log(\frac{1.0 + x}{1.0 - x}) \times 0.5 ).</td>
<td></td>
</tr>
<tr>
<td>( \text{frexp}() )</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>( \text{ldexp}() )</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>( \text{length}(x) )</td>
<td>Inherited from ( \text{sqrt}(\text{dot}(x, x)) ).</td>
<td></td>
</tr>
<tr>
<td>( \text{distance}(x, y) )</td>
<td>Inherited from ( \text{length}(x - y) ).</td>
<td></td>
</tr>
<tr>
<td>( \text{normalize}(x) )</td>
<td>Inherited from ( \text{OpFSub}(\text{OpFMul}, \text{OpFMul}) ).</td>
<td></td>
</tr>
<tr>
<td>( \text{faceforward}(N, I, \text{NRef}) )</td>
<td>Inherited from ( \text{dot}(\text{NRef}, I) &lt; 0.0 \ ? \text{-N} ).</td>
<td></td>
</tr>
<tr>
<td>( \text{reflect}(x, y) )</td>
<td>Inherited from ( x - 2.0 \times \text{dot}(y, x) \times y ).</td>
<td></td>
</tr>
<tr>
<td>( \text{refract}(I, N, \text{eta}) )</td>
<td>Inherited from ( k &lt; 0.0 \ ? \text{(eta} \times \text{dot}(N, I) + \text{sqrt}(k)) \times N ), where ( k = 1 - \text{eta} \times \text{eta} \times (1.0 - \text{dot}(N, I) \times \text{dot}(N, I)) ).</td>
<td></td>
</tr>
<tr>
<td>( \text{round} )</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
</tbody>
</table>
### Instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Single precision, unless decorated with RelaxedPrecision</th>
<th>Half precision</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>roundEven</code></td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td><code>trunc</code></td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td><code>fabs</code></td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td><code>fsign</code></td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td><code>floor</code></td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td><code>ceil</code></td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td><code>frac</code></td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td><code>modf</code></td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td><code>fmin</code></td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td><code>fmax</code></td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td><code>fclamp</code></td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td><code>fmix(x, y, a)</code></td>
<td>Inherited from $x \times (1.0 - a) + y \times a$.</td>
<td></td>
</tr>
<tr>
<td><code>step</code></td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td><code>smoothStep(edge0, edge1, x)</code></td>
<td>Inherited from $t \times t \times (3.0 - 2.0 \times t)$, where $t = clamp(\frac{x - edge0}{edge1 - edge0}, 0.0, 1.0)$.</td>
<td></td>
</tr>
<tr>
<td><code>nmin</code></td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td><code>nmax</code></td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td><code>nclamp</code></td>
<td>Correctly rounded.</td>
<td></td>
</tr>
</tbody>
</table>

GLSL.std.450 extended instructions specifically defined in terms of the above instructions inherit the above errors. GLSL.std.450 extended instructions not listed above and not defined in terms of the above have undefined precision.

For the `OpSRem` and `OpSMod` instructions, if either operand is negative the result is undefined.

**Note**

While the `OpSRem` and `OpSMod` instructions are supported by the Vulkan environment, they require non-negative values and thus do not enable additional functionality beyond what `OpUMod` provides.

### Signedness of SPIR-V Image Accesses

SPIR-V associates a signedness with all integer image accesses. This is required in certain parts of the SPIR-V and the Vulkan image access pipeline to ensure defined results. The signedness is determined from a combination of the access instruction’s `Image Operands` and the underlying image’s `Sampled Type` as follows:

1. If the instruction’s `Image Operands` contains the `SignExtend` operand then the access is signed.
2. If the instruction’s Image Operands contains the ZeroExtend operand then the access is unsigned.

3. Otherwise, the image accesses signedness matches that of the Sampled Type of the OpTypeImage being accessed.

Image Format and Type Matching

When specifying the Image Format of an OpTypeImage, the converted bit width and type, as shown in the table below, must match the Sampled Type. The signedness must match the signedness of any access to the image.

Note
Formatted accesses are always converted from a shader readable type to the resource’s format or vice versa via Format Conversion for reads and Texel Output Format Conversion for writes. As such, the bit width and format below do not necessarily match 1:1 with what might be expected for some formats.

For a given Image Format, the Sampled Type must be the type described in the Type column of the below table, with its Literal Width set to that in the Bit Width column. Every access that is made to the image must have a signedness equal to that in the Signedness column (where applicable).

<table>
<thead>
<tr>
<th>Image Format</th>
<th>Type</th>
<th>Bit Width</th>
<th>Signedness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown</td>
<td>Any</td>
<td>Any</td>
<td>Any</td>
</tr>
<tr>
<td>Rgba32f</td>
<td>OpTypeFloat</td>
<td>32</td>
<td>N/A</td>
</tr>
<tr>
<td>Rg32f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R32f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba16f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg16f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba16Snorm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg16Snorm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16Snorm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgb10A2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R11fG11fB10f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba8Snorm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg8Snorm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8Snorm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Compatibility Between SPIR-V Image Formats And Vulkan Formats

SPIR-V Image Format values are compatible with VkFormat values as defined below:

<table>
<thead>
<tr>
<th>SPIR-V Image Format</th>
<th>OpTypeInt</th>
<th>Bit Width</th>
<th>Signedness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rgba32i</td>
<td>OpTypeInt</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>Rg32i</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R32i</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba16i</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg16i</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16i</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba8i</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg8i</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8i</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba32ui</td>
<td>OpTypeInt</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>Rg32ui</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R32ui</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba16ui</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg16ui</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16ui</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgb10a2ui</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba8ui</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg8ui</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8ui</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R64i</td>
<td>OpTypeInt</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>R64ui</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 65. SPIR-V and Vulkan Image Format Compatibility**

<table>
<thead>
<tr>
<th>SPIR-V Image Format</th>
<th>Compatible Vulkan Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown</td>
<td>Any</td>
</tr>
<tr>
<td>R8</td>
<td>VK_FORMAT_R8_UNORM</td>
</tr>
<tr>
<td>R8Snorm</td>
<td>VK_FORMAT_R8_SNORM</td>
</tr>
<tr>
<td>R8ui</td>
<td>VK_FORMAT_R8_UINT</td>
</tr>
<tr>
<td>R8i</td>
<td>VK_FORMAT_R8_SINT</td>
</tr>
<tr>
<td>Rg8</td>
<td>VK_FORMAT_R8G8_UNORM</td>
</tr>
<tr>
<td>Rg8Snorm</td>
<td>VK_FORMAT_R8G8_SNORM</td>
</tr>
<tr>
<td>Rg8ui</td>
<td>VK_FORMAT_R8G8_UINT</td>
</tr>
<tr>
<td>Rg8i</td>
<td>VK_FORMAT_R8G8_SINT</td>
</tr>
<tr>
<td>SPIR-V Image Format</td>
<td>Compatible Vulkan Format</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>Rgba8</td>
<td>VK_FORMAT_R8G8B8A8_UNORM</td>
</tr>
<tr>
<td>Rgba8Snorm</td>
<td>VK_FORMAT_R8G8B8A8_SNORM</td>
</tr>
<tr>
<td>Rgba8ui</td>
<td>VK_FORMAT_R8G8B8A8_UINT</td>
</tr>
<tr>
<td>Rgba8i</td>
<td>VK_FORMAT_R8G8B8A8_SINT</td>
</tr>
<tr>
<td>Rgb10A2</td>
<td>VK_FORMAT_A2B10G10R10_UNORM_PACK32</td>
</tr>
<tr>
<td>Rgb10a2ui</td>
<td>VK_FORMAT_A2B10G10R10_UINT_PACK32</td>
</tr>
<tr>
<td>R16</td>
<td>VK_FORMAT_R16_UNORM</td>
</tr>
<tr>
<td>R16Snorm</td>
<td>VK_FORMAT_R16_SNORM</td>
</tr>
<tr>
<td>R16ui</td>
<td>VK_FORMAT_R16_UINT</td>
</tr>
<tr>
<td>R16i</td>
<td>VK_FORMAT_R16_SINT</td>
</tr>
<tr>
<td>R16f</td>
<td>VK_FORMAT_R16_SFLOAT</td>
</tr>
<tr>
<td>Rg16</td>
<td>VK_FORMAT_R16G16_UNORM</td>
</tr>
<tr>
<td>Rg16Snorm</td>
<td>VK_FORMAT_R16G16_SNORM</td>
</tr>
<tr>
<td>Rg16ui</td>
<td>VK_FORMAT_R16G16_UINT</td>
</tr>
<tr>
<td>Rg16i</td>
<td>VK_FORMAT_R16G16_SINT</td>
</tr>
<tr>
<td>Rg16f</td>
<td>VK_FORMAT_R16G16_SFLOAT</td>
</tr>
<tr>
<td>Rgba16</td>
<td>VK_FORMAT_R16G16B16A16_UNORM</td>
</tr>
<tr>
<td>Rgba16Snorm</td>
<td>VK_FORMAT_R16G16B16A16_SNORM</td>
</tr>
<tr>
<td>Rgba16ui</td>
<td>VK_FORMAT_R16G16B16A16_UINT</td>
</tr>
<tr>
<td>Rgba16i</td>
<td>VK_FORMAT_R16G16B16A16_SINT</td>
</tr>
<tr>
<td>Rgba16f</td>
<td>VK_FORMAT_R16G16B16A16_SFLOAT</td>
</tr>
<tr>
<td>R32ui</td>
<td>VK_FORMAT_R32_UINT</td>
</tr>
<tr>
<td>R32i</td>
<td>VK_FORMAT_R32_SINT</td>
</tr>
<tr>
<td>R32f</td>
<td>VK_FORMAT_R32_SFLOAT</td>
</tr>
<tr>
<td>Rg32ui</td>
<td>VK_FORMAT_R32G32_UINT</td>
</tr>
<tr>
<td>Rg32i</td>
<td>VK_FORMAT_R32G32_SINT</td>
</tr>
<tr>
<td>Rg32f</td>
<td>VK_FORMAT_R32G32_SFLOAT</td>
</tr>
<tr>
<td>Rgba32ui</td>
<td>VK_FORMAT_R32G32B32A32_UINT</td>
</tr>
<tr>
<td>Rgba32i</td>
<td>VK_FORMAT_R32G32B32A32_SINT</td>
</tr>
<tr>
<td>Rgba32f</td>
<td>VK_FORMAT_R32G32B32A32_SFLOAT</td>
</tr>
<tr>
<td>R64ui</td>
<td>VK_FORMAT_R64_UINT</td>
</tr>
<tr>
<td>R64i</td>
<td>VK_FORMAT_R64_SINT</td>
</tr>
<tr>
<td>R11fG11fB10f</td>
<td>VK_FORMAT_B10G11R11_UFLOAT_PACK32</td>
</tr>
</tbody>
</table>
Appendix B: Memory Model

Agent

*Operation* is a general term for any task that is executed on the system.

*Note*  
An operation is by definition something that is executed. Thus if an instruction is skipped due to control flow, it does not constitute an operation.

Each operation is executed by a particular *agent*. Possible agents include each shader invocation, each host thread, and each fixed-function stage of the pipeline.

Memory Location

A *memory location* identifies unique storage for 8 bits of data. Memory operations access a set of *memory locations* consisting of one or more memory locations at a time, e.g. an operation accessing a 32-bit integer in memory would read/write a set of four memory locations. Memory operations that access whole aggregates may access any padding bytes between elements or members, but no padding bytes at the end of the aggregate. Two sets of memory locations *overlap* if the intersection of their sets of memory locations is non-empty. A memory operation must not affect memory at a memory location not within its set of memory locations.

Memory locations for buffers and images are explicitly allocated in *VkDeviceMemory* objects, and are implicitly allocated for SPIR-V variables in each shader invocation.

Allocation

The values stored in newly allocated memory locations are determined by a SPIR-V variable’s initializer, if present, or else are undefined. At the time an allocation is created there have been no memory operations to any of its memory locations. The initialization is not considered to be a memory operation.

*Note*  
For tessellation control shader output variables, a consequence of initialization not being considered a memory operation is that some implementations may need to insert a barrier between the initialization of the output variables and any reads of those variables.

Memory Operation

For an operation A and memory location M:

- *A reads* M if and only if the data stored in M is an input to A.
- *A writes* M if and only if the data output from A is stored to M.
• A accesses M if and only if it either reads or writes (or both) M.

Note
A write whose value is the same as what was already in those memory locations is still considered to be a write and has all the same effects.

Reference

A reference is an object that a particular agent can use to access a set of memory locations. On the host, a reference is a host virtual address. On the device, a reference is:

• The descriptor that a variable is bound to, for variables in Image, Uniform, or StorageBuffer storage classes. If the variable is an array (or array of arrays, etc.) then each element of the array may be a unique reference.
• The address range for a buffer in PhysicalStorageBuffer storage class, where the base of the address range is queried with vkGetBufferDeviceAddress and the length of the range is the size of the buffer.
• The variable itself for variables in other storage classes.

Two memory accesses through distinct references may require availability and visibility operations as defined below.

Program-Order

A dynamic instance of an instruction is defined in SPIR-V (https://registry.khronos.org/spir-v/specs/unified1/SPIRV.html#DynamicInstance) as a way of referring to a particular execution of a static instruction. Program-order is an ordering on dynamic instances of instructions executed by a single shader invocation:

• (Basic block): If instructions A and B are in the same basic block, and A is listed in the module before B, then the n'th dynamic instance of A is program-ordered before the n'th dynamic instance of B.
• (Branch): The dynamic instance of a branch or switch instruction is program-ordered before the dynamic instance of the OpLabel instruction to which it transfers control.
• (Call entry): The dynamic instance of an OpFunctionCall instruction is program-ordered before the dynamic instances of the OpFunctionParameter instructions and the body of the called function.
• (Call exit): The dynamic instance of the instruction following an OpFunctionCall instruction is program-ordered after the dynamic instance of the return instruction executed by the called function.
• (Transitive Closure): If dynamic instance A of any instruction is program-ordered before dynamic instance B of any instruction and B is program-ordered before dynamic instance C of any instruction then A is program-ordered before C.
• (Complete definition): No other dynamic instances are program-ordered.
For instructions executed on the host, the source language defines the program-order relation (e.g. as “sequenced-before”).

Scope

Atomic and barrier instructions include scopes which identify sets of shader invocations that must obey the requested ordering and atomicity rules of the operation, as defined below.

The various scopes are described in detail in the Shaders chapter.

Atomic Operation

An atomic operation on the device is any SPIR-V operation whose name begins with OpAtomic. An atomic operation on the host is any operation performed with an std::atomic typed object.

Each atomic operation has a memory scope and a semantics. Informally, the scope determines which other agents it is atomic with respect to, and the semantics constrains its ordering against other memory accesses. Device atomic operations have explicit scopes and semantics. Each host atomic operation implicitly uses the CrossDevice scope, and uses a memory semantics equivalent to a C++ std::memory_order value of relaxed, acquire, release, acq_rel, or seq_cst.

Two atomic operations A and B are potentially-mutually-ordered if and only if all of the following are true:

- They access the same set of memory locations.
- They use the same reference.
- A is in the instance of B’s memory scope.
- B is in the instance of A’s memory scope.
- A and B are not the same operation (irreflexive).

Two atomic operations A and B are mutually-ordered if and only if they are potentially-mutually-ordered and any of the following are true:

- A and B are both device operations.
- A and B are both host operations.
- A is a device operation, B is a host operation, and the implementation supports concurrent host- and device-atomics.

Note

If two atomic operations are not mutually-ordered, and if their sets of memory locations overlap, then each must be synchronized against the other as if they were non-atomic operations.
Scoped Modification Order

For a given atomic write A, all atomic writes that are mutually-ordered with A occur in an order known as A's scoped modification order. A's scoped modification order relates no other operations.

Note
Invocations outside the instance of A's memory scope may observe the values at A's set of memory locations becoming visible to it in an order that disagrees with the scoped modification order.

Note
It is valid to have non-atomic operations or atomics in a different scope instance to the same set of memory locations, as long as they are synchronized against each other as if they were non-atomic (if they are not, it is treated as a data race). That means this definition of A's scoped modification order could include atomic operations that occur much later, after intervening non-atomics. That is a bit non-intuitive, but it helps to keep this definition simple and non-circular.

Memory Semantics

Non-atomic memory operations, by default, may be observed by one agent in a different order than they were written by another agent.

Atomics and some synchronization operations include memory semantics, which are flags that constrain the order in which other memory accesses (including non-atomic memory accesses and availability and visibility operations) performed by the same agent can be observed by other agents, or can observe accesses by other agents.

Device instructions that include semantics are OpAtomic*, OpControlBarrier, OpMemoryBarrier, and OpMemoryNamedBarrier. Host instructions that include semantics are some std::atomic methods and memory fences.

SPIR-V supports the following memory semantics:

- Relaxed: No constraints on order of other memory accesses.
- Acquire: A memory read with this semantic performs an acquire operation. A memory barrier with this semantic is an acquire barrier.
- Release: A memory write with this semantic performs a release operation. A memory barrier with this semantic is a release barrier.
- AcquireRelease: A memory read-modify-write operation with this semantic performs both an acquire operation and a release operation, and inherits the limitations on ordering from both of those operations. A memory barrier with this semantic is both a release and acquire barrier.

Note
SPIR-V does not support “consume” semantics on the device.
The memory semantics operand also includes storage class semantics which indicate which storage classes are constrained by the synchronization. SPIR-V storage class semantics include:

- UniformMemory
- WorkgroupMemory
- ImageMemory
- OutputMemory

Each SPIR-V memory operation accesses a single storage class. Semantics in synchronization operations can include a combination of storage classes.

The UniformMemory storage class semantic applies to accesses to memory in the PhysicalStorageBuffer, Uniform and StorageBuffer storage classes. The WorkgroupMemory storage class semantic applies to accesses to memory in the Workgroup storage class. The ImageMemory storage class semantic applies to accesses to memory in the Image storage class. The OutputMemory storage class semantic applies to accesses to memory in the Output storage class.

**Note**
Informally, these constraints limit how memory operations can be reordered, and these limits apply not only to the order of accesses as performed in the agent that executes the instruction, but also to the order the effects of writes become visible to all other agents within the same instance of the instruction’s memory scope.

**Note**
Release and acquire operations in different threads can act as synchronization operations, to guarantee that writes that happened before the release are visible after the acquire. (This is not a formal definition, just an Informative forward reference.)

**Note**
The OutputMemory storage class semantic is only useful in tessellation control shaders, which is the only execution model where output variables are shared between invocations.

The memory semantics operand can also include availability and visibility flags, which apply availability and visibility operations as described in availability and visibility. The availability/visibility flags are:

- MakeAvailable: Semantics must be Release or AcquireRelease. Performs an availability operation before the release operation or barrier.
- MakeVisible: Semantics must be Acquire or AcquireRelease. Performs a visibility operation after the acquire operation or barrier.

The specifics of these operations are defined in Availability and Visibility Semantics.

Host atomic operations may support a different list of memory semantics and synchronization
operations, depending on the host architecture and source language.

**Release Sequence**

After an atomic operation A performs a release operation on a set of memory locations M, the *release sequence headed by A* is the longest continuous subsequence of A's scoped modification order that consists of:

- the atomic operation A as its first element
- atomic read-modify-write operations on M by any agent

**Note**
The atomics in the last bullet **must** be mutually-ordered with A by virtue of being in A's scoped modification order.

**Note**
This intentionally omits “atomic writes to M performed by the same agent that performed A”, which is present in the corresponding C++ definition.

**Synchronizes-With**

*Synchronizes-with* is a relation between operations, where each operation is either an atomic operation or a memory barrier (aka fence on the host).

If A and B are atomic operations, then A synchronizes-with B if and only if all of the following are true:

- A performs a release operation
- B performs an acquire operation
- A and B are mutually-ordered
- B reads a value written by A or by an operation in the release sequence headed by A

*OpControlBarrier*, *OpMemoryBarrier*, and *OpMemoryNamedBarrier* are *memory barrier* instructions in SPIR-V.

If A is a release barrier and B is an atomic operation that performs an acquire operation, then A synchronizes-with B if and only if all of the following are true:

- there exists an atomic write X (with any memory semantics)
- A is program-ordered before X
- X and B are mutually-ordered
- B reads a value written by X or by an operation in the release sequence headed by X
  - If X is relaxed, it is still considered to head a hypothetical release sequence for this rule
- A and B are in the instance of each other’s memory scopes
• X's storage class is in A's semantics.

If A is an atomic operation that performs a release operation and B is an acquire barrier, then A
synchronizes-with B if and only if all of the following are true:

• there exists an atomic read X (with any memory semantics)
• X is program-ordered before B
• X and A are mutually-ordered
• X reads a value written by A or by an operation in the release sequence headed by A
• A and B are in the instance of each other's memory scopes
• X's storage class is in B's semantics.

If A is a release barrier and B is an acquire barrier, then A synchronizes-with B if all of the
following are true:

• there exists an atomic write X (with any memory semantics)
• A is program-ordered before X
• there exists an atomic read Y (with any memory semantics)
• Y is program-ordered before B
• X and Y are mutually-ordered
• Y reads the value written by X or by an operation in the release sequence headed by X
  ◦ If X is relaxed, it is still considered to head a hypothetical release sequence for this rule
• A and B are in the instance of each other's memory scopes
• X's and Y's storage class is in A's and B's semantics.
  ◦ NOTE: X and Y must have the same storage class, because they are mutually ordered.

If A is a release barrier, B is an acquire barrier, and C is a control barrier (where A can equal C, and
B can equal C), then A synchronizes-with B if all of the following are true:

• A is program-ordered before (or equals) C
• C is program-ordered before (or equals) B
• A and B are in the instance of each other's memory scopes
• A and B are in the instance of C's execution scope

Note

This is similar to the barrier-barrier synchronization above, but with a control barrier filling the role of the relaxed atomics.

No other release and acquire barriers synchronize-with each other.
System-Synchronizes-With

System-synchronizes-with is a relation between arbitrary operations on the device or host. Certain operations system-synchronize-with each other, which informally means the first operation occurs before the second and that the synchronization is performed without using application-visible memory accesses.

If there is an execution dependency between two operations A and B, then the operation in the first synchronization scope system-synchronizes-with the operation in the second synchronization scope.

Note

This covers all Vulkan synchronization primitives, including device operations executing before a synchronization primitive is signaled, wait operations happening before subsequent device operations, signal operations happening before host operations that wait on them, and host operations happening before vkQueueSubmit. The list is spread throughout the synchronization chapter, and is not repeated here.

System-synchronizes-with implicitly includes all storage class semantics and has CrossDevice scope.

If A system-synchronizes-with B, we also say A is system-synchronized-before B and B is system-synchronized-after A.

Private vs. Non-Private

By default, non-atomic memory operations are treated as private, meaning such a memory operation is not intended to be used for communication with other agents. Memory operations with the NonPrivatePointer/NonPrivateTexel bit set are treated as non-private, and are intended to be used for communication with other agents.

More precisely, for private memory operations to be Location-Ordered between distinct agents requires using system-synchronizes-with rather than shader-based synchronization. Non-private memory operations still obey program-order.

Atomic operations are always considered non-private.

Inter-Thread-Happens-Before

Let SC be a non-empty set of storage class semantics. Then (using template syntax) operation A inter-thread-happens-before<SC> operation B if and only if any of the following is true:

- A system-synchronizes-with B
- A synchronizes-with B, and both A and B have all of SC in their semantics
- A is an operation on memory in a storage class in SC or that has all of SC in its semantics, B is a release barrier or release atomic with all of SC in its semantics, and A is program-ordered before B
• A is an acquire barrier or acquire atomic with all of SC in its semantics, B is an operation on memory in a storage class in SC or that has all of SC in its semantics, and A is program-ordered before B
• A and B are both host operations and A inter-thread-happens-before B as defined in the host language specification
• A inter-thread-happens-before<SC> some X and X inter-thread-happens-before<SC> B

**Happens-Before**

Operation A *happens-before* operation B if and only if any of the following is true:

• A is program-ordered before B
• A inter-thread-happens-before<SC> B for some set of storage classes SC

*Happens-after* is defined similarly.

> **Note**
> Unlike C++, happens-before is not always sufficient for a write to be visible to a read. Additional *availability* and *visibility* operations may be required for writes to be visible-to other memory accesses.

> **Note**
> Happens-before is not transitive, but each of program-order and inter-thread-happens-before<SC> are transitive. These can be thought of as covering the “single-threaded” case and the “multi-threaded” case, and it is not necessary (and not valid) to form chains between the two.

**Availability and Visibility**

*Availability* and *visibility* are states of a write operation, which (informally) track how far the write has permeated the system, i.e. which agents and references are able to observe the write. Availability state is per *memory domain*. Visibility state is per (agent,reference) pair. Availability and visibility states are per-memory location for each write.

Memory domains are named according to the agents whose memory accesses use the domain. Domains used by shader invocations are organized hierarchically into multiple smaller memory domains which correspond to the different *scopes*. Each memory domain is considered the *dual* of a scope, and vice versa. The memory domains defined in Vulkan include:

• *host* - accessible by host agents
• *device* - accessible by all device agents for a particular device
• *shader* - accessible by shader agents for a particular device, corresponding to the *Device* scope
• *queue family instance* - accessible by shader agents in a single queue family, corresponding to the *QueueFamily* scope.
- **workgroup instance** - accessible by shader agents in the same workgroup, corresponding to the Workgroup scope.

- **subgroup instance** - accessible by shader agents in the same subgroup, corresponding to the Subgroup scope.

The memory domains are nested in the order listed above, with memory domains later in the list nested in the domains earlier in the list.

**Note**

Memory domains do not correspond to storage classes or device-local and host-local VkDeviceMemory allocations, rather they indicate whether a write can be made visible only to agents in the same subgroup, same workgroup, in any shader invocation, or anywhere on the device, or host. The shader, queue family instance, workgroup instance, and subgroup instance domains are only used for shader-based availability/visibility operations, in other cases writes can be made available from/visible to the shader via the device domain.

