# Table of Contents

1. Preamble ................................................................................................. 1  
2. Introduction ............................................................................................ 3  
   2.1. Document Conventions .................................................................. 3  
   2.2. Safety Critical Philosophy ............................................................ 5  
3. Fundamentals .......................................................................................... 7  
   3.1. Host and Device Environment ....................................................... 7  
   3.2. Execution Model .......................................................................... 7  
   3.3. Object Model .............................................................................. 9  
   3.4. Application Binary Interface ......................................................... 13  
   3.5. Command Syntax and Duration ................................................... 13  
   3.6. Threading Behavior ..................................................................... 15  
   3.7. Valid Usage ................................................................................ 21  
   3.8. `VkResult` Return Codes .............................................................. 27  
   3.9. Numeric Representation and Computation .................................... 30  
   3.10. Fixed-Point Data Conversions .................................................... 31  
   3.11. Common Object Types ............................................................... 33  
   3.12. API Name Aliases ..................................................................... 41  
4. Initialization ............................................................................................. 42  
   4.1. Command Function Pointers ......................................................... 42  
   4.2. Instances .................................................................................... 45  
5. Devices and Queues ............................................................................... 52  
   5.1. Physical Devices .......................................................................... 52  
   5.2. Devices ....................................................................................... 76  
   5.3. Queues ....................................................................................... 93  
6. Command Buffers .................................................................................. 101  
   6.1. Command Buffer Lifecycle ......................................................... 101  
   6.2. Command Pools .......................................................................... 103  
   6.3. Command Buffer Allocation and Management ............................. 111  
   6.4. Command Buffer Recording ....................................................... 116  
   6.5. Command Buffer Submission ...................................................... 123  
   6.6. Queue Forward Progress ............................................................. 132  
   6.7. Secondary Command Buffer Execution ...................................... 133  
   6.8. Command Buffer Device Mask .................................................... 136  
7. Synchronization and Cache Control ...................................................... 140  
   7.1. Execution and Memory Dependencies .......................................... 140  
   7.2. Implicit Synchronization Guarantees ............................................ 155  
   7.3. Fences ....................................................................................... 157  
   7.4. Semaphores ............................................................................... 167
## Queries

17. Queries

- 17.1. Query Pools
- 17.2. Query Operation
- 17.3. Occlusion Queries
- 17.4. Pipeline Statistics Queries
- 17.5. Timestamp Queries

## Clear Commands

18. Clear Commands

- 18.1. Clearing Images Outside a Render Pass Instance
- 18.2. Clearing Images Inside a Render Pass Instance
- 18.3. Clear Values
- 18.4. Filling Buffers
- 18.5. Updating Buffers

## Copy Commands

19. Copy Commands

- 19.1. Copying Data Between Buffers
- 19.2. Copying Data Between Images
- 19.3. Copying Data Between Buffers and Images
- 19.4. Image Copies With Scaling
- 19.5. Resolving Multisample Images

## Drawing Commands

20. Drawing Commands

- 20.1. Primitive Topologies
- 20.2. Primitive Order
- 20.3. Programmable Primitive Shading

## Fixed-Function Vertex Processing

21. Fixed-Function Vertex Processing

- 21.1. Vertex Attributes
- 21.2. Vertex Input Description
- 21.3. Vertex Input Address Calculation

## Tessellation

22. Tessellation

- 22.1. Tessellator
- 22.2. Tessellator Patch Discard
- 22.3. Tessellator Spacing
- 22.4. Tessellation Primitive Ordering
- 22.5. Tessellator Vertex Winding Order
- 22.6. Triangle Tessellation
- 22.7. Quad Tessellation
- 22.8. Isoline Tessellation
- 22.9. Tessellation Point Mode
- 22.10. Tessellation Pipeline State

## Geometry Shading

23. Geometry Shading

- 23.1. Geometry Shader Input Primitives
- 23.2. Geometry Shader Output Primitives
Appendix I: Lexicon ................................................................. 1162
Glossary ................................................................................. 1162
Common Abbreviations ....................................................... 1184
Prefixes .................................................................................. 1185
Appendix J: Credits (Informative) ........................................ 1187
Working Group Contributors to Vulkan SC 1.0 ...................... 1187
Working Group Contributors to Vulkan ................................. 1189
Other Credits .......................................................................... 1197
Chapter 1. Preamble

Copyright 2014-2023 The Khronos Group Inc.

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

Khronos grants a conditional copyright license to use and reproduce the unmodified Specification for any purpose, without fee or royalty, EXCEPT no licenses to any patent, trademark or other intellectual property rights are granted under these terms.

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

This Specification has been created under the Khronos Intellectual Property Rights Policy, which is Attachment A of the Khronos Group Membership Agreement available at https://www.khronos.org/files/member_agreement.pdf. Parties desiring to implement the Specification and make use of Khronos trademarks in relation to that implementation, and receive reciprocal patent license protection under the Khronos Intellectual Property Rights Policy must become Adopters and confirm the implementation as conformant under the process defined by Khronos for this Specification; see https://www.khronos.org/adopters.

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

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

Some parts of this Specification are purely informative and so are EXCLUDED 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, and SPIR-V is a trademark of The Khronos Group Inc. OpenCL is a trademark of Apple Inc., used under license by Khronos. OpenGL is a registered trademark and the OpenGL ES logo is a trademark of Hewlett Packard Enterprise, 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


The canonical version of the Specification is available in the official Vulkan SC Registry (https://registry.khronos.org/vulkansc/). The source files used to generate the Vulkan SC specification are stored in the Vulkan SC Documentation Repository (https://github.com/KhronosGroup/VulkanSC-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:

2.2. Safety Critical Philosophy

Vulkan SC 1.0.14 is based on Vulkan 1.2 and, except where explicitly noted, supports all of the same features, properties, and limits as Vulkan 1.2.

Throughout the Vulkan SC specification, changes have been made to the Base Vulkan Specification in order to align it with safety critical use cases and certification. In general changes were made to meet the following categories:

- Deterministic Execution (predictable execution times and results)
- Robustness (error handling, removing ambiguity, clarifying undefined behavior)
- Simplification (changes made to reduce certification effort and challenges)

To simplify capturing the reasoning behind deviations made from the Base Vulkan Specification, the Vulkan SC specification utilizes change identifications to give the reader insight into why the change was made in a concise manner. The change identifications are captured in Change Justification Table. In addition, the Vulkan SC specification contains Vulkan SC Deviations From Base Vulkan which is a complete list of changes between Base Vulkan and Vulkan SC. This is targeted at readers who are familiar with Base Vulkan and would like to understand the differences between Vulkan SC and the Base Vulkan specification.

Vulkan SC follows the Base Vulkan philosophy of requiring valid usage from the application. It is left to each implementation to determine how to ensure safe operation with respect to invalid usage. This may involve determining that certain invalid usage does not pose a safety risk, adding valid usage checks in the driver, requiring valid usage checks in the application, or some
combination of these. Additionally, validation layers are supported during development.

2.2.1. Change Justification Table

The following is a list of the safety critical change identifications used to concisely capture the justification for deviations from the Base Vulkan Specification.

*Table 1. Change Justifications*

<table>
<thead>
<tr>
<th>Change ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCID-1</td>
<td><strong>Deterministic behavior</strong> - no randomness or unpredictability, always produce the same output from a given starting condition or initial state</td>
</tr>
<tr>
<td>SCID-2</td>
<td><strong>Asynchronous calls</strong> - calls initiated by the application but may not execute or use their parameter data until a later time shall be clearly defined when any parameter data is used, especially data which is passed by reference or pointer</td>
</tr>
<tr>
<td>SCID-3</td>
<td><strong>Notification of change of state</strong> - avoid the use of asynchronous events causing code to execute (i.e. callbacks) as this can cause the worst case execution time of a system to be indeterminate</td>
</tr>
<tr>
<td>SCID-4</td>
<td><strong>Garbage collection methods</strong> - avoid the use of garbage collection as this can cause the worst case execution time of a system to be indeterminate. Avoid memory fragmentation by deleting entire buffers instead of individual items within a buffer</td>
</tr>
<tr>
<td>SCID-5</td>
<td><strong>Fully testable</strong> - all behavior of the API must be testable in a repeatable manner, consistent from test run to test run (in some cases this may mean testable by inspection)</td>
</tr>
<tr>
<td>SCID-6</td>
<td><strong>Undefined behavior</strong> - the API must behave as expected under valid input conditions, clearly document conditions that would result in 'fatal error' leaving the system in an unrecoverable state, and document conditions that would result in undefined behavior based on invalid input</td>
</tr>
<tr>
<td>SCID-7</td>
<td><strong>Unique ID</strong> - provide a facility to return a run time implementation unique identifier specific to that runtime so that is may be interrogated at any time. For example, such information could be the version number, name, date, release build number or a combination of these that is unique and comprehensible</td>
</tr>
<tr>
<td>SCID-8</td>
<td><strong>Code complexity</strong> - reducing code complexity to help facilitate certification (for example if there are multiple ways to do the same thing, potentially eliminating one or more of the alternative methods)</td>
</tr>
</tbody>
</table>
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, protected memory management, and transfer.

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.

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 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. 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 \texttt{vkCreate\*} and \texttt{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 \texttt{vkDestroy\*} and \texttt{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.

In Vulkan SC, data structures for objects are reserved by the implementation at device creation time in order to enable implementations to rely solely on static memory management at run-time. The \texttt{VkDeviceObjectReservationCreateInfo} structure provides upper bounds on the simultaneous number of objects of each type that can be allocated during the lifetime of the \texttt{VkDevice}. Most objects can be created and destroyed as needed, provided that no more than the requested number are in existence at any point in time.

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, unless otherwise noted below. 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:

- \texttt{VkPipelineCache}

A \texttt{VkPipelineCache} object created with \texttt{VK_PIPELINE_CACHE_CREATE_USE_APPLICATION_STORAGE_BIT} requires the application to maintain the memory contents pointed to by \texttt{VkPipelineCacheCreateInfo::pInitialData} for the lifetime of the pipeline cache object.

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

A \texttt{VkPipelineLayout} object must not be destroyed while any command buffer that uses it is in the recording state.

\texttt{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 \texttt{vkUpdateDescriptorSets} after the descriptor set layout has been destroyed. Otherwise, a \texttt{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
- VkBuffer
- VkBufferView
- VkImage
- VkImageView
- VkPipeline
- VkSampler
- VkSamplerYcbcrConversion
- VkFramebuffer
- VkRenderPass
- VkCommandBuffer
- 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

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

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

The following Vulkan objects have specific restrictions for when they **can** be destroyed:
• *VkQueue* objects *cannot* be explicitly destroyed. Instead, they are implicitly destroyed when the *VkDevice* object they are retrieved from is destroyed.

• Device memory (*VkDeviceMemory*) allocations, and pool objects (*VkCommandPool, VkDescriptorPool, VkQueryPool*) *cannot* be explicitly freed or destroyed. Instead, they are implicitly freed or destroyed when the *VkDevice* object they are created from is destroyed.

• *VkDevice* objects *can* be destroyed when all *VkQueue* objects retrieved from them are idle, and all objects created from them have been destroyed.

  ◦ This includes the following objects:

    - *VkFence*
    - *VkSemaphore*
    - *VkEvent*
    - *VkBuffer*
    - *VkBufferView*
    - *VkImage*
    - *VkImageView*
    - *VkPipelineCache*
    - *VkPipeline*
    - *VkPipelineLayout*
    - *VkSampler*
    - *VkSamplerYcbcrConversion*
    - *VkDescriptorSetLayout*
    - *VkFramebuffer*
    - *VkRenderPass*
    - *VkCommandBuffer*

  ◦ This does not include objects that do not have corresponding free or destroy commands. If *VkPhysicalDeviceVulkanSC10Properties::deviceDestroyFreesMemory* is *VK_TRUE*, the memory from these objects is returned to the system when the device is destroyed, otherwise it *may* not be returned to the system until the process is terminated.

• *VkPhysicalDevice* objects *cannot* be explicitly destroyed. Instead, they are implicitly destroyed when the *VkInstance* object they are retrieved from is destroyed.

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

### 3.3.2. External Object Handles

As defined above, the scope of object handles created or allocated from a *VkDevice* is limited to that logical device. Objects which are not in scope are said to be external. To bring an external object into scope, an external handle *must* be exported from the object in the source scope and imported into the destination scope.
The scope of external handles and their associated resources may vary according to their type, but they can generally be shared across process and API boundaries.

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.

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 must be set to NULL. Refer to the Memory Allocation chapter for further details.
Commands that allocate Vulkan objects owned by pool objects are of the form `vkAllocate*`, and take `VkAllocateInfo` 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-\texttt{const} qualified memory at any point. Parameters referring to non-\texttt{const} qualified application-owned memory are not marked explicitly as \textit{externally synchronized} in the Specification.

Many object types are \textit{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 \texttt{instance} parameter in \texttt{vkDestroyInstance}
- The \texttt{device} parameter in \texttt{vkDestroyDevice}
- The \texttt{queue} parameter in \texttt{vkQueueSubmit}
- The \texttt{fence} parameter in \texttt{vkQueueSubmit}
- The \texttt{queue} parameter in \texttt{vkQueueWaitIdle}
- The \texttt{memory} parameter in \texttt{vkMapMemory}
- The \texttt{memory} parameter in \texttt{vkUnmapMemory}
- The \texttt{buffer} parameter in \texttt{vkBindBufferMemory}
- The \texttt{image} parameter in \texttt{vkBindImageMemory}
- The \texttt{fence} parameter in \texttt{vkDestroyFence}
- The \texttt{semaphore} parameter in \texttt{vkDestroySemaphore}
- The \texttt{event} parameter in \texttt{vkDestroyEvent}
- The \texttt{event} parameter in \texttt{vkSetEvent}
- The \texttt{event} parameter in \texttt{vkResetEvent}
- The \texttt{buffer} parameter in \texttt{vkDestroyBuffer}
- The \texttt{bufferView} parameter in \texttt{vkDestroyBufferView}
- The \texttt{image} parameter in \texttt{vkDestroyImage}
- The \texttt{imageView} parameter in \texttt{vkDestroyImageView}
- The \texttt{pipelineCache} parameter in \texttt{vkDestroyPipelineCache}
- The \texttt{pipeline} parameter in \texttt{vkDestroyPipeline}
- The \texttt{pipelineLayout} parameter in \texttt{vkDestroyPipelineLayout}
- The \texttt{sampler} parameter in \texttt{vkDestroySampler}
• The `descriptorSetLayout` parameter in `vkDestroyDescriptorSetLayout`
• 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 `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 `vkCmdBindVertexBuffer`
• The `commandBuffer` parameter in `vkCmdDraw`
• The `commandBuffer` parameter in `vkCmdDrawIndexed`
• The `commandBuffer` parameter in `vkCmdDrawIndirect`
• The `commandBuffer` parameter in `vkCmdDrawIndexedIndirect`
• The `commandBuffer` parameter in `vkCmdDispatch`
• The `commandBuffer` parameter in `vkCmdDispatchIndirect`
• The `commandBuffer` parameter in `vkCmdCopyBuffer`
• The `commandBuffer` parameter in `vkCmdCopyImage`
• The `commandBuffer` parameter in `vkCmdBlitImage`
• The `commandBuffer` parameter in `vkCmdCopyBufferToImage`
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 `vkCmdDrawIndexedIndirect`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdDispatch`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdDispatchIndirect`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdCopyBuffer`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdCopyImage`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdBlitImage`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdCopyBufferToImage`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdCopyImageToBuffer`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdUpdateBuffer`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdFillBuffer`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdClearColorImage`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdClearDepthStencilImage`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdClearAttachments`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdResolveImage`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetEvent`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdResetEvent`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdWaitEvents`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdPipelineBarrier`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdBeginQuery`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdEndQuery`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdResetQueryPool`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdWriteTimestamp`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdCopyQueryPoolResults`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdPushConstants`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdBeginRenderPass`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdNextSubpass`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdEndRenderPass`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdExecuteCommands`
- The VkCommandPool that commandBuffer was allocated from, in `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 `vkCmdDrawIndexedIndirectCount2`
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.

Note

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

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

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.

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. Use of an enumerant is valid if the following conditions are true:

• The enumerant is defined as part of the enumerated type.
• The enumerant is not a value suffixed with \_MAX\_ENUM.
  ◦ This value exists only to ensure that C enum types are 32 bits in size and must not be used by applications.
• If the enumerant is used in a function that has a VkInstance as its first parameter and either:
  ◦ it was added by a core version that is supported (as reported by vkEnumerateInstanceVersion) and the value of VkApplicationInfo::apiVersion is greater than or equal to the version that added it; or
  ◦ it was added by an instance extension that was enabled for the instance.
• If the enumerant is used in a function that has a VkPhysicalDevice object as its first parameter and either:
  ◦ it was added by a core version that is supported by that device (as reported by VkPhysicalDeviceProperties::apiVersion);
  ◦ it was added by an instance extension that was enabled for the instance; or
  ◦ it was added by a device extension that is supported by that device.
• If the enumerant is used in a function that has any other dispatchable object as its first parameter and either:
  ◦ it was added by a core version that is supported for the device (as reported by VkPhysicalDeviceProperties::apiVersion); or
  ◦ it was added by a device extension that was enabled for the device.

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
In some special cases, an enumerant is only meaningful if a feature defined by an extension is also enabled, as well as the extension itself. The global “valid enumerant” rule described here does not address such cases.

**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.

An individual bit flag is valid for a `Vk*Flags` type if it would be a valid enumerant when used with the equivalent `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`.

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.
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-level functionality added by a device extension that is dispatched from a `VkDevice`, or from a child object of a `VkDevice` **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
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,
    // 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 VKSC_VERSION_1_0
    VK_ERROR_VALIDATION_FAILED = -1000011001,
    // Provided by VKSC_VERSION_1_0
    VK_ERROR_INVALID_PIPELINE_CACHE_DATA = -1000298000,
    // Provided by VKSC_VERSION_1_0
    VK_ERROR_NO_PIPELINE_MATCH = -1000298001,
} 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

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_VALIDATION_FAILED** A command failed because invalid usage was detected by the implementation or a validation-layer.
- **VK_ERROR_INVALID_PIPELINE_CACHE_DATA** The supplied pipeline cache data was not valid for the current implementation.
• **VK_ERROR_NO_PIPELINE_MATCH** The implementation did not find a match in the pipeline cache for
  the specified pipeline, or **VkPipelineOfflineCreateInfo** was not provided to the
  **vkCreate*Pipelines** function.

• **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. If
**VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations** is **VK_TRUE**, **VK_ERROR_OUT_OF_HOST_MEMORY** **must** not be returned from any physical or logical device command
which explicitly disallows it.

---

**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.

Any command returning a **VkResult** **may** return **VK_ERROR_VALIDATION_FAILED** if a violation of valid
usage is detected, even though commands do not explicitly list this as a possible return code.

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**.