*Availability operations, visibility operations, and memory domain operations* alter the state of the write operations that happen-before them, and which are included in their *source scope* to be available or visible to their *destination scope*.

- For an availability operation, the source scope is a set of (agent, reference, memory location) tuples, and the destination scope is a set of memory domains.

- For a memory domain operation, the source scope is a memory domain and the destination scope is a memory domain.

- For a visibility operation, the source scope is a set of memory domains and the destination scope is a set of (agent, reference, memory location) tuples.

How the scopes are determined depends on the specific operation. Availability and memory domain operations expand the set of memory domains to which the write is available. Visibility operations expand the set of (agent, reference, memory location) tuples to which the write is visible.

Recall that availability and visibility states are per-memory location, and let \( W \) be a write operation to one or more locations performed by agent \( A \) via reference \( R \). Let \( L \) be one of the locations written. \( (W,L) \) (the write \( W \) to \( L \)), is initially not available to any memory domain and only visible to \( (A,R,L) \). An availability operation \( AV \) that happens-after \( W \) and that includes \( (A,R,L) \) in its source scope makes \( (W,L) \) *available* to the memory domains in its destination scope.

A memory domain operation \( DOM \) that happens-after \( AV \) and for which \( (W,L) \) is available in the source scope makes \( (W,L) \) available in the destination memory domain.

A visibility operation \( VIS \) that happens-after \( AV \) (or \( DOM \)) and for which \( (W,L) \) is available in any domain in the source scope makes \( (W,L) \) *visible* to all (agent, reference, \( L \)) tuples included in its destination scope.

If write \( W_2 \) happens-after \( W \), and their sets of memory locations overlap, then \( W \) will not be available/visible to all agents/references for those memory locations that overlap (and future \( AV/DOM/VIS \) ops cannot revive \( W \)’s write to those locations).
Availability, memory domain, and visibility operations are treated like other non-atomic memory accesses for the purpose of memory semantics, meaning they can be ordered by release-acquire sequences or memory barriers.

An availability chain is a sequence of availability operations to increasingly broad memory domains, where element N+1 of the chain is performed in the dual scope instance of the destination memory domain of element N and element N happens-before element N+1. An example is an availability operation with destination scope of the workgroup instance domain that happens-before an availability operation to the shader domain performed by an invocation in the same workgroup. An availability chain AVC that happens-after W and that includes (A,R,L) in the source scope makes (W,L) available to the memory domains in its final destination scope. An availability chain with a single element is just the availability operation.

Similarly, a visibility chain is a sequence of visibility operations from increasingly narrow memory domains, where element N of the chain is performed in the dual scope instance of the source memory domain of element N+1 and element N happens-before element N+1. An example is a visibility operation with source scope of the shader domain that happens-before a visibility operation with source scope of the workgroup instance domain performed by an invocation in the same workgroup. A visibility chain VISC that happens-after AVC (or DOM) and for which (W,L) is available in any domain in the source scope makes (W,L) visible to all (agent,reference,L) tuples included in its final destination scope. A visibility chain with a single element is just the visibility operation.

**Availability, Visibility, and Domain Operations**

The following operations generate availability, visibility, and domain operations. When multiple availability/visibility/domain operations are described, they are system-synchronized-with each other in the order listed.

An operation that performs a memory dependency generates:

- If the source access mask includes `VK_ACCESS_HOST_WRITE_BIT`, then the dependency includes a memory domain operation from host domain to device domain.
- An availability operation with source scope of all writes in the first access scope of the dependency and a destination scope of the device domain.
- A visibility operation with source scope of the device domain and destination scope of the second access scope of the dependency.
- If the destination access mask includes `VK_ACCESS_HOST_READ_BIT` or `VK_ACCESS_HOST_WRITE_BIT`, then the dependency includes a memory domain operation from device domain to host domain.

`vkFlushMappedMemoryRanges` performs an availability operation, with a source scope of (agents,references) = (all host threads, all mapped memory ranges passed to the command), and destination scope of the host domain.

`vkInvalidateMappedMemoryRanges` performs a visibility operation, with a source scope of the host domain and a destination scope of (agents,references) = (all host threads, all mapped memory ranges passed to the command).
vkQueueSubmit performs a memory domain operation from host to device, and a visibility operation with source scope of the device domain and destination scope of all agents and references on the device.

**Availability and Visibility Semantics**

A memory barrier or atomic operation via agent A that includes MakeAvailable in its semantics performs an availability operation whose source scope includes agent A and all references in the storage classes in that instruction's storage class semantics, and all memory locations, and whose destination scope is a set of memory domains selected as specified below. The implicit availability operation is program-ordered between the barrier or atomic and all other operations program-ordered before the barrier or atomic.

A memory barrier or atomic operation via agent A that includes MakeVisible in its semantics performs a visibility operation whose source scope is a set of memory domains selected as specified below, and whose destination scope includes agent A and all references in the storage classes in that instruction's storage class semantics, and all memory locations. The implicit visibility operation is program-ordered between the barrier or atomic and all other operations program-ordered after the barrier or atomic.

The memory domains are selected based on the memory scope of the instruction as follows:

- **Device** scope uses the shader domain
- **QueueFamily** scope uses the queue family instance domain
- **Workgroup** scope uses the workgroup instance domain
- **Subgroup** uses the subgroup instance domain
- **Invocation** perform no availability/visibility operations.

When an availability operation performed by an agent A includes a memory domain D in its destination scope, where D corresponds to scope instance S, it also includes the memory domains that correspond to each smaller scope instance S' that is a subset of S and that includes A. Similarly for visibility operations.

**Per-Instruction Availability and Visibility Semantics**

A memory write instruction that includes MakePointerAvailable, or an image write instruction that includes MakeTexelAvailable, performs an availability operation whose source scope includes the agent and reference used to perform the write and the memory locations written by the instruction, and whose destination scope is a set of memory domains selected by the Scope operand specified in Availability and Visibility Semantics. The implicit availability operation is program-ordered between the write and all other operations program-ordered after the write.

A memory read instruction that includes MakePointerVisible, or an image read instruction that includes MakeTexelVisible, performs a visibility operation whose source scope is a set of memory domains selected by the Scope operand as specified in Availability and Visibility Semantics, and whose destination scope includes the agent and reference used to perform the read and the memory locations read by the instruction. The implicit visibility operation is program-ordered
between read and all other operations program-ordered before the read.

**Note**

Although reads with per-instruction visibility only perform visibility ops from the shader or workgroup instance or subgroup instance domain, they will also see writes that were made visible via the device domain, i.e. those writes previously performed by non-shader agents and made visible via API commands.

**Note**

It is expected that all invocations in a subgroup execute on the same processor with the same path to memory, and thus availability and visibility operations with subgroup scope can be expected to be “free”.

### Location-Ordered

Let X and Y be memory accesses to overlapping sets of memory locations M, where X != Y. Let \((A_X, R_X)\) be the agent and reference used for X, and \((A_Y, R_Y)\) be the agent and reference used for Y. For now, let “→” denote happens-before and “→rcpo” denote the reflexive closure of program-ordered before.

If \(D_1\) and \(D_2\) are different memory domains, then let DOM\((D_1, D_2)\) be a memory domain operation from \(D_1\) to \(D_2\). Otherwise, let DOM\((D, D)\) be a placeholder such that \(X \rightarrow \text{DOM}(D, D) \rightarrow Y\) if and only if \(X \rightarrow Y\).

X is **location-ordered** before Y for a location L in M if and only if any of the following is true:

- \(A_X == A_Y\) and \(R_X == R_Y\) and \(X \rightarrow Y\)
  - NOTE: this case means no availability/visibility ops are required when it is the same (agent,reference).
- X is a read, both X and Y are non-private, and \(X \rightarrow Y\)
- X is a read, and X (transitively) system-synchronizes with Y
- If \(R_X == R_Y\) and \(A_X\) and \(A_Y\) access a common memory domain D (e.g. are in the same workgroup instance if D is the workgroup instance domain), and both X and Y are non-private:
  - X is a write, Y is a write, AVC\((A_X, R_X, D, L)\) is an availability chain making \((X, L)\) available to domain D, and \(X \rightarrow \text{rcpo} \text{AVC}(A_X, R_X, D, L) \rightarrow Y\)
  - X is a write, Y is a read, AVC\((A_X, R_X, D, L)\) is an availability chain making \((X, L)\) available to domain D, VISC\((A_Y, R_Y, D, L)\) is a visibility chain making writes to L available in domain D visible to Y, and \(X \rightarrow \text{rcpo} \text{AVC}(A_X, R_X, D, L) \rightarrow \text{VISC}(A_Y, R_Y, D, L) \rightarrow \text{rcpo} Y\)
  - If
    ```
    VkPhysicalDeviceVulkanMemoryModelFeatures
    ::vulkanMemoryModelAvailabilityVisibilityChains
    is VK_FALSE,
    ```
    then AVC and VISC **must** each only have a single element in the chain, in each sub-bullet above.
- Let \(D_X\) and \(D_Y\) each be either the device domain or the host domain, depending on whether \(A_X\) and \(A_Y\) execute on the device or host:
  - X is a write and Y is a write, and \(X \rightarrow \text{AV}(A_X, R_X, D_X, L) \rightarrow \text{DOM}(D_X, D_Y) \rightarrow Y\)
Data Race

Let X and Y be operations that access overlapping sets of memory locations M, where X ≠ Y, and at least one of X and Y is a write, and X and Y are not mutually-ordered atomic operations. If there does not exist a location-ordered relation between X and Y for each location in M, then there is a data race.

Applications must ensure that no data races occur during the execution of their application.

Visible-To

Let X be a write and Y be a read whose sets of memory locations overlap, and let M be the set of memory locations that overlap. Let M₂ be a non-empty subset of M. Then X is visible-to Y for memory locations M₂ if and only if all of the following are true:

• X is location-ordered before Y for each location L in M₂.
• There does not exist another write Z to any location L in M₂ such that X is location-ordered before Z for location L and Z is location-ordered before Y for location L.

If X is visible-to Y, then Y reads the value written by X for locations M₂.

Acyclicity

Reads-from is a relation between operations, where the first operation is a write, the second operation is a read, and the second operation reads the value written by the first operation. From-reads is a relation between operations, where the first operation is a read, the second operation is a write, and the first operation reads a value written earlier than the second operation in the second operation’s scoped modification order (or the first operation reads from the initial value, and the
second operation is any write to the same locations).

Then the implementation **must** guarantee that no cycles exist in the union of the following relations:

- location-ordered
- scoped modification order (over all atomic writes)
- reads-from
- from-reads

*Note*
This is a “consistency” axiom, which informally guarantees that sequences of operations cannot violate causality.

**Scoped Modification Order Coherence**

Let A and B be mutually-ordered atomic operations, where A is location-ordered before B. Then the following rules are a consequence of acyclicity:

- If A and B are both reads and A does not read the initial value, then the write that A takes its value from **must** be earlier in its own scoped modification order than (or the same as) the write that B takes its value from (no cycles between location-order, reads-from, and from-reads).
- If A is a read and B is a write and A does not read the initial value, then A **must** take its value from a write earlier than B in B's scoped modification order (no cycles between location-order, scope modification order, and reads-from).
- If A is a write and B is a read, then B **must** take its value from A or a write later than A in A's scoped modification order (no cycles between location-order, scoped modification order, and from-reads).
- If A and B are both writes, then A **must** be earlier than B in A's scoped modification order (no cycles between location-order and scoped modification order).
- If A is a write and B is a read-modify-write and B reads the value written by A, then B comes immediately after A in A's scoped modification order (no cycles between scoped modification order and from-reads).

**Shader I/O**

If a shader invocation A in a shader stage other than *Vertex* performs a memory read operation X from an object in storage class *Input*, then X is system-synchronized-after all writes to the corresponding *Output* storage variable(s) in the shader invocation(s) that contribute to generating invocation A, and those writes are all visible-to X.

*Note*
It is not necessary for the upstream shader invocations to have completed execution, they only need to have generated the output that is being read.
Deallocation

A call to `vkFreeMemory` must happen-after all memory operations on all memory locations in that `VkDeviceMemory` object.

**Note**

Normally, device memory operations in a given queue are synchronized with `vkFreeMemory` by having a host thread wait on a fence signalled by that queue, and the wait happens-before the call to `vkFreeMemory` on the host.

The deallocation of SPIR-V variables is managed by the system and happens-after all operations on those variables.

**Descriptions (Informative)**

This subsection offers more easily understandable consequences of the memory model for app/compiler developers.

Let SC be the storage class(es) specified by a release or acquire operation or barrier.

- An atomic write with release semantics must not be reordered against any read or write to SC that is program-ordered before it (regardless of the storage class the atomic is in).
- An atomic read with acquire semantics must not be reordered against any read or write to SC that is program-ordered after it (regardless of the storage class the atomic is in).
- Any write to SC program-ordered after a release barrier must not be reordered against any read or write to SC program-ordered before that barrier.
- Any read from SC program-ordered before an acquire barrier must not be reordered against any read or write to SC program-ordered after the barrier.

A control barrier (even if it has no memory semantics) must not be reordered against any memory barriers.

This memory model allows memory accesses with and without availability and visibility operations, as well as atomic operations, all to be performed on the same memory location. This is critical to allow it to reason about memory that is reused in multiple ways, e.g. across the lifetime of different shader invocations or draw calls. While GLSL (and legacy SPIR-V) applies the “coherent” decoration to variables (for historical reasons), this model treats each memory access instruction as having optional implicit availability/visibility operations. GLSL to SPIR-V compilers should map all (non-atomic) operations on a coherent variable to `Make{Pointer,Texel}{Available}{Visible}` flags in this model.

Atomic operations implicitly have availability/visibility operations, and the scope of those operations is taken from the atomic operation’s scope.
Tessellation Output Ordering

For SPIR-V that uses the Vulkan Memory Model, the OutputMemory storage class is used to synchronize accesses to tessellation control output variables. For legacy SPIR-V that does not enable the Vulkan Memory Model via OpMemoryModel, tessellation outputs can be ordered using a control barrier with no particular memory scope or semantics, as defined below.

Let X and Y be memory operations performed by shader invocations A_x and A_y. Operation X is tessellation-output-ordered before operation Y if and only if all of the following are true:

- There is a dynamic instance of an OpControlBarrier instruction C such that X is program-ordered before C in A_x and C is program-ordered before Y in A_y.
- A_x and A_y are in the same instance of C’s execution scope.

If shader invocations A_x and A_y in the TessellationControl execution model execute memory operations X and Y, respectively, on the Output storage class, and X is tessellation-output-ordered before Y with a scope of Workgroup, then X is location-ordered before Y, and if X is a write and Y is a read then X is visible-to Y.
Appendix C: Compressed Image Formats

The compressed texture formats used by Vulkan are described in the specifically identified sections of the Khronos Data Format Specification, version 1.3.

Unless otherwise described, the quantities encoded in these compressed formats are treated as normalized, unsigned values.

Those formats listed as sRGB-encoded have in-memory representations of R, G and B components which are nonlinearly-encoded as R', G', and B'; any alpha component is unchanged. As part of filtering, the nonlinear R', G', and B' values are converted to linear R, G, and B components; any alpha component is unchanged. The conversion between linear and nonlinear encoding is performed as described in the “KHR_DF_TRANSFER_SRGB” section of the Khronos Data Format Specification.
Block-Compressed Image Formats

BC1, BC2 and BC3 formats are described in “S3TC Compressed Texture Image Formats” chapter of the Khronos Data Format Specification. BC4 and BC5 are described in the “RGTC Compressed Texture Image Formats” chapter. BC6H and BC7 are described in the “BPTC Compressed Texture Image Formats” chapter.

Table 66. Mapping of Vulkan BC formats to descriptions

<table>
<thead>
<tr>
<th>VkFormat</th>
<th>Khronos Data Format Specification description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formats described in the “S3TC Compressed Texture Image Formats” chapter</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGB_UNORM_BLOCK</td>
<td>BC1 with no alpha</td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGB_SRGB_BLOCK</td>
<td>BC1 with no alpha, sRGB-encoded</td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGBA_UNORM_BLOCK</td>
<td>BC1 with alpha</td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGBA_SRGB_BLOCK</td>
<td>BC1 with alpha, sRGB-encoded</td>
</tr>
<tr>
<td>VK_FORMAT_BC2_UNORM_BLOCK</td>
<td>BC2</td>
</tr>
<tr>
<td>VK_FORMAT_BC2_SRGB_BLOCK</td>
<td>BC2, sRGB-encoded</td>
</tr>
<tr>
<td>VK_FORMAT_BC3_UNORM_BLOCK</td>
<td>BC3</td>
</tr>
<tr>
<td>VK_FORMAT_BC3_SRGB_BLOCK</td>
<td>BC3, sRGB-encoded</td>
</tr>
<tr>
<td>Formats described in the “RGTC Compressed Texture Image Formats” chapter</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC4_UNORM_BLOCK</td>
<td>BC4 unsigned</td>
</tr>
<tr>
<td>VK_FORMAT_BC4_SNORM_BLOCK</td>
<td>BC4 signed</td>
</tr>
<tr>
<td>VK_FORMAT_BC5_UNORM_BLOCK</td>
<td>BC5 unsigned</td>
</tr>
<tr>
<td>VK_FORMAT_BC5_SNORM_BLOCK</td>
<td>BC5 signed</td>
</tr>
<tr>
<td>Formats described in the “BPTC Compressed Texture Image Formats” chapter</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC6H_UFLOAT_BLOCK</td>
<td>BC6H (unsigned version)</td>
</tr>
<tr>
<td>VK_FORMAT_BC6H_SFLOAT_BLOCK</td>
<td>BC6H (signed version)</td>
</tr>
<tr>
<td>VK_FORMAT_BC7_UNORM_BLOCK</td>
<td>BC7</td>
</tr>
<tr>
<td>VK_FORMAT_BC7_SRGB_BLOCK</td>
<td>BC7, sRGB-encoded</td>
</tr>
</tbody>
</table>
ETC Compressed Image Formats

The following formats are described in the “ETC2 Compressed Texture Image Formats” chapter of the [Khronos Data Format Specification](https://www.khronos.org/khronos/openvg/khronos_specification/ETC2/ETC2_vulkan.png).

<table>
<thead>
<tr>
<th>VkFormat</th>
<th>Khronos Data Format Specification description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK</td>
<td>RGB ETC2</td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK</td>
<td>RGB ETC2 with sRGB encoding</td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK</td>
<td>RGB ETC2 with punch-through alpha</td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK</td>
<td>RGB ETC2 with punch-through alpha and sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK</td>
<td>RGBA ETC2</td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK</td>
<td>RGBA ETC2 with sRGB encoding</td>
</tr>
<tr>
<td>VK_FORMAT_EAC_R11_UNORM_BLOCK</td>
<td>Unsigned R11 EAC</td>
</tr>
<tr>
<td>VK_FORMAT_EAC_R11_SNORM_BLOCK</td>
<td>Signed R11 EAC</td>
</tr>
<tr>
<td>VK_FORMAT_EAC_R11G11_UNORM_BLOCK</td>
<td>Unsigned RG11 EAC</td>
</tr>
<tr>
<td>VK_FORMAT_EAC_R11G11_SNORM_BLOCK</td>
<td>Signed RG11 EAC</td>
</tr>
</tbody>
</table>
## ASTC Compressed Image Formats

ASTC formats are described in the “ASTC Compressed Texture Image Formats” chapter of the Khronos Data Format Specification.

Table 68. Mapping of Vulkan ASTC formats to descriptions

<table>
<thead>
<tr>
<th>VkFormat</th>
<th>Compressed texel block dimensions</th>
<th>Requested mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_ASTC_4x4_UNORM_BLOCK</td>
<td>4 x 4</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_4x4_SRGB_BLOCK</td>
<td>4 x 4</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x4_UNORM_BLOCK</td>
<td>5 x 4</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x4_SRGB_BLOCK</td>
<td>5 x 4</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x5_UNORM_BLOCK</td>
<td>5 x 5</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x5_SRGB_BLOCK</td>
<td>5 x 5</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x5_UNORM_BLOCK</td>
<td>6 x 5</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x5_SRGB_BLOCK</td>
<td>6 x 5</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x6_UNORM_BLOCK</td>
<td>6 x 6</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x6_SRGB_BLOCK</td>
<td>6 x 6</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x5_UNORM_BLOCK</td>
<td>8 x 5</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x5_SRGB_BLOCK</td>
<td>8 x 5</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x6_UNORM_BLOCK</td>
<td>8 x 6</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x6_SRGB_BLOCK</td>
<td>8 x 6</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x8_UNORM_BLOCK</td>
<td>8 x 8</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x8_SRGB_BLOCK</td>
<td>8 x 8</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x5_UNORM_BLOCK</td>
<td>10 x 5</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x5_SRGB_BLOCK</td>
<td>10 x 5</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x6_UNORM_BLOCK</td>
<td>10 x 6</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x6_SRGB_BLOCK</td>
<td>10 x 6</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x8_UNORM_BLOCK</td>
<td>10 x 8</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x8_SRGB_BLOCK</td>
<td>10 x 8</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x10_UNORM_BLOCK</td>
<td>10 x 10</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x10_SRGB_BLOCK</td>
<td>10 x 10</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x10_UNORM_BLOCK</td>
<td>12 x 10</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x10_SRGB_BLOCK</td>
<td>12 x 10</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x12_UNORM_BLOCK</td>
<td>12 x 12</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VkFormat</td>
<td>Compressed texel block dimensions</td>
<td>Requested mode</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x12_SRGB_BLOCK</td>
<td>12 × 12</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK</td>
<td>4 × 4</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK</td>
<td>5 × 4</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK</td>
<td>5 × 5</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK</td>
<td>6 × 5</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK</td>
<td>6 × 6</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK</td>
<td>8 × 5</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK</td>
<td>8 × 6</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK</td>
<td>8 × 8</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK</td>
<td>10 × 5</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK</td>
<td>10 × 6</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK</td>
<td>10 × 8</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK</td>
<td>10 × 10</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK</td>
<td>12 × 10</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK</td>
<td>12 × 12</td>
<td>HDR</td>
</tr>
</tbody>
</table>

ASTC textures containing HDR block encodings **should** be passed to the API using an ASTC SFLOAT texture format.

**Note**

An HDR block in a texture passed using a LDR UNORM format will return the appropriate ASTC error color if the implementation supports only the ASTC LDR profile, but may result in either the error color or a decompressed HDR color if the implementation supports HDR decoding.

The ASTC decode mode is decode_float16.

Note that an implementation **may** use HDR mode when linear LDR mode is requested.
Appendix D: Core Revisions (Informative)

New minor versions of the Vulkan API are defined periodically by the Khronos Vulkan Working Group. These consist of some amount of additional functionality added to the core API, potentially including both new functionality and functionality promoted from extensions.

It is possible to build the specification for earlier versions, but to aid readability of the latest versions, this appendix gives an overview of the changes as compared to earlier versions.

Version 1.3

Vulkan Version 1.3 promoted a number of key extensions into the core API:

- VK_KHR_copy_commands2
- VK_KHR_dynamic_rendering
- VK_KHR_format_feature_flags2
- VK_KHR_maintenance4
- VK_KHR_shader_integer_dot_product
- VK_KHR_shader_non_semantic_info
- VK_KHR_shader_terminate_invocation
- VK_KHR_synchronization2
- VK_KHR_zero_initialize_workgroup_memory
- VK_EXT_4444_formats
- VK_EXT_extended_dynamic_state
- VK_EXT_extended_dynamic_state2
- VK_EXT_image_robustness
- VK_EXT_inline_uniform_block
- VK_EXT_pipeline_creation_cache_control
- VK_EXT_pipeline_creation_feedback
- VK_EXT_private_data
- VK_EXT_shader_demote_to_helper_invocation
- VK_EXT_subgroup_size_control
- VK_EXT_texel_buffer_alignment
- VK_EXT_texture_compression_astc_hdr
- VK_EXT_tooling_info
- VK_EXT_ycbcr_2plane_444_formats
All differences in behavior between these extensions and the corresponding Vulkan 1.3 functionality are summarized below.

**Differences relative to VK_EXT_4444_formats**

If the VK_EXT_4444_formats extension is not supported, support for all formats defined by it are optional in Vulkan 1.3. There are no members in the VkPhysicalDeviceVulkan13Features structure corresponding to the VkPhysicalDevice4444FormatsFeaturesEXT structure.

**Differences relative to VK_EXT_extended_dynamic_state**

All dynamic state enumerants and entry points defined by VK_EXT_extended_dynamic_state are required in Vulkan 1.3. There are no members in the VkPhysicalDeviceVulkan13Features structure corresponding to the VkPhysicalDeviceExtendedDynamicStateFeaturesEXT structure.

**Differences relative to VK_EXT_extended_dynamic_state2**

The optional dynamic state enumerants and entry points defined by VK_EXT_extended_dynamic_state2 for patch control points and logic op are not promoted in Vulkan 1.3. There are no members in the VkPhysicalDeviceVulkan13Features structure corresponding to the VkPhysicalDeviceExtendedDynamicState2FeaturesEXT structure.

**Differences relative to VK_EXT_texel_buffer_alignment**

The more specific alignment requirements defined by VkPhysicalDeviceTexelBufferAlignmentProperties are required in Vulkan 1.3. There are no members in the VkPhysicalDeviceVulkan13Features structure corresponding to the VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT structure.

**Differences relative to VK_EXT_texture_compression_astc_hdr**

If the VK_EXT_texture_compression_astc_hdr extension is not supported, support for all formats defined by it are optional in Vulkan 1.3. The textureCompressionASTC_HDR member of VkPhysicalDeviceVulkan13Features indicates whether a Vulkan 1.3 implementation supports these formats.

**Differences relative to VK_EXT_ycbcr_2plane_444_formats**

If the VK_EXT_ycbcr_2plane_444_formats extension is not supported, support for all formats defined by it are optional in Vulkan 1.3. There are no members in the VkPhysicalDeviceVulkan13Features structure corresponding to the VkPhysicalDeviceYcbcr2Plane444FormatsFeaturesEXT structure.

**Additional Vulkan 1.3 Feature Support**

In addition to the promoted extensions described above, Vulkan 1.3 added required support for:

- SPIR-V version 1.6
  - SPIR-V 1.6 deprecates (but does not remove) the WorkgroupSize decoration.
• The `bufferDeviceAddress` feature which indicates support for accessing memory in shaders as storage buffers via `vkGetBufferDeviceAddress`.

• The `vulkanMemoryModel` and `vulkanMemoryModelDeviceScope` features, which indicate support for the corresponding Vulkan Memory Model capabilities.

• The `maxInlineUniformTotalSize` limit is added to provide the total size of all inline uniform block bindings in a pipeline layout.

**New Macros**

• `VK_API_VERSION_1_3`

**New Base Types**

• `VkFlags64`

**New Object Types**

• `VkPrivateDataSlot`

**New Commands**

• `vkCmdBeginRendering`
• `vkCmdBindVertexBuffers2`
• `vkCmdBlitImage2`
• `vkCmdCopyBuffer2`
• `vkCmdCopyBufferToImage2`
• `vkCmdCopyImage2`
• `vkCmdCopyImageToBuffer2`
• `vkCmdEndRendering`
• `vkCmdPipelineBarrier2`
• `vkCmdResetEvent2`
• `vkCmdResolveImage2`
• `vkCmdSetCullMode`
• `vkCmdSetDepthBiasEnable`
• `vkCmdSetDepthBoundsTestEnable`
• `vkCmdSetDepthCompareOp`
• `vkCmdSetDepthTestEnable`
• `vkCmdSetDepthWriteEnable`
• `vkCmdSetEvent2`
• `vkCmdSetFrontFace`
• `vkCmdSetPrimitiveRestartEnable`
• `vkCmdSetPrimitiveTopology`
• `vkCmdSetRasterizerDiscardEnable`
• `vkCmdSetScissorWithCount`
• `vkCmdSetStencilOp`
• `vkCmdSetStencilTestEnable`
• `vkCmdSetViewportWithCount`
• `vkCmdWaitEvents2`
• `vkCmdWriteTimestamp2`
• `vkCreatePrivateDataSlot`
• `vkDestroyPrivateDataSlot`
• `vkGetDeviceBufferMemoryRequirements`
• `vkGetDeviceImageMemoryRequirements`
• `vkGetDeviceImageSparseMemoryRequirements`
• `vkGetPhysicalDeviceToolProperties`
• `vkGetPrivateData`
• `vkQueueSubmit2`
• `vkSetPrivateData`

**New Structures**

• `VkBlitImageInfo2`
• `VkBufferCopy2`
• `VkBufferImageCopy2`
• `VkBufferMemoryBarrier2`
• `VkCommandBufferSubmitInfo`
• `VkCopyBufferInfo2`
• `VkCopyBufferToImageInfo2`
• `VkCopyImageInfo2`
• `VkCopyImageToBufferInfo2`
• `VkDependencyInfo`
• `VkDeviceBufferMemoryRequirements`
• `VkDeviceImageMemoryRequirements`
• `VkImageBlit2`
• `VkImageCopy2`
• `VkImageMemoryBarrier2`
• VkImageResolve2
• VkPhysicalDeviceToolProperties
• VkPipelineCreationFeedback
• VkPrivateDataSlotCreateInfo
• VkRenderingAttachmentCreateInfo
• VkRenderingInfo
• VkResolveImageCreateInfo2
• VkSemaphoreSubmitInfo
• VkSubmitInfo2

• Extending VkCommandBufferInheritanceInfo:
  ◦ VkCommandBufferInheritanceRenderingInfo

• Extending VkDescriptorPoolCreateInfo:
  ◦ VkDescriptorPoolInlineUniformBlockCreateInfo

• Extending VkDeviceCreateInfo:
  ◦ VkDevicePrivateDataCreateInfo

• Extending VkFormatProperties2:
  ◦ VkFormatProperties3

• Extending VkGraphicsPipelineCreateInfo:
  ◦ VkPipelineRenderingCreateInfo

• Extending VkGraphicsPipelineCreateInfo, VkComputePipelineCreateInfo,
  VkRayTracingPipelineCreateInfoNV, VkRayTracingPipelineCreateInfoKHR:
  ◦ VkPipelineCreationFeedbackCreateInfo

• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceDynamicRenderingFeatures
  ◦ VkPhysicalDeviceImageRobustnessFeatures
  ◦ VkPhysicalDeviceInlineUniformBlockFeatures
  ◦ VkPhysicalDeviceMaintenance4Features
  ◦ VkPhysicalDevicePipelineCreationCacheControlFeatures
  ◦ VkPhysicalDevicePrivateDataFeatures
  ◦ VkPhysicalDeviceShaderDemoteToHelperInvocationFeatures
  ◦ VkPhysicalDeviceShaderIntegerDotProductFeatures
  ◦ VkPhysicalDeviceShaderTerminateInvocationFeatures
  ◦ VkPhysicalDeviceSubgroupSizeControlFeatures
  ◦ VkPhysicalDeviceSynchronization2Features
  ◦ VkPhysicalDeviceTextureCompressionASTCHDRFeatures
- VkPhysicalDeviceVulkan13Features
- VkPhysicalDeviceZeroInitializeWorkgroupMemoryFeatures

- **Extending VkPhysicalDeviceProperties2:**
  - VkPhysicalDeviceInlineUniformBlockProperties
  - VkPhysicalDeviceMaintenance4Properties
  - VkPhysicalDeviceShaderIntegerDotProductProperties
  - VkPhysicalDeviceSubgroupSizeControlProperties
  - VkPhysicalDeviceTexelBufferAlignmentProperties
  - VkPhysicalDeviceVulkan13Properties

- **Extending VkPipelineShaderStageCreateInfo:**
  - VkPipelineShaderStageRequiredSubgroupSizeCreateInfo

- **Extending VkSubpassDependency2:**
  - VkMemoryBarrier2

- **Extending VkWriteDescriptorSet:**
  - VkWriteDescriptorSetInlineUniformBlock

**New Enums**

- VkAccessFlagBits2
- VkFormatFeatureFlagBits2
- VkPipelineCreationFeedbackFlagBits
- VkPipelineStageFlagBits2
- VkRenderingFlagBits
- VkSubmitFlagBits
- VkToolPurposeFlagBits

**New Bitmasks**

- VkAccessFlags2
- VkFormatFeatureFlags2
- VkPipelineCreationFeedbackFlags
- VkPipelineStageFlags2
- VkPrivateDataSlotCreateFlags
- VkRenderingFlags
- VkSubmitFlags
- VkToolPurposeFlags
New Enum Constants

- Extending `VkAccessFlagBits`:
  - `VK_ACCESS_NONE`

- Extending `VkAttachmentStoreOp`:
  - `VK_ATTACHMENT_STORE_OP_NONE`

- Extending `VkDescriptorType`:
  - `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK`

- Extending `VkDynamicState`:
  - `VK_DYNAMIC_STATE_CULL_MODE`
  - `VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE`
  - `VK_DYNAMIC_STATE_DEPTH_BOUNDS_TEST_ENABLE`
  - `VK_DYNAMIC_STATE_DEPTH_COMPARE_OP`
  - `VK_DYNAMIC_STATE_DEPTH_TEST_ENABLE`
  - `VK_DYNAMIC_STATE_DEPTH_WRITE_ENABLE`
  - `VK_DYNAMIC_STATE_FRONT_FACE`
  - `VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE`
  - `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY`
  - `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE`
  - `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT`
  - `VK_DYNAMIC_STATE_STENCIL_OP`
  - `VK_DYNAMIC_STATE_STENCIL_TEST_ENABLE`
  - `VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE`
  - `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT`

- Extending `VkEventCreateFlagBits`:
  - `VK_EVENT_CREATE_DEVICE_ONLY_BIT`

- Extending `VkFormat`:
  - `VK_FORMAT_A4B4G4R4_UNORM_PACK16`
  - `VK_FORMAT_A4R4G4B4_UNORM_PACK16`
  - `VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK`
  - `VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK`
  - `VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK`
  - `VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK`
  - `VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK`
  - `VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK`
- **Extending** `VkImageAspectFlagBits`:
  - `VK_IMAGE_ASPECT_NONE`

- **Extending** `VkImageLayout`:
  - `VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL`
  - `VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL`

- **Extending** `VkObjectType`:
  - `VK_OBJECT_TYPE_PRIVATE_DATA_SLOT`

- **Extending** `VkPipelineCacheCreateFlagBits`:
  - `VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT`

- **Extending** `VkPipelineCreateFlagBits`:
  - `VK_PIPELINE_CREATE_EARLY_RETURN_ON_FAILURE_BIT`
  - `VK_PIPELINE_CREATE_FAIL_ON_PIPELINE_COMPILE_REQUIRED_BIT`

- **Extending** `VkPipelineShaderStageCreateFlagBits`:
  - `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT`
  - `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT`

- **Extending** `VkPipelineStageFlagBits`:
  - `VK_PIPELINE_STAGE_NONE`

- **Extending** `VkResult`:
  - `VK_PIPELINE_COMPILE_REQUIRED`

- **Extending** `VkStructureType`:
  - `VK_STRUCTURE_TYPE_BLIT_IMAGE_INFO_2`
  - `VK_STRUCTURE_TYPE_BUFFER_COPY_2`
  - `VK_STRUCTURE_TYPE_BUFFER_IMAGE_COPY_2`
- VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER_2
- VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_RENDERING_INFO
- VK_STRUCTURE_TYPE_COMMAND_BUFFER_SUBMIT_INFO
- VK_STRUCTURE_TYPE_COPY_BUFFER_INFO_2
- VK_STRUCTURE_TYPE_COPY_BUFFER_TO_IMAGE_INFO_2
- VK_STRUCTURE_TYPE_COPY_IMAGE_INFO_2
- VK_STRUCTURE_TYPE_COPY_IMAGE_TO_BUFFER_INFO_2
- VK_STRUCTURE_TYPE_DEPENDENCY_INFO
- VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_INLINE_UNIFORM_BLOCK_CREATE_INFO
- VK_STRUCTURE_TYPE_DEVICE_BUFFER_MEMORY_REQUIREMENTS
- VK_STRUCTURE_TYPE_DEVICE_IMAGE_MEMORY_REQUIREMENTS
- VK_STRUCTURE_TYPE_DEVICE_PRIVATE_DATA_CREATE_INFO
- VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_3
- VK_STRUCTURE_TYPE_IMAGE_BLIT_2
- VK_STRUCTURE_TYPE_IMAGE_COPY_2
- VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER_2
- VK_STRUCTURE_TYPE_IMAGE_RESOLVE_2
- VK_STRUCTURE_TYPE_MEMORY_BARRIER_2
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DYNAMIC_RENDERING_FEATURES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_ROBUSTNESS_FEATURES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INLINE_UNIFORM_BLOCK_FEATURES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INLINE_UNIFORM_BLOCK_PROPERTIES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_4_FEATURES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_4_PROPERTIES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PIPELINE_CREATION_CACHE_CONTROL_FEATURES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PRIVATE_DATA_FEATURES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DEMOTE_TO_HELPER_INVOCATION_FEATURES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_INTEGER.Dot_PRODUCT_FEATURES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_TERMINATE_INVOCATION.FeatureS
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_FEATURES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_PROPERTIES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SYNCHRONIZATION_2_FEATURES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_PROPERTIES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXTURE_COMPRESSION_ASTC_HDR_FEATURES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TOOL_PROPERTIES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_3_FEATURES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_3_PROPERTIES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ZERO_INITIALIZE_WORKGROUP_MEMORY_FEATURES
VK_STRUCTURE_TYPE_PIPELINE_CREATION_FEEDBACK_CREATE_INFO
VK_STRUCTURE_TYPE_PIPELINE_RENDERING_CREATE_INFO
VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_REQUIRED_SUBGROUP_SIZE_CREATE_INFO
VK_STRUCTURE_TYPE_PRIVATE_DATA_SLOT_CREATE_INFO
VK_STRUCTURE_TYPE_RENDERING_ATTACHMENT_INFO
VK_STRUCTURE_TYPE_RENDERING_INFO
VK_STRUCTURE_TYPE_RESOLVE_IMAGE_INFO_2
VK_STRUCTURE_TYPE_SEMAPHORE_SUBMIT_INFO
VK_STRUCTURE_TYPE_SUBMIT_INFO_2
VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_INLINE_UNIFORM_BLOCK

Version 1.2

Vulkan Version 1.2 promoted a number of key extensions into the core API:

- VK_KHR_8bit_storage
- VK_KHR_buffer_device_address
- VK_KHR_create_renderpass2
- VK_KHR_depth_stencil_resolve
- VK_KHR_draw_indirect_count
- VK_KHR_driver_properties
- VK_KHR_image_format_list
- VK_KHR_imageless_framebuffer
- VK_KHR_sampler_mirror_clamp_to_edge
- VK_KHR_separate_depth_stencil_layouts
- VK_KHR_shader_atomic_int64
- VK_KHR_shader_float16_int8
- VK_KHR_shader_float_controls
- VK_KHR_shader_subgroup_extended_types
- VK_KHR_spirv_1_4
- VK_KHR_timeline_semaphore
All differences in behavior between these extensions and the corresponding Vulkan 1.2 functionality are summarized below.

**Differences relative to VK_KHR_8bit_storage**

If the `VK_KHR_8bit_storage` extension is not supported, support for the SPIR-V `storageBuffer8BitAccess` capability in shader modules is optional. Support for this feature is defined by `VkPhysicalDeviceVulkan12Features::storageBuffer8BitAccess` when queried via `vkGetPhysicalDeviceFeatures2`.

**Differences relative to VK_KHR_draw_indirect_count**

If the `VK_KHR_draw_indirect_count` extension is not supported, support for the entry points `vkCmdDrawIndirectCount` and `vkCmdDrawIndexedIndirectCount` is optional. Support for this feature is defined by `VkPhysicalDeviceVulkan12Features::drawIndirectCount` when queried via `vkGetPhysicalDeviceFeatures2`.

**Differences relative to VK_KHR_sampler_mirror_clamp_to_edge**

If the `VK_KHR_sampler_mirror_clamp_to_edge` extension is not supported, support for the `VkSamplerAddressMode VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE` is optional. Support for this feature is defined by `VkPhysicalDeviceVulkan12Features::samplerMirrorClampToEdge` when queried via `vkGetPhysicalDeviceFeatures2`.

**Differences relative to VK_EXT_descriptor_indexing**

If the `VK_EXT_descriptor_indexing` extension is not supported, support for the `descriptorIndexing` feature is optional. Support for this feature is defined by `VkPhysicalDeviceVulkan12Features::descriptorIndexing` when queried via `vkGetPhysicalDeviceFeatures2`.

**Differences relative to VK_EXT_scalar_block_layout**

If the `VK_EXT_scalar_block_layout` extension is not supported, support for the `scalarBlockLayout` feature is optional. Support for this feature is defined by `VkPhysicalDeviceVulkan12Features::scalarBlockLayout` when queried via `vkGetPhysicalDeviceFeatures2`.
Differences relative to **VK_EXT_shader_viewport_index_layer**

The `ShaderViewportIndexLayerEXT` SPIR-V capability was replaced with the `ShaderViewportIndex` and `ShaderLayer` capabilities. Declaring both is equivalent to declaring `ShaderViewportIndexLayerEXT`. If the `VK_EXT_shader_viewport_index_layer` extension is not supported, support for the `ShaderViewportIndexLayerEXT` SPIR-V capability is optional. Support for this feature is defined by `VkPhysicalDeviceVulkan12Features::shaderOutputViewportIndex` and `VkPhysicalDeviceVulkan12Features::shaderOutputLayer` when queried via `vkGetPhysicalDeviceFeatures2`.

Differences relative to **VK_KHR_buffer_device_address**

If the `VK_KHR_buffer_device_address` extension is not supported, support for the `bufferDeviceAddress` feature is optional. Support for this feature is defined by `VkPhysicalDeviceVulkan12Features::bufferDeviceAddress` when queried via `vkGetPhysicalDeviceFeatures2`.

Differences relative to **VK_KHR.shader_atomic_int64**

If the `VK_KHR_shader_atomic_int64` extension is not supported, support for the `shaderBufferInt64Atomics` feature is optional. Support for this feature is defined by `VkPhysicalDeviceVulkan12Features::shaderBufferInt64Atomics` when queried via `vkGetPhysicalDeviceFeatures2`.

Differences relative to **VK_KHR.shader_float16_int8**

If the `VK_KHR_shader_float16_int8` extension is not supported, support for the `shaderFloat16` and `shaderInt8` features is optional. Support for these features are defined by `VkPhysicalDeviceVulkan12Features::shaderFloat16` and `VkPhysicalDeviceVulkan12Features::shaderInt8` when queried via `vkGetPhysicalDeviceFeatures2`.

Differences relative to **VK_KHR.vulkan_memory_model**

If the `VK_KHR.vulkan_memory_model` extension is not supported, support for the `vulkanMemoryModel` feature is optional. Support for this feature is defined by `VkPhysicalDeviceVulkan12Features::vulkanMemoryModel` when queried via `vkGetPhysicalDeviceFeatures2`.

**Additional Vulkan 1.2 Feature Support**

In addition to the promoted extensions described above, Vulkan 1.2 added support for:

- SPIR-V version 1.4.
- SPIR-V version 1.5.
- The `samplerMirrorClampToEdge` feature which indicates whether the implementation supports the `VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE` sampler address mode.
- The `ShaderNonUniform` capability in SPIR-V version 1.5.
- The `shaderOutputViewportIndex` feature which indicates that the `ShaderViewportIndex` capability can be used.
• The shaderOutputLayer feature which indicates that the ShaderLayer capability can be used.

• The subgroupBroadcastDynamicId feature which allows the “Id” operand of OpcodeGroupNonUniformBroadcast to be dynamically uniform within a subgroup, and the “Index” operand of OpcodeGroupNonUniformQuadBroadcast to be dynamically uniform within a derivative group, in shader modules of version 1.5 or higher.

• The drawIndirectCount feature which indicates whether the vkCmdDrawIndirectCount and vkCmdDrawIndexedIndirectCount functions can be used.

• The descriptorIndexing feature which indicates the implementation supports the minimum number of descriptor indexing features as defined in the Feature Requirements section.

• The samplerFilterMinmax feature which indicates whether the implementation supports the minimum number of image formats that support the VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT feature bit as defined by the filterMinmaxSingleComponentFormats property minimum requirements.

• The framebufferIntegerColorSampleCounts limit which indicates the color sample counts that are supported for all framebuffer color attachments with integer formats.

New Macros

• VK_API_VERSION_1_2

New Commands

• vkCmdBeginRenderPass2
• vkCmdDrawIndexedIndirectCount
• vkCmdDrawIndirectCount
• vkCmdEndRenderPass2
• vkCmdNextSubpass2
• vkCreateRenderPass2
• vkGetBufferDeviceAddress
• vkGetBufferOpaqueCaptureAddress
• vkGetDeviceMemoryOpaqueCaptureAddress
• vkGetSemaphoreCounterValue
• vkResetQueryPool
• vkSignalSemaphore
• vkWaitSemaphores

New Structures

• VkAttachmentDescription2
• VkAttachmentReference2
• VkBufferDeviceAddressInfo
• VkConformanceVersion
• VkDeviceMemoryOpaqueCaptureAddressInfo
• VkFramebufferAttachmentImageInfo
• VkRenderPassCreateInfo2
• VkSemaphoreSignalInfo
• VkSemaphoreWaitInfo
• VkSubpassBeginInfo
• VkSubpassDependency2
• VkSubpassDescription2
• VkSubpassEndInfo
• Extending VkAttachmentDescription2:
  ◦ VkAttachmentDescriptionStencilLayout
• Extending VkAttachmentReference2:
  ◦ VkAttachmentReferenceStencilLayout
• Extending VkBufferCreateInfo:
  ◦ VkBufferOpaqueCaptureAddressCreateInfo
• Extending VkDescriptorSetAllocateInfo:
  ◦ VkDescriptorSetVariableDescriptorCountAllocateInfo
• Extending VkDescriptorSetLayoutCreateInfo:
  ◦ VkDescriptorSetLayoutBindingFlagsCreateInfo
• Extending VkDescriptorSetLayoutSupport:
  ◦ VkDescriptorSetVariableDescriptorCountLayoutSupport
• Extending VkFramebufferCreateInfo:
  ◦ VkFramebufferAttachmentsCreateInfo
• Extending VkImageCreateInfo, VkPhysicalDeviceImageFormatInfo2:
  ◦ VkImageStencilUsageCreateInfo
• Extending VkImageCreateInfo, VkSwapchainCreateInfoKHR, VkPhysicalDeviceImageFormatInfo2:
  ◦ VkImageFormatListCreateInfo
• Extending VkMemoryAllocateInfo:
  ◦ VkMemoryOpaqueCaptureAddressAllocateInfo
• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDevice8BitStorageFeatures
  ◦ VkPhysicalDeviceBufferDeviceAddressFeatures
  ◦ VkPhysicalDeviceDescriptorIndexingFeatures
  ◦ VkPhysicalDeviceHostQueryResetFeatures
- VkPhysicalDeviceImagelessFramebufferFeatures
- VkPhysicalDeviceScalarBlockLayoutFeatures
- VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures
- VkPhysicalDeviceShaderAtomicInt64Features
- VkPhysicalDeviceShaderFloat16Int8Features
- VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures
- VkPhysicalDeviceTimelineSemaphoreFeatures
- VkPhysicalDeviceUniformBufferStandardLayoutFeatures
- VkPhysicalDeviceVulkan11Features
- VkPhysicalDeviceVulkan12Features
- VkPhysicalDeviceVulkanMemoryModelFeatures

• Extending VkPhysicalDeviceProperties2:
  - VkPhysicalDeviceDepthStencilResolveProperties
  - VkPhysicalDeviceDescriptorIndexingProperties
  - VkPhysicalDeviceDriverProperties
  - VkPhysicalDeviceFloatControlsProperties
  - VkPhysicalDeviceSamplerFilterMinmaxProperties
  - VkPhysicalDeviceTimelineSemaphoreProperties
  - VkPhysicalDeviceVulkan11Properties
  - VkPhysicalDeviceVulkan12Properties

• Extending VkRenderPassBeginInfo:
  - VkRenderPassAttachmentBeginInfo

• Extending VkSamplerCreateInfo:
  - VkSamplerReductionModeCreateInfo

• Extending VkSemaphoreCreateInfo, VkPhysicalDeviceExternalSemaphoreInfo:
  - VkSemaphoreTypeCreateInfo

• Extending VkSubmitInfo, VkBindSparseInfo:
  - VkTimelineSemaphoreSubmitInfo

• Extending VkSubpassDescription2:
  - VkSubpassDescriptionDepthStencilResolve

**New Enums**

• VkDescriptorBindingFlagBits
• VkDriverId
• VkResolveModeFlagBits
• VkSamplerReductionMode
• VkSemaphoreType
• VkSemaphoreWaitFlagBits
• VkShaderFloatControlsIndependence

New Bitmasks

• VkDescriptorBindingFlags
• VkResolveModeFlags
• VkSemaphoreWaitFlags

New Enum Constants

• VK_MAX_DRIVER_INFO_SIZE
• VK_MAX_DRIVER_NAME_SIZE

Extending VkBufferCreateFlagBits:
  ◦ VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT

Extending VkBufferUsageFlagBits:
  ◦ VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT

Extending VkDescriptorPoolCreateFlagBits:
  ◦ VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT

Extending VkDescriptorSetLayoutCreateFlagBits:
  ◦ VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT

Extending VkFormatFeatureFlagBits:
  ◦ VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT

Extending VkFramebufferCreateFlagBits:
  ◦ VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT

Extending VkImageLayout:
  ◦ VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL
  ◦ VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL
  ◦ VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL
  ◦ VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL

Extending VkMemoryAllocateFlagBits:
  ◦ VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT
  ◦ VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT

Extending VkResult:
  ◦ VK_ERROR_FRAGMENTATION
- VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS

- **Extending VkSamplerAddressMode:**
  - VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE

- **Extending VkStructureType:**
  - VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_2
  - VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_STENCIL_LAYOUT
  - VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_2
  - VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_STENCIL_LAYOUT
  - VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO
  - VK_STRUCTURE_TYPE_BUFFER_OPAQUE_CAPTURE_ADDRESS_CREATE_INFO
  - VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_BINDING_FLAGS_CREATE_INFO
  - VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_ALLOCATE_INFO
  - VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_LAYOUT_SUPPORT
  - VK_STRUCTURE_TYPE_DEVICE_MEMORY_OPAQUE_CAPTURE_ADDRESS_INFO
  - VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENTS_CREATE_INFO
  - VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENT_IMAGE_INFO
  - VK_STRUCTURE_TYPE_IMAGE_FORMAT_LIST_CREATE_INFO
  - VK_STRUCTURE_TYPE_IMAGE_STENCIL_USAGE_CREATE_INFO
  - VK_STRUCTURE_TYPE_MEMORY_OPAQUE_CAPTURE_ADDRESS_ALLOCATE_INFO
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_8BIT_STORAGE_FEATURES
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEPTH_STENCIL_RESOLVE_PROPERTIES
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_FEATURES
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_PROPERTIES
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DRIVER_PROPERTIES
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FLOAT_CONTROLS_PROPERTIES
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_HOST_QUERY_RESET_FEATURES
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGELESS_FRAMEBUFFER_FEATURES
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_FILTER_MINMAX_PROPERTIES
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SCALAR_BLOCK_LAYOUT_FEATURES
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SEPARATE_DEPTH_STENCIL_LAYOUTS_FEATURES
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_INT64_FEATURES
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_FLOAT16_INT8_FEATURES
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SUBGROUP_EXTENDED_TYPES_FEATURES
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_FEATURES
Version 1.1

Vulkan Version 1.1 promoted a number of key extensions into the core API:

- VK_KHR_16bit_storage
- VK_KHR_bind_memory2
- VK_KHR_dedicated_allocation
- VK_KHR_descriptor_update_template
- VK_KHR_device_group
- VK_KHR_device_group_creation
- VK_KHR_external_fence
- VK_KHR_external_fence_capabilities
- VK_KHR_external_memory
- VK_KHR_external_memory_capabilities
- VK_KHR_external_semaphore
You mentioned a list of Vulkan extensions, including:

- VK_KHR_external_semaphore_capabilities
- VK_KHR_get_memory_requirements2
- VK_KHR_get_physical_device_properties2
- VK_KHR_maintenance1
- VK_KHR_maintenance2
- VK_KHR_maintenance3
- VK_KHR_multiview
- VK_KHR_relaxed_block_layout
- VK_KHR_sampler_ycbcr_conversion
- VK_KHR_shader_draw_parameters
- VK_KHR_storage_buffer_storage_class
- VK_KHR_variable_pointers

All differences in behavior between these extensions and the corresponding Vulkan 1.1 functionality are summarized below.