---

**Note**

Implementations can also use **Fault Handling** to report runtime errors where
suitable return values are not available or to provide more prompt notification of
an error.
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 $2^{32}$.

\[
\begin{align*}
x \times 0 &= 0 	imes x = 0 & \text{for any non-infinite and non-NaN } x. \\
1 \times x &= x \times 1 = x. \\
x + 0 &= 0 + x = x. \\
0^0 &= 1.
\end{align*}
\]

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

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

Only the range \([-2^{b-1} + 1, 2^b - 1]\) 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 {
    uint32_t width;
    uint32_t height;
} VkExtent2D;
```

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

A three-dimensional extent is defined by the structure:

```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_TYPE_DEVICE_CREATE_INFO = 3,
    VK_STRUCTURE_TYPE_SUBMIT_INFO = 4,
    VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO = 5,
    VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE = 6,
    VK_STRUCTURE_TYPE_FENCE_CREATE_INFO = 8,
    VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO = 9,
    VK_STRUCTURE_TYPE_EVENT_CREATE_INFO = 10,
    VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO = 11,
    VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO = 12,
    VK_STRUCTURE_TYPE_BUFFER_VIEW_CREATE_INFO = 13,
    VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO = 14,
    VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO = 15,
    VK_STRUCTURE_TYPE_PIPELINE_CACHE_CREATE_INFO = 17,
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO = 18,
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO = 19,
    VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO = 20,
    VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO = 21,
    VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO = 22,
    VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO = 23,
    VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO = 24,
    VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO = 25,
    VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO = 26,
    VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO = 27,
    VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO = 28,
    VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO = 29,
    VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO = 30,
    VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO = 31,
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO = 32,
    VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO = 33,
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO = 34,
    VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET = 35,
    VK_STRUCTURE_TYPE_COPY_DESCRIPTOR_SET = 36,
    VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO = 37,
    VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO = 38,
    VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO = 39,
};
```
VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO = 40,
VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_INFO = 41,
VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO = 42,
VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO = 43,
VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER = 44,
VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER = 45,
VK_STRUCTURE_TYPE_MEMORY_BARRIER = 46,
VK_STRUCTURE_TYPE_LOADER_INSTANCE_CREATE_INFO = 47,
VK_STRUCTURE_TYPE_LOADER_DEVICE_CREATE_INFO = 48,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_PROPERTIES = 100004000,
// 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_TYPEDEVICE_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_BIND_BUFFER_MEMORY_DEVICE_GROUP_INFO = 1000060013,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORYDEVICE_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_MEMORY_REQUIREMENTS_2 = 1000146003,
// 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,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_2 = 1000059002,
// Provided by VK_VERSION_1_1
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_PHYSICAL_DEVICE_POINT_CLIPPING_PROPERTIES = 1000117000,
VK_STRUCTURE_TYPE_RENDER_PASS_INPUT_ATTACHMENT_ASPECT_CREATE_INFO = 1000117001,
VK_STRUCTURE_TYPE_IMAGE_VIEW_USAGE_CREATE_INFO = 1000117002,
VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_DOMAIN_ORIGIN_STATE_CREATE_INFO = 1000117003,
VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO = 1000053000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_FEATURES = 1000053001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PROPERTIES = 1000053002,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTERS_FEATURES = 1000120000,
VK_STRUCTURE_TYPE_PROTECTED_SUBMIT_INFO = 1000145000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_FEATURES = 1000145001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_PROPERTIES = 1000145002,
VK_STRUCTURE_TYPE_DEVICE_QUEUE_INFO_2 = 1000145003,
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_CREATE_INFO = 1000156000,
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_INFO = 1000156001,
VK_STRUCTURE_TYPE_BIND_IMAGE_PLANE_MEMORY_INFO = 1000156002,
VK_STRUCTURE_TYPE_IMAGE_PLANE_MEMORY_REQUIREMENTS_INFO = 1000156003,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_YCBCR_CONVERSION_FEATURES = 1000156004,
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_IMAGE_FORMAT_PROPERTIES = 1000156005,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_IMAGE_FORMAT_INFO = 1000071000,
VK_STRUCTURE_TYPE_EXTERNAL_IMAGE_FORMAT_PROPERTIES = 1000071001,
VK_STRUCTURE_TYPE_PHYSICALDEVICE_EXTERNALBUFFER_INFO = 1000071002,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_PROPERTIES = 1000071004,
VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO = 1000072000,
VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO = 1000072001,
VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO = 1000072002,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_FENCE_INFO = 1000112000,
VK_STRUCTURE_TYPE_EXTERNAL_FENCE_PROPERTIES = 1000112001,
VK_STRUCTURE_TYPE_EXPORT_FENCE_CREATE_INFO = 1000113000,
VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_CREATE_INFO = 1000077000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SEMAPHORE_INFO = 1000076000,
VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES = 1000076001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_3_PROPERTIES = 1000168000,
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_SUPPORT = 1000168001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DRAW_PARAMETERS_FEATURES = 1000063000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_1_FEATURES = 49,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_1_PROPERTIES = 50,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_2_FEATURES = 51,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_2_PROPERTIES = 52,
VK_STRUCTURE_TYPE_IMAGE_FORMAT_LIST_CREATE_INFO = 1000147000,
VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_2 = 1000109000,
VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_2 = 1000109001,
VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_2 = 1000109002,
VK_STRUCTURE_TYPE_SUBPASS_DEPENDENCY_2 = 1000109003,
VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO_2 = 1000109004,
VK_STRUCTURE_TYPE_SUBPASS_BEGIN_INFO = 1000109005,
VK_STRUCTURE_TYPE_SUBPASS_END_INFO = 1000109006,
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,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_RENDER_PASS_ATTACHMENT_BEGIN_INFO = 1000108003,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_UNIFORM_BUFFER_STANDARD_LAYOUT_FEATURES = 1000253000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICALDEVICE_SHADER_SUBGROUP_EXTENDED_TYPES_FEATURES = 1000175000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SEPARATE_DEPTH_STENCIL_LAYOUTS_FEATURES = 1000253001,
VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_STENCIL_LAYOUT = 1000241000,
VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_STENCIL_LAYOUT = 1000241001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_HOST_QUERY_RESET_FEATURES = 1000261000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_FEATURES = 1000207000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_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_SC_1_0_FEATURES = 1000298000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_SC_1_0_PROPERTIES = 1000298001,
VK_STRUCTURE_TYPE_DEVICE_OBJECT_RESERVATION_CREATE_INFO = 1000298002,
VK_STRUCTURE_TYPE_COMMAND_POOL_MEMORY_RESERVATION_CREATE_INFO = 1000298003,
VK_STRUCTURE_TYPE_COMMAND_POOL_MEMORY_CONSUMPTION = 1000298004,
VK_STRUCTURE_TYPE_PIPELINE_POOL_SIZE = 1000298005,
VK_STRUCTURE_TYPE_FAULT_DATA = 1000298007,
VK_STRUCTURE_TYPE_FAULT_CALLBACK_INFO = 1000298008,
VK_STRUCTURE_TYPE_PIPELINE_OFFLINE_CREATE_INFO = 1000298010,
} VkStructureType;
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 by calling:

```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><code>NULL</code></td>
<td><code>NULL</code></td>
<td>undefined</td>
</tr>
<tr>
<td>invalid non-NULL instance</td>
<td><code>*1</code></td>
<td>undefined</td>
</tr>
<tr>
<td><code>global command</code>²</td>
<td><code>fp</code></td>
<td></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&lt;sup&gt;4&lt;/sup&gt; 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

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 by calling:

```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 3. `vkGetDeviceProcAddr` behavior

<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</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 version²</td>
<td>fp⁴</td>
</tr>
<tr>
<td></td>
<td>device-level dispatchable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>command³</td>
<td></td>
</tr>
<tr>
<td>device</td>
<td>enabled extension</td>
<td>fp⁴</td>
</tr>
<tr>
<td></td>
<td>device-level dispatchable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>command³</td>
<td></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

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. If a function pointer is returned, it 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

In Vulkan SC 1.0, 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:

```c
// Provided by VK_VERSION_1_0
```
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-null
  `pAllocator` must be NULL
- 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;
    const void*pNext;
    VkInstanceCreateFlags flags;
} VkInstanceCreateInfo;
```
const VkApplicationInfo* pApplicationInfo;
uint32_t enabledLayerCount;
const char* const* ppEnabledLayerNames;
uint32_t enabledExtensionCount;
const char* const* ppEnabledExtensionNames;
} VkInstanceCreateInfo;

• **sType** is a VkStructureType value identifying 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:

```c
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 a VkStructureType value identifying 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. The variant version of the instance must match that 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, unless an incompatible variant is requested.

Note
Vulkan SC 1.0 is based on Vulkan 1.2 and thus instance creation may only fail with VK_ERROR_INCOMPATIBLE_DRIVER if an incompatible variant is requested - that is if the Vulkan SC API is requested from a Vulkan implementation or if the Vulkan API is requested from a Vulkan SC implementation.

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

Valid Usage

- VUID-VkApplicationInfo-apiVersion-05021
  If apiVersion is not 0 and its variant is VKSC_API_VARIANT, then it must be greater than or equal to VKSC_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

Valid Usage (Implicit)

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

- VUID-vkDestroyInstance-pAllocator-null
  pAllocator must be NULL

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.

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

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

If the vendor has a PCI \texttt{vendor ID}, the low 16 bits of \texttt{vendorID} \textbf{must} contain that PCI vendor ID, and the remaining bits \textbf{must} be set to zero. Otherwise, the value returned \textbf{must} be a valid Khronos vendor ID, obtained as described in the \textit{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 \texttt{VkVendorId} type.

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

\begin{quote}
\textit{Note} \\
The same device ID \textbf{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 \textbf{should} use the same device ID, even if those uses occur in different SoCs.
\end{quote}

Khronos vendor IDs which \textbf{may} be returned in \texttt{VkPhysicalDeviceProperties::vendorID} are:

\begin{verbatim}
// 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,
    VK_VENDOR_ID_MOBILEYE = 0x10007,
} VkVendorId;
\end{verbatim}

\begin{quote}
\textit{Note} \\
Khronos vendor IDs may be allocated by vendors at any time. Only the latest canonical versions of this Specification, of the corresponding \texttt{vk.xml} API Registry,
and of the corresponding `vulkan_sc_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
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 a `VkStructureType` value identifying 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  
  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  
  `VkPhysicalDeviceDepthStencilResolveProperties`,  
  `VkPhysicalDeviceDescriptorIndexingProperties`,  
  `VkPhysicalDeviceDriverProperties`,  
  `VkPhysicalDeviceFloatControlsProperties`,  
  `VkPhysicalDeviceIDProperties`,  
  `VkPhysicalDeviceMaintenance3Properties`,  
  `VkPhysicalDeviceMultiviewProperties`,  
  `VkPhysicalDevicePointClippingProperties`,  
  `VkPhysicalDeviceProtectedMemoryProperties`,  
  `VkPhysicalDeviceSamplerFilterMinmaxProperties`,  
  `VkPhysicalDeviceSubgroupProperties`,  
  `VkPhysicalDeviceTimelineSemaphoreProperties`,  
  `VkPhysicalDeviceVulkan11Properties`,  
  `VkPhysicalDeviceVulkan12Properties`, or `VkPhysicalDeviceVulkanSC10Properties`
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];
    uint32_t deviceNodeMask;
    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 a `VkStructureType` value identifying 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:
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 shaderRoundingModeRTZFloat16;
    VkBool32 shaderRoundingModeRTZFloat32;
    VkBool32 shaderRoundingModeRTZFloat64;
    uint32_t maxUpdateAfterBindDescriptorsInAllPools;
    VkBool32 shaderUniformBufferArrayNonUniformIndexingNative;
    VkBool32 shaderSampledImageArrayNonUniformIndexingNative;
    VkBool32 shaderStorageBufferArrayNonUniformIndexingNative;
    VkBool32 shaderStorageImageArrayNonUniformIndexingNative;
    VkBool32 shaderInputAttachmentArrayNonUniformIndexingNative;
    VkBool32 robustBufferAccessUpdateAfterBind;
    VkBool32 quadDivergentImplicitLod;
    uint32_t maxPerStageDescriptorUpdateAfterBindSamplers;
    uint32_t maxPerStageUpdateAfterBindResources;
    uint32_t maxPerStageDescriptorUpdateAfterBindUniformBuffers;
    uint32_t maxPerStageDescriptorUpdateAfterBindStorageBuffers;
    uint32_t maxPerStageDescriptorUpdateAfterBindSampledImages;
    uint32_t maxPerStageDescriptorUpdateAfterBindStorageImages;
    uint32_t maxPerStageDescriptorUpdateAfterBindInputAttachments;
}
uint32_t maxDescriptorSetUpdateAfterBindSamplers;
uint32_t maxDescriptorSetUpdateAfterBindUniformBuffers;
uint32_t maxDescriptorSetUpdateAfterBindUniformBuffersDynamic;
uint32_t maxDescriptorSetUpdateAfterBindStorageBuffers;
uint32_t maxDescriptorSetUpdateAfterBindStorageBuffersDynamic;
uint32_t maxDescriptorSetUpdateAfterBindSampledImages;
uint32_t maxDescriptorSetUpdateAfterBindStorageImages;
uint32_t maxDescriptorSetUpdateAfterBindInputAttachments;
VkResolveModeFlags supportedDepthResolveModes;
VkResolveModeFlags supportedStencilResolveModes;
VkBool32 independentResolveNone;
VkBool32 independentResolve;
VkBool32 filterMinmaxSingleComponentFormats;
VkBool32 filterMinmaxImageComponentMapping;
uint64_t maxTimelineSemaphoreValueDifference;
VkSampleCountFlags framebufferIntegerColorSampleCounts;
}

VkPhysicalDeviceVulkan12Properties;

• **sType** is a VkStructureType value identifying 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.

- 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** 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 LOD 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. 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. A value of VK_RESOLVE_MODE_NONE indicates that depth resolve operations are disallowed [SCID-8]. If any bits are set then 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. A value of VK_RESOLVE_MODE_NONE indicates that stencil resolve operations are disallowed [SCID-8]. If any bits are set then 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,
VkPhysicalDeviceDepthStencilResolveProperties,
VkPhysicalDeviceSamplerFilterMinmaxProperties,
VkPhysicalDeviceTimelineSemaphoreProperties.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceVulkan12Properties-sType-sType sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_2_PROPERTIES
The `VkPhysicalDeviceVulkanSC10Properties` structure is defined as:

```c
// Provided by VKSC_VERSION_1_0
typedef struct VkPhysicalDeviceVulkanSC10Properties {
    VkStructureType sType;
    void* pNext;
    VkBool32 deviceNoDynamicHostAllocations;
    VkBool32 deviceDestroyFreesMemory;
    VkBool32 commandPoolMultipleCommandBuffersRecording;
    VkBool32 commandPoolResetCommandBuffer;
    VkBool32 commandBufferSimultaneousUse;
    VkBool32 secondaryCommandBufferNullOrImagelessFramebuffer;
    VkBool32 recycleDescriptorSetMemory;
    VkBool32 recyclePipelineMemory;
    uint32_t maxRenderPassSubpasses;
    uint32_t maxRenderPassDependencies;
    uint32_t maxSubpassInputAttachments;
    uint32_t maxSubpassPreserveAttachments;
    uint32_t maxFramebufferAttachments;
    uint32_t maxDescriptorSetLayoutBindings;
    uint32_t maxQueryFaultCount;
    uint32_t maxCallbackFaultCount;
    uint32_t maxCommandPoolCommandBuffers;
    VkDeviceSize maxCommandBufferSize;
} VkPhysicalDeviceVulkanSC10Properties;
```

- `sType` is a `VkStructureType` value identifying this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `deviceNoDynamicHostAllocations` indicates whether the implementation will perform dynamic host memory allocations for physical or logical device commands. If `deviceNoDynamicHostAllocations` is `VK_TRUE` the implementation will allocate host memory for objects based on the provided `VkDeviceObjectReservationCreateInfo` limits during `vkCreateDevice`. Under valid API usage, `VK_ERROR_OUT_OF_HOST_MEMORY` may only be returned by commands which do not explicitly disallow it.
- `deviceDestroyFreesMemory` indicates whether destroying the device frees all memory resources back to the system.
- `commandPoolMultipleCommandBuffersRecording` indicates whether multiple command buffers from the same command pool can be in the recording state at the same time.
- `commandPoolResetCommandBuffer` indicates whether command buffers support `vkResetCommandBuffer`, and `vkBeginCommandBuffer` when not in the initial state.
- `commandBufferSimultaneousUse` indicates whether command buffers support `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT`.
- `secondaryCommandBufferNullOrImagelessFramebuffer` indicates whether the framebuffer member of `VkCommandBufferInheritanceInfo` may be equal to `VK_NULL_HANDLE` or be created with a `VkFramebufferCreateInfo::flags` value that includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT` if the command buffer will be executed within a render pass instance.
• `recycleDescriptorSetMemory` indicates whether descriptor pools are able to immediately reuse pool memory from descriptor sets that have been freed. If this is `VK_FALSE`, then memory may only be reallocated after `vkResetDescriptorPool` is called.

• `recyclePipelineMemory` indicates whether the memory for a pipeline is available for reuse by new pipelines after the pipeline is destroyed.

• `maxRenderPassSubpasses` is the maximum number of subpasses in a render pass.

• `maxRenderPassDependencies` is the maximum number of dependencies in a render pass.

• `maxSubpassInputAttachments` is the maximum number of input attachments in a subpass.

• `maxSubpassPreserveAttachments` is the maximum number of preserve attachments in a subpass.

• `maxFramebufferAttachments` is the maximum number of attachments in a framebuffer, as well as the maximum number of attachments in a render pass.

• `maxDescriptorSetLayoutBindings` is the maximum number of bindings in a descriptor set layout.

• `maxQueryFaultCount` is the maximum number of faults that the implementation can record, to be reported via `vkGetFaultData`.

• `maxCallbackFaultCount` is the maximum number of faults that the implementation can report via a single call to `PFN_vkFaultCallbackFunction`.

• `maxCommandPoolCommandBuffers` is the maximum number of command buffers that can be allocated from a single command pool.

• `maxCommandBufferSize` is the maximum supported size of a single command buffer in bytes. Applications can use `vkGetCommandPoolMemoryConsumption` to compare a command buffer’s current memory usage to this limit.

**Note**

Implementations that do not have a fixed upper bound on the number of command buffers that may be allocated from a command pool can report `0xFFFFFFFFU` for `maxCommandPoolCommandBuffers`.

Implementations that do not have a fixed upper bound on the command buffer size can report `UINT64_MAX` for `maxCommandBufferSize`.

If the `VkPhysicalDeviceVulkanSC10Properties` 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 SC 1.0 functionality.

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceVulkanSC10Properties-sType-sType`  
  * `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_SC_1_0_PROPERTIES`  

The `VkPhysicalDeviceIDProperties` structure is defined as:
```c
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 a `VkStructureType` value identifying 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`.

```c
#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`.

```c
#define VK_LUID_SIZE                      8U
```

The `VkPhysicalDeviceDriverProperties` structure is defined as:

```c
//@ 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 a `VkStructureType` value identifying 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,
    VK_DRIVER_ID_MESA_AGXV = 25,
    VK_DRIVER_ID_MESA_VENUS = 26,
} 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 example, different platforms, proprietary or open source, etc.). Only the latest canonical versions of this Specification, of the corresponding vk.xml API Registry, and of the corresponding vulkan_sc_core.h header file must contain all reserved Khronos driver IDs.
Only driver IDs registered with Khronos are given symbolic names. There may be unregistered driver IDs returned.

VK_MAX_DRIVER_NAME_SIZE is the length in char values of an array containing a driver name string, as returned in VkPhysicalDeviceDriverProperties::driverName.

```
#define VK_MAX_DRIVER_NAME_SIZE 256U
```

VK_MAX_DRIVER_INFO_SIZE is the length in char values of an array containing a driver information string, as returned in VkPhysicalDeviceDriverProperties::driverInfo.

```
#define VK_MAX_DRIVER_INFO_SIZE 256U
```

The conformance test suite version an implementation is compliant with is described with the VkConformanceVersion structure:

```
// Provided by VK_VERSION_1_2
typedef struct VkConformanceVersion {
    uint8_t major;
    uint8_t minor;
    uint8_t subminor;
    uint8_t patch;
} VkConformanceVersion;
```

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

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

```
// 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-pQueueFamilyProperties-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 `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 \(x + width\) must equal the width of the image subresource corresponding to the parameter.
  - \(height\) of a \(VkExtent3D\) parameter must be an integer multiple of \(A_y\), or else \(y + height\) must equal the height of the image subresource corresponding to the parameter.
  - \(depth\) of a \(VkExtent3D\) parameter must be an integer multiple of \(A_z\), or else \(z + depth\) must equal the depth of the image subresource corresponding to the parameter.
  - If the format of the image corresponding to the parameters is one of the block-compressed formats then for the purposes of the above calculations the granularity must be scaled up by the compressed texel block dimensions.

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

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

For physical device feature queries see the Features chapter.

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

```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,
    // 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. This flag is not supported in Vulkan SC [SCID-8].

• **VK_QUEUE_PROTECTED_BIT** specifies that queues in this queue family support the **VK_DEVICE_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.

```
// 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:

```
// 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 is not NULL, pQueueFamilyProperties must be a valid pointer to an array of pQueueFamilyPropertyCount VkQueueFamilyProperties2 structures

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 a VkStructureType value identifying 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 vkCreateInstance
    uint32_t* pPhysicalDeviceGroupCount,     // Pointer to an integer related to the number of device groups available or queried
    VkPhysicalDeviceGroupProperties*         // Pointer to an array of VkPhysicalDeviceGroupProperties structures
);                                      // 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
```

• 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.
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 a `VkStructureType` value identifying 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**
  - `pNext` must be `NULL`

**VK_MAX_DEVICE_GROUP_SIZE** is the length of an array containing `VkPhysicalDevice` handle values representing all physical devices in a group, as returned in `VkPhysicalDeviceGroupProperties::physicalDevices`.