**Differences relative to VK_KHR_16bit_storage**

If the VK_KHR_16bit_storage extension is not supported, support for the storageBuffer16BitAccess feature is optional. Support for this feature is defined by VkPhysicalDevice16BitStorageFeatures::storageBuffer16BitAccess or VkPhysicalDeviceVulkan11Features::storageBuffer16BitAccess when queried via vkGetPhysicalDeviceFeatures2.

**Differences relative to VK_KHR_sampler_ycbcr_conversion**

If the VK_KHR_sampler_ycbcr_conversion extension is not supported, support for the samplerYcbcrConversion feature is optional. Support for this feature is defined by VkPhysicalDeviceSamplerYcbcrConversionFeatures::samplerYcbcrConversion or VkPhysicalDeviceVulkan11Features::samplerYcbcrConversion when queried via vkGetPhysicalDeviceFeatures2.

**Differences relative to VK_KHR_shader_draw_parameters**

If the VK_KHR_shader_draw_parameters extension is not supported, support for the SPV_KHR_shader_draw_parameters SPIR-V extension is optional. Support for this feature is defined by VkPhysicalDeviceShaderDrawParametersFeatures::shaderDrawParameters or VkPhysicalDeviceVulkan11Features::shaderDrawParameters when queried via vkGetPhysicalDeviceFeatures2.

**Differences relative to VK_KHR_variable_pointers**

If the VK_KHR_variable_pointers extension is not supported, support for the variablePointersStorageBuffer feature is optional. Support for this feature is defined by
Additional Vulkan 1.1 Feature Support

In addition to the promoted extensions described above, Vulkan 1.1 added support for:

- The `group operations` and `subgroup scope`.
- The `protected memory` feature.
- A new command to enumerate the instance version: `vkEnumerateInstanceVersion`.
- The `VkPhysicalDeviceShaderDrawParametersFeatures` feature query struct (where the `VK_KHR_shader_draw_parameters` extension did not have one).

New Macros

- `VK_API_VERSION_1_1`

New Object Types

- `VkDescriptorUpdateTemplate`
- `VkSamplerYcbcrConversion`

New Commands

- `vkBindBufferMemory2`
- `vkBindImageMemory2`
- `vkCmdDispatchBase`
- `vkCmdSetDeviceMask`
- `vkCreateDescriptorUpdateTemplate`
- `vkCreateSamplerYcbcrConversion`
- `vkDestroyDescriptorUpdateTemplate`
- `vkDestroySamplerYcbcrConversion`
- `vkEnumerateInstanceVersion`
- `vkEnumeratePhysicalDeviceGroups`
- `vkGetBufferMemoryRequirements2`
- `vkGetDeviceGroupPeerMemoryFeatures`
- `vkGetDeviceQueue2`
- `vkGetImageMemoryRequirements2`
- `vkGetImageSparseMemoryRequirements2`
• vkGetPhysicalDeviceExternalBufferProperties
• vkGetPhysicalDeviceExternalFenceProperties
• vkGetPhysicalDeviceExternalSemaphoreProperties
• vkGetPhysicalDeviceFeatures2
• vkGetPhysicalDeviceFormatProperties2
• vkGetPhysicalDeviceImageFormatProperties2
• vkGetPhysicalDeviceMemoryProperties2
• vkGetPhysicalDeviceProperties2
• vkGetPhysicalDeviceQueueFamilyProperties2
• vkGetPhysicalDeviceSparseImageFormatProperties2
• vkTrimCommandPool
• vkUpdateDescriptorSetWithTemplate

**New Structures**

• VkBindBufferMemoryInfo
• VkBindImageMemoryInfo
• VkBufferMemoryRequirementsInfo2
• VkDescriptorSetLayoutSupport
• VkDescriptorUpdateTemplateCreateInfo
• VkDescriptorUpdateTemplateEntry
• VkDeviceQueueInfo2
• VkExternalBufferProperties
• VkExternalFenceProperties
• VkExternalMemoryProperties
• VkExternalSemaphoreProperties
• VkFormatProperties2
• VkImageFormatProperties2
• VkImageMemoryRequirementsInfo2
• VkImageSparseMemory RequirementsInfo2
• VkInputAttachmentAspectReference
• VkMemoryRequirements2
• VkPhysicalDeviceExternalBufferInfo
• VkPhysicalDeviceExternalFenceInfo
• VkPhysicalDeviceExternalSemaphoreInfo
• VkPhysicalDeviceGroupProperties
- `VkPhysicalDeviceImageFormatInfo2`
- `VkPhysicalDeviceMemoryProperties2`
- `VkPhysicalDeviceProperties2`
- `VkPhysicalDeviceSparseImageFormatInfo2`
- `VkQueueFamilyProperties2`
- `VkSamplerYcbcrConversionCreateInfo`
- `VkSparseImageFormatProperties2`
- `VkSparseImageMemoryRequirements2`

Extending `VkBindBufferMemoryInfo`:
  - `VkBindBufferMemoryDeviceGroupInfo`

Extending `VkBindImageMemoryInfo`:
  - `VkBindImageMemoryDeviceGroupInfo`
  - `VkBindImagePlaneMemoryInfo`

Extending `VkBindSparseInfo`:
  - `VkDeviceGroupBindSparseInfo`

Extending `VkBufferCreateInfo`:
  - `VkExternalMemoryBufferCreateInfo`

Extending `VkCommandBufferBeginInfo`:
  - `VkDeviceGroupCommandBufferBeginInfo`

Extending `VkDeviceCreateInfo`:
  - `VkDeviceGroupDeviceCreateInfo`
  - `VkPhysicalDeviceFeatures2`

Extending `VkFenceCreateInfo`:
  - `VkExportFenceCreateInfo`

Extending `VkImageCreateInfo`:
  - `VkExternalMemoryImageCreateInfo`

Extending `VkImageFormatProperties2`:
  - `VkExternalImageFormatProperties`
  - `VkSamplerYcbcrConversionImageFormatProperties`

Extending `VkImageMemoryRequirementsInfo2`:
  - `VkImagePlaneMemoryRequirementsInfo`

Extending `VkImageViewCreateInfo`:
  - `VkImageViewUsageCreateInfo`

Extending `VkMemoryAllocateInfo`:
  - `VkExportMemoryAllocateInfo`
• VkMemoryAllocateFlagsInfo
• VkMemoryDedicatedAllocateInfo

• Extending VkMemoryRequirements2:
  • VkMemoryDedicatedRequirements

• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  • VkPhysicalDevice16BitStorageFeatures
  • VkPhysicalDeviceMultiviewFeatures
  • VkPhysicalDeviceProtectedMemoryFeatures
  • VkPhysicalDeviceSamplerYcbcrConversionFeatures
  • VkPhysicalDeviceShaderDrawParameterFeatures
  • VkPhysicalDeviceShaderDrawParametersFeatures
  • VkPhysicalDeviceVariablePointerFeatures
  • VkPhysicalDeviceVariablePointersFeatures

• Extending VkPhysicalDeviceImageFormatInfo2:
  • VkPhysicalDeviceExternalImageFormatInfo

• Extending VkPhysicalDeviceProperties2:
  • VkPhysicalDeviceIDProperties
  • VkPhysicalDeviceMaintenance3Properties
  • VkPhysicalDeviceMultiviewProperties
  • VkPhysicalDevicePointClippingProperties
  • VkPhysicalDeviceProtectedMemoryProperties
  • VkPhysicalDeviceSubgroupProperties

• Extending VkPipelineTessellationStateCreateInfo:
  • VkPipelineTessellationDomainOriginStateCreateInfo

• Extending VkRenderPassBeginInfo, VkRenderingInfo:
  • VkDeviceGroupRenderPassBeginInfo

• Extending VkRenderPassCreateInfo:
  • VkRenderPassInputAttachmentAspectCreateInfo
  • VkRenderPassMultiviewCreateInfo

• Extending VkSamplerCreateInfo, VkImageViewCreateInfo:
  • VkSamplerYcbcrConversionInfo

• Extending VkSemaphoreCreateInfo:
  • VkExportSemaphoreCreateInfo

• Extending VkSubmitInfo:
  • VkDeviceGroupSubmitInfo
New Enums

• VkChromaLocation
• VkDescriptorUpdateTemplateType
• VkDeviceQueueCreateFlagBits
• VkExternalFenceFeatureFlagBits
• VkExternalFenceHandleTypeFlagBits
• VkExternalMemoryFeatureFlagBits
• VkExternalMemoryHandleTypeFlagBits
• VkExternalSemaphoreFeatureFlagBits
• VkExternalSemaphoreHandleTypeFlagBits
• VkFenceImportFlagBits
• VkMemoryAllocateFlagBits
• VkPeerMemoryFeatureFlagBits
• VkPointClippingBehavior
• VkSamplerYcbcrModelConversion
• VkSamplerYcbcrRange
• VkSemaphoreImportFlagBits
• VkSubgroupFeatureFlagBits
• VkTessellationDomainOrigin

New Bitmasks

• VkCommandPoolTrimFlags
• VkDescriptorUpdateTemplateCreateFlags
• VkExternalFenceFeatureFlags
• VkExternalFenceHandleTypeFlags
• VkExternalMemoryFeatureFlags
• VkExternalMemoryHandleTypeFlags
• VkExternalSemaphoreFeatureFlags
• VkExternalSemaphoreHandleTypeFlags
• VkFenceImportFlags
• VkMemoryAllocateFlags
• VkPeerMemoryFeatureFlags
• VkSemaphoreImportFlags
• VkSemaphoreImportFlagBits
New Enum Constants

- **VK_LUID_SIZE**
- **VK_MAX_DEVICE_GROUP_SIZE**
- **VK_QUEUE_FAMILY_EXTERNAL**
- Extending **VkBufferCreateFlagBits**:
  - **VK_BUFFER_CREATE_PROTECTED_BIT**
- Extending **VkCommandPoolCreateFlagBits**:
  - **VK_COMMAND_POOL_CREATE_PROTECTED_BIT**
- Extending **VkDependencyFlagBits**:
  - **VK_DEPENDENCY_DEVICE_GROUP_BIT**
  - **VK_DEPENDENCY_VIEW_LOCAL_BIT**
- Extending **VkDeviceQueueCreateFlagBits**:
  - **VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT**
- Extending **VkFormat**:
  - **VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16**
  - **VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16**
  - **VK_FORMAT_B16G16R16G16_422_UNORM**
  - **VK_FORMAT_B8G8R8G8_422_UNORM**
  - **VK_FORMAT_G10X6B10X6G10X6R10X6_2PLANE_420_UNORM_3PACK16**
  - **VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16**
  - **VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16**
  - **VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16**
  - **VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16**
  - **VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16**
  - **VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16**
  - **VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16**
  - **VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16**
  - **VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16**
  - **VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16**
  - **VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16**
  - **VK_FORMAT_G16B16G16R16_422_UNORM**
  - **VK_FORMAT_G16_B16R16_2PLANE_420_UNORM**
  - **VK_FORMAT_G16_B16R16_2PLANE_422_UNORM**
- VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM
- VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM
- VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM
- VK_FORMAT_G8B8G8R8_422_UNORM
- VK_FORMAT_G8_B8R8_2PLANE_420_UNORM
- VK_FORMAT_G8_B8R8_2PLANE_422_UNORM
- VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM
- VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM
- VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM
- VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16
- VK_FORMAT_R10X6G10X6X6B10X6A10X6_UNORM_4PACK16
- VK_FORMAT_R10X6_G10X6_UNORM_PACK16
- VK_FORMAT_R12X4G12X4B12X4A12X4_UNORM_4PACK16
- VK_FORMAT_R12X4G12X4B12X4A12X4X6B10X6A10X6_UNORM_4PACK16
- VK_FORMAT_R12X4_G12X4_UNORM_PACK16
- VK_FORMAT_R12X4_G12X4B12X4A12X4X6B10X6A10X6_UNORM_4PACK16
- VK_FORMAT_R12X4_UNORM_PACK16

- **Extending VkFormatFeatureFlagBits:**
  - VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT
  - VK_FORMAT_FEATURE_DISJOINT_BIT
  - VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT
  - VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT
  - VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT
  - VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT
  - VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT
  - VK_FORMAT_FEATURE_TRANSFER_DST_BIT
  - VK_FORMAT_FEATURE_TRANSFER_SRC_BIT

- **Extending VkImageAspectFlagBits:**
  - VK_IMAGE_ASPECT_PLANE_0_BIT
  - VK_IMAGE_ASPECT_PLANE_1_BIT
  - VK_IMAGE_ASPECT_PLANE_2_BIT

- **Extending VkImageCreateFlagBits:**
  - VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT
  - VK_IMAGE_CREATE_ALIAS_BIT
  - VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT
  - VK_IMAGE_CREATE_DISJOINT_BIT
- VK_IMAGE_CREATE_EXTENDED_USAGE_BIT
- VK_IMAGE_CREATE_PROTECTED_BIT
- VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT

**Extending VkImageLayout:**
- VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL
- VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL

**Extending VkMemoryHeapFlagBits:**
- VK_MEMORY_HEAP_MULTI_INSTANCE_BIT

**Extending VkMemoryPropertyFlagBits:**
- VK_MEMORY_PROPERTY_PROTECTED_BIT

**Extending VkObjectType:**
- VK_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE
- VK_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION

**Extending VkPipelineCreateFlagBits:**
- VK_PIPELINE_CREATE_DISPATCH_BASE
- VK_PIPELINE_CREATE_DISPATCH_BASE_BIT
- VK_PIPELINE_CREATE_VIEW_INDEX_FROM_DEVICE_INDEX_BIT

**Extending VkQueueFlagBits:**
- VK_QUEUE_PROTECTED_BIT

**Extending VkResult:**
- VK_ERROR_INVALID_EXTERNAL_HANDLE
- VK_ERROR_OUT_OF_POOL_MEMORY

**Extending VkStructureType:**
- VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORYDEVICE_GROUP_INFO
- VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO
- VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORYDEVICE_GROUP_INFO
- VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_INFO
- VK_STRUCTURE_TYPE_BIND_IMAGE_PLANE_MEMORY_INFO
- VK_STRUCTURE_TYPE_BUFFER_MEMORY_REQUIREMENTS_INFO_2
- VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_SUPPORT
- VK_STRUCTURE_TYPE_DESCRIPTORUPDATE_TEMPLATE_CREATE_INFO
- VK_STRUCTURE_TYPEDEVICEGROUPBINDSPARSE_INFO
- VK_STRUCTURE_TYPEDEVICEGROUPCOMMANDBUFFERBEGIN_INFO
- VK_STRUCTURE_TYPEDEVICEGROUPDEVICECREATE_INFO
- VK_STRUCTURE_TYPEDEVICEGROUPRENDERPASSBEGIN_INFO
- VK_STRUCTURE_TYPE_DEVICE_GROUP_SUBMIT_INFO
- VK_STRUCTURE_TYPE_DEVICE_QUEUE_INFO_2
- VK_STRUCTURE_TYPE_EXPORT_FENCE_CREATE_INFO
- VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO
- VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO
- VK_STRUCTURE_TYPE_EXTERNAL_BUFFER_PROPERTIES
- VK_STRUCTURE_TYPE_EXTERNAL_FENCE_PROPERTIES
- VK_STRUCTURE_TYPE_EXTERNAL_IMAGE_FORMAT_PROPERTIES
- VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO
- VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO
- VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES
- VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_2
- VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2
- VK_STRUCTURE_TYPE_IMAGE_MEMORY_REQUIREMENTS_INFO_2
- VK_STRUCTURE_TYPE_IMAGE_PLANE_MEMORY_REQUIREMENTS_INFO
- VK_STRUCTURE_TYPE_IMAGE_SPARSE_MEMORY_REQUIREMENTS_INFO_2
- VK_STRUCTURE_TYPE_IMAGE_VIEW_USAGE_CREATE_INFO
- VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_FLAGS_INFO
- VK_STRUCTURE_TYPE_MEMORY_DEDICATED_ALLOCATE_INFO
- VK_STRUCTURE_TYPE_MEMORY_DEDICATED_REQUIREMENTS
- VK_STRUCTURE_TYPE_MEMORY_REQUIREMENTS_2
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_16BIT_STORAGE_FEATURES
- VK_STRUCTURE_TYPE_PHYSICALDEVICE_EXTERNAL_BUFFER_INFO
- VK_STRUCTURE_TYPE_PHYSICALDEVICE_EXTERNAL_FENCE_INFO
- VK_STRUCTURE_TYPE_PHYSICALDEVICE_EXTERNAL_IMAGE_FORMAT_INFO
- VK_STRUCTURE_TYPE_PHYSICALDEVICE_EXTERNAL_SEMAPHORE_INFO
- VK_STRUCTURE_TYPE_PHYSICALDEVICE_FEATURES_2
- VK_STRUCTURE_TYPE_PHYSICALDEVICE_IMAGE_FORMAT_INFO_2
- VK_STRUCTURE_TYPE_PHYSICALDEVICE_MAINTENANCE_3_PROPERTIES
- VK_STRUCTURE_TYPE_PHYSICALDEVICE_MEMORY_PROPERTIES_2
- VK_STRUCTURE_TYPE_PHYSICALDEVICE_MULTIVIEW_FEATURES
- VK_STRUCTURE_TYPE_PHYSICALDEVICE_MULTIVIEW_PROPERTIES
- VK_STRUCTURE_TYPE_PHYSICALDEVICE_POINT_CLIPPING_PROPERTIES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_FEATURES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_PROPERTIES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_YCBCR_CONVERSION_FEATURES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DRAW_PARAMETERS_FEATURES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DRAW_PARAMETER_FEATURES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SPARSE_IMAGE_FORMAT_INFO_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_PROPERTIES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTERS_FEATURES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTER_FEATURES
VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_DOMAIN_ORIGIN_STATE_CREATE_INFO
VK_STRUCTURE_TYPE_PROTECTED_SUBMIT_INFO
VK_STRUCTURE_TYPE_QUEUE_FAMILY_PROPERTIES_2
VK_STRUCTURE_TYPE_RENDER_PASS_INPUT_ATTACHMENT_ASPECT_CREATE_INFO
VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_CREATE_INFO
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_IMAGE_FORMAT_PROPERTIES
VK_STRUCTURE_TYPE_SPARSE_IMAGE_FORMAT_PROPERTIES_2
VK_STRUCTURE_TYPE_SPARSE_IMAGE_MEMORY_REQUIREMENTS_2

Version 1.0

Vulkan Version 1.0 was the initial release of the Vulkan API.

New Macros

- VK_API_VERSION
- VK_API_VERSION_1_0
- VK_API_VERSION_MAJOR
- VK_API_VERSION_MINOR
- VK_API_VERSION_PATCH
- VK_API_VERSION_VARIANT
- VK_DEFINE_HANDLE
- VK_DEFINE_NON_DISPATCHABLE_HANDLE
- VK_HEADER_VERSION
- VK_HEADER_VERSION_COMPLETE
• VK_MAKE_API_VERSION
• VK_MAKE_VERSION
• VK_NULL_HANDLE
• VK_USE_64_BIT_PTR_DEFINES
• VK_VERSION_MAJOR
• VK_VERSION_MINOR
• VK_VERSION_PATCH

New Base Types

• VkBool32
• VkDeviceAddress
• VkDeviceSize
• VkFlags
• VkSampleMask

New Object Types

• VkBuffer
• VkBufferView
• VkCommandBuffer
• VkCommandPool
• VkDescriptorPool
• VkDescriptorSet
• VkDescriptorSetLayout
• VkDevice
• VkDeviceMemory
• VkEvent
• VkFence
• VkFramebuffer
• VkImage
• VkImageView
• VkInstance
• VkPhysicalDevice
• VkPipeline
• VkPipelineCache
• VkPipelineLayout
- VkQueryPool
- VkQueue
- VkRenderPass
- VkSampler
- VkSemaphore
- VkShaderModule

**New Commands**

- vkAllocateCommandBuffers
- vkAllocateDescriptorSets
- vkAllocateMemory
- vkBeginCommandBuffer
- vkBindBufferMemory
- vkBindImageMemory
- vkCmdBeginQuery
- vkCmdBeginRenderPass
- vkCmdBindDescriptorSets
- vkCmdBindIndexBuffer
- vkCmdBindPipeline
- vkCmdBindVertexBuffers
- vkCmdBlitImage
- vkCmdClearAttachments
- vkCmdClearColorImage
- vkCmdClearDepthStencilImage
- vkCmdCopyBuffer
- vkCmdCopyBufferToImage
- vkCmdCopyImage
- vkCmdCopyImageToBuffer
- vkCmdCopyQueryPoolResults
- vkCmdDispatch
- vkCmdDispatchIndirect
- vkCmdDraw
- vkCmdDrawIndexed
- vkCmdDrawIndexedIndirect
- vkCmdDrawIndirect
• vkCmdEndQuery
• vkCmdEndRenderPass
• vkCmdExecuteCommands
• vkCmdFillBuffer
• vkCmdNextSubpass
• vkCmdPipelineBarrier
• vkCmdPushConstants
• vkCmdResetEvent
• vkCmdResetQueryPool
• vkCmdResolveImage
• vkCmdSetBlendConstants
• vkCmdSetDepthBias
• vkCmdSetDepthBounds
• vkCmdSetEvent
• vkCmdSetLineWidth
• vkCmdSetScissor
• vkCmdSetStencilCompareMask
• vkCmdSetStencilReference
• vkCmdSetStencilWriteMask
• vkCmdSetViewport
• vkCmdUpdateBuffer
• vkCmdWaitEvents
• vkCmdWriteTimestamp
• vkCreateBuffer
• vkCreateBufferView
• vkCreateCommandPool
• vkCreateComputePipelines
• vkCreateDescriptorPool
• vkCreateDescriptorSetLayout
• vkCreateDevice
• vkCreateEvent
• vkCreateFence
• vkCreateFramebuffer
• vkCreateGraphicsPipelines
• vkCreateImage
• vkCreateImageView
• vkCreateInstance
• vkCreatePipelineCache
• vkCreatePipelineLayout
• vkCreateQueryPool
• vkCreateRenderPass
• vkCreateSampler
• vkCreateSemaphore
• vkCreateShaderModule
• vkDestroyBuffer
• vkDestroyBufferView
• vkDestroyCommandPool
• vkDestroyDescriptorPool
• vkDestroyDescriptorSetLayout
• vkDestroyDevice
• vkDestroyEvent
• vkDestroyFence
• vkDestroyFramebuffer
• vkDestroyImage
• vkDestroyImageView
• vkDestroyInstance
• vkDestroyPipeline
• vkDestroyPipelineCache
• vkDestroyPipelineLayout
• vkDestroyQueryPool
• vkDestroyRenderPass
• vkDestroySampler
• vkDestroySemaphore
• vkDestroyShaderModule
• vkDeviceWaitIdle
• vkEndCommandBuffer
• vkEnumerateDeviceExtensionProperties
• vkEnumerateDeviceLayerProperties
• vkEnumerateInstanceExtensionProperties
• vkEnumerateInstanceLayerProperties
- vkEnumeratePhysicalDevices
- vkFlushMappedMemoryRanges
- vkFreeCommandBuffers
- vkFreeDescriptorSets
- vkFreeMemory
- vkGetBufferMemoryRequirements
- vkGetDeviceMemoryCommitment
- vkGetDeviceProcAddr
- vkGetDeviceQueue
- vkGetEventStatus
- vkGetFenceStatus
- vkGetImageMemoryRequirements
- vkGetImageSparseMemoryRequirements
- vkGetImageSubresourceLayout
- vkGetInstanceProcAddr
- vkGetPhysicalDeviceFeatures
- vkGetPhysicalDeviceFormatProperties
- vkGetPhysicalDeviceImageFormatProperties
- vkGetPhysicalDeviceMemoryProperties
- vkGetPhysicalDeviceProperties
- vkGetPhysicalDeviceQueueFamilyProperties
- vkGetPhysicalDeviceSparseImageFormatProperties
- vkGetPipelineCacheData
- vkGetQueryPoolResults
- vkGetRenderAreaGranularity
- vkInvalidateMappedMemoryRanges
- vkMapMemory
- vkMergePipelineCaches
- vkQueueBindSparse
- vkQueueSubmit
- vkQueueWaitIdle
- vkResetCommandBuffer
- vkResetCommandPool
- vkResetDescriptorPool
- vkResetEvent
• vkResetFences
• vkSetEvent
• vkUnmapMemory
• vkUpdateDescriptorSets
• vkWaitForFences