```c
#define VK_MAX_DEVICE_GROUP_SIZE 32U
```

### 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.

vkCreateDevice also verifies that mandatory structures and features for Vulkan SC are present and enabled:

- The pNext chain must include a VkDeviceObjectReservationCreateInfo structure.
- The pNext chain must include a VkPhysicalDeviceVulkanSC10Features structure.

If any of these conditions are not met, vkCreateDevice must return VK_ERROR_INITIALIZATION_FAILED.

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

An implementation may allow multiple logical devices to be created from the same physical device. Logical device creation may fail due to lack of device-specific resources, including too many other logical devices, in addition to other errors. If that occurs, vkCreateDevice will return VK_ERROR_TOO_MANY_OBJECTS.

If the pipeline cache data pointed to by the pInitialData member of any element of VkDeviceObjectReservationCreateInfo::pPipelineCacheCreateInfos is not compatible with the device, then vkCreateDevice will return VK_ERROR_INVALID_PIPELINE_CACHE_DATA.

---

**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

- VUID-vkCreateDevice-deviceMemoryRequestCount-05095  
  The sum of deviceMemoryRequestCount over all VkDeviceObjectReservationCreateInfo structures included in the VkDeviceCreateInfo::pNext chain must be less than or equal to VkPhysicalDeviceLimits::maxMemoryAllocationCount

- VUID-vkCreateDevice-samplerRequestCount-05096  
  The sum of samplerRequestCount over all VkDeviceObjectReservationCreateInfo structures included in the VkDeviceCreateInfo::pNext chain must be less than or equal to VkPhysicalDeviceLimits::maxSamplerAllocationCount

**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-null
  pAllocator must be NULL

• 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
  • VK_ERROR_INVALID_PIPELINE_CACHE_DATA

The VkDeviceCreateInfo structure is defined as:

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

• sType is a VkStructureType value identifying 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, VkPhysicalDeviceShaderFloat16Int8Features, or VkPhysicalDeviceDescriptorIndexingFeatures structure.

VkPhysicalDeviceScalarBlockLayoutFeatures,
VkPhysicalDeviceImagelessFramebufferFeatures,
VkPhysicalDeviceUniformBufferStandardLayoutFeatures,
VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures,
VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures,
VkPhysicalDeviceHostQueryResetFeatures,
VkPhysicalDeviceTimelineSemaphoreFeatures,
VkPhysicalDeviceBufferDeviceAddressFeatures,
VkPhysicalDeviceVulkanMemoryModelFeatures structure

Valid Usage (Implicit)

- VUID-VkDeviceCreateInfo-sType-sType
  
  The **sType** value of each struct in the **pNext** chain must be either

  - VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO
  - VkDeviceGroupDeviceCreateInfo, VkDeviceObjectReservationCreateInfo, VkFaultCallbackInfo,
    VkPhysicalDevice16BitStorageFeatures, VkPhysicalDevice8BitStorageFeatures,
    VkPhysicalDeviceBufferDeviceAddressFeatures, VkPhysicalDeviceDescriptorIndexingFeatures,
    VkPhysicalDeviceFeatures2, VkPhysicalDeviceHostQueryResetFeatures,
    VkPhysicalDeviceImagelessFramebufferFeatures, VkPhysicalDeviceMultiviewFeatures,
    VkPhysicalDeviceProtectedMemoryFeatures, VkPhysicalDeviceSamplerYcbcrConversionFeatures,
    VkPhysicalDeviceScalarBlockLayoutFeatures, VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures,
    VkPhysicalDeviceShaderAtomicInt64Features, VkPhysicalDeviceShaderDrawParametersFeatures,
    VkPhysicalDeviceShaderFloat16Int8Features, VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures,
    VkPhysicalDeviceTimelineSemaphoreFeatures, VkPhysicalDeviceUniformBufferStandardLayoutFeatures,
    VkPhysicalDeviceVariablePointersFeatures, VkPhysicalDeviceVulkan11Features,
    VkPhysicalDeviceVulkan12Features, VkPhysicalDeviceVulkanMemoryModelFeatures, or
    VkPhysicalDeviceVulkanSC10Features

- 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, VkDeviceObjectReservationCreateInfo, VkFaultCallbackInfo,
    - VkPhysicalDevice16BitStorageFeatures, VkPhysicalDevice8BitStorageFeatures,
    - VkPhysicalDeviceBufferDeviceAddressFeatures, VkPhysicalDeviceDescriptorIndexingFeatures,
    - VkPhysicalDeviceFeatures2, VkPhysicalDeviceHostQueryResetFeatures,
    - VkPhysicalDeviceImagelessFramebufferFeatures, VkPhysicalDeviceMultiviewFeatures,
    - VkPhysicalDeviceProtectedMemoryFeatures, VkPhysicalDeviceSamplerYcbcrConversionFeatures,
    - VkPhysicalDeviceScalarBlockLayoutFeatures, VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures,
    - VkPhysicalDeviceShaderAtomicInt64Features, VkPhysicalDeviceShaderDrawParametersFeatures,
    - VkPhysicalDeviceShaderFloat16Int8Features, VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures,
    - VkPhysicalDeviceTimelineSemaphoreFeatures, VkPhysicalDeviceUniformBufferStandardLayoutFeatures,
    - VkPhysicalDeviceVariablePointersFeatures, VkPhysicalDeviceVulkan11Features,
    - VkPhysicalDeviceVulkan12Features, VkPhysicalDeviceVulkanMemoryModelFeatures, or
    - VkPhysicalDeviceVulkanSC10Features

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

  - VkDeviceObjectReservationCreateInfo

- VUID-VkDeviceCreateInfo-flags-zerobitmask
  
  The **flags** must be 0

- VUID-VkDeviceCreateInfo-pQueueCreateInfos-parameter
  
  The **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.

```c
// 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:

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

- `sType` is a `VkStructureType` value identifying 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 `vkEnumeratePhysicalDeviceGroups`

- VUID-VkDeviceGroupDeviceCreateInfo-physicalDeviceCount-00377
  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

Data structures for objects are reserved by the implementation at device creation time. The application must provide upper bounds on numbers of objects and other limits at device creation time. To reserve data structures for use by objects created from this device, add a `VkDeviceObjectReservationCreateInfo` structure to the `pNext` chain of the `VkDeviceCreateInfo` structure.

```c
// Provided by VKSC_VERSION_1_0
typedef struct VkDeviceObjectReservationCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t pipelineCacheCreateInfoCount;
    const VkPipelineCacheCreateInfo* pPipelineCacheCreateInfos;
    uint32_t pipelinePoolSizeCount;
    const VkPipelinePoolSize* pPipelinePoolSizes;
    uint32_t semaphoreRequestCount;
    uint32_t commandBufferRequestCount;
    uint32_t fenceRequestCount;
    uint32_t deviceMemoryRequestCount;
    uint32_t bufferRequestCount;
    uint32_t imageRequestCount;
    uint32_t eventRequestCount;
    uint32_t queryPoolRequestCount;
    uint32_t bufferViewRequestCount;
    uint32_t imageViewRequestCount;
} VkDeviceObjectReservationCreateInfo;
```
uint32_t layeredImageViewRequestCount;
uint32_t pipelineCacheRequestCount;
uint32_t pipelineLayoutRequestCount;
uint32_t renderPassRequestCount;
uint32_t graphicsPipelineRequestCount;
uint32_t computePipelineRequestCount;
uint32_t descriptorSetLayoutRequestCount;
uint32_t samplerRequestCount;
uint32_t descriptorPoolRequestCount;
uint32_t descriptorSetRequestCount;
uint32_t framebufferRequestCount;
uint32_t commandPoolRequestCount;
uint32_t samplerYcbcrConversionRequestCount;
uint32_t surfaceRequestCount;
uint32_t swapchainRequestCount;
uint32_t displayModeRequestCount;
uint32_t subpassDescriptionRequestCount;
uint32_t attachmentDescriptionRequestCount;
uint32_t descriptorSetLayoutBindingRequestCount;
uint32_t descriptorSetLayoutBindingLimit;
uint32_t maxImageViewMipLevels;
uint32_t maxImageViewArrayLayers;
uint32_t maxLayeredImageViewMipLevels;
uint32_t maxOcclusionQueriesPerPool;
uint32_t maxPipelineStatisticsQueriesPerPool;
uint32_t maxTimestampQueriesPerPool;
uint32_t maxImmutableSamplersPerDescriptorSetLayout;

VkDeviceObjectReservationCreateInfo;

• **sType** is a *VkStructureType* value identifying this structure.

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

• **pipelineCacheCreateInfoCount** is the length of the *pPipelineCacheCreateInfos* array.

• **pPipelineCacheCreateInfos** is a pointer to an array of *VkPipelineCacheCreateInfo* structures that contain the creation information of the pipeline caches that can be created on this device.

• **pipelinePoolSizeCount** is the length of the *pPipelinePoolSizes* array.

• **pPipelinePoolSizes** is a pointer to an array of *VkPipelinePoolSize* structures requesting memory be reserved for pipelines of the specified sizes.

• **semaphoreRequestCount** is the requested maximum number of *VkSemaphore* objects that can exist at the same time.

• **commandBufferRequestCount** is the requested maximum number of *VkCommandBuffer* objects that can be reserved by all *VkCommandPool* objects.

• **fenceRequestCount** is the requested maximum number of *VkFence* objects that can exist at the same time.

• **deviceMemoryRequestCount** is the requested maximum number of *VkDeviceMemory* objects that can exist at the same time.
bufferRequestCount is the requested maximum number of VkBuffer objects that can exist at the same time.

imageRequestCount is the requested maximum number of VkImage objects that can exist at the same time.

eventRequestCount is the requested maximum number of VkEvent objects that can exist at the same time.

queryPoolRequestCount is the requested maximum number of VkQueryPool objects that can exist at the same time.

bufferViewRequestCount is the requested maximum number of VkBufferView objects that can exist at the same time.

imageViewRequestCount is the requested maximum number of VkImageView objects that can exist at the same time.

layeredImageViewRequestCount is the requested maximum number VkImageView objects created with VkImageViewCreateInfo::subresourceRange.layerCount greater than 1 that can exist at the same time.

pipelineCacheRequestCount is the requested maximum number of VkPipelineCache objects that can exist at the same time.

pipelineLayoutRequestCount is the requested maximum number of VkPipelineLayout objects that can exist at the same time.

renderPassRequestCount is the requested maximum number of VkRenderPass objects that can exist at the same time.

graphicsPipelineRequestCount is the requested maximum number of graphics VkPipeline objects that can exist at the same time.

computePipelineRequestCount is the requested maximum number of compute VkPipeline objects that can exist at the same time.

descriptorSetLayoutRequestCount is the requested maximum number of VkDescriptorSetLayout objects that can exist at the same time.

samplerRequestCount is the requested maximum number of VkSampler objects that can exist at the same time.

descriptorPoolRequestCount is the requested maximum number of VkDescriptorPool objects that can exist at the same time.

descriptorSetRequestCount is the requested maximum number of VkDescriptorSet objects that can exist at the same time.

framebufferRequestCount is the requested maximum number of VkFramebuffer objects that can exist at the same time.

commandPoolRequestCount is the requested maximum number of VkCommandPool objects that can exist at the same time.

samplerYcbcrConversionRequestCount is the requested maximum number of VkSamplerYcbcrConversion objects that can exist at the same time.

surfaceRequestCount is deprecated and implementations must ignore it.
• `swapchainRequestCount` is the requested maximum number of `VkSwapchainKHR` objects that can exist at the same time.

• `displayModeRequestCount` is deprecated and implementations must ignore it.

• `subpassDescriptionRequestCount` is the requested maximum sum of all `VkRenderPassCreateInfo2::subpassCount` values across all `VkRenderPass` objects that can exist at the same time.

• `attachmentDescriptionRequestCount` is the requested maximum sum of all `VkRenderPassCreateInfo2::attachmentCount` values across all `VkRenderPass` objects that can exist at the same time.

• `descriptorSetLayoutBindingRequestCount` is the requested maximum sum of all `VkDescriptorSetLayoutCreateInfo::bindingCount` values across all `VkDescriptorSetLayout` objects that can exist at the same time.

• `descriptorSetLayoutBindingLimit` is one greater than the maximum value of `VkDescriptorSetLayoutBinding::binding` that can be used.

• `maxImageViewMipLevels` is the maximum value of `VkImageViewCreateInfo::subresourceRange.levelCount` that can be used.

• `maxImageViewArrayLayers` is the maximum value of `VkImageViewCreateInfo::subresourceRange.layerCount` that can be used.

• `maxLayeredImageViewMipLevels` is the maximum value of `VkImageViewCreateInfo::subresourceRange.levelCount` that can be used when `VkImageViewCreateInfo::subresourceRange.layerCount` is greater than 1.

• `maxOcclusionQueriesPerPool` is the requested maximum number of `VK_QUERY_TYPE_OCCLUSION` queries that can exist at the same time in a single query pool.

• `maxPipelineStatisticsQueriesPerPool` is the requested maximum number of `VK_QUERY_TYPE_PIPELINE_STATISTICS` queries that can exist at the same time in a single query pool.

• `maxTimestampQueriesPerPool` is the requested maximum number of `VK_QUERY_TYPE_TIMESTAMP` queries that can exist at the same time in a single query pool.

• `maxImmutableSamplersPerDescriptorSetLayout` is the requested maximum number of immutable samplers that can be used across all bindings in a descriptor set layout.

Multiple `VkDeviceObjectReservationCreateInfo` structures can be chained together. The maximum value from all instances of `maxImageViewMipLevels`, `maxImageViewArrayLayers`, `maxLayeredImageViewMipLevels`, `descriptorSetLayoutBindingLimit`, `maxOcclusionQueriesPerPool`, `maxPipelineStatisticsQueriesPerPool`, `maxTimestampQueriesPerPool`, and `maxImmutableSamplersPerDescriptorSetLayout` will be reserved. For the remaining members, the sum of the requested resources from all instances of `VkDeviceObjectReservationCreateInfo` will be reserved.

If `VkPhysicalDeviceVulkanSC10Properties::deviceDestroyFreesMemory` is `VK_TRUE`, the reserved memory is returned to the system when the device is destroyed, otherwise it may not be returned to the system until the process is terminated.
Valid Usage

- **VUID-VkDeviceObjectReservationCreateInfo-maxImageViewArrayLayers-05014**
  
  `maxImageViewArrayLayers` **must** be less than or equal to `VkPhysicalDeviceLimits::maxImageArrayLayers`.

- **VUID-VkDeviceObjectReservationCreateInfo-maxImageViewMipLevels-05015**
  
  `maxImageViewMipLevels` **must** be less than or equal to the number of levels in the complete mipmap chain based on the maximum of `VkPhysicalDeviceLimits::maxImageDimension1D`, `maxImageDimension2D`, `maxImageDimension3D`, and `maxImageDimensionCube`.

- **VUID-VkDeviceObjectReservationCreateInfo-maxLayeredImageViewMipLevels-05016**
  
  `maxLayeredImageViewMipLevels` **must** be less than or equal to the number of levels in the complete mipmap chain based on `VkPhysicalDeviceLimits::maxImageDimension1D`, `maxImageDimension2D`, `maxImageDimension3D`, and `maxImageDimensionCube`.

- **VUID-VkDeviceObjectReservationCreateInfo-subpassDescriptionRequestCount-05017**
  
  `subpassDescriptionRequestCount` **must** be less than or equal to `renderPassRequestCount` multiplied by `VkPhysicalDeviceVulkanSC10Properties::maxRenderPassSubpasses`.

- **VUID-VkDeviceObjectReservationCreateInfo-attachmentDescriptionRequestCount-05018**
  
  `attachmentDescriptionRequestCount` **must** be less than or equal to `renderPassRequestCount` multiplied by `VkPhysicalDeviceVulkanSC10Properties::maxFramebufferAttachments`.

Valid Usage (Implicit)

- **VUID-VkDeviceObjectReservationCreateInfo-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_OBJECT_RESERVATION_CREATE_INFO`.

- **VUID-VkDeviceObjectReservationCreateInfo-pPipelineCacheCreateInfos-parameter**
  
  If `pipelineCacheCreateInfoCount` is not 0, `pPipelineCacheCreateInfos` **must** be a valid pointer to an array of `pipelineCacheCreateInfoCount` valid `VkPipelineCacheCreateInfo` structures.

- **VUID-VkDeviceObjectReservationCreateInfo-pPipelinePoolSizes-parameter**
  
  If `pipelinePoolSizeCount` is not 0, `pPipelinePoolSizes` **must** be a valid pointer to an array of `pipelinePoolSizeCount` valid `VkPipelinePoolSize` structures.

Memory for pipelines is reserved by the implementation at device creation time. The application specifies sizes to be reserved and a count for each size, and when a pipeline is created the application specifies which size to use.

```c
// Provided by VKSC_VERSION_1_0
typedef struct VkPipelinePoolSize {
    VkStructureType          sType;
    const void*              pNext;
    VkDeviceSize             poolEntrySize;
    uint32_t                  poolEntryCount;
};
```
VkPipelinePoolSize;

• sType is a VkStructureType value identifying this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• poolEntrySize is the size to reserve for each entry.
• poolEntryCount is the number of entries to reserve.

Valid Usage (Implicit)

• VUID-VkPipelinePoolSize-sType-sType
  sType must be VK_STRUCTURE_TYPE_PIPELINE_POOL_SIZE
• VUID-VkPipelinePoolSize-pNext-pNext
  pNext must be NULL

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

Fault Handling can be used by the implementation to provide more information on the cause of a device becoming lost. Allowing applications to take appropriate corrective behavior for the cause of the device lost.

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

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

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

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

If `VkPhysicalDeviceVulkanSC10Properties::deviceDestroyFreesMemory` is VK_TRUE, the reserved
memory for child objects without explicit free or destroy commands is returned to the system when
the device is destroyed, otherwise it may not be returned to the system until the process is
terminated.

**Valid Usage**

- VUID-vkDestroyDevice-device-05137
  All child objects created on device, except those with no explicit free or destroy
  command, must have been destroyed prior to destroying device
Valid Usage (Implicit)

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

- VUID-vkDestroyDevice-pAllocator-null
  `pAllocator` must be `NULL`

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:
The `VkDeviceQueueCreateInfo` structure is defined as:

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

- `sType` is a `VkStructureType` value identifying 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  
  **pQueue** must be a valid pointer to a **VkQueue** handle

To retrieve a handle to a **VkQueue** object with specific **VkDeviceQueueCreateInfo** creation flags, call:

```c
// Provided by VK_VERSION_1_1
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 a VkStructureType value identifying 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 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.

### 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:

// 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.

![Figure 1. Lifecycle of a command buffer](image)

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.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is **VK_TRUE**, `vkCreateCommandPool` **must not** return `VK_ERROR_OUT_OF_HOST_MEMORY`. 
Valid Usage

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

- VUID-vkCreateCommandPool-device-05068
  The number of command pools currently allocated from device plus 1 must be less than or equal to the total number of command pools requested via VkDeviceObjectReservationCreateInfo::commandPoolRequestCount specified when device was created

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-null
  pAllocator must be NULL

- 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 a VkStructureType value identifying 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

• VUID-VkCommandPoolCreateInfo-pNext-05002
  The pNext chain must include a VkCommandPoolMemoryReservationCreateInfo structure

### 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 or a pointer to a valid instance of VkCommandPoolMemoryReservationCreateInfo

• VUID-VkCommandPoolCreateInfo-sType-unique
  The sType value of each struct in the pNext chain must be unique

• 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 \texttt{vkBeginCommandBuffer}. If this flag is not set on a pool, then \texttt{vkResetCommandBuffer must} not be called for any command buffer allocated from that pool.

- \texttt{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;
```

\texttt{VkCommandPoolCreateFlags} is a bitmask type for setting a mask of zero or more \texttt{VkCommandPoolCreateFlagBits}.

The \texttt{pNext} chain of \texttt{VkCommandPoolCreateInfo} must include a \texttt{VkCommandPoolMemoryReservationCreateInfo} structure. This structure controls how much memory is allocated at command pool creation time to be used for all command buffers recorded from this pool.

The \texttt{VkCommandPoolMemoryReservationCreateInfo} structure is defined as:

```c
// Provided by VKSC_VERSION_1_0
typedef struct VkCommandPoolMemoryReservationCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkDeviceSize commandPoolReservedSize;
    uint32_t commandPoolMaxCommandBuffers;
} VkCommandPoolMemoryReservationCreateInfo;
```

- \texttt{sType} is a \texttt{VkStructureType} value identifying this structure.
- \texttt{pNext} is \texttt{NULL} or a pointer to a structure extending this structure.
- \texttt{commandPoolReservedSize} is the number of bytes to be allocated for all command buffer data recorded into this pool.
- \texttt{commandPoolMaxCommandBuffers} is the maximum number of command buffers that can be allocated from this command pool.

The number of command buffers reserved using \texttt{commandPoolMaxCommandBuffers} is permanently counted against the total number of command buffers requested via \texttt{VkDeviceObjectReservationCreateInfo::commandBufferRequestCount} even if the command buffers are freed at a later time.

Each command recorded into a command buffer has an implementation-dependent size that counts against \texttt{commandPoolReservedSize}. There is no minimum command pool size, but some sizes may be too small for any commands to be recorded in them on a given implementation. Applications are expected to estimate their worst-case command buffer memory usage at development time using \texttt{vkGetCommandPoolMemoryConsumption} and reserve large enough command buffers. This command can also be used at runtime to verify expected memory usage.

While the memory consumption of a particular command is implementation-dependent, it is a
deterministic function of the parameters to the command and of the objects used by the command (including the command buffer itself). Two command buffers will consume the same amount of pool memory if:

- all numerical parameters to each command match exactly,
- all objects used by each command are identically defined, and
- the order of the commands is the same.

\[\text{Note}\]

The rules for identically defined objects apply recursively, implying for example that if the command buffers are created in different devices that those devices must have been created with the same features enabled.

Each command buffer \textbf{may} require some base alignment in the pool, so the total pool memory will match if each command buffer’s consumption matches and the command buffers are recorded one at a time and in the same order.

If all these criteria are satisfied, then a command pool memory consumption returned by \texttt{vkGetCommandPoolMemoryConsumption} will be sufficient to record the same command buffers again.

<table>
<thead>
<tr>
<th>Valid Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>- VUID-VkCommandPoolMemoryReservationCreateInfo-commandPoolReservedSize-05003 \ntextbf{commandPoolReservedSize} must be greater than 0</td>
</tr>
<tr>
<td>- VUID-VkCommandPoolMemoryReservationCreateInfo-commandPoolMaxCommandBuffers-05004 \ntextbf{commandPoolMaxCommandBuffers} must be greater than 0</td>
</tr>
<tr>
<td>- VUID-VkCommandPoolMemoryReservationCreateInfo-commandPoolMaxCommandBuffers-05090 \ntextbf{commandPoolMaxCommandBuffers} must be less than or equal to \texttt{VkPhysicalDeviceVulkanSC10Properties::maxCommandPoolCommandBuffers}</td>
</tr>
<tr>
<td>- VUID-VkCommandPoolMemoryReservationCreateInfo-commandPoolMaxCommandBuffers-05074 The number of command buffers reserved by all command pools plus \textbf{commandPoolMaxCommandBuffers} must be less than or equal to the total number of command buffers requested via \texttt{VkDeviceObjectReservationCreateInfo::commandBufferRequestCount}</td>
</tr>
</tbody>
</table>

\[\text{Valid Usage (Implicit)}\]

- VUID-VkCommandPoolMemoryReservationCreateInfo-sType-sType \textbf{sType} must be \texttt{VK_STRUCTURE_TYPE_COMMAND_POOL_MEMORY_RESERVATION_CREATE_INFO}

To get memory usage information for a command pool object, call:
void vkGetCommandPoolMemoryConsumption(
    VkDevice device,            
    VkCommandPool commandPool,  
    VkCommandBuffer commandBuffer,  
    VkCommandPoolMemoryConsumption* pConsumption);

- `device` is the logical device that owns the command pool.
- `commandPool` is the command pool from which to query the memory usage.
- `commandBuffer` is an optional command buffer from which to query the memory usage.
- `pConsumption` is a pointer to a `VkCommandPoolMemoryConsumption` structure where the memory usage is written.