New Structures

• VkAllocationCallbacks
• VkApplicationInfo
• VkAttachmentDescription
• VkAttachmentReference
• VkBaseInStructure
• VkBaseOutStructure
• VkBindSparseInfo
• VkBufferCopy
• VkBufferCreateInfo
• VkBufferImageCopy
• VkBufferMemoryBarrier
• VkBufferViewCreateInfo
• VkClearAttachment
• VkClearDepthStencilValue
• VkClearRect
• VkCommandBufferAllocateInfo
• VkCommandBufferBeginInfo
• VkCommandBufferInheritanceInfo
• VkCommandPoolCreateInfo
• VkComponentMapping
• VkComputePipelineCreateInfo
• VkCopyDescriptorSet
• VkDescriptorBufferInfo
• VkDescriptorImageInfo
• VkDescriptorPoolCreateInfo
• VkDescriptorPoolSize
• VkDescriptorSetAllocateInfo
• VkDescriptorSetLayoutBinding
- VkDescriptorSetLayoutCreateInfo
- VkDeviceCreateInfo
- VkDeviceQueueCreateInfo
- VkDispatchIndirectCommand
- VkDrawIndexedIndirectCommand
- VkDrawIndirectCommand
- VkEventCreateInfo
- VkExtensionProperties
- VkExtent2D
- VkExtent3D
- VkFenceCreateInfo
- VkFormatProperties
- VkFramebufferCreateInfo
- VkGraphicsPipelineCreateInfo
- VkImageBlit
- VkImageCopy
- VkImageCreateInfo
- VkImageFormatProperties
- VkImageMemoryBarrier
- VkImageResolve
- VkImageSubresource
- VkImageSubresourceLayers
- VkImageSubresourceRange
- VkImageViewCreateInfo
- VkInstanceCreateInfo
- VkLayerProperties
- VkMappedMemoryRange
- VkMemoryAllocateInfo
- VkMemoryBarrier
- VkMemoryHeap
- VkMemoryRequirements
- VkMemoryType
- VkOffset2D
- VkOffset3D
- VkPhysicalDeviceFeatures
• VkPhysicalDeviceLimits
• VkPhysicalDeviceMemoryProperties
• VkPhysicalDeviceProperties
• VkPhysicalDeviceSparseProperties
• VkPipelineCacheCreateInfo
• VkPipelineCacheHeaderVersionOne
• VkPipelineColorBlendAttachmentState
• VkPipelineColorBlendStateCreateInfo
• VkPipelineDepthStencilStateCreateInfo
• VkPipelineDynamicStateCreateInfo
• VkPipelineInputAssemblyStateCreateInfo
• VkPipelineLayoutCreateInfo
• VkPipelineMultisampleStateCreateInfo
• VkPipelineRasterizationStateCreateInfo
• VkPipelineShaderStageCreateInfo
• VkPipelineTessellationStateCreateInfo
• VkPipelineVertexInputStateCreateInfo
• VkPipelineViewportStateCreateInfo
• VkPushConstantRange
• VkQueryPoolCreateInfo
• VkQueueFamilyProperties
• VkRect2D
• VkRenderPassBeginInfo
• VkRenderPassCreateInfo
• VkSamplerCreateInfo
• VkSemaphoreCreateInfo
• VkSparseBufferMemoryBindInfo
• VkSparseImageFormatProperties
• VkSparseImageMemoryBind
• VkSparseImageMemoryBindInfo
• VkSparseImageMemoryRequirements
• VkSparseImageOpaqueMemoryBindInfo
• VkSparseMemoryBind
• VkSpecializationInfo
• VkSpecializationMapEntry
• VkStencilOpState
• VkSubmitInfo
• VkSubpassDependency
• VkSubpassDescription
• VkSubresourceLayout
• VkVertexInputAttributeDescription
• VkVertexInputBindingDescription
• VkViewport
• VkWriteDescriptorSet
• Extending VkPipelineShaderStageCreateInfo:
  ◦ VkShaderModuleCreateInfo

New Unions
• VkClearColorValue
• VkClearValue

New Function Pointers
• PFN_vkAllocationFunction
• PFN_vkFreeFunction
• PFN_vkInternalAllocationNotification
• PFN_vkInternalFreeNotification
• PFN_vkReallocationFunction
• PFN_vkVoidFunction

New Enums
• VkAccessFlagBits
• VkAttachmentDescriptionFlagBits
• VkAttachmentLoadOp
• VkAttachmentStoreOp
• VkBlendFactor
• VkBlendOp
• VkBorderColor
• VkBufferCreateFlagBits
• VkBufferUsageFlagBits
• VkColorComponentFlagBits
• VkCommandBufferLevel
• VkCommandBufferResetFlagBits
• VkCommandBufferUsageFlagBits
• VkCommandPoolCreateFlagBits
• VkCommandPoolResetFlagBits
• VkCompareOp
• VkComponentSwizzle
• VkCullModeFlagBits
• VkDependencyFlagBits
• VkDescriptorPoolCreateFlagBits
• VkDescriptorSetLayoutCreateFlagBits
• VkDescriptorType
• VkDynamicState
• VkEventCreateFlagBits
• VkFenceCreateFlagBits
• VkFilter
• VkFormat
• VkFormatFeatureFlagBits
• VkFramebufferCreateFlagBits
• VkFrontFace
• VkImageAspectFlagBits
• VkImageCreateFlagBits
• VkImageLayout
• VkImageTiling
• VkImageType
• VkImageUsageFlagBits
• VkImageViewCreateFlagBits
• VkImageViewType
• VkIndexType
• VkInstanceCreateFlagBits
• VkInternalAllocationType
• VkLogicOp
• VkMemoryHeapFlagBits
• VkMemoryPropertyFlagBits
• VkObjectType
• VkPhysicalDeviceType
• VkPipelineBindPoint
• VkPipelineCacheHeaderVersion
• VkPipelineCreateFlagBits
• VkPipelineShaderStageCreateFlagBits
• VkPipelineStageFlagBits
• VkPolygonMode
• VkPrimitiveTopology
• VkQueryControlFlagBits
• VkQueryPipelineStatisticFlagBits
• VkQueryResultFlagBits
• VkQueryType
• VkQueueFlagBits
• VkRenderPassCreateFlagBits
• VkResult
• VkSampleCountFlagBits
• VkSamplerAddressMode
• VkSamplerCreateFlagBits
• VkSamplerMipmapMode
• VkShaderStageFlagBits
• VkSharingMode
• VkSparseImageFormatFlagBits
• VkSparseMemoryBindFlagBits
• VkStencilFaceFlagBits
• VkStencilOp
• VkStructureType
• VkSubpassContents
• VkSubpassDescriptionFlagBits
• VkSystemAllocationScope
• VkVendorId
• VkVertexInputRate

**New Bitmasks**

• VkAccessFlags
• VkAttachmentDescriptionFlags
- VkBufferCreateFlags
- VkBufferUsageFlags
- VkBufferViewCreateFlags
- VkColorComponentFlags
- VkCommandBufferResetFlags
- VkCommandBufferUsageFlags
- VkCommandPoolCreateFlags
- VkCommandPoolResetFlags
- VkCullModeFlags
- VkDependencyFlags
- VkDescriptorPoolCreateFlags
- VkDescriptorPoolResetFlags
- VkDescriptorSetLayoutCreateFlags
- VkDeviceCreateFlags
- VkDeviceQueueCreateFlags
- VkEventCreateFlags
- VkFenceCreateFlags
- VkFormatFeatureFlags
- VkFramebufferCreateFlags
- VkImageAspectFlags
- VkImageCreateFlags
- VkImageUsageFlags
- VkImageViewCreateFlags
- VkInstanceCreateFlags
- VkMemoryHeapFlags
- VkMemoryMapFlags
- VkMemoryPropertyFlags
- VkPipelineCacheCreateFlags
- VkPipelineColorBlendStateCreateFlags
- VkPipelineCreateFlags
- VkPipelineDepthStencilStateCreateFlags
- VkPipelineDynamicStateCreateFlags
- VkPipelineInputAssemblyStateCreateFlags
- VkPipelineLayoutCreateFlags
- VkPipelineMultisampleStateCreateFlags
• VkPipelineRasterizationStateCreateFlags
• VkPipelineShaderStageCreateFlags
• VkPipelineStageFlags
• VkPipelineTessellationStateCreateFlags
• VkPipelineVertexInputStateCreateFlags
• VkPipelineViewportStateCreateFlags
• VkQueryControlFlags
• VkQueryPipelineStatisticFlags
• VkQueryPoolCreateFlags
• VkQueryResultFlags
• VkQueueFlags
• VkRenderPassCreateFlags
• VkSampleCountFlags
• VkSamplerCreateFlags
• VkSemaphoreCreateFlags
• VkShaderModuleCreateFlags
• VkShaderStageFlags
• VkSparseImageFormatFlags
• VkSparseMemoryBindFlags
• VkStencilFaceFlags
• VkSubpassDescriptionFlags

New Headers
• vk_platform

New Enum Constants
• VK_ATTACHMENT_UNUSED
• VK_FALSE
• VK_LOD_CLAMP_NONE
• VK_MAX_DESCRIPTION_SIZE
• VK_MAX_EXTENSION_NAME_SIZE
• VK_MAX_MEMORY_HEAPS
• VK_MAX_MEMORY_TYPES
• VK_MAX_PHYSICAL_DEVICE_NAME_SIZE
• VK_QUEUE_FAMILY_IGNORED
• VK_REMAINING_ARRAY_LAYERS
• VK_REMAINING_MIP_LEVELS
• VK_SUBPASS_EXTERNAL
• VK_TRUE
• VK_UUID_SIZE
• VK_WHOLE_SIZE
Extensions to the Vulkan API can be defined by authors, groups of authors, and the Khronos Vulkan Working Group. In order not to compromise the readability of the Vulkan Specification, the core Specification does not incorporate most extensions. The online Registry of extensions is available at URL

https://registry.khronos.org/vulkan/

and allows generating versions of the Specification incorporating different extensions.

Most of the content previously in this appendix does not specify use of specific Vulkan extensions and layers, but rather specifies the processes by which extensions and layers are created. As of version 1.0.21 of the Vulkan Specification, this content has been migrated to the Vulkan Documentation and Extensions document. Authors creating extensions and layers must follow the mandatory procedures in that document.

The remainder of this appendix documents a set of extensions chosen when this document was built. Versions of the Specification published in the Registry include:

- Core API + mandatory extensions required of all Vulkan implementations.
- Core API + all registered and published Khronos (KHR) extensions.
- Core API + all registered and published extensions.

Extensions are grouped as Khronos KHR, multivendor EXT, and then alphabetically by author ID. Within each group, extensions are listed in alphabetical order by their name.

Note

As of the initial Vulkan 1.1 public release, the KHX author ID is no longer used. All KHX extensions have been promoted to KHR status. Previously, this author ID was used to indicate that an extension was experimental, and is being considered for standardization in future KHR or core Vulkan API versions. We no longer use this mechanism for exposing experimental functionality.

Some vendors may use an alternate author ID ending in X for some of their extensions. The exact meaning of such an author ID is defined by each vendor, and may not be equivalent to KHX, but it is likely to indicate a lesser degree of interface stability than a non-X extension from the same vendor.

List of Extensions
Appendix F: Vulkan Roadmap Milestones

Roadmap milestones are intended to be supported by mid-to-high-end smartphones, tablets, laptops, consoles, and desktop devices.

Each milestone indicates support for a set of extensions, features, limits, and formats across these devices, and should be supported by all such new hardware shipping by the end of the target year or shortly thereafter.

Roadmap 2022

The Roadmap 2022 milestone is intended to be supported by newer mid-to-high-end devices shipping in 2022 or shortly thereafter across mainstream smartphone, tablet, laptops, console and desktop devices.

Required API versions

This profile requires Vulkan 1.3.

Required Features

The following core optional features are required to be supported:

- Vulkan 1.0 Optional Features
  - `fullDrawIndexUint32`
  - `imageCubeArray`
  - `independentBlend`
  - `sampleRateShading`
  - `drawIndirectFirstInstance`
  - `depthClamp`
  - `depthBiasClamp`
  - `samplerAnisotropy`
  - `occlusionQueryPrecise`
  - `fragmentStoresAndAtomics`
  - `shaderStorageImageExtendedFormats`
  - `shaderUniformBufferArrayDynamicIndexing`
  - `shaderSampledImageArrayDynamicIndexing`
  - `shaderStorageBufferArrayDynamicIndexing`
  - `shaderStorageImageArrayDynamicIndexing`

- Vulkan 1.1 Optional Features
  - `samplerYcbcrConversion`
• Vulkan 1.2 Optional Features
  ◦ samplerMirrorClampToEdge
  ◦ descriptorIndexing
  ◦ shaderUniformTexelBufferArrayDynamicIndexing
  ◦ shaderStorageTexelBufferArrayDynamicIndexing
  ◦ shaderUniformBufferArrayNonUniformIndexing
  ◦ shaderSampledImageArrayNonUniformIndexing
  ◦ shaderStorageBufferArrayNonUniformIndexing
  ◦ shaderStorageImageArrayNonUniformIndexing
  ◦ descriptorBindingSampledImageUpdateAfterBind
  ◦ descriptorBindingStorageImageUpdateAfterBind
  ◦ descriptorBindingStorageBufferUpdateAfterBind
  ◦ descriptorBindingUniformTexelBufferUpdateAfterBind
  ◦ descriptorBindingStorageTexelBufferUpdateAfterBind
  ◦ descriptorBindingUpdateUnusedWhilePending
  ◦ descriptorBindingPartiallyBound
  ◦ descriptorBindingVariableDescriptorCount
  ◦ runtimeDescriptorArray
  ◦ scalarBlockLayout

**Required Limits**

The following core increased limits are **required**

*Table 69. Vulkan 1.0 Limits*

<table>
<thead>
<tr>
<th>Limit Name</th>
<th>Unsupported Limit</th>
<th>Core Limit</th>
<th>Profile Limit</th>
<th>Limit Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxImageDimension1D</td>
<td>-</td>
<td>4096</td>
<td>8192</td>
<td>min</td>
</tr>
<tr>
<td>maxImageDimension2D</td>
<td>-</td>
<td>4096</td>
<td>8192</td>
<td>min</td>
</tr>
<tr>
<td>maxImageDimensionCube</td>
<td>-</td>
<td>4096</td>
<td>8192</td>
<td>min</td>
</tr>
<tr>
<td>maxImageArrayLayers</td>
<td>-</td>
<td>256</td>
<td>2048</td>
<td>min</td>
</tr>
<tr>
<td>maxUniformBufferRange</td>
<td>-</td>
<td>16384</td>
<td>65536</td>
<td>min</td>
</tr>
<tr>
<td>bufferImageGranularity</td>
<td>-</td>
<td>131072</td>
<td>4096</td>
<td>max</td>
</tr>
<tr>
<td>maxPerStageDescriptorSamplers</td>
<td>-</td>
<td>16</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>Limit Name</td>
<td>Unsupported Limit</td>
<td>Core Limit</td>
<td>Profile Limit</td>
<td>Limit Type¹</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>-------------------</td>
<td>------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>maxPerStageDescriptorUniformBuffers</td>
<td>-</td>
<td>12</td>
<td>15</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorStorageBuffers</td>
<td>-</td>
<td>4</td>
<td>30</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorSampledImages</td>
<td>-</td>
<td>16</td>
<td>200</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorStorageImages</td>
<td>-</td>
<td>4</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageResources</td>
<td>-</td>
<td>128</td>
<td>200</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetSamplers</td>
<td>-</td>
<td>96</td>
<td>576</td>
<td>min, n × PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetUniformBuffers</td>
<td>-</td>
<td>72</td>
<td>90</td>
<td>min, n × PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetStorageBuffers</td>
<td>-</td>
<td>24</td>
<td>96</td>
<td>min, n × PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetSampledImages</td>
<td>-</td>
<td>96</td>
<td>1800</td>
<td>min, n × PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetStorageImages</td>
<td>-</td>
<td>24</td>
<td>144</td>
<td>min, n × PerStage</td>
</tr>
<tr>
<td>maxFragmentCombinedOutputResources</td>
<td>-</td>
<td>4</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxComputeWorkGroupInvocations</td>
<td>-</td>
<td>128</td>
<td>256</td>
<td>min</td>
</tr>
<tr>
<td>maxComputeWorkGroupSize</td>
<td>-</td>
<td>(128,128,64)</td>
<td>(256,256,64)</td>
<td>min</td>
</tr>
<tr>
<td>subTexelPrecisionBits</td>
<td>-</td>
<td>4</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td>mipmapPrecisionBits</td>
<td>-</td>
<td>4</td>
<td>6</td>
<td>min</td>
</tr>
<tr>
<td>maxSamplerLodBias</td>
<td>-</td>
<td>2</td>
<td>14</td>
<td>min</td>
</tr>
<tr>
<td>pointSizeGranularity</td>
<td>0.0</td>
<td>1.0</td>
<td>0.125</td>
<td>max, fixed point increment</td>
</tr>
<tr>
<td>lineWidthGranularity</td>
<td>0.0</td>
<td>1.0</td>
<td>0.5</td>
<td>max, fixed point increment</td>
</tr>
<tr>
<td>standardSampleLocations</td>
<td>-</td>
<td>-</td>
<td>VK_TRUE</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>maxColorAttachments</td>
<td>-</td>
<td>4</td>
<td>7</td>
<td>min</td>
</tr>
</tbody>
</table>

Table 70. Vulkan 1.1 Limits
<table>
<thead>
<tr>
<th>Limit Name</th>
<th>Unsupported Limit</th>
<th>Core Limit</th>
<th>Profile Limit</th>
<th>Limit Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>subgroupSize</td>
<td>-</td>
<td>1/4</td>
<td>4</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>subgroupSupportedStages</td>
<td>-</td>
<td>VK_SHADER_STAGE_COMPUTE_BIT</td>
<td>VK_SHADER_STAGE_COMPUTE_BIT VK_SHADER_STAGE_FRAGMENT_BIT</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>subgroupSupportedOperations</td>
<td>-</td>
<td>VK_SUBGROUP_FEATURE_BASIC_BIT</td>
<td>VK_SUBGROUP_FEATURE_BASIC_BIT VK_SUBGROUP_FEATURE_VOTE_BIT VK_SUBGROUP_FEATURE_ARITHMETIC_BIT VK_SUBGROUP_FEATURE_BALLOT_BIT VK_SUBGROUP_FEATURE_SHUFFLE_BIT VK_SUBGROUP_FEATURE_SHUFFLE_RELATIVE_BIT VK_SUBGROUP_FEATURE_QUAD_BIT</td>
<td>implementation-dependent</td>
</tr>
</tbody>
</table>

Table 71. Vulkan 1.2 Limits

<table>
<thead>
<tr>
<th>Limit Name</th>
<th>Unsupported Limit</th>
<th>Core Limit</th>
<th>Profile Limit</th>
<th>Limit Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>shaderSignedZeroInfNanPreserveFloat16</td>
<td>-</td>
<td>-</td>
<td>VK_TRUE</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>shaderSignedZeroInfNanPreserveFloat32</td>
<td>-</td>
<td>-</td>
<td>VK_TRUE</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindInputAttachments</td>
<td>0</td>
<td>4</td>
<td>7</td>
<td>min</td>
</tr>
</tbody>
</table>

Table 72. Vulkan 1.3 Limits
<table>
<thead>
<tr>
<th>Limit Name</th>
<th>Unsupported Limit</th>
<th>Core Limit</th>
<th>Profile Limit</th>
<th>Limit Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxSubgroupSize</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
</tbody>
</table>

**Required extensions**

The following extensions are **required**

*VK_KHR_global_priority*
Appendix G: API Boilerplate

This appendix defines Vulkan API features that are infrastructure required for a complete functional description of Vulkan, but do not logically belong elsewhere in the Specification.

Vulkan Header Files

Vulkan is defined as an API in the C99 language. Khronos provides a corresponding set of header files for applications using the API, which may be used in either C or C++ code. The interface descriptions in the specification are the same as the interfaces defined in these header files, and both are derived from the \texttt{vk.xml} XML API Registry, which is the canonical machine-readable description of the Vulkan API. The Registry, scripts used for processing it into various forms, and documentation of the registry schema are available as described at https://registry.khronos.org/vulkan/#apiregistry.

Language bindings for other languages can be defined using the information in the Specification and the Registry. Khronos does not provide any such bindings, but third-party developers have created some additional bindings.

Vulkan Combined API Header \texttt{vulkan.h} (Informative)

Applications normally will include the header \texttt{vulkan.h}. In turn, \texttt{vulkan.h} always includes the following headers:

- \texttt{vk_platform.h}, defining platform-specific macros and headers.
- \texttt{vulkan_core.h}, defining APIs for the Vulkan core and all registered extensions other than window system-specific and provisional extensions, which are included in separate header files.

In addition, specific preprocessor macros defined at the time \texttt{vulkan.h} is included cause header files for the corresponding window system-specific and provisional interfaces to be included, as described below.

Vulkan Platform-Specific Header \texttt{vk_platform.h} (Informative)

Platform-specific macros and interfaces are defined in \texttt{vk_platform.h}. These macros are used to control platform-dependent behavior, and their exact definitions are under the control of specific platforms and Vulkan implementations.

Platform-Specific Calling Conventions

On many platforms the following macros are empty strings, causing platform- and compiler-specific default calling conventions to be used.

\texttt{VKAPI_ATTR} is a macro placed before the return type in Vulkan API function declarations. This macro controls calling conventions for C++11 and GCC/Clang-style compilers.

\texttt{VKAPI_CALL} is a macro placed after the return type in Vulkan API function declarations. This macro controls calling conventions for MSVC-style compilers.
VKAPI_PTR is a macro placed between the '(' and '*' in Vulkan API function pointer declarations. This macro also controls calling conventions, and typically has the same definition as VKAPI_ATTR or VKAPI_CALL, depending on the compiler.

With these macros, a Vulkan function declaration takes the form of:

```c
VKAPI_ATTR <return_type> VKAPI_CALL <command_name>(<command_parameters>);
```

Additionally, a Vulkan function pointer type declaration takes the form of:

```c
typedef <return_type> (VKAPI_PTR *PFN_<command_name>)(<command_parameters>);
```

Platform-Specific Header Control

If the VK_NO_STDINT_H macro is defined by the application at compile time, extended integer types used by the Vulkan API, such as `uint8_t`, **must** also be defined by the application. Otherwise, the Vulkan headers will not compile. If VK_NO_STDINT_H is not defined, the system `<stdint.h>` is used to define these types. There is a fallback path when Microsoft Visual Studio version 2008 and earlier versions are detected at compile time.

If the VK_NO_STDDEF_H macro is defined by the application at compile time, `size_t`, **must** also be defined by the application. Otherwise, the Vulkan headers will not compile. If VK_NO_STDDEF_H is not defined, the system `<stddef.h>` is used to define this type.

Vulkan Core API Header `vulkan_core.h`

Applications that do not make use of window system-specific extensions may simply include `vulkan_core.h` instead of `vulkan.h`, although there is usually no reason to do so. In addition to the Vulkan API, `vulkan_core.h` also defines a small number of C preprocessor macros that are described below.

Vulkan Header File Version Number

`VK_HEADER_VERSION` is the version number of the `vulkan_core.h` header. This value is kept synchronized with the patch version of the released Specification.

```c
// Provided by VK_VERSION_1_0
// Version of this file
#define VK_HEADER_VERSION 237
```

`VK_HEADER_VERSION_COMPLETE` is the complete version number of the `vulkan_core.h` header, comprising the major, minor, and patch versions. The major/minor values are kept synchronized with the complete version of the released Specification. This value is intended for use by automated tools to identify exactly which version of the header was used during their generation.

Applications should not use this value as their `VkApplicationInfo::apiVersion`. Instead applications
should explicitly select a specific fixed major/minor API version using, for example, one of the `VK_API_VERSION_*_*` values.

```c
// Provided by VK_VERSION_1_0
// Complete version of this file
#define VK_HEADER_VERSION_COMPLETE VK_MAKE_API_VERSION(0, 1, 3, VK_HEADER_VERSION)
```

`VK_API_VERSION` is now commented out of `vulkan_core.h` and cannot be used.

```c
// Provided by VK_VERSION_1_0
// DEPRECATED: This define has been removed. Specific version defines (e.g. VK_API_VERSION_1_0), or the VK_MAKE_VERSION macro, should be used instead.
#ifndef VK_API_VERSION
#define VK_API_VERSION VK_MAKE_VERSION(1, 0, 0) // Patch version should always be set to 0
#endif
```

### Vulkan Handle Macros

**`VK_DEFINE_HANDLE`** defines a **dispatchable handle** type.

```c
// Provided by VK_VERSION_1_0
#define VK_DEFINE_HANDLE(object) typedef struct object##_T* object;
```

- **object** is the name of the resulting C type.

The only dispatchable handle types are those related to device and instance management, such as `VkDevice`.

**`VK_DEFINE_NON_DISPATCHABLE_HANDLE`** defines a **non-dispatchable handle** type.

```c
// Provided by VK_VERSION_1_0
#ifndef VK_DEFINE_NON_DISPATCHABLE_HANDLE
#define VK_DEFINE_NON_DISPATCHABLE_HANDLE(object)
    typedef struct object##_T object;
#else
    typedef uint64_t object;
#endif
```

- **object** is the name of the resulting C type.

Most Vulkan handle types, such as `VkBuffer`, are non-dispatchable.