**Valid Usage (Implicit)**

- VUID-vkGetCommandPoolMemoryConsumption-device-parameter
  `device` must be a valid `VkDevice` handle
- VUID-vkGetCommandPoolMemoryConsumption-commandPool-parameter
  `commandPool` must be a valid `VkCommandPool` handle
- VUID-vkGetCommandPoolMemoryConsumption-commandBuffer-parameter
  If `commandBuffer` is not NULL, `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkGetCommandPoolMemoryConsumption-pConsumption-parameter
  `pConsumption` must be a valid pointer to a `VkCommandPoolMemoryConsumption` structure
- VUID-vkGetCommandPoolMemoryConsumption-commandPool-parent
  `commandPool` must have been created, allocated, or retrieved from `device`
- VUID-vkGetCommandPoolMemoryConsumption-commandBuffer-parent
  If `commandBuffer` is a valid handle, it must have been created, allocated, or retrieved from `commandPool`

**Host Synchronization**

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

The `VkCommandPoolMemoryConsumption` structure is defined as:

```c
// Provided by VKSC_VERSION_1_0
typedef struct VkCommandPoolMemoryConsumption {
    VkStructureType sType;
    void* pNext;
}
```
VkDeviceSize commandPoolAllocated;
VkDeviceSize commandPoolReservedSize;
VkDeviceSize commandBufferAllocated;
} VkCommandPoolMemoryConsumption;

- **sType** is a `VkStructureType` value identifying this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **commandPoolAllocated** is the number of bytes currently allocated from this pool for command buffer data.
- **commandPoolReservedSize** is the total number of bytes available for all command buffer data recorded into this pool. This is equal to the value requested in `VkCommandPoolMemoryReservationCreateInfo::commandPoolReservedSize`.
- **commandBufferAllocated** is the number of bytes currently allocated from this pool for the specified command buffer's data. This number will be less than or equal to `VkPhysicalDeviceVulkanSC10Properties::maxCommandBufferSize`. If no command buffer is specified, then this is set to zero.

### Valid Usage (Implicit)

- VUID-VkCommandPoolMemoryConsumption-sType-sType
  
  **sType** **must** be `VK_STRUCTURE_TYPE_COMMAND_POOL_MEMORY_CONSUMPTION`

- VUID-VkCommandPoolMemoryConsumption-pNext-pNext
  
  **pNext** **must** be NULL

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-zero bitmask
  flags must be 0
• 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:

// Provided by VK_VERSION_1_0
typedef enum VkCommandPoolResetFlagBits {
} VkCommandPoolResetFlagBits;

• VK_COMMAND_POOL_RESET_RELEASE_RESOURCES_BIT is not supported in Vulkan SC [SCID-4].

// Provided by VK_VERSION_1_0
typedef VkFlags VkCommandPoolResetFlags;

VkCommandPoolResetFlags is a bitmask type for setting a mask of zero or more VkCommandPoolResetFlagBits.
Command pools cannot be destroyed or trimmed \[SCID-4\]. If $\text{VkPhysicalDeviceVulkanSC10Properties::deviceDestroyFreesMemory}$ is \text{VK_TRUE}, then the memory used by command pools is returned to the system when the device is destroyed.

### 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);
```

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

\text{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 \text{pCommandBuffers} array to \text{NULL} and return the error.

\text{Note}

Filling \text{pCommandBuffers} with \text{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.

If $\text{VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations}$ is \text{VK_TRUE}, \text{vkAllocateCommandBuffers} must not return \text{VK_ERROR_OUT_OF_HOST_MEMORY}.

**Valid Usage (Implicit)**

- \text{VUID-vkAllocateCommandBuffers-device-parameter} \text{device} must be a valid \text{VkDevice} handle
- \text{VUID-vkAllocateCommandBuffers-pAllocateInfo-parameter} \text{pAllocateInfo} must be a valid pointer to a valid \text{VkCommandBufferAllocateInfo} structure
- \text{VUID-vkAllocateCommandBuffers-pCommandBuffers-parameter} \text{pCommandBuffers} must be a valid pointer to an array of \text{pAllocateInfo->commandBufferCount} \text{VkCommandBuffer} handles
The 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;
```

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

The number of command buffers allocated using `commandBufferCount` counts against the maximum number of command buffers reserved via VkCommandPoolMemoryReservationCreateInfo::commandPoolMaxCommandBuffers specified when `commandPool` was created. Once command buffers are freed with `vkFreeCommandBuffers`, they can be allocated from `commandPool` again.

Valid Usage

- VUID-VkCommandBufferAllocateInfo-commandPool-05006
  The number of command buffers currently allocated from `commandPool` plus `commandBufferCount` must be less than or equal to the value of
VkCommandPoolMemoryReservationCreateInfo::commandPoolMaxCommandBuffers specified when commandPool was created

Valid Usage (Implicit)

- VUID-VkCommandBufferAllocateInfo-sType-sType sType must be VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO
- VUID-VkCommandBufferAllocateInfo-pNext-pNext 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
// Provided by VK_VERSION_1_0
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
// Provided by VK_VERSION_1_0
VkResult vkResetCommandBuffer(
    VkCommandBuffer commandBuffer,
    VkCommandBufferResetFlags flags);
```

- commandBuffer is the command buffer to reset. The command buffer can be in any state other than pending, and is moved into the initial state.
- flags is a bitmask 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
commandBuffer must have been allocated from a pool that was created with the VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT

commandPoolResetCommandBuffer must be supported

Valid Usage (Implicit)

commandBuffer must be a valid VkCommandBuffer handle

flags must be a valid combination of VkCommandBufferResetFlagBits values

Host Synchronization

Host access to commandBuffer must be externally synchronized

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, 
    VkCommandPool commandPool, 
    uint32_t commandBufferCount, 
    const VkCommandBuffer* pCommandBuffers);
```

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

Any primary command buffer that is in the recording or executable state and has any element of **pCommandBuffers** recorded into it, becomes invalid.

Freeing a command buffer does not return the memory used by command recording back to its parent command pool. This memory will be reclaimed the next time vkResetCommandPool is called.

### 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.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkBeginCommandBuffer` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

### 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-commandPoolResetCommandBuffer-05136
  If `commandPoolResetCommandBuffer` is not supported, `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

- VUID-vkBeginCommandBuffer-commandPoolMultipleCommandBuffersRecording-05007
  If commandPoolMultipleCommandBuffersRecording is VK_FALSE, then the command pool that
  commandBuffer was created from must have no other command buffers in the recording
  state

- VUID-vkBeginCommandBuffer-commandBufferSimultaneousUse-05008
  If commandBufferSimultaneousUse is VK_FALSE, then pBeginInfo->flags must not include
  VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT

---

**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 a `VkStructureType` value identifying 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-09123**
  If `flags` contains `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT`, the `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations.

- **VUID-VkCommandBufferBeginInfo-flags-05009**
  If `flags` contains `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT` and `secondaryCommandBufferNullOrImagelessFramebuffer` is `VK_TRUE`, 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-05010**
  If `flags` contains `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT` and `secondaryCommandBufferNullOrImagelessFramebuffer` is `VK_FALSE`, the framebuffer member of `pInheritanceInfo` and must not have been created with a `VkFramebufferCreateInfo::flags` value that includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`.

- **VUID-VkCommandBufferBeginInfo-flags-06000**
  If `flags` contains `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT` the `renderPass` member of `pInheritanceInfo` must be a valid `VkRenderPass`.

- **VUID-VkCommandBufferBeginInfo-flags-06001**
  If `flags` contains `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT` the subpass member of `pInheritanceInfo` must be a valid subpass index within the `renderPass` member of `pInheritanceInfo`.

### 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`. 

118
The `sType` value of each struct in the `pNext` chain must be unique.

The `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 any queue of the same queue family 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:

```c
// 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;
};
```
• `sType` is a `VkStructureType` value identifying 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, `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
If the inheritedQueries feature is not enabled, queryFlags must be 0.

If the pipelineStatisticsQuery feature is enabled, pipelineStatistics must be a valid combination of VkQueryPipelineStatisticFlagBits values.

If the pipelineStatisticsQuery feature is not enabled, pipelineStatistics must be 0.

Valid Usage (Implicit)

- sType must be VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_INFO
- pNext must be NULL
- Both of framebuffer, and renderPass that are valid handles 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.

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.

The command buffer must have been in the recording state, and, if successful, is moved to the executable state.

If there was an error during recording, the application will be notified by an unsuccessful return
code returned by `vkEndCommandBuffer`, and the command buffer will be moved to the invalid state.

If recording a command would exceed the amount of command pool memory reserved by `VkCommandPoolMemoryReservationCreateInfo::commandPoolReservedSize`, the implementation may report a `VK_FAULT_TYPE_COMMAND_BUFFER_FULL` fault. The command buffer remains in the recording state until `vkEndCommandBuffer` is called. When `vkEndCommandBuffer` is called on a command buffer for which the command pool memory reservation was exceeded during recording, it must return `VK_ERROR_OUT_OF_DEVICE_MEMORY`.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkEndCommandBuffer` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

### 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`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
When a command buffer is in the executable state, it can be submitted to a queue for execution.

### 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` as possible.

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...
If \texttt{VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations} is \texttt{VK_TRUE}, \texttt{vkQueueSubmit} must not return \texttt{VK_ERROR_OUT_OF_HOST_MEMORY}.

### Valid Usage

- **VUID-vkQueueSubmit-fence-00063**
  If \textit{fence} is not \texttt{VK_NULL_HANDLE}, \textit{fence} must be unsigned

- **VUID-vkQueueSubmit-fence-00064**
  If \textit{fence} is not \texttt{VK_NULL_HANDLE}, \textit{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 \texttt{vkCmdSetEvent}, \texttt{vkCmdResetEvent} or \texttt{vkCmdWaitEvents} that have been recorded into any of the command buffer elements of the \texttt{pCommandBuffers} member of any element of \textit{pSubmits}, must not reference any \textit{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 \textit{pending state}

- **VUID-vkQueueSubmit-pWaitDstStageMask-00066**
  Any stage flag included in any element of the \texttt{pWaitDstStageMask} member of any element of \textit{pSubmits} must be a pipeline stage supported by one of the capabilities of \textit{queue}, as specified in the \textit{table of supported pipeline stages}

- **VUID-vkQueueSubmit-pSignalSemaphores-00067**
  Each binary semaphore element of the \texttt{pSignalSemaphores} member of any element of \textit{pSubmits} must be unsigned 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 \texttt{pWaitSemaphores} member of any element of \textit{pSubmits} executes on \textit{queue}, there must be no other queues waiting on the same semaphore

- **VUID-vkQueueSubmit-pWaitSemaphores-03238**
  All elements of the \texttt{pWaitSemaphores} member of all elements of \textit{pSubmits} created with a \textit{VkSemaphoreType} of \texttt{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 must have also been submitted for execution

- **VUID-vkQueueSubmit-pCommandBuffers-00070**
  Each element of the \texttt{pCommandBuffers} member of each element of \textit{pSubmits} must be in the \textit{pending} or \textit{executable state}

- **VUID-vkQueueSubmit-pCommandBuffers-00071**
  If any element of the \texttt{pCommandBuffers} member of any element of \textit{pSubmits} was not recorded with the \texttt{VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT}, it must not be in the \textit{pending state}
Any secondary command buffers recorded into any element of the `pCommandBuffers` member of any element of `pSubmits` must be in the pending or executable state.

If any secondary command buffers recorded into any element of the `pCommandBuffers` member of any element of `pSubmits` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT`, it must not be in the pending state.

Each element of the `pCommandBuffers` member of each element of `pSubmits` must have been allocated from a `VkCommandPool` that was created for the same queue family `queue` belongs to.

If any element of `pSubmits->pCommandBuffers` includes a Queue Family Transfer Acquire Operation, there must exist a previously submitted Queue Family Transfer Release Operation on a queue in the queue family identified by the acquire operation, with parameters matching the acquire operation as defined in the definition of such acquire operations, and which happens-before the acquire operation.

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.

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.

If `queue` was not created with `VKDEVICE_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)**

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

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

Command Properties

<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>-</td>
<td>-</td>
<td>Any</td>
<td>-</td>
</tr>
</tbody>
</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
typedef struct VkSubmitInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t waitSemaphoreCount;
    const VkSemaphore* pWaitSemaphores;
    const VkPipelineStageFlags* pWaitDstStageMask;
    uint32_t commandBufferCount;
    const VkCommandBuffer* pCommandBuffers;
    uint32_t signalSemaphoreCount;
    const VkSemaphore* pSignalSemaphores;
} VkSubmitInfo;
```

- `sType` is a VkStructureType value identifying 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-04996**
  pWaitDstStageMask must not be 0

- **VUID-VkSubmitInfo-pCommandBuffers-00075**
  Each element of pCommandBuffers must not have been allocated with VK_COMMAND_BUFFER_LEVEL_SECONDARY

- **VUID-VkSubmitInfo-pWaitDstStageMask-00078**
  Each element of pWaitDstStageMask must not include VK_PIPELINE_STAGE_HOST_BIT

- **VUID-VkSubmitInfo-pWaitSemaphores-03239**
  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-VkSubmitInfo-pNext-03240**
  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-VkSubmitInfo-pNext-03241**
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-VkSubmitInfo-pSignalSemaphores-03242
  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-VkSubmitInfo-pWaitSemaphores-03243
  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

- VUID-VkSubmitInfo-pSignalSemaphores-03244
  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

- VUID-VkSubmitInfo-pNext-04120
  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

- VUID-VkSubmitInfo-pNext-04148
  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

**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`.
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 a `VkStructureType` value identifying 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 `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;
} VkDeviceGroupSubmitInfo;
```
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 a VkStructureType value identifying 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_TYPE_DEVICE_GROUP_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

Secondary command buffers **must** not be directly submitted to a queue. To record a secondary command buffer to execute as part of a primary command buffer, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdExecuteCommands(
    VkCommandBuffer commandBuffer, 
    uint32_t commandBufferCount, 
    const VkCommandBuffer* pCommandBuffers);
```

- `commandBuffer` is a handle to a primary command buffer that the secondary command buffers are executed in.
- `commandBufferCount` is the length of the `pCommandBuffers` array.
- `pCommandBuffers` is a pointer to an array of `commandBufferCount` secondary command buffer handles, which are recorded to execute in the primary command buffer in the order they are listed in the array.

If any element of `pCommandBuffers` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flag, and it was recorded into any other primary command buffer which is currently in the executable or recording state, that primary command buffer becomes **invalid**.

## Valid Usage

- **VUID-vkCmdExecuteCommands-pCommandBuffers-00088**
  Each element of `pCommandBuffers` **must** have been allocated with a level of `VK_COMMAND_BUFFER_LEVEL_SECONDARY`

- **VUID-vkCmdExecuteCommands-pCommandBuffers-00089**
  Each element of `pCommandBuffers` **must** be in the pending or executable state

- **VUID-vkCmdExecuteCommands-pCommandBuffers-00091**
  If any element of `pCommandBuffers` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flag, it **must** not be in the pending state

- **VUID-vkCmdExecuteCommands-pCommandBuffers-00092**
  If any element of `pCommandBuffers` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flag, it **must** not have already been recorded to `commandBuffer`

- **VUID-vkCmdExecuteCommands-pCommandBuffers-00093**
  If any element of `pCommandBuffers` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flag, it **must** not appear more than once in `pCommandBuffers`

- **VUID-vkCmdExecuteCommands-pCommandBuffers-00094**
  Each element of `pCommandBuffers` **must** have been allocated from a `VkCommandPool` that was created for the same queue family as the `VkCommandPool` from which `commandBuffer` was
allocated

- VUID-vkCmdExecuteCommands-pCommandBuffers-00096
  If `vkCmdExecuteCommands` is being called within a render pass instance, each element of
  `pCommandBuffers` must have been recorded with the
  `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT`

- VUID-vkCmdExecuteCommands-pCommandBuffers-00099
  If `vkCmdExecuteCommands` is being called within a render pass instance, and any element of
  `pCommandBuffers` was recorded with `VkCommandBufferInheritanceInfo::framebuffer` not
  equal to `VK_NULL_HANDLE`, that `VkFramebuffer` must match the `VkFramebuffer` used in the
  current render pass instance

- VUID-vkCmdExecuteCommands-contents-06018
  If `vkCmdExecuteCommands` is being called within a render pass instance begun with
  `vkCmdBeginRenderPass`, its `contents` parameter must have been set to
  `VK_SUBPASS_CONTENTS_SECONDARY_COMMAND BUFFERS`

- VUID-vkCmdExecuteCommands-pCommandBuffers-06019
  If `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_PIPELINE_STATISTICS` 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-commandBuffer-09375
  `commandBuffer` **must** not be a secondary command buffer

---

**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-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

---

**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</td>
<td>Indirection</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

---

**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 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 a `VkStructureType` value identifying 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
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-cmddpool
  
  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, Secondary</td>
<td>Both</td>
<td>Graphics, Compute, Transfer</td>
<td>State</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, controlling resource access between host and device.

**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. Events can be used to control resource access within a single queue.

**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. Pipeline barriers can be used to control resource access within a single queue.

**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. Render pass objects can be used to control resource access within a single queue.

### 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 $\text{Scope}_{1\text{st}}$ and $\text{Scope}_{2\text{nd}}$ be the synchronization scopes of $\text{Sync}$.
- Let $\text{ScopedOps}_1$ be the intersection of sets $\text{Ops}_1$ and $\text{Scope}_{1\text{st}}$.
- Let $\text{ScopedOps}_2$ be the intersection of sets $\text{Ops}_2$ and $\text{Scope}_{2\text{nd}}$.
- 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 $\text{Scope}_{2\text{nd}}$ in the first dependency and $\text{Scope}_{1\text{st}}$ 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 $\text{Scope}_{1\text{st}}$ be the first synchronization scope of the first command in $\text{Sync}$.
- Let $\text{Scope}_{2\text{nd}}$ 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 \texttt{ScopedOps}_1 is made available. Any type of access that is in a memory dependency’s second access scope and occurs in \texttt{ScopedOps}_2 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 \texttt{MemOps}_1 be the set of memory accesses performed by \texttt{ScopedOps}_1.
- Let \texttt{MemOps}_2 be the set of memory accesses performed by \texttt{ScopedOps}_2.
- Let \texttt{AccessScope}_\text{1st} be the first access scope of the first command in the \texttt{Sync} chain.
- Let \texttt{AccessScope}_\text{2nd} be the second access scope of the last command in the \texttt{Sync} chain.
- Let \texttt{ScopedMemOps}_1 be the intersection of sets \texttt{MemOps}_1 and \texttt{AccessScope}_\text{1st}.
- Let \texttt{ScopedMemOps}_2 be the intersection of sets \texttt{MemOps}_2 and \texttt{AccessScope}_\text{2nd}.
- Submitting \texttt{Ops}_1, \texttt{Sync}, and \texttt{Ops}_2 for execution, in that order, will result in a memory dependency \texttt{MemDep} between \texttt{ScopedOps}_1 and \texttt{ScopedOps}_2.

• Memory dependency \texttt{MemDep} guarantees that:
  ◦ Memory writes in \texttt{ScopedMemOps}_1 are made available.
  ◦ Available memory writes, including those from \texttt{ScopedMemOps}_1, are made visible to \texttt{ScopedMemOps}_2.

\textbf{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.

\textbf{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.

**Note**

Image layout transitions with VK_IMAGE_LAYOUT_UNDEFINED allow the implementation to discard the image subresource range, which can provide performance or power benefits. Tile-based architectures may be able to avoid flushing tile data to memory, and immediate style renderers may be able to achieve fast metadata clears to reinitialize frame buffer compression state, or similar.

If the contents of an attachment are not needed after a render pass completes, then applications should use VK_ATTACHMENT_STORE_OP_DONT_CARE.

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 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.
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 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,
} VkPipelineStageFlagBits;
```

- **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 render pass 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 render pass 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 includes blending, logic operations, render pass load and store operations for color attachments, render pass multisample resolve operations, 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
  ◦ vkCmdBlitImage
  ◦ 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 any **logically earlier** stages. Its first **access scope** only includes memory accesses performed by pipeline stages explicitly specified in the source stage 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 any **logically later** stages. Its second **access scope** only includes memory accesses performed by pipeline stages explicitly specified in the destination stage mask.

**Note**

Note that **access scopes** do not interact with the logically earlier or later stages for either scope - only the stages the app specifies 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 4. 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_2_NONE</td>
<td>None required</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_TOP_OF_PIPE_BIT</td>
<td>None required</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>Pipeline stage flag</td>
<td>Required queue capability flag</td>
</tr>
<tr>
<td>--------------------------------------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_COMPUTE_SHADER_BIT</td>
<td>VK_QUEUE_COMPUTE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT or VK_QUEUE_TRANSFER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_BOTTOM_OF_PIPE_BIT</td>
<td>None required</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_HOST_BIT</td>
<td>None required</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT</td>
<td>None required</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_COPY_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT or VK_QUEUE_TRANSFER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_RESOLVE_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT or VK_QUEUE_TRANSFER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_BLIT_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT or VK_QUEUE_TRANSFER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_CLEAR_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT or VK_QUEUE_TRANSFER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_INDEX_INPUT_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>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_2_PRE_RASTERIZATION_SHADERS_BIT</td>
<td>VK_QUEUE_GRAPHICS_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 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_2_DRAW_INDIRECT_BIT`
- `VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT`
- `VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_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_EARLY_FRAGMENT_TESTS_BIT`
- `VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT`
- `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT`
- `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT`

For the compute pipeline, the following stages occur in this order:

- `VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT`
- `VK_PIPELINE_STAGE_2_COMPUTE_SHADER_BIT`

For the transfer pipeline, the following stages occur in this order:

- `VK_PIPELINE_STAGE_2_TRANSFER_BIT`

For host operations, only one pipeline stage occurs, so no order is guaranteed:

- `VK_PIPELINE_STAGE_2_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 VkSubpassDependency, VkMemoryBarrier, VkBufferMemoryBarrier, and VkImageMemoryBarrier, specifying access behavior, are:

```c
// 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,
} VkAccessFlagBits;
```

- **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 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 certain render pass load operations. Such access occurs in the **VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT** pipeline stage.

• **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 render pass 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 certain render pass load operations. Such access occurs in the **VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT** or **VK_PIPELINE_STAGE_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 certain render pass load and store operations. Such access occurs in the **VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT** or **VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT** pipeline stages.

• **VK_ACCESS_TRANSFER_READ_BIT** specifies read access to an image or buffer in a copy operation.

• **VK_ACCESS_TRANSFER_WRITE_BIT** specifies write access to an image or buffer in a clear or copy operation.

• **VK_ACCESS_HOST_READ_BIT** specifies read access by a host operation. Accesses of this type are not performed through a resource, but directly on memory. 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>
<thead>
<tr>
<th>Access flag</th>
<th>Supported pipeline stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_ACCESS_2_NONE</td>
<td>Any</td>
</tr>
<tr>
<td>VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT</td>
<td>VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_2_INDEX_READ_BIT</td>
<td>VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT, VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT</td>
<td>VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT, VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_2_UNIFORM_READ_BIT</td>
<td>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_COMPUTE_SHADER_BIT,</td>
</tr>
<tr>
<td>VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_2_SHADER_READ_BIT</td>
<td>VK_PIPELINE_STAGE_2_FRAGMENT_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_COMPUTE_SHADER_BIT,</td>
</tr>
<tr>
<td>VK_ACCESS_2_SHADER_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_2_FRAGMENT_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_COMPUTE_SHADER_BIT,</td>
</tr>
<tr>
<td>VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT, VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT,</td>
</tr>
<tr>
<td>VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT, VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT, VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT, VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>Access flag</td>
<td>Supported pipeline stages</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VK_ACCESS_2_TRANSFER_READ_BIT</td>
<td>VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT, VK_PIPELINE_STAGE_2_COPY_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_2_RESOLVE_BIT, VK_PIPELINE_STAGE_2_BLIT_BIT,</td>
</tr>
<tr>
<td>VK_ACCESS_2_TRANSFER_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT, VK_PIPELINE_STAGE_2_COPY_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_2_RESOLVE_BIT, VK_PIPELINE_STAGE_2_BLIT_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_2_CLEAR_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_2_HOST_READ_BIT</td>
<td>VK_PIPELINE_STAGE_2_HOST_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_2_HOST_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_2_HOST_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_2_MEMORY_READ_BIT</td>
<td>Any</td>
</tr>
<tr>
<td>VK_ACCESS_2_MEMORY_WRITE_BIT</td>
<td>Any</td>
</tr>
<tr>
<td>VK_ACCESS_2_SHADER_SAMPLED_READ_BIT</td>
<td>VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT, VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_2_COMPUTE_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_2_SHADER_STORAGE_READ_BIT</td>
<td>VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT, VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_2_COMPUTE_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT, VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_2_COMPUTE_SHADER_BIT</td>
</tr>
</tbody>
</table>

// Provided by VK_VERSION_1_0

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, 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 \((x, y, \text{layer}, \text{sample})\) 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` commands are executed on the host, for a single queue, from first to last.
2. The order in which `VkSubmitInfo` structures are specified in the `pSubmits` parameter of `vkQueueSubmit`, from lowest index to highest.
3. The order in which command buffers are specified in the `pCommandBuffers` member of `VkSubmitInfo` from lowest index to highest.

4. The order in which commands were recorded to a command buffer on the host, from first to last:
   - For commands recorded outside a render pass, this includes all other commands recorded outside a render pass, including `vkCmdBeginRenderPass` and `vkCmdEndRenderPass` commands; it does not directly include commands inside a render pass.
   - For commands recorded inside a render pass, this includes all other commands recorded inside the same subpass, including the `vkCmdBeginRenderPass` and `vkCmdEndRenderPass` commands that delimit the same render pass instance; it does not include commands recorded to other subpasses. **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` commands are executed on the host, for a single queue, from first to last.
   2. The order in which `VkSubmitInfo` structures are specified in the `pSubmits` parameter of `vkQueueSubmit`, from lowest index to highest.
   3. The fence signal operation defined by the `fence` parameter of a `vkQueueSubmit` command is ordered after all semaphore signal operations defined by that command.

   Semaphore signal operations defined by a single `VkSubmitInfo` 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.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkCreateFence` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

### Valid Usage

- VUID-vkCreateFence-device-05068
  The number of fences currently allocated from `device` plus 1 must be less than or equal to the total number of fences requested via `VkDeviceObjectReservationCreateInfo::fenceRequestCount` specified when `device` was created

### 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-null
  `pAllocator` must be NULL
- 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 a `VkStructureType` value identifying 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

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.

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:

typedef struct VkExportFenceCreateInfo {

```c
// Provided by VK_VERSION_1_1
typedef struct VkExportFenceCreateInfo {
```

```c
```
VkExportFenceCreateInfo

- **sType** is a *VkStructureType* value identifying 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
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
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-null**
  
  *pAllocator* must be **NULL**

- **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>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_SUCCESS</td>
<td>The fence specified by <em>fence</em> is signaled.</td>
</tr>
<tr>
<td>VK_NOT_READY</td>
<td>The fence specified by <em>fence</em> is unsignaled.</td>
</tr>
<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**.

If **VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations** is **VK_TRUE**, **vkGetFenceStatus** must not return **VK_ERROR_OUT_OF_HOST_MEMORY**.

### 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 is the logical device that owns the fences.
    uint32_t fenceCount,  // fenceCount is the number of fences to reset.
    const VkFence* pFences);  // 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 \texttt{pFences} is already in the unsignaled state when \texttt{vkResetFences} is executed, then \texttt{vkResetFences} has no effect on that fence.

### Valid Usage

- VUID-vkResetFences-pFences-01123
  Each element of \texttt{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
  \texttt{device} must be a valid \texttt{VkDevice} handle

- VUID-vkResetFences-pFences-parameter
  \texttt{pFences} must be a valid pointer to an array of \texttt{fenceCount} valid \texttt{VkFence} handles

- VUID-vkResetFences-fenceCount-arraylength
  \texttt{fenceCount} must be greater than \texttt{0}

- VUID-vkResetFences-pFences-parent
  Each element of \texttt{pFences} must have been created, allocated, or retrieved from \texttt{device}

### Host Synchronization

- Host access to each member of \texttt{pFences} must be externally synchronized

### Return Codes

**Success**
- \texttt{VK_SUCCESS}

**Failure**
- \texttt{VK_ERROR_OUT_OF_DEVICE_MEMORY}

When a fence is submitted to a queue as part of a \texttt{queue submission} command, it defines a memory dependency on the batches that were submitted as part of that command, and defines a \textit{fence signal operation} which sets the fence to the signaled state.

The first \texttt{synchronization scope} includes every batch submitted in the same \texttt{queue submission} command. Fence signal operations that are defined by \texttt{vkQueueSubmit} additionally include in the first synchronization scope all commands that occur earlier in \texttt{submission order}. Fence signal operations that are defined by \texttt{vkQueueSubmit} additionally include in the first synchronization scope any semaphore and fence signal operations that occur earlier in \texttt{signal operation order}.

The second \texttt{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:

```
// 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.

If **VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations** is **VK_TRUE**,
vkWaitForFences must not return VK_ERROR_OUT_OF_HOST_MEMORY.

### 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 \texttt{vkResetFences}.

\begin{quote}
\textbf{Note}

Restoring a fence to its prior permanent payload is a distinct operation from resetting a fence payload. See \texttt{vkResetFences} for more detail.
\end{quote}

Performing a subsequent temporary import on a fence before resetting it has no effect on this requirement; the next unsignal of the fence \textbf{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 \texttt{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 \textit{permanence}. Each handle type supports either one or both types of permanence.

The implementation \textbf{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 \textit{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 \textbf{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 \textbf{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.

\textbf{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 \texttt{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:

\begin{itemize}
  \item 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.
  \item 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.
  \item 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 \textbf{may} become signaled again when the command completes execution.
\end{itemize}
• 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 vkWaitSemaphores command.

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:

```cpp
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSemaphore)
```

To create a semaphore, call:

```cpp
// 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.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkCreateSemaphore` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

**Valid Usage**

- **VUID-vkCreateSemaphore-device-05068**
  The number of semaphores currently allocated from `device` plus 1 must be less than or equal to the total number of semaphores requested via `VkDeviceObjectReservationCreateInfo::semaphoreRequestCount` specified when `device` was created.

**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-null**
  `pAllocator` must be `NULL`
- **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 a *VkStructureType* value identifying 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:

```c
// Provided by VK_VERSION_1_2
typedef struct VkSemaphoreTypeCreateInfo {
    VkStructureType sType;
    const void*   pNext;
    VkSemaphoreType semaphoreType;
    uint64_t      initialValue;
} VkSemaphoreTypeCreateInfo;
```

- `sType` is a `VkStructureType` value identifying 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:

```c
// Provided by VK_VERSION_1_2
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 initial value 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
typedef struct VkExportSemaphoreCreateInfo {
    VkStructureType sType;
    const void*pNext;
    VkExternalSemaphoreHandleTypeFlags handleTypes;
} VkExportSemaphoreCreateInfo;
```

- **sType** is a VkStructureType value identifying 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:
// Provided by VK_VERSION_1_0

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

- `device` is the logical device that destroys the semaphore.
- `semaphore` is the handle of the semaphore to destroy.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage

- VUID-vkDestroySemaphore-semaphore-05149
  All submitted batches that refer to `semaphore` must have completed execution

### 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-null
  `pAllocator` must be `NULL`

- 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.
signal operations that are defined by `vkQueueSubmit` additionally include all commands that occur earlier in submission order. Semaphore signal operations that are defined by `vkQueueSubmit` 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`. Also, in the case of `vkQueueSubmit`, the second synchronization scope additionally includes all commands that occur later in submission order.

The first access scope is empty.

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

The semaphore wait operation happens-after the first set of operations in the execution dependency, and happens-before the second set of operations in the execution dependency.

**Note**

Unlike timeline semaphores, fences or events, the act of waiting for a binary semaphore also unsignals that semaphore. Applications **must** ensure that between two such wait operations, the semaphore is signaled again, with execution dependencies used to ensure these occur in order. Binary semaphore waits and signals should thus occur in discrete 1:1 pairs.
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,          // device is the logical device that owns the semaphore.
    VkSemaphore semaphore,    // semaphore is the handle of the semaphore to query.
    uint64_t* pValue);         // 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.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkGetSemaphoreCounterValue` **must** not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

174
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`.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkWaitSemaphores` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

### 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 a `VkStructureType` value identifying 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.

```
// 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:

```
// 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.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is VK_TRUE, `vkSignalSemaphore` must not return VK_ERROR_OUT_OF_HOST_MEMORY.

**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

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

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 a `VkStructureType` value identifying 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`
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:

```cpp
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkEvent)
```

To create an event, call:

```cpp
// 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.

If VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations is VK_TRUE, vkCreateEvent must not return VK_ERROR_OUT_OF_HOST_MEMORY.

Valid Usage

- VUID-vkCreateEvent-device-05068
  The number of events currently allocated from device plus 1 must be less than or equal to the total number of events requested via VkDeviceObjectReservationCreateInfo::eventRequestCount specified when device was created
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-null
pAllocator must be NULL

- 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 a VkStructureType value identifying 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-zerobitmask
  flags must be 0
typedef enum VkEventCreateFlagBits {
} VkEventCreateFlagBits;

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 VkEventCreateFlags;

VkEventCreateFlags is a bitmask type for setting a mask of VkEventCreateFlagBits.

To destroy an event, call:

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

• device is the logical device that destroys the event.
• event is the handle of the event to destroy.
• pAllocator controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

• VUID-vkDestroyEvent-event-01145
  All submitted commands that refer to event must have completed execution

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-null
pAllocator must be NULL

• 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:

```cpp
// Provided by VK_VERSION_1_0
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.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkGetEventStatus` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

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
- VK_EVENT_RESET

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST

To set the state of an event to signaled from the host, call:

```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.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is VK_TRUE, `vkSetEvent` must not return VK_ERROR_OUT_OF_HOST_MEMORY.

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

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

```c
// Provided by VK_VERSION_1_0
void vkCmdSetEvent(
    VkCommandBuffer commandBuffer,  // commandBuffer is the command buffer into which the command is recorded.
    VkEvent event,  // event is the event that will be signaled.
    VkPipelineStageFlags stageMask);  // stageMask specifies the source stage mask used to determine the first synchronization scope.
```

When `vkCmdSetEvent` is submitted to a queue, it defines an execution dependency on commands that were submitted before it, and defines an event signal operation which sets the event to the signaled state.

The first **synchronization scope** includes all commands that occur earlier in submission order. The synchronization scope is limited to operations on the pipeline stages determined by the source stage mask specified by `stageMask`.

The second **synchronization scope** includes only the event signal operation.

If `event` is already in the signaled state when `vkCmdSetEvent` is executed on the device, then `vkCmdSetEvent` has no effect, no event signal operation occurs, and no execution dependency is generated.

Valid Usage

- VUID-vkCmdSetEvent-stageMask-04090

  If the **geometryShader** feature is not enabled, `stageMask` **must** not contain
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

stageMask must not contain VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT

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

stageMask must not include VK_PIPELINE_STAGE_HOST_BIT

The current device mask of commandBuffer must include exactly one physical device

Valid Usage (Implicit)

commandBuffer must be a valid VkCommandBuffer handle

event must be a valid VkEvent handle

stageMask must be a valid combination of VkPipelineStageFlagBits values

commandBuffer must be in the recording state

The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations

commandBuffer must only be called outside of a render pass instance

Both of commandBuffer, and event must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

Host access to commandBuffer must be externally synchronized

Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized
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.

When `vkCmdResetEvent` is submitted to a queue, it defines an execution dependency on commands that were submitted before it, and defines an event unsignal operation which resets the event to the unsignaled state.

The first synchronization scope includes all commands that occur earlier in submission order. The synchronization scope is limited to operations on the pipeline stages determined by the source stage mask specified by `stageMask`.

The second synchronization scope includes only the event unsignal operation.

If `event` is already in the unsignaled state when `vkCmdResetEvent` is executed on the device, then `vkCmdResetEvent` has no effect, no event unsignal operation occurs, and no execution dependency is generated.

**Valid Usage**

- 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-04996
**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 `VkCommandPoolCreateInfo` structure that was used to create the `VkCommandPool` 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-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 `VkCommandBuffer` handle.

- VUID-vkCmdResetEvent-event-parameter
  `event` must be a valid `VkEvent` handle.

- VUID-vkCmdResetEvent-stageMask-parameter
  **stageMask** must be a valid combination of `VkPipelineStageFlagBits` values.

- VUID-vkCmdResetEvent-commandBuffer-recording
  `commandBuffer` must be in the recording state.

- VUID-vkCmdResetEvent-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute 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.
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.

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.

---

**Note**

vkCmdWaitEvents is used with vkCmdSetEvent to define a memory dependency between two sets of action commands, roughly in the same way as pipeline barriers, but split into two commands such that work between the two may execute unhindered.

Unlike vkCmdPipelineBarrier, a queue family ownership transfer cannot be performed using vkCmdWaitEvents.

**Note**

Applications should be careful to avoid race conditions when using events. There is no direct ordering guarantee between vkCmdWaitEvents and vkCmdResetEvent, or vkCmdSetEvent. Another execution dependency (e.g. a pipeline barrier or semaphore with VK_PIPELINE_STAGE_ALL_COMMANDS_BIT) is needed to prevent such a race condition.

---

**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 VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
• VUID-vkCmdWaitEvents-srcStageMask-04996
  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-04996
  dstStageMask must not be 0

• VUID-vkCmdWaitEvents-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-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.

**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 `VkMemoryBarrier` structures.

- **VUID-vkCmdWaitEvents-pBufferMemoryBarriers-parameter**
  If `bufferMemoryBarrierCount` is not 0, `pBufferMemoryBarriers` must be a valid pointer to an array of `VkBufferMemoryBarrier` structures.
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, or
  compute 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>
</tbody>
</table>

7.6. Pipeline Barriers

To record a pipeline barrier, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdPipelineBarrier(
    VkCommandBuffer commandBuffer,
    VkPipelineStageFlags srcStageMask,
    VkPipelineStageFlags dstStageMask,
    VkDependencyFlags dependencyFlags,
    const VkMemoryBarrier* memoryBarrierCount,
    VkMemoryBarrier* pMemoryBarriers,
    uint32_t bufferMemoryBarrierCount,
    const VkBufferMemoryBarrier* pBufferMemoryBarriers,
);```
`void vkCmdPipelineBarrier
    (uint32_t commandBuffer,
     const VkPipelineStageFlags srcStageMask,
     const VkPipelineStageFlags dstStageMask,
     const 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 VkPipelineStageFlags specifying the source stages.
• **dstStageMask** is a bitmask of VkPipelineStageFlags specifying the destination stages.
• **dependencyFlags** is a bitmask of VkDependencyFlags specifying how execution and memory dependencies are formed.
• **memoryBarrierCount** is the length of the pMemoryBarriers array.
• **pMemoryBarriers** is a pointer to an array of VkMemoryBarrier structures.
• **bufferMemoryBarrierCount** is the length of the pBufferMemoryBarriers array.
• ** pBufferMemoryBarriers** is a pointer to an array of VkBufferMemoryBarrier structures.
• **imageMemoryBarrierCount** is the length of the pImageMemoryBarriers array.
• **pImageMemoryBarriers** is a pointer to an array of VkImageMemoryBarrier structures.

When vkCmdPipelineBarrier is submitted to a queue, it defines a memory dependency between commands that were submitted to the same queue before it, and those submitted to the same queue 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 \( \text{dependencyFlags} \) includes \( \text{VK_DEPENDENCY_BY_REGION_BIT} \), then any dependency between \( \text{framebuffer-space} \) pipeline stages is \( \text{framebuffer-local} \) - otherwise it is \( \text{framebuffer-global} \).

### Valid Usage

- **VUID-vkCmdPipelineBarrier-srcStageMask-04090**
  - If the \( \text{geometryShader} \) feature is not enabled, \( \text{srcStageMask} \) **must** not contain \( \text{VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT} \).

- **VUID-vkCmdPipelineBarrier-srcStageMask-04091**
  - If the \( \text{tessellationShader} \) feature is not enabled, \( \text{srcStageMask} \) **must** not contain \( \text{VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT} \) or \( \text{VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT} \).

- **VUID-vkCmdPipelineBarrier-srcStageMask-04996**
  - \( \text{srcStageMask} \) **must** not be \( \emptyset \).

- **VUID-vkCmdPipelineBarrier-dstStageMask-04090**
  - If the \( \text{geometryShader} \) feature is not enabled, \( \text{dstStageMask} \) **must** not contain \( \text{VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT} \).

- **VUID-vkCmdPipelineBarrier-dstStageMask-04091**
  - If the \( \text{tessellationShader} \) feature is not enabled, \( \text{dstStageMask} \) **must** not contain \( \text{VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT} \) or \( \text{VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT} \).

- **VUID-vkCmdPipelineBarrier-dstStageMask-04996**
  - \( \text{dstStageMask} \) **must** not be \( \emptyset \).

- **VUID-vkCmdPipelineBarrier-srcAccessMask-02815**
  - The \( \text{srcAccessMask} \) member of each element of \( \text{pMemoryBarriers} \) **must** only include access flags that are supported by one or more of the pipeline stages in \( \text{srcStageMask} \), as specified in the table of supported access types.

- **VUID-vkCmdPipelineBarrier-dstAccessMask-02816**
  - The \( \text{dstAccessMask} \) member of each element of \( \text{pMemoryBarriers} \) **must** only include access flags that are supported by one or more of the pipeline stages in \( \text{dstStageMask} \), as specified in the table of supported access types.

- **VUID-vkCmdPipelineBarrier-pBufferMemoryBarriers-02817**
  - For any element of \( \text{pBufferMemoryBarriers} \), if its \( \text{srcQueueFamilyIndex} \) and \( \text{dstQueueFamilyIndex} \) members are equal, or if its \( \text{srcQueueFamilyIndex} \) is the queue family index that was used to create the command pool that \( \text{commandBuffer} \) was allocated from, then its \( \text{srcAccessMask} \) member **must** only contain access flags that are supported by one or more of the pipeline stages in \( \text{srcStageMask} \), as specified in the table of supported access types.

- **VUID-vkCmdPipelineBarrier-pBufferMemoryBarriers-02818**
  - For any element of \( \text{pBufferMemoryBarriers} \), if its \( \text{srcQueueFamilyIndex} \) and \( \text{dstQueueFamilyIndex} \) members are equal, or if its \( \text{dstQueueFamilyIndex} \) is the queue family index that was used to create the command pool that \( \text{commandBuffer} \) was allocated from, then its \( \text{dstAccessMask} \) 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 `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-vkCmdPipelineBarrier-None-07889**
  If `vkCmdPipelineBarrier` is called within a render pass instance using a `VkRenderPass` object, the render pass must have been created with at least one subpass dependency that expresses a dependency from the current subpass to itself, does not include `VK_DEPENDENCY_BY_REGION_BIT` if this command does not, does not include `VK_DEPENDENCY_VIEW_LOCAL_BIT` if this command does not, and has 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 using a `VkRenderPass` object, it must not include any buffer memory barriers.

- **VUID-vkCmdPipelineBarrier-image-04073**
  If `vkCmdPipelineBarrier` is called within a render pass instance using a `VkRenderPass` object, 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 memory barrier included in this command must be equal.

- **VUID-vkCmdPipelineBarrier-None-07890**
  If `vkCmdPipelineBarrier` is called within a render pass instance, and the source stage masks of any memory barriers include `framebuffer-space stages`, destination stage masks of all memory barriers must only include `framebuffer-space stages`.

- **VUID-vkCmdPipelineBarrier-dependencyFlags-07891**
If `vkCmdPipelineBarrier` is called within a render pass instance, and and the source stage masks of any memory barriers include framebuffer-space stages, then `dependencyFlags` must include `VK_DEPENDENCY_BY_REGION_BIT`.

- **VUID-vkCmdPipelineBarrier-None-07892**
  If `vkCmdPipelineBarrier` is called within a render pass instance, the source and destination stage masks of any memory barriers must only include graphics pipeline stages.

- **VUID-vkCmdPipelineBarrier-dependencyFlags-01186**
  If `vkCmdPipelineBarrier` is called outside of a render pass instance, the dependency flags must not include `VK_DEPENDENCY_VIEW_LOCAL_BIT`.

- **VUID-vkCmdPipelineBarrier-None-07893**
  If `vkCmdPipelineBarrier` is called inside a render pass instance, and there is more than one view in the current subpass, dependency flags must include `VK_DEPENDENCY_VIEW_LOCAL_BIT`.

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-srcStageMask-06461**
  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-vkCmdPipelineBarrier-dstStageMask-06462**
  Any pipeline stage included in `dependencyFlags` must be a valid combination of `VkDependencyFlagBits` values.

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.
**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</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:

```cpp
// 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 dependencies will be view-local.
- `VK_DEPENDENCY_DEVICE_GROUP_BIT` specifies that dependencies are non-device-local.

```cpp
// Provided by VK_VERSION_1_0
typedef VkFlags VkDependencyFlags;
```

`VkDependencyFlags` is a bitmask type for setting a mask of zero or more `VkDependencyFlagBits`. 
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 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 a VkStructureType value identifying 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 buffer range. Buffer memory barriers can also be used to define a queue family ownership transfer for the specified buffer range.