---

**Note**

- **object** is the name of the resulting C type.
The `vulkan_core.h` header allows the `VK_DEFINE_NON_DISPATCHABLE_HANDLE` and `VK_NULL_HANDLE` definitions to be overridden by the application. If `VK_DEFINE_NON_DISPATCHABLE_HANDLE` is already defined when `vulkan_core.h` is compiled, the default definitions for `VK_DEFINE_NON_DISPATCHABLE_HANDLE` and `VK_NULL_HANDLE` are skipped. This allows the application to define a binary-compatible custom handle which **may** provide more type-safety or other features needed by the application. Applications **must** not define handles in a way that is not binary compatible - where binary compatibility is platform dependent.

`VK_NULL_HANDLE` is a reserved value representing a non-valid object handle. It may be passed to and returned from Vulkan commands only when specifically allowed.

```c
// Provided by VK_VERSION_1_0

#ifndef VK_DEFINE_NON_DISPATCHABLE_HANDLE
    #if (VK_USE_64_BIT_PTR_DEFINES==1)
        #if (defined(__cplusplus) && (__cplusplus >= 201103L)) || (defined(_MSVC_LANG) && (_MSVC_LANG >= 201103L))
            #define VK_NULL_HANDLE nullptr
        #else
            #define VK_NULL_HANDLE ((void*)0)
        #endif
    #else
        #define VK_NULL_HANDLE 0ULL
    #endif
#else
    #define VK_NULL_HANDLE 0
#endif
```

**VK_USE_64_BIT_PTR_DEFINES** defines whether the default non-dispatchable handles are declared using either a 64-bit pointer type or a 64-bit unsigned integer type.

**VK_USE_64_BIT_PTR_DEFINES** is set to '1' to use a 64-bit pointer type or any other value to use a 64-bit unsigned integer type.

```c
// Provided by VK_VERSION_1_0

#ifndef VK_USE_64_BIT_PTR_DEFINES
    #if defined(__LP64__) || defined(_WIN64) || (defined(__x86_64__) && !defined(__ILP32__)) || defined(_M_X64) || defined(__ia64) || defined (_M_IA64) || defined(__aarch64__) || defined(__powerpc64__) ||
        #define VK_USE_64_BIT_PTR_DEFINES 1
    #else
        #define VK_USE_64_BIT_PTR_DEFINES 0
    #endif
#endif
```
The `vulkan_core.h` header allows the `VK_USE_64_BIT_PTR_DEFINES` definition to be overridden by the application. This allows the application to select either a 64-bit pointer type or a 64-bit unsigned integer type for non-dispatchable handles in the case where the predefined preprocessor check does not identify the desired configuration.

This macro was introduced starting with the Vulkan 1.2.174 headers, and its availability can be checked at compile time by requiring `VK_HEADER_VERSION >= 174`. It is not available if you are using older headers, such as may be shipped with an older Vulkan SDK. Developers requiring this functionality may wish to include a copy of the current Vulkan headers with their project in this case.

**Window System-Specific Header Control (Informative)**

To use a Vulkan extension supporting a platform-specific window system, header files for that window system must be included at compile time, or platform-specific types must be forward-declared. The Vulkan header files are unable to determine whether or not an external header is available at compile time, so platform-specific extensions are provided in separate headers from the core API and platform-independent extensions, allowing applications to decide which ones they need to be defined and how the external headers are included.

Extensions dependent on particular sets of platform headers, or that forward-declare platform-specific types, are declared in a header named for that platform. Before including these platform-specific Vulkan headers, applications must include both `vulkan_core.h` and any external native headers the platform extensions depend on.

As a convenience for applications that do not need the flexibility of separate platform-specific Vulkan headers, `vulkan.h` includes `vulkan_core.h`, and then conditionally includes platform-specific Vulkan headers and the external headers they depend on. Applications control which platform-specific headers are included by defining macros before including `vulkan.h`.

The correspondence between platform-specific extensions, external headers they require, the platform-specific header which declares them, and the preprocessor macros which enable inclusion by `vulkan.h` are shown in the following table.

<table>
<thead>
<tr>
<th>Extension Name</th>
<th>Window System Name</th>
<th>Platform-specific Header</th>
<th>Required External Headers</th>
<th>Controlling <code>vulkan.h</code> Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VK_KHR_android_surface</code></td>
<td>Android</td>
<td><code>vulkan_android.h</code></td>
<td>None</td>
<td><code>VK_USE_PLATFORM_ANDROID_KHR</code></td>
</tr>
<tr>
<td><code>VK_KHR_wayland_surface</code></td>
<td>Wayland</td>
<td><code>vulkan_wayland.h</code></td>
<td><code>&lt;wayland-client.h&gt;</code></td>
<td><code>VK_USE_PLATFORM_WAYLAND_KHR</code></td>
</tr>
<tr>
<td>Extension Name</td>
<td>Window System Name</td>
<td>Platform-specific Header</td>
<td>Required External Headers</td>
<td>Controlling Vulkan.h Macro</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------</td>
<td>--------------------------</td>
<td>--------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>VK_KHR_xcb_surface</td>
<td>X11 Xcb</td>
<td>vulkan_xcb.h</td>
<td>&lt;xcb/xcb.h&gt;</td>
<td>VK_USE_PLATFORM_XCB_KHR</td>
</tr>
<tr>
<td>VK_KHR_xlib_surface</td>
<td>X11 Xlib</td>
<td>vulkan_xlib.h</td>
<td>&lt;X11/Xlib.h&gt;</td>
<td>VK_USE_PLATFORM_XLIB_KHR</td>
</tr>
<tr>
<td>VK_EXT_directfb_surface</td>
<td>DirectFB</td>
<td>vulkan_directfb.h</td>
<td>&lt;directfb/directfb.h&gt;</td>
<td>VK_USE_PLATFORM_DIRECTFB_EXT</td>
</tr>
<tr>
<td>VK_EXT_acquire_xlib_display</td>
<td>X11 XRandR</td>
<td>vulkan_xlib_xrandr.h</td>
<td>&lt;X11/Xlib.h&gt;, &lt;X11/extensions/Xrandr.h&gt;</td>
<td>VK_USE_PLATFORM_XLIB_XRANDBR_EXT</td>
</tr>
<tr>
<td>VK_GGP_stream_descriptor_surface, VK_GGP_frame_token</td>
<td>Google Games Platform</td>
<td>vulkan_ggp.h</td>
<td>&lt;ggp_c/vulkan_types.h&gt;</td>
<td>VK_USE_PLATFORM_GGP</td>
</tr>
<tr>
<td>VK_MVK_ios_surface</td>
<td>iOS</td>
<td>vulkan_ios.h</td>
<td>None</td>
<td>VK_USE_PLATFORM_IOS_MVK</td>
</tr>
<tr>
<td>VK_MVK_macos_surface</td>
<td>macOS</td>
<td>vulkan_macos.h</td>
<td>None</td>
<td>VK_USE_PLATFORM_MACOS_MVK</td>
</tr>
<tr>
<td>VK_NN_vi_surface</td>
<td>VI</td>
<td>vulkan_vi.h</td>
<td>None</td>
<td>VK_USE_PLATFORM_VI_NN</td>
</tr>
<tr>
<td>VK_FUCHSIA_imagepipe_surface</td>
<td>Fuchsia</td>
<td>vulkan_fuchsia.h</td>
<td>&lt;zircon/types.h&gt;</td>
<td>VK_USE_PLATFORM_FUCHSIA</td>
</tr>
<tr>
<td>VK_EXT_metal_surface</td>
<td>Metal on CoreAnimation</td>
<td>vulkan_metal.h</td>
<td>None</td>
<td>VK_USE_PLATFORM_METAL_EXT</td>
</tr>
<tr>
<td>VK_QNX_screen_surface</td>
<td>QNX Screen</td>
<td>vulkan_screen.h</td>
<td>&lt;screen/screen.h&gt;</td>
<td>VK_USE_PLATFORM_SCREEN_QNX</td>
</tr>
</tbody>
</table>

**Note**

This section describes the purpose of the headers independently of the specific underlying functionality of the window system extensions themselves. Each extension name will only link to a description of that extension when viewing a specification built with that extension included.
Provisional Extension Header Control (Informative)

_Provisional extensions should_ not be used in production applications. The functionality defined by such extensions _may_ change in ways that break backwards compatibility between revisions, and before final release of a non-provisional version of that extension.

Provisional extensions are defined in a separate _provisional header_, `vulkan_beta.h`, allowing applications to decide whether or not to include them. The mechanism is similar to _window system-specific headers_: before including `vulkan_beta.h`, applications _must_ include `vulkan_core.h`.

**Note**

Sometimes a provisional extension will include a subset of its interfaces in `vulkan_core.h`. This may occur if the provisional extension is promoted from an existing vendor or EXT extension and some of the existing interfaces are defined as aliases of the provisional extension interfaces. All other interfaces of that provisional extension which are not aliased will be included in `vulkan_beta.h`.

As a convenience for applications, `vulkan.h` conditionally includes `vulkan_beta.h`. Applications _can_ control inclusion of `vulkan_beta.h` by _defining_ the macro `VK_ENABLE_BETA_EXTENSIONS` before including `vulkan.h`.

**Note**

Starting in version 1.2.171 of the Specification, all provisional enumerants are protected by the macro `VK_ENABLE_BETA_EXTENSIONS`. Applications needing to use provisional extensions must always define this macro, even if they are explicitly including `vulkan_beta.h`. This is a minor change to behavior, affecting only provisional extensions.

**Note**

This section describes the purpose of the provisional header independently of the specific provisional extensions which are contained in that header at any given time. The extension appendices for provisional extensions note their provisional status, and link back to this section for more information. Provisional extensions are intended to provide early access for bleeding-edge developers, with the understanding that extension interfaces may change in response to developer feedback. Provisional extensions are very likely to eventually be updated and released as non-provisional extensions, but there is no guarantee this will happen, or how long it will take if it does happen.

Video Std Headers (Informative)

Performing video coding operations usually involves the application having to provide various parameters, data structures, or other syntax elements specific to the particular video compression standard used, and the associated semantics are covered by the specification of those.

The interface descriptions of these are available in the header files derived from the `video.xml` XML
file, which includes a machine-readable definition of data structures and enumerations that are associated with the externally-provided video compression standards, and are non-normatively referred to in the Vulkan Specification.

Table 74. Video Std Headers

<table>
<thead>
<tr>
<th>Video Std Header Name</th>
<th>Description</th>
<th>Header File</th>
<th>Related Extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>vulkan_video_codecs_common</td>
<td>Codec-independent common definitions</td>
<td>&lt;vk_video/vulkan_video_codecs_common.h&gt;</td>
<td>-</td>
</tr>
<tr>
<td>vulkan_video_codec_h264std</td>
<td>ITU-T H.264 common definitions</td>
<td>&lt;vk_video/vulkan_video_codec_h264std.h&gt;</td>
<td>VK_EXT_video_decode_h264, VK_EXT_video_encode_h264</td>
</tr>
<tr>
<td>vulkan_video_codec_h264std_decode</td>
<td>ITU-T H.264 decode-specific definitions</td>
<td>&lt;vk_video/vulkan_video_codec_h264std_decode.h&gt;</td>
<td>VK_EXT_video_decode_h264</td>
</tr>
<tr>
<td>vulkan_video_codec_h264std_encode</td>
<td>ITU-T H.264 encode-specific definitions</td>
<td>&lt;vk_video/vulkan_video_codec_h264std_encode.h&gt;</td>
<td>VK_EXT_video_encode_h264</td>
</tr>
<tr>
<td>vulkan_video_codec_h265std</td>
<td>ITU-T H.265 common definitions</td>
<td>&lt;vk_video/vulkan_video_codec_h265std.h&gt;</td>
<td>VK_EXT_video_decode_h265, VK_EXT_video_encode_h265</td>
</tr>
<tr>
<td>vulkan_video_codec_h265std_decode</td>
<td>ITU-T H.265 decode-specific definitions</td>
<td>&lt;vk_video/vulkan_video_codec_h265std_decode.h&gt;</td>
<td>VK_EXT_video_decode_h265</td>
</tr>
<tr>
<td>vulkan_video_codec_h265std_encode</td>
<td>ITU-T H.265 encode-specific definitions</td>
<td>&lt;vk_video/vulkan_video_codec_h265std_encode.h&gt;</td>
<td>VK_EXT_video_encode_h265</td>
</tr>
</tbody>
</table>
Appendix H: Invariance

The Vulkan specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different Vulkan implementations. However, the specification does specify exact matches, in some cases, for images produced by the same implementation. The purpose of this appendix is to identify and provide justification for those cases that require exact matches.

Repeatability

The obvious and most fundamental case is repeated issuance of a series of Vulkan commands. For any given Vulkan and framebuffer state vector, and for any Vulkan command, the resulting Vulkan and framebuffer state must be identical whenever the command is executed on that initial Vulkan and framebuffer state. This repeatability requirement does not apply when using shaders containing side effects (image and buffer variable stores and atomic operations), because these memory operations are not guaranteed to be processed in a defined order.

One purpose of repeatability is avoidance of visual artifacts when a double-buffered scene is redrawn. If rendering is not repeatable, swapping between two buffers rendered with the same command sequence may result in visible changes in the image. Such false motion is distracting to the viewer. Another reason for repeatability is testability.

Repeatability, while important, is a weak requirement. Given only repeatability as a requirement, two scenes rendered with one (small) polygon changed in position might differ at every pixel. Such a difference, while within the law of repeatability, is certainly not within its spirit. Additional invariance rules are desirable to ensure useful operation.

Multi-pass Algorithms

Invariance is necessary for a whole set of useful multi-pass algorithms. Such algorithms render multiple times, each time with a different Vulkan mode vector, to eventually produce a result in the framebuffer. Examples of these algorithms include:

- “Erasing” a primitive from the framebuffer by redrawing it, either in a different color or using the XOR logical operation.
- Using stencil operations to compute capping planes.

Invariance Rules

For a given Vulkan device:

Rule 1 For any given Vulkan and framebuffer state vector, and for any given Vulkan command, the resulting Vulkan and framebuffer state must be identical each time the command is executed on that initial Vulkan and framebuffer state.

Rule 2 Changes to the following state values have no side effects (the use of any other state value is not affected by the change):
Required:

- Color and depth/stencil attachment contents
- Scissor parameters (other than enable)
- Write masks (color, depth, stencil)
- Clear values (color, depth, stencil)

Strongly suggested:

- Stencil parameters (other than enable)
- Depth test parameters (other than enable)
- Blend parameters (other than enable)
- Logical operation parameters (other than enable)

**Corollary 1** Fragment generation is invariant with respect to the state values listed in Rule 2.

**Rule 3** The arithmetic of each per-fragment operation is invariant except with respect to parameters that directly control it.

**Corollary 2** Images rendered into different color attachments of the same framebuffer, either simultaneously or separately using the same command sequence, are pixel identical.

**Rule 4** Identical pipelines will produce the same result when run multiple times with the same input. The wording “Identical pipelines” means VkPipeline objects that have been created with identical SPIR-V binaries and identical state, which are then used by commands executed using the same Vulkan state vector. Invariance is relaxed for shaders with side effects, such as performing stores or atomics.

**Rule 5** All fragment shaders that either conditionally or unconditionally assign FragCoord.z to FragDepth are depth-invariant with respect to each other, for those fragments where the assignment to FragDepth actually is done.

If a sequence of Vulkan commands specifies primitives to be rendered with shaders containing side effects (image and buffer variable stores and atomic operations), invariance rules are relaxed. In particular, rule 1, corollary 2, and rule 4 do not apply in the presence of shader side effects.

The following weaker versions of rules 1 and 4 apply to Vulkan commands involving shader side effects:

**Rule 6** For any given Vulkan and framebuffer state vector, and for any given Vulkan command, the contents of any framebuffer state not directly or indirectly affected by results of shader image or buffer variable stores or atomic operations must be identical each time the command is executed on that initial Vulkan and framebuffer state.

**Rule 7** Identical pipelines will produce the same result when run multiple times with the same input as long as:

- shader invocations do not use image atomic operations;
• no framebuffer memory is written to more than once by image stores, unless all such stores write the same value; and

• no shader invocation, or other operation performed to process the sequence of commands, reads memory written to by an image store.

Note

The OpenGL specification has the following invariance rule: Consider a primitive \( p' \) obtained by translating a primitive \( p \) through an offset \( (x, y) \) in window coordinates, where \( x \) and \( y \) are integers. As long as neither \( p' \) nor \( p \) is clipped, it must be the case that each fragment \( f' \) produced from \( p' \) is identical to a corresponding fragment \( f \) from \( p \) except that the center of \( f' \) is offset by \( (x, y) \) from the center of \( f \).

This rule does not apply to Vulkan and is an intentional difference from OpenGL.

When any sequence of Vulkan commands triggers shader invocations that perform image stores or atomic operations, and subsequent Vulkan commands read the memory written by those shader invocations, these operations must be explicitly synchronized.

Tessellation Invariance

When using a pipeline containing tessellation evaluation shaders, the fixed-function tessellation primitive generator consumes the input patch specified by an application and emits a new set of primitives. The following invariance rules are intended to provide repeatability guarantees. Additionally, they are intended to allow an application with a carefully crafted tessellation evaluation shader to ensure that the sets of triangles generated for two adjacent patches have identical vertices along shared patch edges, avoiding “cracks” caused by minor differences in the positions of vertices along shared edges.

Rule 1 When processing two patches with identical outer and inner tessellation levels, the tessellation primitive generator will emit an identical set of point, line, or triangle primitives as long as the pipeline used to process the patch primitives has tessellation evaluation shaders specifying the same tessellation mode, spacing, vertex order, and point mode decorations. Two sets of primitives are considered identical if and only if they contain the same number and type of primitives and the generated tessellation coordinates for the vertex numbered \( m \) of the primitive numbered \( n \) are identical for all values of \( m \) and \( n \).

Rule 2 The set of vertices generated along the outer edge of the subdivided primitive in triangle and quad tessellation, and the tessellation coordinates of each, depend only on the corresponding outer tessellation level and the spacing decorations in the tessellation shaders of the pipeline.

Rule 3 The set of vertices generated when subdividing any outer primitive edge is always symmetric. For triangle tessellation, if the subdivision generates a vertex with tessellation coordinates of the form \((0, x, 1-x), (x, 0, 1-x), \) or \((x, 1-x, 0)\), it will also generate a vertex with coordinates of exactly \((0, 1-x, x), (1-x, 0, x), \) or \((1-x, x, 0)\), respectively. For quad tessellation, if the subdivision generates a vertex with coordinates of \((x, 0)\) or \((0, x)\), it will also generate a vertex with coordinates of exactly \((1-x, 0)\) or \((0, 1-x)\), respectively. For isoline tessellation, if it generates vertices at \((0, x)\) and \((1, x)\) where \( x \) is not zero, it will also generate vertices at exactly \((0, 1-x)\) and \((1, 1-x)\), respectively.
Rule 4 The set of vertices generated when subdividing outer edges in triangular and quad tessellation must be independent of the specific edge subdivided, given identical outer tessellation levels and spacing. For example, if vertices at (x, 1 - x, 0) and (1-x, x, 0) are generated when subdividing the w = 0 edge in triangular tessellation, vertices must be generated at (x, 0, 1-x) and (1-x, 0, x) when subdividing an otherwise identical v = 0 edge. For quad tessellation, if vertices at (x, 0) and (1-x, 0) are generated when subdividing the v = 0 edge, vertices must be generated at (0, x) and (0, 1-x) when subdividing an otherwise identical u = 0 edge.

Rule 5 When processing two patches that are identical in all respects enumerated in rule 1 except for vertex order, the set of triangles generated for triangle and quad tessellation must be identical except for vertex and triangle order. For each triangle n1 produced by processing the first patch, there must be a triangle n2 produced when processing the second patch each of whose vertices has the same tessellation coordinates as one of the vertices in n1.

Rule 6 When processing two patches that are identical in all respects enumerated in rule 1 other than matching outer tessellation levels and/or vertex order, the set of interior triangles generated for triangle and quad tessellation must be identical in all respects except for vertex and triangle order. For each interior triangle n1 produced by processing the first patch, there must be a triangle n2 produced when processing the second patch each of whose vertices has the same tessellation coordinates as one of the vertices in n1. A triangle produced by the tessellator is considered an interior triangle if none of its vertices lie on an outer edge of the subdivided primitive.

Rule 7 For quad and triangle tessellation, the set of triangles connecting an inner and outer edge depends only on the inner and outer tessellation levels corresponding to that edge and the spacing decorations.

Rule 8 The value of all defined components of TessCoord will be in the range [0, 1]. Additionally, for any defined component x of TessCoord, the results of computing 1.0-x in a tessellation evaluation shader will be exact. If any floating-point values in the range [0, 1] fail to satisfy this property, such values must not be used as tessellation coordinate components.
Appendix I: Lexicon

This appendix defines terms, abbreviations, and API prefixes used in the Specification.

Glossary

The terms defined in this section are used consistently throughout the Specification and may be used with or without capitalization.

**Accessible (Descriptor Binding)**

A descriptor binding is accessible to a shader stage if that stage is included in the `stageFlags` of the descriptor binding. Descriptors using that binding can only be used by stages in which they are accessible.

**Acquire Operation (Resource)**

An operation that acquires ownership of an image subresource or buffer range.

**Adjacent Vertex**

A vertex in an adjacency primitive topology that is not part of a given primitive, but is accessible in geometry shaders.

**Alias (API type/command)**

An identical definition of another API type/command with the same behavior but a different name.

**Aliased Range (Memory)**

A range of a device memory allocation that is bound to multiple resources simultaneously.

**Allocation Scope**

An association of a host memory allocation to a parent object or command, where the allocation's lifetime ends before or at the same time as the parent object is freed or destroyed, or during the parent command.

**Aspect (Image)**

Some image types contain multiple kinds (called “aspects”) of data for each pixel, where each aspect is used in a particular way by the pipeline and may be stored differently or separately from other aspects. For example, the color components of an image format make up the color aspect of the image, and can be used as a framebuffer color attachment. Some operations, like depth testing, operate only on specific aspects of an image.

**Attachment (Render Pass)**

A zero-based integer index name used in render pass creation to refer to a framebuffer attachment that is accessed by one or more subpasses. The index also refers to an attachment description which includes information about the properties of the image view that will later be attached.
Availability Operation
An operation that causes the values generated by specified memory write accesses to become available for future access.

Available
A state of values written to memory that allows them to be made visible.

Back-Facing
See Facingness.

Batch
A single structure submitted to a queue as part of a queue submission command, describing a set of queue operations to execute.

Backwards Compatibility
A given version of the API is backwards compatible with an earlier version if an application, relying only on valid behavior and functionality defined by the earlier specification, is able to correctly run against each version without any modification. This assumes no active attempt by that application to not run when it detects a different version.

Binary Semaphore
A semaphore with a boolean payload indicating whether the semaphore is signaled orunsigneded. Represented by a VkSemaphore object created with a semaphore type of VK_SEMAPHORE_TYPE_BINARY.

Binding (Memory)
An association established between a range of a resource object and a range of a memory object. These associations determine the memory locations affected by operations performed on elements of a resource object. Memory bindings are established using the vkBindBufferMemory command for non-sparse buffer objects, using the vkBindImageMemory command for non-sparse image objects, and using the vkQueueBindSparse command for sparse resources.

Blend Constant
Four floating point (RGBA) values used as an input to blending.

Blending
Arithmetic operations between a fragment color value and a value in a color attachment that produce a final color value to be written to the attachment.

Buffer
A resource that represents a linear array of data in device memory. Represented by a VkBuffer object.

Buffer Device Address
A 64-bit value used in a shader to access buffer memory through the PhysicalStorageBuffer storage class.
Buffer View
An object that represents a range of a specific buffer, and state controlling how the contents are interpreted. Represented by a \texttt{VkBufferView} object.

Built-In Variable
A variable decorated in a shader, where the decoration makes the variable take values provided by the execution environment or values that are generated by fixed-function pipeline stages.

Built-In Interface Block
A block defined in a shader containing only variables decorated with built-in decorations, and is used to match against other shader stages.

Clip Coordinates
The homogeneous coordinate space in which vertex positions (\texttt{Position} decoration) are written by \texttt{pre-rasterization} shader stages.

Clip Distance
A built-in output from \texttt{pre-rasterization} shader stages defining a clip half-space against which the primitive is clipped.

Clip Volume
The intersection of the view volume with all clip half-spaces.

Color Attachment
A subpass attachment point, or image view, that is the target of fragment color outputs and blending.

Color Renderable Format
A \texttt{VkFormat} where \texttt{VK\_FORMAT\_FEATURE\_COLOR\_ATTACHMENT\_BIT} is set in one of the following, depending on the image's tiling:

- \texttt{VkFormatProperties::linearTilingFeatures}
- \texttt{VkFormatProperties::optimalTilingFeatures}

Combined Image Sampler
A descriptor type that includes both a sampled image and a sampler.

Command Buffer
An object that records commands to be submitted to a queue. Represented by a \texttt{VkCommandBuffer} object.

Command Pool
An object that command buffer memory is allocated from, and that owns that memory. Command pools aid multithreaded performance by enabling different threads to use different allocators, without internal synchronization on each use. Represented by a \texttt{VkCommandPool} object.
Compatible Allocator
When allocators are compatible, allocations from each allocator can be freed by the other allocator.

Compatible Image Formats
When formats are compatible, images created with one of the formats can have image views created from it using any of the compatible formats. Also see Size-Compatible Image Formats.

Compatible Queues
Queues within a queue family. Compatible queues have identical properties.

Complete Mipmap Chain
The entire set of mipmap levels that can be provided for an image, from the largest application specified mipmap level size down to the minimum mipmap size. See Image Mipmap Sizing.

Component (Format)
A distinct part of a format. Color components are represented with R, G, B, and A. Depth and stencil components are represented with D and S. Formats can have multiple instances of the same component. Some formats have other notations such as E or X which are not considered a component of the format.

Compressed Texel Block
An element of an image having a block-compressed format, comprising a rectangular block of texel values that are encoded as a single value in memory. Compressed texel blocks of a particular block-compressed format have a corresponding width, height, and depth defining the dimensions of these elements in units of texels, and a size in bytes of the encoding in memory.