The VkBufferMemoryBarrier structure is defined as:

```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 a VkStructureType value identifying 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 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-buffer-09095**
  
  If `buffer` was created with a sharing mode of `VK_SHARING_MODE_EXCLUSIVE`, and `srcQueueFamilyIndex` and `dstQueueFamilyIndex` are not equal, `srcQueueFamilyIndex must be VK_QUEUE_FAMILY_EXTERNAL, or a valid queue family`

- **VUID-VkBufferMemoryBarrier-buffer-09096**
  
  If `buffer` was created with a sharing mode of `VK_SHARING_MODE_EXCLUSIVE`, and `srcQueueFamilyIndex` and `dstQueueFamilyIndex` are not equal, `dstQueueFamilyIndex must be VK_QUEUE_FAMILY_EXTERNAL, or a valid queue family`

- **VUID-VkBufferMemoryBarrier-srcQueueFamilyIndex-04087**
  
  If `srcQueueFamilyIndex` is not equal to `dstQueueFamilyIndex`, at least one of `srcQueueFamilyIndex` or `dstQueueFamilyIndex must not be VK_QUEUE_FAMILY_EXTERNAL`

- **VUID-VkBufferMemoryBarrier-None-09049**
  
  If `buffer` was created with a sharing mode of `VK_SHARING_MODE_CONCURRENT`, at least one of `srcQueueFamilyIndex` and `dstQueueFamilyIndex must be VK_QUEUE_FAMILY_IGNORED`

- **VUID-VkBufferMemoryBarrier-None-09050**
  
  If `buffer` was created with a sharing mode of `VK_SHARING_MODE_CONCURRENT`, `srcQueueFamilyIndex must be VK_QUEUE_FAMILY_IGNORED or VK_QUEUE_FAMILY_EXTERNAL`

- **VUID-VkBufferMemoryBarrier-None-09051**
  
  If `buffer` was created with a sharing mode of `VK_SHARING_MODE_CONCURRENT`, `dstQueueFamilyIndex must be VK_QUEUE_FAMILY_IGNORED or VK_QUEUE_FAMILY_EXTERNAL`
Valid Usage (Implicit)

- VUID-VkBufferMemoryBarrier-sType-sType
  *sType* must be *VK_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.

```c
#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 *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 a *VkStructureType* value identifying 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.

**oldLayout** and **newLayout** define an image layout transition for the specified image subresource range.

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-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

- **VUID-VkImageMemoryBarrier-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-VkImageMemoryBarrier-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-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
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_OPTIMAL then image must have been created with VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT set

• 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-image-09117
  If image was created with a sharing mode of VK_SHARING_MODE_EXCLUSIVE, and srcQueueFamilyIndex and dstQueueFamilyIndex are not equal, srcQueueFamilyIndex must be VK_QUEUE_FAMILY_EXTERNAL, or a valid queue family

• VUID-VkImageMemoryBarrier-image-09118
  If image was created with a sharing mode of VK_SHARING_MODE_EXCLUSIVE, and srcQueueFamilyIndex and dstQueueFamilyIndex are not equal, dstQueueFamilyIndex must be VK_QUEUE_FAMILY_EXTERNAL, or a valid queue family

• VUID-VkImageMemoryBarrier-srcQueueFamilyIndex-04070
  If srcQueueFamilyIndex is not equal to dstQueueFamilyIndex, at least one of srcQueueFamilyIndex or dstQueueFamilyIndex must not be VK_QUEUE_FAMILY_EXTERNAL

• VUID-VkImageMemoryBarrier-subresourceRange-01486
  subresourceRange.baseMipLevel must be less than the mipLevels specified in VkImageCreateInfo when image was created

• 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-image-09241
  If image has a color format that is single-plane, then the aspectMask member of subresourceRange must be VK_IMAGE_ASPECT_COLOR_BIT

- VUID-VkImageMemoryBarrier-image-09242
  If image has a color format and 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 at least one multi-planar aspect mask bit or VK_IMAGE_ASPECT_COLOR_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-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-aspectMask-08702
  If the aspectMask member of subresourceRange includes VK_IMAGE_ASPECT_DEPTH_BIT, oldLayout and newLayout must not be one of VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL

- VUID-VkImageMemoryBarrier-aspectMask-08703
  If the aspectMask member of subresourceRange includes VK_IMAGE_ASPECT_STENCIL_BIT, oldLayout and newLayout must not be one of VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL

- VUID-VkImageMemoryBarrier-None-09052
  If image was created with a sharing mode of VK_SHARING_MODE_CONCURRENT, at least one of srcQueueFamilyIndex and dstQueueFamilyIndex must be VK_QUEUE_FAMILY_IGNORED

- VUID-VkImageMemoryBarrier-None-09053
  If image was created with a sharing mode of VK_SHARING_MODE_CONCURRENT, srcQueueFamilyIndex must be VK_QUEUE_FAMILY_IGNORED or VK_QUEUE_FAMILY_EXTERNAL

- VUID-VkImageMemoryBarrier-None-09054
  If image was created with a sharing mode of VK_SHARING_MODE_CONCURRENT, dstQueueFamilyIndex must be VK_QUEUE_FAMILY_IGNORED or VK_QUEUE_FAMILY_EXTERNAL
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.

```c
#define VK_QUEUE_FAMILY_EXTERNAL          (~1U)
```

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 required to match on both sides - but this was subsequently relaxed. These masks should be set to 0.

If the transfer is via an image memory barrier, and an image layout transition is desired, then the values of oldLayout and newLayout in the release operation’s memory barrier must be equal to values of oldLayout and newLayout in the 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 release operation and happens-before the acquire operation.
If the values of `srcQueueFamilyIndex` and `dstQueueFamilyIndex` are equal, no ownership transfer is performed, and the barrier operates as if they were both set to `VK_QUEUE_FAMILY_IGNORED`.

Queue family ownership transfers **may** perform read and write accesses on all memory bound to the image subresource or buffer range, so applications **must** ensure that all memory writes have been made available before a queue family ownership transfer is executed. Available memory is automatically made visible to queue family release and acquire operations, and writes performed by those operations are automatically made available.

Once a queue family has acquired ownership of a buffer range or image subresource range of a `VK_SHARING_MODE_EXCLUSIVE` resource, its contents are undefined to other queue families unless ownership is transferred. The contents of any portion of another resource which aliases memory that is bound to the transferred buffer or image subresource range are undefined after a release or acquire operation.

**Note**
Because events **cannot** be used directly for inter-queue synchronization, and because `vkCmdSetEvent` does not have the queue family index or memory barrier parameters needed by a *release operation*, the release and acquire operations of a queue family ownership transfer **can** only be performed using `vkCmdPipelineBarrier`.

### 7.8. Wait Idle Operations

To wait on the host for the completion of outstanding queue operations for a given queue, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkQueueWaitIdle(  
  VkQueue queue);
```

- `queue` is the queue on which to wait.

`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 `vkWaitForFences` with an infinite timeout and `waitAll` set to `VK_TRUE`.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkQueueWaitIdle` **must** not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

#### 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>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>Any</td>
<td>-</td>
</tr>
</tbody>
</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`.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is VK_TRUE, `vkDeviceWaitIdle` must not return VK_ERROR_OUT_OF_HOST_MEMORY.

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 includes execution of vkQueueSubmit on the host and anything that happened-before it, as defined by the host memory model.

Note
Some systems allow writes that do not directly integrate with the host memory model; these have to be synchronized by the application manually. One example of this is non-temporal store instructions on x86; to ensure these happen-before submission, applications should call __mm_sfence().

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 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.

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:

```cpp
// 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.

```c
#define VK_ATTACHMENT_UNUSED (~0U)
```

### 8.1. Render Pass Creation

To create a render pass, call:

```c
// 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
If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkCreateRenderPass` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

### Valid Usage

- **VUID-vkCreateRenderPass-device-05068**
  The number of render passes currently allocated from `device` plus 1 must be less than or equal to the total number of render passes requested via `VkDeviceObjectReservationCreateInfo::renderPassRequestCount` specified when `device` was created.

- **VUID-vkCreateRenderPass-subpasses-device-05089**
  The number of subpasses currently allocated from `device` across all `VkRenderPass` objects plus `pCreateInfo->subpassCount` must be less than or equal to the total number of subpasses requested via `VkDeviceObjectReservationCreateInfo::subpassDescriptionRequestCount` specified when `device` was created.

- **VUID-vkCreateRenderPass-attachments-device-05089**
  The number of attachments currently allocated from `device` across all `VkRenderPass` objects plus `pCreateInfo->attachmentCount` must be less than or equal to the total number of attachments requested via `VkDeviceObjectReservationCreateInfo::attachmentDescriptionRequestCount` specified when `device` was created.

### 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-null**
  `pAllocator` must be NULL.

- **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:

```c
// 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 a VkStructureType value identifying 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
• 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 `loadOp` equal to `VK_ATTACHMENT_LOAD_OP_CLEAR`, the first use of that attachment must not specify a layout equal to `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`.

• VUID-VkRenderPassCreateInfo-pAttachments-01567
  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`.

• VUID-VkRenderPassCreateInfo-pNext-01929
  If the `pNext` chain includes a `VkRenderPassMultiviewCreateInfo` structure, if its `dependencyCount` member is not zero, it must be equal to `dependencyCount`.

• VUID-VkRenderPassCreateInfo-pNext-01930
  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.

• VUID-VkRenderPassCreateInfo-pNext-02512
  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

- VUID-VkRenderPassCreateInfo-pNext-02513
  If the pNext chain includes a VkRenderPassMultiviewCreateInfo structure, elements of its pViewMasks member must either all be 0, or all not be 0

- VUID-VkRenderPassCreateInfo-pNext-02514
  If the pNext chain includes a VkRenderPassMultiviewCreateInfo structure, and each element of its pViewMasks member is 0, the dependencyFlags member of each element of pDependencies must not include VK_DEPENDENCY_VIEW_LOCAL_BIT

- VUID-VkRenderPassCreateInfo-pNext-02515
  If the pNext chain includes a VkRenderPassMultiviewCreateInfo structure, and each element of its pViewMasks member is 0, its correlationMaskCount member must be 0

- VUID-VkRenderPassCreateInfo-pDependencies-00837
  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

- VUID-VkRenderPassCreateInfo-pDependencies-00838
  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

- VUID-VkRenderPassCreateInfo-pDependencies-06866
  For any element of pDependencies, if its srcSubpass is not VK_SUBPASS_EXTERNAL, it must be less than subpassCount

- VUID-VkRenderPassCreateInfo-pDependencies-06867
  For any element of pDependencies, if its dstSubpass is not VK_SUBPASS_EXTERNAL, it must be less than subpassCount

- VUID-VkRenderPassCreateInfo-subpassCount-05050
  subpassCount must be less than or equal to maxRenderPassSubpasses

- VUID-VkRenderPassCreateInfo-dependencyCount-05051
  dependencyCount must be less than or equal to maxRenderPassDependencies

- VUID-VkRenderPassCreateInfo-attachmentCount-05052
  attachmentCount must be less than or equal to maxFramebufferAttachments

Valid Usage (Implicit)

- VUID-VkRenderPassCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO

- VUID-VkRenderPassCreateInfo-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 VkRenderPassInputAttachmentAspectCreateInfo or
VkRenderPassMultiviewCreateInfo

- **VUID-VkRenderPassCreateInfo-sType-unique**
  The `sType` value of each struct in the `pNext` chain must be unique

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

- **VUID-VkRenderPassCreateInfo-pAttachments-parameter**
  If `attachmentCount` is not 0, `pAttachments` must be a valid pointer to an array of `attachmentCount` valid `VkAttachmentDescription` structures

- **VUID-VkRenderPassCreateInfo-pSubpasses-parameter**
  `pSubpasses` must be a valid pointer to an array of `subpassCount` valid `VkSubpassDescription` structures

- **VUID-VkRenderPassCreateInfo-pDependencies-parameter**
  If `dependencyCount` is not 0, `pDependencies` must be a valid pointer to an array of `dependencyCount` valid `VkSubpassDependency` structures

- **VUID-VkRenderPassCreateInfo-subpassCount-arraylength**
  `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:

```c
// Provided by VK_VERSION_1_1
typedef struct VkRenderPassMultiviewCreateInfo {
    VkStructureType sType;
} VkRenderPassMultiviewCreateInfo;
```
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 a VkStructureType value identifying 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 \texttt{pCorrelationMasks} are a set of masks of views indicating that views in the same mask \textbf{may} exhibit spatial coherency between the views, making it more efficient to render them concurrently. Correlation masks \textbf{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 \texttt{vkCmdBeginRenderPass} or \texttt{vkCmdNextSubpass} is called the graphics pipeline \textbf{must} be bound, any relevant descriptor sets or vertex/index buffers \textbf{must} be bound, and any relevant dynamic state or push constants \textbf{must} be set before they are used.

### Valid Usage

- \textbf{VUID-VkRenderPassMultiviewCreateInfo-pCorrelationMasks-00841}
  Each view index \textbf{must} not be set in more than one element of \texttt{pCorrelationMasks}

- \textbf{VUID-VkRenderPassMultiviewCreateInfo-multiview-06555}
  If the multiview feature is not enabled, each element of \texttt{pViewMasks} \textbf{must} be 0

- \textbf{VUID-VkRenderPassMultiviewCreateInfo-pViewMasks-06697}
  The index of the most significant bit in each element of \texttt{pViewMasks} \textbf{must} be less than \texttt{maxMultiviewViewCount}

### Valid Usage (Implicit)

- \textbf{VUID-VkRenderPassMultiviewCreateInfo-sType-sType}
  \texttt{sType} \textbf{must} be \texttt{VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO}

- \textbf{VUID-VkRenderPassMultiviewCreateInfo-pViewMasks-parameter}
  If \texttt{subpassCount} is not 0, \texttt{pViewMasks} \textbf{must} be a valid pointer to an array of \texttt{subpassCount uint32_t} values

- \textbf{VUID-VkRenderPassMultiviewCreateInfo-pViewOffsets-parameter}
  If \texttt{dependencyCount} is not 0, \texttt{pViewOffsets} \textbf{must} be a valid pointer to an array of \texttt{dependencyCount int32_t} values

- \textbf{VUID-VkRenderPassMultiviewCreateInfo-pCorrelationMasks-parameter}
  If \texttt{correlationMaskCount} is not 0, \texttt{pCorrelationMasks} \textbf{must} be a valid pointer to an array of \texttt{correlationMaskCount uint32_t} values

The \texttt{VkAttachmentDescription} structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkAttachmentDescription {
    VkAttachmentDescriptionFlags flags;
    VkFormat format;
    VkSampleCountFlagBits samples;
    VkAttachmentLoadOp loadOp;
} VkAttachmentDescription;
```
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** for the attachment. **storeOp** and **stencilStoreOp** define the **store operations** for the attachment. 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.

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.

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-06699
  If format includes a color or depth component 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 format is a color format, initialLayout must not be
  VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL or
  VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL.

• VUID-VkAttachmentDescription-format-06488
  If format is a color format, finalLayout must not be
  VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL or
  VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL.

• VUID-VkAttachmentDescription-separateDepthStencilLayouts-03284
  If the separateDepthStencilLayouts feature is not enabled, 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.

• VUID-VkAttachmentDescription-separateDepthStencilLayouts-03285
  If the separateDepthStencilLayouts feature is not enabled, 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.

• VUID-VkAttachmentDescription-format-03286
  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.

• VUID-VkAttachmentDescription-format-03287
  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.

• VUID-VkAttachmentDescription-format-06906
  If format is a depth/stencil format which includes both depth and stencil components, initialLayout must not be
  VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL or
  VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL.

• VUID-VkAttachmentDescription-format-06907
  If format is a depth/stencil format which includes both depth and stencil components,
**finalLayout** must not be `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`

- **VUID-VkAttachmentDescription-format-03290**
  If `format` is a depth/stencil format which includes only the depth component, **initialLayout** must not be `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`

- **VUID-VkAttachmentDescription-format-03291**
  If `format` is a depth/stencil format which includes only the depth component, **finalLayout** must not be `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`

- **VUID-VkAttachmentDescription-samples-08745**
  samples must be a bit value that is set in `imageCreateSampleCounts` (as defined in `Image Creation Limits`) for the given `format`

- **VUID-VkAttachmentDescription-format-06698**
  **format** must not be `VK_FORMAT_UNDEFINED`

- **VUID-VkAttachmentDescription-format-06700**
  If `format` includes a stencil component 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 component, **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 component, **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 components, **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 components, **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.

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 a VkStructureType value identifying 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-aspectMask-01964
  aspectMask must not include VK_IMAGE_ASPECT_METADATA_BIT

Valid Usage (Implicit)

• VUID-VkInputAttachmentAspectReference-aspectMask-parameter
  aspectMask must be a valid combination of VkImageAspectFlagBits values
• VUID-VkInputAttachmentAspectReference-aspectMask-requiredbitmask
  aspectMask must not be 0

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 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 operation** for depth and stencil are defined by `VkSubpassDescriptionDepthStencilResolve::depthResolveMode` and `VkSubpassDescriptionDepthStencilResolve::stencilResolveMode`, respectively. If `VkSubpassDescriptionDepthStencilResolve::depthResolveMode` is `VK_RESOLVE_MODE_NONE` or the `pDepthStencilResolveAttachment` does not have a depth aspect, no resolve operation is performed for the depth attachment. If `VkSubpassDescriptionDepthStencilResolve::stencilResolveMode` is `VK_RESOLVE_MODE_NONE` or the `pDepthStencilResolveAttachment` does not have a stencil aspect, no resolve operation is performed for the stencil attachment.

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 $S_1$ 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_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_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_READ_ONLY_OPTIMAL` or `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_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`, `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_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL,
VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL,
VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL

- VUID-VkSubpassDescription-pipelineBindPoint-04952
  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-09430
  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 `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`.

- VUID-VkSubpassDescription-pDepthStencilAttachment-01418
  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.

- VUID-VkSubpassDescription-inputAttachmentCount-05053
  `inputAttachmentCount` must be less than or equal to `maxSubpassInputAttachments`.

- VUID-VkSubpassDescription-pColorAttachments-parameter
  If `pColorAttachments` is not `NULL`, `pColorAttachments` must be a valid pointer to an array of `colorAttachmentCount` valid `VkAttachmentReference` structures.

- VUID-VkSubpassDescription-pResolveAttachments-parameter
  If `colorAttachmentCount` is not 0, any `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
  Each element of `pPreserveAttachments` must not be `VK_ATTACHMENT_UNUSED`.

---

**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
  Each element of `pPreserveAttachments` must not be `VK_ATTACHMENT_UNUSED`.
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:

```c
// Provided by VK_VERSION_1_0
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.

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

`VkSubpassDescriptionFlags` is a bitmask type for setting a mask of zero or more `VkSubpassDescriptionFlagBits`.

The `VkAttachmentReference` structure is defined as:

```c
// Provided by VK_VERSION_1_0
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`.  

---

236
**Valid Usage (Implicit)**

- **VUID-VkAttachmentReference-layout-parameter**
  - *layout* must be a valid *VkImageLayout* value

---

**VK_SUBPASS_EXTERNAL** is a special subpass index value expanding synchronization scope outside a subpass. It is described in more detail by *VkSubpassDependency*.

```c
#define VK_SUBPASS_EXTERNAL (~0U)
```

The **VkSubpassDependency** structure is defined as:

```c
// 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;
```

- **srcSubpass** is the subpass index of the first subpass in the dependency, or **VK_SUBPASS_EXTERNAL**.
- **dstSubpass** is the subpass index of the second subpass in the dependency, or **VK_SUBPASS_EXTERNAL**.
- **srcStageMask** is a bitmask of **VkPipelineStageFlagBits** specifying the source stage mask.
- **dstStageMask** is a bitmask of **VkPipelineStageFlagBits** specifying the destination stage mask.
- **srcAccessMask** is a bitmask of **VkAccessFlagBits** specifying a source access mask.
- **dstAccessMask** is a bitmask of **VkAccessFlagBits** specifying a destination access mask.
- **dependencyFlags** is a bitmask of **VkDependencyFlagBits**.

If **srcSubpass** is equal to **dstSubpass** then the **VkSubpassDependency** does not directly define a dependency. Instead, it enables pipeline barriers to be used in a render pass instance within the identified subpass, where the scopes of one pipeline barrier must be a subset of those described by one subpass dependency. Subpass dependencies specified in this way that include framebuffer-space stages in the **srcStageMask** must only include framebuffer-space stages in **dstStageMask**, and must include **VK_DEPENDENCY_BY_REGION_BIT**. When a subpass dependency is specified in this way for a subpass that has more than one view in its view mask, its **dependencyFlags** must include **VK_DEPENDENCY_VIEW_LOCAL_BIT**.

If **srcSubpass** and **dstSubpass** are not equal, when a render pass instance which includes a subpass dependency is submitted to a queue, it defines a dependency between the subpasses identified by **srcSubpass** and **dstSubpass**.
If `srcSubpass` is equal to `VK_SUBPASS_EXTERNAL`, the first synchronization scope includes commands that occur earlier in submission order than the `vkCmdBeginRenderPass` used to begin the render pass instance. Otherwise, the first set of commands includes all commands submitted as part of the subpass instance identified by `srcSubpass` and any load, store, or multisample resolve operations on attachments used in `srcSubpass`. In either case, the first synchronization scope is limited to operations on the pipeline stages determined by the source stage mask specified by `srcStageMask`.

If `dstSubpass` is equal to `VK_SUBPASS_EXTERNAL`, the second synchronization scope includes commands that occur later in submission order than the `vkCmdEndRenderPass` used to end the render pass instance. Otherwise, the second set of commands includes all commands submitted as part of the subpass instance identified by `dstSubpass` and any load, store, and multisample resolve operations on attachments used in `dstSubpass`. In either case, the second synchronization scope is limited to operations on the pipeline stages determined by the destination stage mask specified by `dstStageMask`.

The first access scope is limited to 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`
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

srcStageMask must not be Ø

If the geometryShader feature is not enabled, dstStageMask must not contain
VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT

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

dstStageMask must not be Ø

srcSubpass must be less than or equal to dstSubpass, unless one of them is
VK_SUBPASS_EXTERNAL, to avoid cyclic dependencies and ensure a valid execution order

srcSubpass and dstSubpass must not both be equal to VK_SUBPASS_EXTERNAL

If srcSubpass is equal to dstSubpass and srcStageMask includes a framebuffer-space stage,
dstStageMask must only contain framebuffer-space stages

Any access flag included in srcAccessMask must be supported by one of the pipeline stages in
srcStageMask, as specified in the table of supported access types

Any access flag included in dstAccessMask must be supported by one of the pipeline stages in
dstStageMask, as specified in the table of supported access types

If srcSubpass equals dstSubpass, and srcStageMask and dstStageMask both include a
framebuffer-space stage, then dependencyFlags must include VK_DEPENDENCY_BY_REGION_BIT

If dependencyFlags includes VK_DEPENDENCY_VIEW_LOCAL_BIT, srcSubpass must not be equal to
VK_SUBPASS_EXTERNAL

If dependencyFlags includes VK_DEPENDENCY_VIEW_LOCAL_BIT, dstSubpass must not be equal to
VK_SUBPASS_EXTERNAL

If srcSubpass equals dstSubpass and that subpass has more than one bit set in the view
mask, then dependencyFlags must include VK_DEPENDENCY_VIEW_LOCAL_BIT
Valid Usage (Implicit)

- **VUID-VkSubpassDependency-srcStageMask-parameter**
  
  *srcStageMask must* be a valid combination of *VkPipelineStageFlagBits* values

- **VUID-VkSubpassDependency-dstStageMask-parameter**
  
  *dstStageMask must* be a valid combination of *VkPipelineStageFlagBits* values

- **VUID-VkSubpassDependency-srcAccessMask-parameter**
  
  *srcAccessMask must* be a valid combination of *VkAccessFlagBits* values

- **VUID-VkSubpassDependency-dstAccessMask-parameter**
  
  *dstAccessMask must* be a valid combination of *VkAccessFlagBits* values

- **VUID-VkSubpassDependency-dependencyFlags-parameter**
  
  *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:

```c
VkSubpassDependency implicitDependency = {
```
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_BOTTOM_OF_PIPE_BIT,
    .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 \texttt{srcSubpass}.

Automatic layout transitions into the layout used in a subpass happen-before the visibility operations for all dependencies with that subpass as the \texttt{dstSubpass}.

Automatic layout transitions away from initialLayout happen-after the availability operations for all dependencies with a \texttt{srcSubpass} equal to \texttt{VK_SUBPASS_EXTERNAL}, where \texttt{dstSubpass} uses the attachment that will be transitioned. For attachments created with \texttt{VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT}, automatic layout transitions away from initialLayout happen-after the availability operations for all dependencies with a \texttt{srcSubpass} equal to \texttt{VK_SUBPASS_EXTERNAL}, where \texttt{dstSubpass} uses any aliased attachment.

Automatic layout transitions into finalLayout happen-before the visibility operations for all dependencies with a \texttt{dstSubpass} equal to \texttt{VK_SUBPASS_EXTERNAL}, where \texttt{srcSubpass} uses the attachment that will be transitioned. For attachments created with \texttt{VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT}, automatic layout transitions into finalLayout happen-before the visibility operations for all dependencies with a \texttt{dstSubpass} equal to \texttt{VK_SUBPASS_EXTERNAL}, where \texttt{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 \textbf{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:

```cpp
// 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 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 substructures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkCreateRenderPass2` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

**Valid Usage**

- **VUID-vkCreateRenderPass2-device-05068**
  The number of render passes currently allocated from `device` plus 1 must be less than or equal to the total number of render passes requested via `VkDeviceObjectReservationCreateInfo::renderPassRequestCount` specified when `device` was created.

- **VUID-vkCreateRenderPass2-subpasses-device-05089**
  The number of subpasses currently allocated from `device` across all `VkRenderPass` objects plus `pCreateInfo->subpassCount` must be less than or equal to the total number of subpasses requested via `VkDeviceObjectReservationCreateInfo::subpassDescriptionRequestCount` specified when `device` was created.

- **VUID-vkCreateRenderPass2-attachments-device-05089**
  The number of attachments currently allocated from `device` across all `VkRenderPass` objects plus `pCreateInfo->attachmentCount` must be less than or equal to the total number of attachments requested via `VkDeviceObjectReservationCreateInfo::attachmentDescriptionRequestCount` specified when `device` was created.

**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-null**
  
  `pAllocator` must be `NULL`

- **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 a `VkStructureType` value identifying 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 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 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,
VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_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,
  VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL

• VUID-VkRenderPassCreateInfo2-pDependencies-03054
  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

• VUID-VkRenderPassCreateInfo2-pDependencies-03055
  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

• VUID-VkRenderPassCreateInfo2-pCorrelatedViewMasks-03056
  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

• VUID-VkRenderPassCreateInfo2-viewMask-03057
  If the VkSubpassDescription2::viewMask member of all elements of pSubpasses is 0,
correlatedViewMaskCount must be 0

• VUID-VkRenderPassCreateInfo2-viewMask-03058
  The VkSubpassDescription2::viewMask member of all elements of pSubpasses must either
  all be 0, or all not be 0

• VUID-VkRenderPassCreateInfo2-viewMask-03059
  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

• VUID-VkRenderPassCreateInfo2-pDependencies-03060
  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

• VUID-VkRenderPassCreateInfo2-attachment-02525
  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

• VUID-VkRenderPassCreateInfo2-srcSubpass-02526
  The srcSubpass member of each element of pDependencies must be less than subpassCount
• **VUID-VkRenderPassCreateInfo2-dstSubpass-02527**
  The `dstSubpass` member of each element of `pDependencies` must be less than `subpassCount`

• **VUID-VkRenderPassCreateInfo2-subpassCount-05055**
  `subpassCount` must be less than or equal to `maxRenderPassSubpasses`

• **VUID-VkRenderPassCreateInfo2-dependencyCount-05056**
  `dependencyCount` must be less than or equal to `maxRenderPassDependencies`

• **VUID-VkRenderPassCreateInfo2-attachmentCount-05057**
  `attachmentCount` must be less than or equal to `maxFramebufferAttachments`

• **VUID-VkRenderPassCreateInfo2-attachment-06244**
  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 components

• **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 component

• **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 component

### 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
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 a **VkStructureType** value identifying 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 **format** is a depth/stencil format, and either **initialLayout** or **finalLayout** does not specify a layout for the stencil aspect, then the application **must** specify the initial and final layouts of the stencil aspect by including a **VkAttachmentDescriptionStencilLayout** structure in the **pNext** chain.

### Valid Usage

- **VUID-VkAttachmentDescription2-format-06699**
  
  If **format** includes a color or depth component and **loadOp** is **VK_ATTACHMENT_LOAD_OP_LOAD**, then **initialLayout** **must** not be **VK_IMAGE_LAYOUT_UNDEFINED**

- **VUID-VkAttachmentDescription2-finalLayout-00843**
  
  **finalLayout** **must** not be **VK_IMAGE_LAYOUT_UNDEFINED** or **VK_IMAGE_LAYOUT_PREINITIALIZED**

- **VUID-VkAttachmentDescription2-format-03280**
  
  If **format** is a color format, **initialLayout** **must** not be
  
  - **VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL**
  - **VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL**

- **VUID-VkAttachmentDescription2-format-03281**
  
  If **format** is a depth/stencil format, **initialLayout** **must** not be **VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL**

- **VUID-VkAttachmentDescription2-format-03282**
  
  If **format** is a color format, **finalLayout** **must** not be
  
  - **VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL**
  - **VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL**

- **VUID-VkAttachmentDescription2-format-03283**
  
  If **format** is a depth/stencil format, **finalLayout** **must** not be **VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL**

- **VUID-VkAttachmentDescription2-format-06487**
  
  If **format** is a color format, **initialLayout** **must** not be
  
  - **VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL**
  - **VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_ATTACHMENT_OPTIMAL**

249
If format is a color format, finalLayout must not be
VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL or
VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL

If the separateDepthStencilLayouts feature is not enabled, 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, 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 depth/stencil format which includes both depth and stencil components,
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 components,
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 component,
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 component, finalLayout must not be
VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL or
VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL

samples must be a bit value that is set in imageCreateSampleCounts (as defined in Image
Creation Limits) for the given format
If the `pNext` chain does not include a `VkAttachmentDescriptionStencilLayout` structure, `format` includes a stencil component, 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 component, 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 components, and `initialLayout` is `VK_IMAGE_LAYER_2D` or `VK_IMAGE_LAYER_2D_FORMAT_ATTACHMENT_OPTIMAL`, or `VK_IMAGE_LAYER_2D_FORMAT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`, the `pNext` chain must include a `VkAttachmentDescriptionStencilLayout` structure.

If `format` is a depth/stencil format which includes both depth and stencil components, and `finalLayout` is `VK_IMAGE_LAYER_2D` or `VK_IMAGE_LAYER_2D_FORMAT_ATTACHMENT_OPTIMAL`, or `VK_IMAGE_LAYER_2D_FORMAT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`, the `pNext` chain must include a `VkAttachmentDescriptionStencilLayout` structure.

If the `pNext` chain does not include a `VkAttachmentDescriptionStencilLayout` structure and `format` only includes a stencil component, `initialLayout` must not be `VK_IMAGE_LAYER_2D` or `VK_IMAGE_LAYER_2D_FORMAT_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYER_2D_FORMAT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`.

If the `pNext` chain does not include a `VkAttachmentDescriptionStencilLayout` structure and `format` only includes a stencil component, `finalLayout` must not be `VK_IMAGE_LAYER_2D` or `VK_IMAGE_LAYER_2D_FORMAT_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYER_2D_FORMAT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`.

`format` must not be `VK_FORMAT_UNDEFINED`.

Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_2`.
- `pNext` must be `NULL` or a pointer to a valid instance of `VkAttachmentDescriptionStencilLayout`.
- The `sType` value of each struct in the `pNext` chain must be unique.
- `flags` must be a valid combination of `VkAttachmentDescriptionFlagBits` values.
- `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:

```c
// Provided by VK_VERSION_1_2
typedef struct VkAttachmentDescriptionStencilLayout {
    VkStructureType sType;
    void* pNext;
    VkImageLayout stencilInitialLayout;
    VkImageLayout stencilFinalLayout;
} VkAttachmentDescriptionStencilLayout;
```

• *sType* is a *VkStructureType* value identifying 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_READ_ONLY_OPTIMAL*, *VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL*, *VK_IMAGE_LAYOUT_DEPTH_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_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, 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.

- VUID-VkAttachmentDescriptionStencilLayout-stencilFinalLayout-parameter

stencilFinalLayout 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 a VkStructureType value identifying 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**, its **layout** member **must** not be **VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL** or **VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL**
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

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

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_OPTIMAL or
VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL

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_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 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 VkAttachmentReferenceStencilLayout 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
• **VUID-VkSubpassDescription2-pResolveAttachments-03066**
  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-VkSubpassDescription2-pResolveAttachments-03068**
  Each element of `pResolveAttachments` **must** have the same `VkFormat` as its corresponding color attachment.

• **VUID-VkSubpassDescription2-pResolveAttachments-03067**
  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-VkSubpassDescription2-pInputAttachments-02897**
  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-VkSubpassDescription2-pColorAttachments-02898**
  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**
  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**
  Each 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-inputAttachmentCount-05058**
  `inputAttachmentCount` must be less than or equal to `maxSubpassInputAttachments`

- **VUID-VkSubpassDescription2-preserveAttachmentCount-05059**
  `preserveAttachmentCount` must be less than or equal to `maxSubpassPreserveAttachments`

- **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 `inputAttachmentCount` valid `VkAttachmentReference2` structures

- **VUID-VkSubpassDescription2-pColorAttachments-parameter**
  If `colorAttachmentCount` is not 0, `pColorAttachments` must be a valid pointer to an array of `colorAttachmentCount` valid `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
If `preserveAttachmentCount` is not 0, `pPreserveAttachments` must be a valid pointer to an array of `preserveAttachmentCount` `uint32_t` values.

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 a `VkStructureType` value identifying 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 the `pNext` chain of `VkSubpassDescription2` includes a `VkSubpassDescriptionDepthStencilResolve` structure, then that structure describes multisample resolve operations for the depth/stencil attachment in a subpass. If this structure is not included in the `pNext` chain of `VkSubpassDescription2`, or if it is and either `pDepthStencilResolveAttachment` is NULL or 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 numeric format.

- **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 numeric format.

- **VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-03183**
  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-pDepthStencilResolveAttachment-03184**
  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`.

- **VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-03185**
  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.
Valid Usage (Implicit)

- VUID-VkSubpassDescriptionDepthStencilResolve-sType-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

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 a VkStructureType value identifying 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 pNext chain. Otherwise, layout describes the layout for all relevant image aspects.
Valid Usage

- **VUID-VkAttachmentReference2-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-VkAttachmentReference2-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-VkAttachmentReference2-sType-sType**
  sType must be VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_2

- **VUID-VkAttachmentReference2-pNext-pNext**
  pNext must be NULL or a pointer to a valid instance of VkAttachmentReferenceStencilLayout

- **VUID-VkAttachmentReference2-sType-unique**
  The sType value of each struct in the pNext chain must be unique

- **VUID-VkAttachmentReference2-layout-parameter**
  layout must be a valid VkImageLayout value

The VkAttachmentReferenceStencilLayout structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkAttachmentReferenceStencilLayout {
    VkStructureType sType;
    void* pNext;
    VkImageLayout stencilLayout;
} VkAttachmentReferenceStencilLayout;
```

- **sType** is a VkStructureType value identifying this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **stencilLayout** is a 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,
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
typedef struct VK_VERSION_1_2
    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 a VkStructureType value identifying 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.

### 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-04996**
  `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-04996**
  `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-srcAccessMask-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.

- **VUID-VkSubpassDependency2-dstAccessMask-03089**
  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.

- **VUID-VkSubpassDependency2-dependencyFlags-03090**
  If `dependencyFlags` includes `VK_DEPENDENCY_VIEW_LOCAL_BIT`, `srcSubpass` must not be equal to `VK_SUBPASS_EXTERNAL`. 
• **VUID-VkSubpassDependency2-dependencyFlags-03091**
  If `dependencyFlags` includes `VK_DEPENDENCY_VIEW_LOCAL_BIT`, `dstSubpass` must not be equal to `VK_SUBPASS_EXTERNAL`

• **VUID-VkSubpassDependency2-srcSubpass-02245**
  If `srcSubpass` equals `dstSubpass`, and `srcStageMask` and `dstStageMask` both include a framebuffer-space stage, then `dependencyFlags` must include `VK_DEPENDENCY_BY_REGION_BIT`

• **VUID-VkSubpassDependency2-viewOffset-02530**
  If `viewOffset` is not equal to 0, `srcSubpass` must not be equal to `dstSubpass`

• **VUID-VkSubpassDependency2-dependencyFlags-03092**
  If `dependencyFlags` does not include `VK_DEPENDENCY_VIEW_LOCAL_BIT`, `viewOffset` must be 0

### Valid Usage (Implicit)

• **VUID-VkSubpassDependency2-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_SUBPASS_DEPENDENCY_2`

• **VUID-VkSubpassDependency2-pNext-pNext**
  `pNext` must be `NULL`

• **VUID-VkSubpassDependency2-srcStageMask-parameter**
  `srcStageMask` must be a valid combination of `VkPipelineStageFlagBits` values

• **VUID-VkSubpassDependency2-dstStageMask-parameter**
  `dstStageMask` must be a valid combination of `VkPipelineStageFlagBits` values

• **VUID-VkSubpassDependency2-srcAccessMask-parameter**
  `srcAccessMask` must be a valid combination of `VkAccessFlagBits` values

• **VUID-VkSubpassDependency2-dstAccessMask-parameter**
  `dstAccessMask` must be a valid combination of `VkAccessFlagBits` values

• **VUID-VkSubpassDependency2-dependencyFlags-parameter**
  `dependencyFlags` must be a valid combination of `VkDependencyFlagBits` values

To destroy a render pass, call:

```c
// Provided by VK_VERSION_1_0
define 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

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-null
  pAllocator must be NULL

- 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.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkCreateFramebuffer` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

---

**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`

- VUID-vkCreateFramebuffer-device-05068
  The number of framebuffers currently allocated from `device` plus 1 must be less than or equal to the total number of framebuffers requested via `VkDeviceObjectReservationCreateInfo::framebufferRequestCount` specified when `device` was created
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-null
  pAllocator must be NULL

- 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 a VkStructureType value identifying 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.

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. In all such cases, rasterizationSamples must be a bit value that is set in
VkPhysicalDeviceLimits::framebufferNoAttachmentsSampleCounts.

---

**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
If \textit{renderPass} was specified with non-zero view masks, \textit{layers} \textbf{must} be 1

If \textit{flags} does not include \texttt{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT}, each element of \textit{pAttachments} that is a 2D or 2D array image view taken from a 3D image \textbf{must} not be a depth/stencil format

If the \texttt{imagelessFramebuffer} feature is not enabled, \textit{flags} \textbf{must} not include \texttt{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT}

If \textit{flags} includes \texttt{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT}, the \textit{pNext} chain \textbf{must} include a \texttt{VkFramebufferAttachmentsCreateInfo} structure

If \textit{flags} includes \texttt{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT}, the \texttt{attachmentImageInfoCount} member of a \texttt{VkFramebufferAttachmentsCreateInfo} structure in the \textit{pNext} chain \textbf{must} be equal to either zero or \texttt{attachmentCount}

If \textit{flags} includes \texttt{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT}, the \texttt{width} member of any element of the \texttt{pAttachmentImageInfos} member of a \texttt{VkFramebufferAttachmentsCreateInfo} structure in the \textit{pNext} chain that is used as an input, color, resolve or depth/stencil attachment in \textit{renderPass} \textbf{must} be greater than or equal to \texttt{width}

If \textit{flags} includes \texttt{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT}, the \texttt{height} member of any element of the \texttt{pAttachmentImageInfos} member of a \texttt{VkFramebufferAttachmentsCreateInfo} structure in the \textit{pNext} chain that is used as an input, color, resolve or depth/stencil attachment in \textit{renderPass} \textbf{must} be greater than or equal to \texttt{height}

If multiview is enabled for \textit{renderPass} and \textit{flags} includes \texttt{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT}, the \texttt{layerCount} member of any element of the \texttt{pAttachmentImageInfos} member of a \texttt{VkFramebufferAttachmentsCreateInfo} structure included in the \textit{pNext} chain used as an input, color, resolve, or depth/stencil attachment in \textit{renderPass} \textbf{must} be greater than the maximum bit index set in the view mask in the subpasses in which it is used in \textit{renderPass}

If multiview is not enabled for \textit{renderPass} and \textit{flags} includes \texttt{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT}, the \texttt{layerCount} member of any element of the \texttt{pAttachmentImageInfos} member of a \texttt{VkFramebufferAttachmentsCreateInfo} structure included in the \textit{pNext} chain used as an input, color, resolve, or depth/stencil attachment in \textit{renderPass} \textbf{must} be greater than or equal to \texttt{layers}

If \textit{flags} includes \texttt{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT}, the \texttt{usage} member of any element of the \texttt{pAttachmentImageInfos} member of a \texttt{VkFramebufferAttachmentsCreateInfo} structure included in the \textit{pNext} chain that refers to an attachment used as a color attachment or resolve attachment by \textit{renderPass} \textbf{must} include
VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT

- VUID-VkFramebufferCreateInfo-flags-03202
  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

- VUID-VkFramebufferCreateInfo-flags-03204
  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

- VUID-VkFramebufferCreateInfo-flags-03205
  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

- VUID-VkFramebufferCreateInfo-flags-04113
  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

- VUID-VkFramebufferCreateInfo-attachmentCount-05060
  attachmentCount must be less than or equal to maxFramebufferAttachments

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:
typedef struct VkFramebufferAttachmentsCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t attachmentImageInfoCount;
    const VkFramebufferAttachmentImageInfo* pAttachmentImageInfos;
} VkFramebufferAttachmentsCreateInfo;

- **sType** is a *VkStructureType* value identifying 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-pAttachmentImageInfos-parameter
  
  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:

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 a *VkStructureType* value identifying 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-required bitmask**
  - **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
typedef VkFlags VkFramebufferCreateFlags;
```

`VkFramebufferCreateFlags` is a bitmask type for setting a mask of zero or more `VkFramebufferCreateFlagBits`.

To destroy a framebuffer, call:

```c
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

### Valid Usage (Implicit)

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

- `VUID-vkDestroyFramebuffer-framebuffer-parameter`
  If `framebuffer` is not `VK_NULL_HANDLE`, `framebuffer` must be a valid `VkFramebuffer` handle

- `VUID-vkDestroyFramebuffer-pAllocator-null`
  `pAllocator` must be `NULL`

- `VUID-vkDestroyFramebuffer-framebuffer-parent`
  If `framebuffer` is a valid handle, it must have been created, allocated, or retrieved from `device`

### Host Synchronization

- Host access to `framebuffer` must be externally synchronized
8.4. Render Pass Load Operations

Render pass load operations define the initial values of an attachment during a render pass instance.