Constant Integral Expressions
A SPIR-V constant instruction whose type is OpTypeInt. See Constant Instruction in section 2.2.1 “Instructions” of the Khronos SPIR-V Specification.

Coverage Index
The index of a sample in the coverage mask.

Coverage Mask
A bitfield associated with a fragment representing the samples that were determined to be covered based on the result of rasterization, and then subsequently modified by fragment operations or the fragment shader.

Cull Distance
A built-in output from pre-rasterization shader stages defining a cull half-space where the primitive is rejected if all vertices have a negative value for the same cull distance.

Cull Volume
The intersection of the view volume with all cull half-spaces.

Decoration (SPIR-V)
Auxiliary information such as built-in variables, stream numbers, invariance, interpolation type,
relaxed precision, etc., added to variables or structure-type members through decorations.

**Deprecated (feature)**

A feature is deprecated if it is no longer recommended as the correct or best way to achieve its intended purpose.

**Depth/Stencil Attachment**

A subpass attachment point, or image view, that is the target of depth and/or stencil test operations and writes.

**Depth/Stencil Format**

A `VkFormat` that includes depth and/or stencil components.

**Depth/Stencil Image (or ImageView)**

A `VkImage` (or `VkImageView`) with a depth/stencil format.

**Depth/Stencil Resolve Attachment**

A subpass attachment point, or image view, that is the target of a multisample resolve operation from the corresponding depth/stencil attachment at the end of the subpass.

**Derivative Group**

A set of fragment shader invocations that cooperate to compute derivatives, including implicit derivatives for sampled image operations.

**Descriptor**

Information about a resource or resource view written into a descriptor set that is used to access the resource or view from a shader.

**Descriptor Binding**

An entry in a descriptor set layout corresponding to zero or more descriptors of a single descriptor type in a set. Defined by a `VkDescriptorsetLayoutBinding` structure.

**Descriptor Pool**

An object that descriptor sets are allocated from, and that owns the storage of those descriptor sets. Descriptor pools aid multithreaded performance by enabling different threads to use different allocators, without internal synchronization on each use. Represented by a `VkDescriptorPool` object.

**Descriptor Set**

An object that resource descriptors are written into via the API, and that can be bound to a command buffer such that the descriptors contained within it can be accessed from shaders. Represented by a `VkDescriptorSet` object.

**Descriptor Set Layout**

An object defining the set of resources (types and counts) and their relative arrangement (in the binding namespace) within a descriptor set. Used when allocating descriptor sets and when creating pipeline layouts. Represented by a `VkDescriptorsetLayout` object.
Device
The processor(s) and execution environment that perform tasks requested by the application via the Vulkan API.

Device Group
A set of physical devices that support accessing each other’s memory and recording a single command buffer that can be executed on all the physical devices.

Device Index
A zero-based integer that identifies one physical device from a logical device. A device index is valid if it is less than the number of physical devices in the logical device.

Device Mask
A bitmask where each bit represents one device index. A device mask value is valid if every bit that is set in the mask is at a bit position that is less than the number of physical devices in the logical device.

Device Memory
Memory accessible to the device. Represented by a VkDeviceMemory object.

Device-Level Command
Any command that is dispatched from a logical device, or from a child object of a logical device.

Device-Level Functionality
All device-level commands and objects, and their structures, enumerated types, and enumerants.

Device-Level Object
Logical device objects and their child objects. For example, VkDevice, VkQueue, and VkCommandBuffer objects are device-level objects.

Device-Local Memory
Memory that is connected to the device, and may be more performant for device access than host-local memory.

Direct Drawing Commands
Drawing commands that take all their parameters as direct arguments to the command (and not sourced via structures in buffer memory as the indirect drawing commands). Includes vkCmdDraw, and vkCmdDrawIndexed.

Disjoint
Disjoint planes are image planes to which memory is bound independently. A disjoint image consists of multiple disjoint planes, and is created with the VK_IMAGE_CREATE_DISJOINT_BIT bit set.

Dispatchable Command
A non-global command. The first argument to each dispatchable command is a dispatchable handle type.
Dispatchable Handle
A handle of a pointer handle type which may be used by layers as part of intercepting API commands.

Dispatching Commands
Commands that provoke work using a compute pipeline. Includes `vkCmdDispatch` and `vkCmdDispatchIndirect`.

Drawing Commands
Commands that provoke work using a graphics pipeline. Includes `vkCmdDraw`, `vkCmdDrawIndexed`, `vkCmdDrawIndirectCount`, `vkCmdDrawIndexedIndirectCount`, `vkCmdDrawIndirect`, and `vkCmdDrawIndexedIndirect`.

Duration (Command)
The duration of a Vulkan command refers to the interval between calling the command and its return to the caller.

Dynamic Storage Buffer
A storage buffer whose offset is specified each time the storage buffer is bound to a command buffer via a descriptor set.

Dynamic Uniform Buffer
A uniform buffer whose offset is specified each time the uniform buffer is bound to a command buffer via a descriptor set.

Dynamically Uniform
See Dynamically Uniform in section 2.2 “Terms” of the Khronos SPIR-V Specification.

Element
Arrays are composed of multiple elements, where each element exists at a unique index within that array. Used primarily to describe data passed to or returned from the Vulkan API.

Explicitly-Enabled Layer
A layer enabled by the application by adding it to the enabled layer list in `vkCreateInstance` or `vkCreateDevice`.

Event
A synchronization primitive that is signaled when execution of previous commands completes through a specified set of pipeline stages. Events can be waited on by the device and polled by the host. Represented by a `VkEvent` object.

Executable State (Command Buffer)
A command buffer that has ended recording commands and can be executed. See also Initial State and Recording State.

Execution Dependency
A dependency that guarantees that certain pipeline stages’ work for a first set of commands has completed execution before certain pipeline stages’ work for a second set of commands begins.
execution. This is accomplished via pipeline barriers, subpass dependencies, events, or implicit ordering operations.

**Execution Dependency Chain**
A sequence of execution dependencies that transitively act as a single execution dependency.

**Explicit chroma reconstruction**
An implementation of sampler Y’C_bC_r conversion which reconstructs reduced-resolution chroma samples to luma resolution and then separately performs texture sample interpolation. This is distinct from an implicit implementation, which incorporates chroma sample reconstruction into texture sample interpolation.

**Extension Scope**
The set of objects and commands that **can** be affected by an extension. Extensions are either device scope or instance scope.

**Extending Structure**
A structure type which may appear in the **pNext** chain of another structure, extending the functionality of the other structure. Extending structures may be defined by either core API versions or extensions.

**External Handle**
A resource handle which has meaning outside of a specific Vulkan device or its parent instance. External handles **may** be used to share resources between multiple Vulkan devices in different instances, or between Vulkan and other APIs. Some external handle types correspond to platform-defined handles, in which case the resource **may** outlive any particular Vulkan device or instance and **may** be transferred between processes, or otherwise manipulated via functionality defined by the platform for that handle type.

**External synchronization**
A type of synchronization **required** of the application, where parameters defined to be externally synchronized **must** not be used simultaneously in multiple threads.

**Facingness (Polygon)**
A classification of a polygon as either front-facing or back-facing, depending on the orientation (winding order) of its vertices.

**Facingness (Fragment)**
A fragment is either front-facing or back-facing, depending on the primitive it was generated from. If the primitive was a polygon (regardless of polygon mode), the fragment inherits the facingness of the polygon. All other fragments are front-facing.

**Fence**
A synchronization primitive that is signaled when a set of batches or sparse binding operations complete execution on a queue. Fences **can** be waited on by the host. Represented by a **VkFence** object.
Flat Shading
A property of a vertex attribute that causes the value from a single vertex (the provoking vertex) to be used for all vertices in a primitive, and for interpolation of that attribute to return that single value unaltered.

Format Features
A set of features from VkFormatFeatureFlagBits that a VkFormat is capable of using for various commands. The list is determined by factors such as VkImageTiling.

Fragment
A rectangular framebuffer region with associated data produced by rasterization and processed by fragment operations including the fragment shader.

Fragment Area
The width and height, in pixels, of a fragment.

Fragment Input Attachment Interface
Variables with UniformConstant storage class and a decoration of InputAttachmentIndex that are statically used by a fragment shader's entry point, which receive values from input attachments.

Fragment Output Interface
A fragment shader entry point's variables with Output storage class, which output to color and/or depth/stencil attachments.

Framebuffer
A collection of image views and a set of dimensions that, in conjunction with a render pass, define the inputs and outputs used by drawing commands. Represented by a VkFramebuffer object.

Framebuffer Attachment
One of the image views used in a framebuffer.

Framebuffer Coordinates
A coordinate system in which adjacent pixels' coordinates differ by 1 in x and/or y, with (0,0) in the upper left corner and pixel centers at half-integers.

Framebuffer-Space
Operating with respect to framebuffer coordinates.

Framebuffer-Local
A framebuffer-local dependency guarantees that only for a single framebuffer region, the first set of operations happens-before the second set of operations.

Framebuffer-Global
A framebuffer-global dependency guarantees that for all framebuffer regions, the first set of operations happens-before the second set of operations.
Framebuffer Region
A framebuffer region is a set of sample (x, y, layer, sample) coordinates that is a subset of the entire framebuffer.

Front-Facing
See Facingness.

Full Compatibility
A given version of the API is fully compatible with another version if an application, relying only on valid behavior and functionality defined by either of those specifications, is able to correctly run against each version without any modification. This assumes no active attempt by that application to not run when it detects a different version.

Global Command
A Vulkan command for which the first argument is not a dispatchable handle type.

Global Workgroup
A collection of local workgroups dispatched by a single dispatching command.

Handle
An opaque integer or pointer value used to refer to a Vulkan object. Each object type has a unique handle type.

Happen-after, happens-after
A transitive, irreflexive and antisymmetric ordering relation between operations. An execution dependency with a source of A and a destination of B enforces that B happens-after A. The inverse relation of happens-before.

Happen-before, happens-before
A transitive, irreflexive and antisymmetric ordering relation between operations. An execution dependency with a source of A and a destination of B enforces that A happens-before B. The inverse relation of happens-after.

Helper Invocation
A fragment shader invocation that is created solely for the purposes of evaluating derivatives for use in non-helper fragment shader invocations, and which does not have side effects.

Host
The processor(s) and execution environment that the application runs on, and that the Vulkan API is exposed on.

Host Mapped Device Memory
Device memory that is mapped for host access using vkMapMemory.

Host Memory
Memory not accessible to the device, used to store implementation data structures.
**Host-Accessible Subresource**
A buffer, or a linear image subresource in either the **VK_IMAGE_LAYOUT_PREINITIALIZED** or **VK_IMAGE_LAYOUT_GENERAL** layout. Host-accessible subresources have a well-defined addressing scheme which can be used by the host.

**Host-Local Memory**
Memory that is not local to the device, and **may** be less performant for device access than device-local memory.

**Host-Visible Memory**
Device memory that **can** be mapped on the host and **can** be read and written by the host.

**ICD**
Installable Client Driver. An ICD is represented as a **VkPhysicalDevice**.

**Identically Defined Objects**
Objects of the same type where all arguments to their creation or allocation functions, with the exception of **pAllocator**, are

1. Vulkan handles which refer to the same object or
2. identical scalar or enumeration values or
3. Host pointers which point to an array of values or structures which also satisfy these three constraints.

**Image**
A resource that represents a multi-dimensional formatted interpretation of device memory. Represented by a **VkImage** object.

**Image Subresource**
A specific mipmap level, layer, and set of aspects of an image.

**Image Subresource Range**
A set of image subresources that are contiguous mipmap levels and layers.

**Image View**
An object that represents an image subresource range of a specific image, and state controlling how the contents are interpreted. Represented by a **VkImageView** object.

**Immutable Sampler**
A sampler descriptor provided at descriptor set layout creation time for a specific binding. This sampler is then used for that binding in all descriptor sets allocated with the layout, and it **cannot** be changed.

**Implicit chroma reconstruction**
An implementation of sampler Y’C₆₆C₇ conversion which reconstructs the reduced-resolution chroma samples directly at the sample point, as part of the normal texture sampling operation. This is distinct from an **explicit chroma reconstruction** implementation, which reconstructs the reduced-resolution chroma samples to the resolution of the luma samples, then filters the result.
as part of texture sample interpolation.

**Implicitly-Enabled Layer**
A layer enabled by a loader-defined mechanism outside the Vulkan API, rather than explicitly by the application during instance or device creation.

**Index Buffer**
A buffer bound via `vkCmdBindIndexBuffer` which is the source of index values used to fetch vertex attributes for a `vkCmdDrawIndexed` or `vkCmdDrawIndexedIndirect` command.

**Indexed Drawing Commands**
*Drawing commands* which use an *index buffer* as the source of index values used to fetch vertex attributes for a *drawing command*. Includes `vkCmdDrawIndexed`, `vkCmdDrawIndexedIndirectCount`, and `vkCmdDrawIndexedIndirect`.

**Indirect Commands**
Drawing or dispatching commands that source some of their parameters from structures in buffer memory. Includes `vkCmdDrawIndirect`, `vkCmdDrawIndexedIndirect`, `vkCmdDrawIndirectCount`, `vkCmdDrawIndexedIndirectCount`, and `vkCmdDispatchIndirect`.

**Indirect Drawing Commands**
*Drawing commands* that source some of their parameters from structures in buffer memory. Includes `vkCmdDrawIndirect`, `vkCmdDrawIndirectCount`, `vkCmdDrawIndexedIndirectCount`, and `vkCmdDrawIndexedIndirect`.

**Initial State (Command Buffer)**
A command buffer that has not begun recording commands. See also Recording State and Executable State.

**Inline Uniform Block**
A descriptor type that represents uniform data stored directly in descriptor sets, and supports read-only access in a shader.

**Input Attachment**
A descriptor type that represents an image view, and supports unfiltered read-only access in a shader, only at the fragment’s location in the view.

**Instance**
The top-level Vulkan object, which represents the application’s connection to the implementation. Represented by a `VkInstance` object.

**Instance-Level Command**
Any command that is dispatched from an instance, or from a child object of an instance, except for physical devices and their children.

**Instance-Level Functionality**
All instance-level commands and objects, and their structures, enumerated types, and enumerants.
**Instance-Level Object**
High-level Vulkan objects, which are not physical devices, nor children of physical devices. For example, `VkInstance` is an instance-level object.

**Instance (Memory)**
In a logical device representing more than one physical device, some device memory allocations have the requested amount of memory allocated multiple times, once for each physical device in a device mask. Each such replicated allocation is an instance of the device memory.

**Instance (Resource)**
In a logical device representing more than one physical device, buffer and image resources exist on all physical devices but can be bound to memory differently on each. Each such replicated resource is an instance of the resource.

**Internal Synchronization**
A type of synchronization **required** of the implementation, where parameters not defined to be externally synchronized **may** require internal mutexing to avoid multithreaded race conditions.

**Invocation (Shader)**
A single execution of an entry point in a SPIR-V module. For example, a single vertex's execution of a vertex shader or a single fragment's execution of a fragment shader.

**Invocation Group**
A set of shader invocations that are executed in parallel and that **must** execute the same control flow path in order for control flow to be considered dynamically uniform.

**Linear Resource**
A resource is *linear* if it is one of the following:

- a `VkBuffer`
- a `VkImage` created with `VK_IMAGE_TILING_LINEAR`

A resource is *non-linear* if it is one of the following:

- a `VkImage` created with `VK_IMAGE_TILING_OPTIMAL`

**Local Workgroup**
A collection of compute shader invocations invoked by a single dispatching command, which share data via `WorkgroupLocal` variables and can synchronize with each other.

**Logical Device**
An object that represents the application’s interface to the physical device. The logical device is the parent of most Vulkan objects. Represented by a `VkDevice` object.

**Logical Operation**
Bitwise operations between a fragment color value and a value in a color attachment, that produce a final color value to be written to the attachment.
Lost Device
A state that a logical device may be in as a result of unrecoverable implementation errors, or other exceptional conditions.

Mappable
See Host-Visible Memory.

Memory Dependency
A memory dependency is an execution dependency which includes availability and visibility operations such that:

• The first set of operations happens-before the availability operation
• The availability operation happens-before the visibility operation
• The visibility operation happens-before the second set of operations

Memory Domain
A memory domain is an abstract place to which memory writes are made available by availability operations and memory domain operations. The memory domains correspond to the set of agents that the write can then be made visible to. The memory domains are host, device, shader, workgroup instance (for workgroup instance there is a unique domain for each compute workgroup) and subgroup instance (for subgroup instance there is a unique domain for each subgroup).

Memory Domain Operation
An operation that makes the writes that are available to one memory domain available to another memory domain.

Memory Heap
A region of memory from which device memory allocations can be made.

Memory Type
An index used to select a set of memory properties (e.g. mappable, cached) for a device memory allocation.

Minimum Miplevel Size
The smallest size that is permitted for a miplevel. For conventional images this is 1x1x1. See Image Miplevel Sizing.

Mip Tail Region
The set of mipmap levels of a sparse residency texture that are too small to fill a sparse block, and that must all be bound to memory collectively and opaquely.

Multi-planar
A multi-planar format (or “planar format”) is an image format consisting of more than one plane, identifiable with a _2PLANE or _3PLANE component to the format name and listed in Formats requiring sampler Y’C_bC_r conversion for VK_IMAGE_ASPECT_COLOR_BIT image views. A multi-planar image (or “planar image”) is an image of a multi-planar format.
Non-Dispatchable Handle
A handle of an integer handle type. Handle values may not be unique, even for two objects of the same type.

Non-Indexed Drawing Commands
Drawing commands for which the vertex attributes are sourced in linear order from the vertex input attributes for a drawing command (i.e. they do not use an index buffer). Includes vkCmdDraw, vkCmdDrawIndirectCount, and vkCmdDrawIndirect.

Normalized
A value that is interpreted as being in the range [0,1] as a result of being implicitly divided by some other value.

Normalized Device Coordinates
A coordinate space after perspective division is applied to clip coordinates, and before the viewport transformation converts them to framebuffer coordinates.

Obsoleted (feature)
A feature is obsolete if it can no longer be used.

Opaque Capture Address
A 64-bit value representing the device address of a buffer or memory object that is expected to be used by trace capture/replay tools in combination with the bufferDeviceAddress feature.

Overlapped Range (Aliased Range)
The aliased range of a device memory allocation that intersects a given image subresource of an image or range of a buffer.

Ownership (Resource)
If an entity (e.g. a queue family) has ownership of a resource, access to that resource is well-defined for access by that entity.

Packed Format
A format whose components are stored as a single texel block in memory, with their relative locations defined within that element.

Payload
Importable or exportable reference to the internal data of an object in Vulkan.

Peer Memory
An instance of memory corresponding to a different physical device than the physical device performing the memory access, in a logical device that represents multiple physical devices.

Physical Device
An object that represents a single device in the system. Represented by a VkPhysicalDevice object.
**Physical-Device-Level Command**

Any command that is dispatched from a physical device.

**Physical-Device-Level Functionality**

All physical-device-level commands and objects, and their structures, enumerated types, and enumerants.

**Physical-Device-Level Object**

Physical device objects. For example, `VkPhysicalDevice` is a physical-device-level object.

**Pipeline**

An object controlling how graphics or compute work is executed on the device. A pipeline includes one or more shaders, as well as state controlling any non-programmable stages of the pipeline. Represented by a `VkPipeline` object.

**Pipeline Barrier**

An execution and/or memory dependency recorded as an explicit command in a command buffer, that forms a dependency between the previous and subsequent commands.

**Pipeline Cache**

An object that can be used to collect and retrieve information from pipelines as they are created, and can be populated with previously retrieved information in order to accelerate pipeline creation. Represented by a `VkPipelineCache` object.

**Pipeline Layout**

An object defining the set of resources (via a collection of descriptor set layouts) and push constants used by pipelines that are created using the layout. Used when creating a pipeline and when binding descriptor sets and setting push constant values. Represented by a `VkPipelineLayout` object.

**Pipeline Stage**

A logically independent execution unit that performs some of the operations defined by an action command.

**pNext Chain**

A set of structures chained together through their `pNext` members.

**Planar**

See multi-planar.

**Plane**

An image plane is part of the representation of an image, containing a subset of the color components necessary to represent the texels in the image and with a contiguous mapping of coordinates to bound memory. Most images consist only of a single plane, but some formats spread the components across multiple image planes. The host-accessible properties of each image plane are accessible for a linear layout using `vkGetImageSubresourceLayout`. If a multi-planar image is created with the `VK_IMAGE_CREATE_DISJOINT_BIT` bit set, the image is described as disjoint, and its planes are therefore bound to memory independently.
Point Sampling (Rasterization)
A rule that determines whether a fragment sample location is covered by a polygon primitive by testing whether the sample location is in the interior of the polygon in framebuffer-space, or on the boundary of the polygon according to the tie-breaking rules.

Potential Format Features
The union of all `VkFormatFeatureFlagBits` that the implementation supports for a specified `VkFormat`, over all supported image tilings.

Pre-rasterization
Operations that execute before rasterization, and any state associated with those operations.

Preserve Attachment
One of a list of attachments in a subpass description that is not read or written by the subpass, but that is read or written on earlier and later subpasses and whose contents must be preserved through this subpass.

Primary Command Buffer
A command buffer that can execute secondary command buffers, and can be submitted directly to a queue.

Primitive Topology
State controlling how vertices are assembled into primitives, e.g. as lists of triangles, strips of lines, etc.

Promoted (feature)
A feature from an older extension is considered promoted if it is made available as part of a new core version or newer extension with wider support.

Protected Buffer
A buffer to which protected device memory can be bound.

Protected-capable Device Queue
A device queue to which protected command buffers can be submitted.

Protected Command Buffer
A command buffer which can be submitted to a protected-capable device queue.

Protected Device Memory
Device memory which can be visible to the device but must not be visible to the host.

Protected Image
An image to which protected device memory can be bound.

Provisional
A feature is released provisionally in order to get wider feedback on the functionality before it is finalized. Provisional features may change in ways that break backwards compatibility, and thus are not recommended for use in production applications.
**Provoking Vertex**

The vertex in a primitive from which flat shaded attribute values are taken. This is generally the “first” vertex in the primitive, and depends on the primitive topology.

**Push Constants**

A small bank of values writable via the API and accessible in shaders. Push constants allow the application to set values used in shaders without creating buffers or modifying and binding descriptor sets for each update.

**Push Constant Interface**

The set of variables with `PushConstant` storage class that are statically used by a shader entry point, and which receive values from push constant commands.

**Descriptor Update Template**

An object specifying a mapping from descriptor update information in host memory to elements in a descriptor set, which helps enable more efficient descriptor set updates.

**Query Pool**

An object containing a number of query entries and their associated state and results. Represented by a `VkQueryPool` object.

**Queue**

An object that executes command buffers and sparse binding operations on a device. Represented by a `VkQueue` object.

**Queue Family**

A set of queues that have common properties and support the same functionality, as advertised in `VkQueueFamilyProperties`.

**Queue Operation**

A unit of work to be executed by a specific queue on a device, submitted via a queue submission command. Each queue submission command details the specific queue operations that occur as a result of calling that command. Queue operations typically include work that is specific to each command, and synchronization tasks.

**Queue Submission**

Zero or more batches and an optional fence to be signaled, passed to a command for execution on a queue. See the Devices and Queues chapter for more information.

**Recording State (Command Buffer)**

A command buffer that is ready to record commands. See also Initial State and Executable State.

**Release Operation (Resource)**

An operation that releases ownership of an image subresource or buffer range.

**Render Pass**

An object that represents a set of framebuffer attachments and phases of rendering using those attachments. Represented by a `VkRenderPass` object.
**Render Pass Instance**
A use of a render pass in a command buffer.

**Required Extensions**
Extensions that **must** be enabled alongside extensions dependent on them (see Extension Dependencies).

**Reset (Command Buffer)**
Resetting a command buffer discards any previously recorded commands and puts a command buffer in the initial state.

**Residency Code**
An integer value returned by sparse image instructions, indicating whether any sparse unbound texels were accessed.

**Resolve Attachment**
A subpass attachment point, or image view, that is the target of a multisample resolve operation from the corresponding color attachment at the end of the subpass.

**Sample Index**
The index of a sample within a single set of samples.

**Sample Shading**
Invoking the fragment shader multiple times per fragment, with the covered samples partitioned among the invocations.

**Sampled Image**
A descriptor type that represents an image view, and supports filtered (sampled) and unfiltered read-only access in a shader.

**Sampler**
An object containing state controlling how sampled image data is sampled (or filtered) when accessed in a shader. Also a descriptor type describing the object. Represented by a VkSampler object.

**Secondary Command Buffer**
A command buffer that **can** be executed by a primary command buffer, and **must** not be submitted directly to a queue.

**Self-Dependency**
A subpass dependency from a subpass to itself, i.e. with srcSubpass equal to dstSubpass. A self-dependency is not automatically performed during a render pass instance, rather a subset of it **can** be performed via vkCmdPipelineBarrier during the subpass.

**Semaphore**
A synchronization primitive that supports signal and wait operations, and **can** be used to synchronize operations within a queue or across queues. Represented by a VkSemaphore object.
Shader
Instructions selected (via an entry point) from a shader module, which are executed in a shader stage.

Shader Code
A stream of instructions used to describe the operation of a shader.

Shader Module
A collection of shader code, potentially including several functions and entry points, that is used to create shaders in pipelines. Represented by a VkShaderModule object.

Shader Stage
A stage of the graphics or compute pipeline that executes shader code.

Side Effect
A store to memory or atomic operation on memory from a shader invocation.

Single-plane format
A format that is not multi-planar.

Size-Compatible Image Formats
When a compressed image format and an uncompressed image format are size-compatible, it means that the texel block size of the uncompressed format must equal the texel block size of the compressed format.

Sparse Block
An element of a sparse resource that can be independently bound to memory. Sparse blocks of a particular sparse resource have a corresponding size in bytes that they use in the bound memory.

Sparse Image Block
A sparse block in a sparse partially-resident image. In addition to the sparse block size in bytes, sparse image blocks have a corresponding width, height, and depth defining the dimensions of these elements in units of texels or compressed texel blocks, the latter being used in case of sparse images having a block-compressed format.

Sparse Unbound Texel
A texel read from a region of a sparse texture that does not have memory bound to it.

Static Use
An object in a shader is statically used by a shader entry point if any function in the entry point’s call tree contains an instruction using the object. A reference in the entry point’s interface list does not constitute a static use. Static use is used to constrain the set of descriptors used by a shader entry point.

Storage Buffer
A descriptor type that represents a buffer, and supports reads, writes, and atomics in a shader.
Storage Image
A descriptor type that represents an image view, and supports unfiltered loads, stores, and atomics in a shader.

Storage Texel Buffer
A descriptor type that represents a buffer view, and supports unfiltered, formatted reads, writes, and atomics in a shader.

Subgroup
A set of shader invocations that can synchronize and share data with each other efficiently. In compute shaders, the local workgroup is a superset of the subgroup.

Subgroup Mask
A bitmask for all invocations in the current subgroup with one bit per invocation, starting with the least significant bit in the first vector component, continuing to the last bit (less than SubgroupSize) in the last required vector component.

Subpass
A phase of rendering within a render pass, that reads and writes a subset of the attachments.

Subpass Dependency
An execution and/or memory dependency between two subpasses described as part of render pass creation, and automatically performed between subpasses in a render pass instance. A subpass dependency limits the overlap of execution of the pair of subpasses, and can provide guarantees of memory coherence between accesses in the subpasses.

Subpass Description
Lists of attachment indices for input attachments, color attachments, depth/stencil attachment, resolve attachments, depth/stencil resolve, and preserve attachments used by the subpass in a render pass.

Subset (Self-Dependency)
A subset of a self-dependency is a pipeline barrier performed during the subpass of the self-dependency, and whose stage masks and access masks each contain a subset of the bits set in the identically named mask in the self-dependency.

Texel Block
A single addressable element of an image with an uncompressed VkFormat, or a single compressed block of an image with a compressed VkFormat.

Texel Block Size
The size (in bytes) used to store a texel block of a compressed or uncompressed image.

Texel Coordinate System
One of three coordinate systems (normalized, unnormalized, integer) defining how texel coordinates are interpreted in an image or a specific mipmap level of an image.
Timeline Semaphore
A semaphore with a strictly increasing 64-bit unsigned integer payload indicating whether the semaphore is signaled with respect to a particular reference value. Represented by a VkSemaphore object created with a semaphore type of VK_SEMAPHORE_TYPE_TIMELINE.

Uniform Texel Buffer
A descriptor type that represents a buffer view, and supports unfiltered, formatted, read-only access in a shader.

Uniform Buffer
A descriptor type that represents a buffer, and supports read-only access in a shader.

Units in the Last Place (ULP)
A measure of floating-point error loosely defined as the smallest representable step in a floating-point format near a given value. For the precise definition see Precision and Operation of SPIR-V instructions or Jean-Michel Muller, “On the definition of ulp(x)”, RR-5504, INRIA. Other sources may also use the term “unit of least precision”.

Unnormalized
A value that is interpreted according to its conventional interpretation, and is not normalized.

Unprotected Buffer
A buffer to which unprotected device memory can be bound.

Unprotected Command Buffer
A command buffer which can be submitted to an unprotected device queue or a protected-capable device queue.

Unprotected Device Memory
Device memory which can be visible to the device and can be visible to the host.

Unprotected Image
An image to which unprotected device memory can be bound.

User-Defined Variable Interface
A shader entry point’s variables with Input or Output storage class that are not built-in variables.

Vertex Input Attribute
A graphics pipeline resource that produces input values for the vertex shader by reading data from a vertex input binding and converting it to the attribute’s format.

Variable-Sized Descriptor Binding
A descriptor binding whose size will be specified when a descriptor set is allocated using this layout.

Vertex Input Binding
A graphics pipeline resource that is bound to a buffer and includes state that affects addressing calculations within that buffer.
Vertex Input Interface

A vertex shader entry point's variables with `Input` storage class, which receive values from vertex input attributes.

View Mask

When multiview is enabled, a view mask is a property of a subpass controlling which views the rendering commands are broadcast to.

View Volume

A subspace in homogeneous coordinates, corresponding to post-projection x and y values between -1 and +1, and z values between 0 and +1.

Viewport Transformation

A transformation from normalized device coordinates to framebuffer coordinates, based on a viewport rectangle and depth range.

Visibility Operation

An operation that causes available values to become visible to specified memory accesses.

Visible

A state of values written to memory that allows them to be accessed by a set of operations.

Common Abbreviations

The abbreviations and acronyms defined in this section are sometimes used in the Specification and the API where they are considered clear and commonplace.

Src

Source

Dst

Destination

Min

Minimum

Max

Maximum

Rect

Rectangle

Info

Information

LOD

Level of Detail
Log
Logarithm

ID
Identifier

UUID
 Universally Unique Identifier

Op
 Operation

R
 Red color component

G
 Green color component

B
 Blue color component

A
 Alpha color component

RTZ
 Round towards zero

RTE
 Round to nearest even

Prefixes

Prefixes are used in the API to denote specific semantic meaning of Vulkan names, or as a label to avoid name clashes, and are explained here:

VK/Vk/vk
 Vulkan namespace
 All types, commands, enumerants and defines in this specification are prefixed with these two characters.

PFN/pfn
 Function Pointer
 Denotes that a type is a function pointer, or that a variable is of a pointer type.

P
 Pointer
 Variable is a pointer.
**vkCmd**

Commands that record commands in command buffers
These API commands do not result in immediate processing on the device. Instead, they record the requested action in a command buffer for execution when the command buffer is submitted to a queue.

**s**

Structure
Used to denote the `VK_STRUCTURE_TYPE*` member of each structure in `sType`
Appendix J: Credits (Informative)

Vulkan 1.3 is the result of contributions from many people and companies participating in the Khronos Vulkan Working Group, as well as input from the Vulkan Advisory Panel.

Members of the Working Group, including the company that they represented at the time of their most recent contribution, are listed in the following section. Some specific contributions made by individuals are listed together with their name.

Working Group Contributors to Vulkan

- Aaron Greig, Codeplay Software Ltd. (version 1.1)
- Aaron Hagan, AMD (version 1.1)
- Adam Jackson, Red Hat (versions 1.0, 1.1)
- Adam Śmigielski, Mobica (version 1.0)
- Aditi Verma, Qualcomm (version 1.3)
- Ahmed Abdelkhalek, AMD (version 1.3)
- Aidan Fabius, Core Avionics & Industrial Inc. (version 1.2)
- Alan Baker, Google (versions 1.1, 1.2, 1.3)
- Alan Ward, Google (versions 1.1, 1.2)
- Alejandro Piñeiro, Igalia (version 1.1)
- Alex Bourd, Qualcomm Technologies, Inc. (versions 1.0, 1.1)
- Alex Crabb, Caster Communications (versions 1.2, 1.3)
- Alex Walters, Imagination Technologies (versions 1.2, 1.3)
- Alexander Galazin, Arm (versions 1.0, 1.1, 1.2, 1.3)
- Alexey Sachkov, Intel (version 1.3)
- Allan MacKinnon, Google (version 1.3)
- Allen Hux, Intel (version 1.0)
- Alon Or-bach, Google (versions 1.0, 1.1, 1.2, 1.3) (WSI technical sub-group chair)
- Anastasia Stulova, Arm (versions 1.2, 1.3)
- Andreas Vasilakis, Think Silicon (version 1.2)
- Andres Gomez, Igalia (version 1.1)
- Andrew Cox, Samsung Electronics (version 1.0)
- Andrew Ellem, Google (version 1.3)
- Andrew Garrard, Imagination Technologies (versions 1.0, 1.1, 1.2, 1.3) (format wrangler)
- Andrew Poole, Samsung Electronics (version 1.0)
- Andrew Rafter, Samsung Electronics (version 1.0)
• Andrew Richards, Codeplay Software Ltd. (version 1.0)
• Andrew Woloszyn, Google (versions 1.0, 1.1)
• Ann Thorsnes, Khronos (versions 1.2, 1.3)
• Antoine Labour, Google (versions 1.0, 1.1)
• Aras Pranckevičius, Unity Technologies (version 1.0)
• Arseny Kapoulkine, Roblox (version 1.3)
• Ashwin Kolhe, NVIDIA (version 1.0)
• Baldur Karlsson, Valve Software (versions 1.1, 1.2, 1.3)
• Barthold Lichtenbelt, NVIDIA (version 1.1)
• Bas Nieuwenhuizen, Google (versions 1.1, 1.2)
• Ben Bowman, Imagination Technologies (version 1.0)
• Benj Lipchak, Unknown (version 1.0)
• Bill Hollings, Brenwill (versions 1.0, 1.1, 1.2, 1.3)
• Bill Licea-Kane, Qualcomm Technologies, Inc. (versions 1.0, 1.1)
• Blaine Kohl, Khronos (versions 1.2, 1.3)
• Bob Fraser, Google (version 1.3)
• Boris Zanin, Mobica (versions 1.2, 1.3)
• Brent E. Insko, Intel (version 1.0)
• Brian Ellis, Qualcomm Technologies, Inc. (version 1.0)
• Brian Paul, VMware (versions 1.2, 1.3)
• Caio Marcelo de Oliveira Filho, Intel (versions 1.2, 1.3)
• Cass Everitt, Oculus VR (versions 1.0, 1.1)
• Cemil Azizoglu, Canonical (version 1.0)
• Chad Versace, Google (versions 1.0, 1.1, 1.2)
• Chang-Hyo Yu, Samsung Electronics (version 1.0)
• Charles Giessen, LunarG (version 1.3)
• Chia-I Wu, LunarG (version 1.0)
• Chris Frascati, Qualcomm Technologies, Inc. (version 1.0)
• Chris Glover, Google (version 1.3)
• Christian Forfang, Arm (version 1.3)
• Christoph Kubisch, NVIDIA (version 1.3)
• Christophe Riccio, Unity Technologies (versions 1.0, 1.1)
• Cody Northrop, Unity Technologies (version 1.0)
• Colin Riley, AMD (version 1.1)
• Cort Stratton, Google (versions 1.1, 1.2)
• Courtney Goeltzenleuchter, Google (versions 1.0, 1.1, 1.3)
• Craig Davies, Huawei (version 1.2)
• Dae Kim, Imagination Technologies (version 1.1)
• Damien Leone, NVIDIA (version 1.0)
• Dan Baker, Oxide Games (versions 1.0, 1.1)
• Dan Ginsburg, Valve Software (versions 1.0, 1.1, 1.2, 1.3)
• Daniel Johnston, Intel (versions 1.0, 1.1)
• Daniel Koch, NVIDIA (versions 1.0, 1.1, 1.2, 1.3)
• Daniel Rakos, AMD (versions 1.0, 1.1, 1.2, 1.3)
• Daniel Stone, Collabora (versions 1.1, 1.2)
• Daniel Vetter, Intel (version 1.2)
• David Airlie, Red Hat (versions 1.0, 1.1, 1.2, 1.3)
• David Mao, AMD (versions 1.0, 1.2)
• David Miller, Miller & Mattson (versions 1.0, 1.1) (Vulkan reference card)
• David Neto, Google (versions 1.0, 1.1, 1.2, 1.3)
• David Pankratz, Huawei (version 1.3)
• David Wilkinson, AMD (version 1.2)
• David Yu, Pixar (version 1.0)
• Dejan Mircevski, Google (version 1.1)
• Diego Novillo, Google (version 1.3)
• Dimitris Georgakakis, Think Silicon (version 1.3)
• Dominik Witczak, AMD (versions 1.0, 1.1, 1.3)
• Donald Scorgie, Imagination Technologies (version 1.2)
• Dzmitry Malyshau, Mozilla (versions 1.1, 1.2, 1.3)
• Ed Hutchins, Oculus (version 1.2)
• Emily Stearns, Khronos (versions 1.2, 1.3)
• François Duranleau, Gameloft (version 1.3)
• Frank (LingJun) Chen, Qualcomm Technologies, Inc. (version 1.0)
• Fred Liao, Mediatek (version 1.0)
• Gabe Dagani, Freescale (version 1.0)
• Gabor Sines, AMD (version 1.2)
• Graeme Leese, Broadcom (versions 1.0, 1.1, 1.2, 1.3)
• Graham Connor, Imagination Technologies (version 1.0)
• Graham Sellers, AMD (versions 1.0, 1.1)
• Graham Wihlidal, Electronic Arts (version 1.3)
• Greg Fischer, LunarG (version 1.1)
• Gregory Grebe, AMD (version 1.3)
• Hai Nguyen, Google (versions 1.2, 1.3)
• Hans-Kristian Arntzen, Valve Software (versions 1.1, 1.2, 1.3)
• Henri Verbeet, Codeweavers (version 1.2)
• Huei Long Wang, Huawei (version 1.3)
• Hwanyong Lee, Kyungpook National University (version 1.0)
• Iago Toral, Igalia (versions 1.1, 1.2)
• Ian Elliott, Google (versions 1.0, 1.1, 1.2)
• Ian Romanick, Intel (versions 1.0, 1.1, 1.3)
• Ivan Briano, Intel (version 1.3)
• James Fitzpatrick, Imagination (version 1.3)
• James Hughes, Oculus VR (version 1.0)
• James Jones, NVIDIA (versions 1.0, 1.1, 1.2, 1.3)
• James Riordon, Khronos (versions 1.2, 1.3)
• Jamie Madill, Google (version 1.3)
• Jan Hermes, Continental Corporation (versions 1.0, 1.1)
• Jan-Harald Fredriksen, Arm (versions 1.0, 1.1, 1.2, 1.3)
• Jason Ekstrand, Intel (versions 1.0, 1.1, 1.2, 1.3)
• Jean-François Roy, Google (versions 1.1, 1.2, 1.3)
• Jeff Bolz, NVIDIA (versions 1.0, 1.1, 1.2, 1.3)
• Jeff Juliano, NVIDIA (versions 1.0, 1.1, 1.2)
• Jeff Leger, Qualcomm Technologies, Inc. (versions 1.1, 1.3)
• Jeff Phillips, Khronos (version 1.3)
• Jeff Vigil, Samsung Electronics (versions 1.0, 1.1, 1.2, 1.3)
• Jens Owen, Google (versions 1.0, 1.1)
• Jeremy Hayes, LunarG (version 1.0)
• Jesse Barker, Unity Technologies (versions 1.0, 1.1, 1.2, 1.3)
• Jesse Hall, Google (versions 1.0, 1.1, 1.2, 1.3)
• Joe Davis, Samsung Electronics (version 1.1)
• Johannes van Waveren, Oculus VR (versions 1.0, 1.1)
• John Anthony, Arm (version 1.2, 1.3)
• John Kessenich, Google (versions 1.0, 1.1, 1.2, 1.3) (SPIR-V and GLSL for Vulkan spec author)
• John McDonald, Valve Software (versions 1.0, 1.1, 1.2, 1.3)
• John Zulauf, LunarG (versions 1.1, 1.2, 1.3)
• Jon Ashburn, LunarG (version 1.0)
• Jon Leech, Independent (versions 1.0, 1.1, 1.2, 1.3) (XML toolchain, normative language, release wrangler)
• Jonas Gustavsson, Samsung Electronics (versions 1.0, 1.1)
• Jonas Meyer, Epic Games (versions 1.2, 1.3)
• Jonathan Hamilton, Imagination Technologies (version 1.0)
• Jordan Justen, Intel (version 1.1)
• Joshua Ashton, Valve Software (version 1.3)
• Jungwoo Kim, Samsung Electronics (versions 1.0, 1.1)
• Jörg Wagner, Arm (version 1.1)
• Kalle Raita, Google (version 1.1)
• Karen Ghavam, LunarG (versions 1.1, 1.2, 1.3)
• Karl Schultz, LunarG (versions 1.1, 1.2)
• Kathleen Mattson, Khronos (versions 1.0, 1.1, 1.2)
• Kaye Mason, Google (version 1.2)
• Keith Packard, Valve (version 1.2)
• Kenneth Benzie, Codeplay Software Ltd. (versions 1.0, 1.1)
• Kenneth Russell, Google (version 1.1)
• Kerch Holt, NVIDIA (versions 1.0, 1.1)
• Kevin O’Neil, AMD (version 1.1)
• Kevin Petit, Arm (version 1.3)
• Kris Rose, Khronos (versions 1.2, 1.3)
• Kristian Kristensen, Intel (versions 1.0, 1.1)
• Krzysztof Iwanicki, Samsung Electronics (version 1.0)
• Larry Seiler, Intel (version 1.0)
• Laura Shubel, Caster Communications (version 1.3)
• Lauri Ilola, Nokia (version 1.1)
• Lei Zhang, Google (version 1.2)
• Lenny Komow, LunarG (versions 1.1, 1.2)
• Liam Middlebrook, NVIDIA (version 1.3)
• Lionel Landwerlin, Intel (versions 1.1, 1.2)
• Lisie Aartsen, Khronos (version 1.3)
• Liz Maitral, Khronos (version 1.2)
• Lou Kramer, AMD (version 1.3)
• Lutz Latta, Lucasfilm (version 1.0)
• Maciej Jesionowski, AMD (version 1.1)
• Mais Alnasser, AMD (version 1.1)
• Marcin Kantoch, AMD (version 1.3)
• Marcin Rogucki, Mobica (version 1.1)
• Maria Rovatsou, Codeplay Software Ltd. (version 1.0)
• Mariusz Merecki, Intel (version 1.3)
• Mark Bellamy, Arm (version 1.2, 1.3)
• Mark Callow, Independent (versions 1.0, 1.1, 1.2, 1.3)
• Mark Kilgard, NVIDIA (versions 1.1, 1.2)
• Mark Lobodzinski, LunarG (versions 1.0, 1.1, 1.2)
• Mark Young, LunarG (versions 1.1, 1.3)
• Markus Tavenrath, NVIDIA (version 1.1)
• Marty Johnson, Khronos (version 1.3)
• Mateusz Przybylkski, Intel (version 1.0)
• Mathias Heyer, NVIDIA (versions 1.0, 1.1)
• Mathias Schott, NVIDIA (versions 1.0, 1.1)
• Mathieu Robart, Arm (version 1.2)
• Matt Netsch, Qualcomm Technologies, Inc. (version 1.1)
• Matthew Rusch, NVIDIA (version 1.3)
• Matthäus Chajdas, AMD (versions 1.1, 1.2, 1.3)
• Maurice Ribble, Qualcomm Technologies, Inc. (versions 1.0, 1.1)
• Maxim Lukyanov, Samsung Electronics (version 1.0)
• Michael Blumenkrantz, Self (version 1.3)
• Michael Lentine, Google (version 1.0)
• Michael O'Hara, AMD (version 1.1)
• Michael Phillip, Samsung Electronics (version 1.2)
• Michael Wong, Codeplay Software Ltd. (version 1.1)
• Michael Worcester, Imagination Technologies (versions 1.0, 1.1)
• Michal Pietrasiuk, Intel (versions 1.0, 1.3)
• Mika Isojarvi, Google (versions 1.0, 1.1)
• Mike Schuchardt, LunarG (versions 1.1, 1.2)
• Mike Stroyan, LunarG (version 1.0)
• Mike Weiblen, LunarG (versions 1.1, 1.2, 1.3)
• Minyoung Son, Samsung Electronics (version 1.0)
• Mitch Singer, AMD (versions 1.0, 1.1, 1.2, 1.3)
• Mythri Venugopal, Samsung Electronics (version 1.0)
• Naveen Leekha, Google (version 1.0)
• Neil Henning, AMD (versions 1.0, 1.1, 1.2, 1.3)
• Neil Hickey, Arm (version 1.2)
• Neil Trevett, NVIDIA (versions 1.0, 1.1, 1.2, 1.3)
• Nick Penwarden, Epic Games (version 1.0)
• Nicolai Hähnle, AMD (version 1.1)
• Niklas Smedberg, Unity Technologies (version 1.0)
• Norbert Nopper, Independent (versions 1.0, 1.1)
• Nuno Subtil, NVIDIA (versions 1.1, 1.2, 1.3)
• Pat Brown, NVIDIA (version 1.0)
• Patrick Cozzi, Independent (version 1.1)
• Patrick Doane, Blizzard Entertainment (version 1.0)
• Peter Lohrmann, AMD (versions 1.0, 1.2)
• Petros Bantolas, Imagination Technologies (version 1.1)
• Philip Rebohle, Valve Software (version 1.3)
• Pierre Boudier, NVIDIA (versions 1.0, 1.1, 1.2, 1.3)
• Pierre-Loup Griffais, Valve Software (versions 1.0, 1.1, 1.2, 1.3)
• Piers Daniell, NVIDIA (versions 1.0, 1.1, 1.2, 1.3)
• Ping Liu, Intel (version 1.3)
• Piotr Bialecki, Intel (version 1.0)
• Piotr Byszewski, Mobica (version 1.3)
• Prabindh Sundareson, Samsung Electronics (version 1.0)
• Pyry Haulos, Google (versions 1.0, 1.1) (Vulkan conformance test subcommittee chair)
• Rachel Bradshaw, Caster Communications (version 1.3)
• Rajeev Rao, Qualcomm (version 1.2)
• Ralph Potter, Samsung Electronics (versions 1.1, 1.2, 1.3)
• Raun Krisch, Samsung Electronics (version 1.3)
• Ray Smith, Arm (versions 1.0, 1.1, 1.2)
• Ricardo Garcia, Igalia (version 1.3)
• Richard Huddy, Samsung Electronics (versions 1.2, 1.3)
• Rob Barris, NVIDIA (version 1.1)
• Rob Stepinski, Transgaming (version 1.0)
• Robert Simpson, Qualcomm Technologies, Inc. (versions 1.0, 1.1, 1.3)
• Rolando Caloca Olivares, Epic Games (versions 1.0, 1.1, 1.2, 1.3)
• Ronan Keryell, Xilinx (version 1.3)
• Roy Ju, Mediatek (version 1.0)
• Rufus Hamade, Imagination Technologies (version 1.0)
• Ruihao Zhang, Qualcomm Technologies, Inc. (versions 1.1, 1.2, 1.3)
• Samuel Huang, Mediatek (version 1.3)
• Samuel Iglesias Gonsalvez, Igalia (version 1.3)
• Sascha Willems, Self (version 1.3)
• Sean Ellis, Arm (version 1.0)
• Sean Harmer, KDAB Group (versions 1.0, 1.1)
• Shannon Woods, Google (versions 1.0, 1.1, 1.2, 1.3)
• Slawomir Cygan, Intel (versions 1.0, 1.1, 1.3)
• Slawomir Grajewski, Intel (versions 1.0, 1.1, 1.3)
• Sorel Bosan, AMD (version 1.1)
• Spencer Fricke, Samsung Electronics (versions 1.2, 1.3)
• Stefanus Du Toit, Google (version 1.0)
• Stephen Huang, Mediatek (version 1.1)
• Steve Hill, Broadcom (versions 1.0, 1.2)
• Steve Viggers, Core Avionics & Industrial Inc. (versions 1.0, 1.2)
• Steve Winston, Holochip (version 1.3)
• Stuart Smith, AMD (versions 1.0, 1.1, 1.2, 1.3)
• Sujeevan Rajayogam, Google (version 1.3)
• Tilmann Scheller, Samsung Electronics (version 1.1)
• Tim Foley, Intel (version 1.0)
• Tim Lewis, Khronos (version 1.3)
• Timo Suoranta, AMD (version 1.0)
• Timothy Lottes, AMD (versions 1.0, 1.1)
• Tobias Hector, AMD (versions 1.0, 1.1, 1.2, 1.3) (validity language and toolchain)
• Tobin Ehlis, LunarG (version 1.0)
• Tom Olson, Arm (versions 1.0, 1.1, 1.2, 1.3) (Working Group chair)
• Tomasz Bednarz, Independent (version 1.1)
• Tomasz Kubale, Intel (version 1.0)
• Tony Barbour, LunarG (versions 1.0, 1.1, 1.2)
• Tony Zlatinski, NVIDIA (version 1.3)
• Victor Eruhimov, Unknown (version 1.1)
• Vikram Kushwaha, NVIDIA (version 1.3)
• Vincent Hindriksen, Stream HPC (versions 1.2, 1.3)
• Wasim Abbas, Arm (version 1.3)
• Wayne Lister, Imagination Technologies (version 1.0)
• Wolfgang Engel, Unknown (version 1.1)
• Yanjun Zhang, VeriSilicon (versions 1.0, 1.1, 1.2, 1.3)
• Yunxing Zhu, Huawei (version 1.3)

**Other Credits**

The Vulkan Advisory Panel members provided important real-world usage information and advice that helped guide design decisions.

The wider Vulkan community have provided useful feedback, questions and specification changes that have helped improve the quality of the Specification via [GitHub](https://github.com).

Administrative support to the Working Group for Vulkan 1.1, 1.2, and 1.3 was provided by Khronos staff including Ann Thorsnes, Blaine Kohl, Dominic Agoro-Ombaka, Emily Stearns, Jeff Phillips, Lisie Aartsen, Liz Maitral, Marty Johnson, Tim Lewis, and Xiao-Yu CHENG; and by Alex Crabb, Laura Shubel, and Rachel Bradshaw of Caster Communications.

Administrative support for Vulkan 1.0 was provided by Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, Kathleen Mattson and Michelle Clark of Gold Standard Group.

Technical support was provided by James Riordon, site administration of Khronos.org and OpenGL.org.