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 load operation for each sample in an attachment happens-before any recorded command which accesses the sample in that render pass instance via that attachment or an alias.

**Note**
Because load operations always happen first, external synchronization with attachment access only needs to synchronize the load operations with previous commands; not the operations within the render pass instance.

Load operations only update values within the defined render area for the render pass instance. However, any writes performed by a load operation (as defined by its access masks) to a given attachment may read and write back any memory locations within the image subresource bound for that attachment. For depth/stencil images, writes to one aspect may also result in read-modify-write operations for the other aspect.

**Note**
As entire subresources could be accessed by load operations, applications cannot safely access values outside of the render area during a render pass instance when a load operation that modifies values is used.

Load operations that can be used for a render pass 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 as the initial values. 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`.

During a render pass instance, input and 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*.

### 8.5. Render Pass Store Operations

Render pass store operations define how values written to an attachment during a render pass instance are stored to memory.

Store operations for attachments with a depth/stencil format execute in the `VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT` pipeline stage. Store operations for attachments with a color format execute in the `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` pipeline stage. The store operation for each sample in an attachment happens-after any recorded command which accesses the sample via that attachment or an alias.

**Note**

Because store operations always happen after other accesses in a render pass instance, external synchronization with attachment access in an earlier render pass only needs to synchronize with the store operations; not the operations within the render pass instance.

Store operations only update values within the defined render area for the render pass instance. However, any writes performed by a store operation (as defined by its access masks) to a given attachment **may** read and write back any memory locations within the image subresource bound for that attachment. For depth/stencil images writes to one aspect **may** also result in read-modify-write operations for the other aspect.

**Note**

As entire subresources could be accessed by store operations, applications cannot safely access values outside of the render area via aliased resources during a render pass instance when a store operation that modifies values is used.

Possible values of `VkAttachmentDescription::storeOp` and `stencilStoreOp`, specifying how the contents of the attachment are treated, are:

```c
// Provided by VK_VERSION_1_0
```
```c
typedef enum VkAttachmentStoreOp {
    VK_ATTACHMENT_STORE_OP_STORE = 0,
    VK_ATTACHMENT_STORE_OP_DONT_CARE = 1,
} 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`.

**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.

### 8.6. Render Pass Multisample Resolve Operations

Render pass multisample resolve operations combine sample values from a single pixel in a multisample attachment and store the result to the corresponding pixel in a single sample attachment.

Multisample resolve operations for attachments execute in the `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` pipeline stage. A final resolve operation for all pixels in the render area happens-after any recorded command which writes a pixel via the multisample attachment to be resolved or an explicit alias of it in the subpass that it is specified. Any single sample attachment specified for use in a multisample resolve operation may have its contents modified at any point once rendering begins for the render pass instance. Reads from the multisample attachment can be synchronized with `VK_ACCESS_COLOR_ATTACHMENT_READ_BIT`. Access to the single sample attachment can be synchronized with `VK_ACCESS_COLOR_ATTACHMENT_READ_BIT` and `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`. These pipeline stage and access types are used whether the attachments are color or depth/stencil attachments.

When using render pass objects, a subpass dependency specified with the above pipeline stages and access flags will ensure synchronization with multisample resolve operations for any attachments that were last accessed by that subpass. This allows later subpasses to read resolved values as input attachments.

Resolve operations only update values within the defined render area for the render pass instance. However, any writes performed by a resolve operation (as defined by its access masks) to a given attachment may read and write back any memory locations within the image subresource bound for that attachment. For depth/stencil images writes to one aspect may also result in read-modify-write operations for the other aspect.
As entire subresources could be accessed by multisample resolve operations, applications cannot safely access values outside of the render area via aliased resources during a render pass instance when a multisample resolve operation is performed.

Multisample values in a multisample attachment are combined according to the resolve mode used:

```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.

If no resolve mode is otherwise specified, **VK_RESOLVE_MODE_AVERAGE_BIT** is used.

```c
// Provided by VK_VERSION_1_2
typedef VkFlags VkResolveModeFlags;
```

VkResolveModeFlags is a bitmask type for setting a mask of zero or more VkResolveModeFlagBits.

### 8.7. 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:

```c
// Provided by VK_VERSION_1_0
void vkCmdBeginRenderPass(
```
VkCommandBuffer
const VkRenderPassBeginInfo* pRenderPassBegin,
VkSubpassContents contents);

- `commandBuffer` is the command buffer in which to record the command.
- `pRenderPassBegin` is a pointer to a `VkRenderPassBeginInfo` structure specifying the render pass to begin an instance of, and the framebuffer the instance uses.
- `contents` is a `VkSubpassContents` value specifying how the commands in the first subpass will be provided.

After beginning a render pass instance, the command buffer is ready to record the commands for the first subpass of that render pass.

**Valid Usage**

- **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`, or `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 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

- **VUID-vkCmdBeginRenderPass-framebuffer-09045**
  If any attachments specified in `framebuffer` are used by `renderPass` and are bound to overlapping memory locations, there **must** be only one that is used as a color attachment, depth/stencil, or resolve attachment in any subpass

---

**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
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.
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, or 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-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-03098
  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-03099
  If any of the initialLayout or finalLayout member of the VkAttachmentDescription structures or the layout member of the VkAttachmentReference structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL then the corresponding attachment image view of the framebuffer specified in the framebuffer member of pRenderPassBegin must have been created with a usage value including VK_IMAGE_USAGE_TRANSFER_DST_BIT.
If 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`.

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.

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.

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.

If any attachments specified in `framebuffer` are used by `renderPass` and are bound to overlapping memory locations, there must be only one that is used as a color attachment, depth/stencil, or resolve attachment in any subpass.

**Valid Usage (Implicit)**

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `pRenderPassBegin` must be a valid pointer to a valid `VkRenderPassBeginInfo` structure
- `pSubpassBeginInfo` must be a valid pointer to a valid `VkSubpassBeginInfo` structure
- `commandBuffer` must be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations
- This command must only be called outside of a render pass instance
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></td>
<td></td>
<td></td>
<td>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 a `VkStructureType` value identifying 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_OP_CLEAR`. The array is indexed by attachment.
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.

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-None-08996
If the pNext chain does not contain VkDeviceGroupRenderPassBeginInfo or its deviceRenderAreaCount member is equal to 0, renderArea.extent.width must be greater than 0

- VUID-VkRenderPassBeginInfo-None-08997
If the pNext chain does not contain VkDeviceGroupRenderPassBeginInfo or its deviceRenderAreaCount member is equal to 0, renderArea.extent.height must be greater than 0

- 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

- VUID-VkRenderPassBeginInfo-framebuffer-02780
  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

- VUID-VkRenderPassBeginInfo-framebuffer-03209
  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

- VUID-VkRenderPassBeginInfo-framebuffer-04627
  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
• VUID-VkRenderPassBeginInfo-framebuffer-03211
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

• VUID-VkRenderPassBeginInfo-framebuffer-03212
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

• VUID-VkRenderPassBeginInfo-framebuffer-03213
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

• VUID-VkRenderPassBeginInfo-framebuffer-03214
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

• 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-09047
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:

```c
// Provided by VK_VERSION_1_2
typedef struct VK_VERSION_1_2
VkSubpassBeginInfo {
    VkStructureType sType;
    const void* pNext;
    VkSubpassContents contents;
} VkSubpassBeginInfo;
```

- sType is a VkStructureType value identifying 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
  ...
**VkSubpassContents** value

Possible values of **vkCmdBeginRenderPass::contents**, specifying how the commands in the first subpass will be provided, are:

```plaintext
// 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 in the command buffer until **vkCmdNextSubpass** or **vkCmdEndRenderPass**.

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

The **VkDeviceGroupRenderPassBeginInfo** structure is defined as:

```plaintext
// 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 a **VkStructureType** value identifying 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 **load**, **store**, and **multisample 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 to 0`
- **VUID-VkDeviceGroupRenderPassBeginInfo-offset-06167**
  The `offset.y` member of any element of `pDeviceRenderAreas must be greater than or equal to 0`
- **VUID-VkDeviceGroupRenderPassBeginInfo-offset-06168**
  The sum of the `offset.x` and `extent.width` members of any element of `pDeviceRenderAreas must be less than or equal to maxFramebufferWidth`
- **VUID-VkDeviceGroupRenderPassBeginInfo-offset-06169**
  The sum of the `offset.y` and `extent.height` members of any element of `pDeviceRenderAreas must be less than or equal to maxFramebufferHeight`
- **VUID-VkDeviceGroupRenderPassBeginInfo-extent-08998**
  The `extent.width` member of any element of `pDeviceRenderAreas must be greater than 0`
- **VUID-VkDeviceGroupRenderPassBeginInfo-extent-08999**
  The `extent.height` member of any element of `pDeviceRenderAreas must be greater than 0`

### Valid Usage (Implicit)

- **VUID-VkDeviceGroupRenderPassBeginInfo-sType-sType**
**sType** must be `VK_STRUCTURE_TYPE_DEVICE_GROUP_RENDER_PASS_BEGIN_INFO`

- VUID-VkDeviceGroupRenderPassBeginInfo-pDeviceRenderAreas-parameter
  If `deviceRenderAreaCount` is not 0, `pDeviceRenderAreas` must be a valid pointer to an array of `deviceRenderAreaCount` `VkRect2D` structures

The `VkRenderPassAttachmentBeginInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkRenderPassAttachmentBeginInfo {
    VkStructureType       sType;
    const void*            pNext;
    uint32_t                attachmentCount;
    const VkImageView*     pAttachments;
} VkRenderPassAttachmentBeginInfo;
```

- **sType** is a `VkStructureType` value identifying 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**

- VUID-VkRenderPassAttachmentBeginInfo-pAttachments-03218
  Each element of `pAttachments` must only specify a single mip level
- VUID-VkRenderPassAttachmentBeginInfo-pAttachments-03219
  Each element of `pAttachments` must have been created with the identity swizzle
- VUID-VkRenderPassAttachmentBeginInfo-pAttachments-04114
  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
```
```c
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
- VUID-vkGetRenderAreaGranularity-renderPass-parameter
  - `renderPass` must be a valid `VkRenderPass` handle
- VUID-vkGetRenderAreaGranularity-pGranularity-parameter
  - `pGranularity` must be a valid pointer to a `VkExtent2D` structure
- VUID-vkGetRenderAreaGranularity-renderPass-parent
  - `renderPass` must have been created, allocated, or retrieved from `device`

To transition to the next subpass in the render pass instance after recording the commands for a subpass, call:

```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.

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
- VUID-vkCmdNextSubpass-contents-parameter  
  *contents* must be a valid *VkSubpassContents* value
- VUID-vkCmdNextSubpass-commandBuffer-recording  
  *commandBuffer* must be in the recording state
- VUID-vkCmdNextSubpass-commandBuffer-cmdpool  
  The *VkCommandPool* that *commandBuffer* was allocated from must support graphics operations
- VUID-vkCmdNextSubpass-renderpass  
  This command must only be called inside of a render pass instance
- VUID-vkCmdNextSubpass-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
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</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:

```
// 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-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 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_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-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
  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:
typedef struct VkSubpassEndInfo {
    VkStructureType sType;
    const void* pNext;
} VkSubpassEndInfo;

• sType is a VkStructureType value identifying 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

8.8. Common Render Pass Data Races (Informative)

Due to the complexity of how rendering is performed, there are several ways an application can accidentally introduce a data race, usually by doing something that may seem benign but actually cannot be supported. This section indicates a number of the more common cases as guidelines to help avoid them.

8.8.1. Sampling From a Read-only Attachment

Vulkan includes read-only layouts for depth/stencil images, that allow the images to be both read during a render pass for the purposes of depth/stencil tests, and read as a non-attachment.

However, because VK_ATTACHMENT_STORE_OP_STORE and VK_ATTACHMENT_STORE_OP_DONT_CARE may perform write operations, even if no recorded command writes to an attachment, reading from an image while also using it as an attachment with these store operations can result in a data race. If the reads from the non-attachment are performed in a fragment shader where the accessed samples match those covered by the fragment shader, no data race will occur as store operations are guaranteed to operate after fragment shader execution for the set of samples the fragment covers. Notably, input attachments can also be used for this case. Reading other samples or in any other shader stage can result in unexpected behavior due to the potential for a data race, and validation errors should be generated for doing so. In practice, many applications have shipped reading samples outside of the covered fragment without any observable issue, but there is no guarantee that this will always work, and it is not advisable to rely on this in new or re-worked code bases.

8.8.2. Non-overlapping Access Between Resources

When relying on non-overlapping accesses between attachments and other resources, it is important to note that load and store operations have fairly wide alignment requirements -
potentially affecting entire subresources and adjacent depth/stencil aspects. This makes it invalid to access a non-attachment subresource that is simultaneously being used as an attachment where either access performs a write operation.

8.8.3. Depth/Stencil and Input Attachments

When rendering to only the depth OR stencil aspect of an image, an input attachment accessing the other aspect will always result in a data race.

8.8.4. Synchronization Options

There are several synchronization options available to synchronize between accesses to resources within a render pass. Some of the options are outlined below:

- A **VkSubpassDependency** in a render pass object can synchronize attachment writes and multisample resolve operations from a prior subpass for subsequent input attachment reads.
- A **vkCmdPipelineBarrier** inside a subpass can synchronize prior attachment writes in the subpass with subsequent input attachment reads.
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:

```cpp
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkShaderModule)
```

Shader modules are not used in Vulkan SC, but the type has been retained for compatibility [SCID-8].

In Vulkan SC, the shader modules and pipeline state are supplied to an offline compiler which creates a pipeline cache entry which is loaded at pipeline creation time.

9.2. Binding Shaders

Before a shader can be used it must be first bound to the command buffer.
Calling `vkCmdBindPipeline` binds all stages corresponding to the `VkPipelineBindPoint`.

The following table describes the relationship between shader stages and pipeline bind points:

<table>
<thead>
<tr>
<th>Shader stage</th>
<th>Pipeline bind point</th>
<th>behavior controlled</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <code>VK_SHADER_STAGE_VERTEX_BIT</code></td>
<td><code>VK_PIPELINE_BIND_POINT_GRAPHICS</code></td>
<td>all drawing commands</td>
</tr>
<tr>
<td>• <code>VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• <code>VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• <code>VK_SHADER_STAGE_GEOMETRY_BIT</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• <code>VK_SHADER_STAGE_FRAGMENT_BIT</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• <code>VK_SHADER_STAGE_COMPUTE_BIT</code></td>
<td><code>VK_PIPELINE_BIND_POINT_COMPUTE</code></td>
<td>all dispatch commands</td>
</tr>
</tbody>
</table>

### 9.3. 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.

### 9.3.1. 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.
9.4. 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 SequentiallyConsistent is treated as AcquireRelease. SequentiallyConsistent should not be used.

9.5. 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.
9.6. 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.

9.6.1. 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).

9.7. 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.

9.7.1. 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 \texttt{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.

### 9.8. 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.

#### 9.8.1. 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.

### 9.9. 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.

#### 9.9.1. 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 \texttt{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.

### 9.10. Fragment Shaders

Fragment shaders are invoked as a \texttt{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.

### 9.11. Compute Shaders

Compute shaders are invoked via \texttt{vkCmdDispatch} and \texttt{vkCmdDispatchIndirect} commands. In general, they have access to similar resources as shader stages executing as part of a graphics pipeline.

Compute workloads are formed from groups of work items called workgroups and processed by the
compute shader in the current compute pipeline. A workgroup is a collection of shader invocations that execute the same shader, potentially in parallel. Compute shaders execute in *global workgroups* which are divided into a number of *local workgroups* with a size that can be set by assigning a value to the `LocalSize` execution mode or via an object decorated by the `WorkgroupSize` decoration. An invocation within a local workgroup can share data with other members of the local workgroup through shared variables and issue memory and control flow barriers to synchronize with other members of the local workgroup.

### 9.12. 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`.
9.13. 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. A shader entry point also *statically uses* any variables explicitly declared in its interface.


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.

9.14.2. 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.

9.14.3. 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`. 

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.

### 9.14.5. 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.


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.

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.)
9.14.7. 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.

*Note*

In shaders, there are two kinds of uniformity that are of primary interest to applications: uniform within an invocation group (a.k.a. dynamically uniform), and uniform within a subgroup scope.

While one could make the assumption that being uniform in invocation group implies being uniform in subgroup scope, it is not necessarily the case for shader stages without defined workgroups.

For shader stages with defined workgroups however, the relationship between invocation group and subgroup scope is well defined as a subgroup is a subset of the workgroup, and the workgroup is the invocation group. If a value is uniform in invocation group, it is by definition also uniform in subgroup scope. This is important if writing code like:

```c
uniform texture2D Textures[];
uint dynamicallyUniformValue = gl_WorkGroupID.x;
vec4 value = texelFetch(Textures[dynamicallyUniformValue], coord, 0);
// subgroupUniformValue is guaranteed to be uniform within the subgroup.
// This value also happens to be dynamically uniform.
vec4 subgroupUniformValue = subgroupBroadcastFirst(dynamicallyUniformValue);
```

In shader stages without defined workgroups, this gets complicated. Due to scoping rules, there is no guarantee that a subgroup is a subset of the invocation group, which in turn defines the scope for dynamically uniform. In graphics, the invocation group is a single draw command, except for multi-draw situations, and indirect draws with drawCount > 1, where there are multiple invocation groups, one per **DrawIndex**.

```c
// Assume SubgroupSize = 8, where 3 draws are packed together.
```
Another problematic scenario is when a shader attempts to help the compiler notice that a value is uniform in subgroup scope to potentially improve performance.

For implementations where subgroups are packed across draws, the implementation must make sure to handle descriptor indexing correctly. From the specification's point of view, a dynamically uniform index does not require NonUniform decoration, and such an implementation will likely either promote descriptor indexing into NonUniform on its own, or handle non-uniformity implicitly.

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.

9.14.9. 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.

9.15. Group Operations

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
communication between invocations in a particular scope instance.

Group operations all take a Scope defining the desired scope instance to operate within. Only the Subgroup scope can be used for these operations; the subgroupSupportedOperations limit defines which types of operation can be used.

### 9.15.1. Basic Group Operations

Basic group operations include the use of OpGroupNonUniformElect, OpControlBarrier, OpMemoryBarrier, and atomic operations.

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 true.

The Memory Model appendix defines the operation of barriers and atomics.

### 9.15.2. 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?

**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.

### 9.15.3. 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.

### 9.15.4. Ballot Group Operations

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.

9.15.5. Shuffle Group Operations

The shuffle group operations allow invocations to read values from other invocations within a group.

9.15.6. Shuffle Relative Group Operations

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.

9.15.7. Clustered Group Operations

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.

9.16. 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.

9.17. 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 \( \text{DPdx}(P_i) \) 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 \( \text{DPdy}(P_i) \) 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 \( \text{DPdx} \) and \( \text{DPdy} \)), 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 LOD 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:

\[
\text{DPdx}(P_n)' = \frac{\text{DPdx}(P_n)}{w}
\]

\[
\text{DPdy}(P_n)' = \frac{\text{DPdy}(P_n)}{h}
\]
where \( w \) and \( h \) are the size of the fragments in the quad, and \( \text{DPdx}(P_n)' \) and \( \text{DPdy}(P_n)' \) 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 \( \text{OpImage}^*\text{Sample}^*\text{ImplicitLod} \) 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.

### 9.18. 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.
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.
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 an offline compiled pipeline provided in `pipelineCache` and the pipeline identified by `VkPipelineOfflineCreateInfo` structure in the `pNext` chain of `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 the handle of a valid pipeline cache object.
- `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.

If a pipeline creation fails due to:

- The identified pipeline not being present in `pipelineCache`
- The `pNext` chain not including a `VkPipelineOfflineCreateInfo` structure

the operation will continue as specified in Multiple Pipeline Creation and the command will return `VK_ERROR_NO_PIPELINE_MATCH`.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkCreateComputePipelines` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

### Valid Usage

- **VUID-vkCreateComputePipelines-device-05068**
  The number of compute pipelines currently allocated from `device` plus `createInfoCount` must be less than or equal to the total number of compute pipelines requested via `VkDeviceObjectReservationCreateInfo::computePipelineRequestCount` specified when `device` was created.

### Valid Usage (Implicit)

- **VUID-vkCreateComputePipelines-device-parameter**
  `device` must be a valid `VkDevice` handle
- **VUID-vkCreateComputePipelines-pipelineCache-parameter**
  `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-null
  pAllocator must be NULL

• 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
  pipelineCache 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_NO_PIPELINE_MATCH
• VK_ERROR_OUT_OF_POOL_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 a VkStructureType value identifying 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. This is not used in Vulkan SC [SCID-8].
• `basePipelineIndex` is an index into the pCreateInfos parameter to use as a pipeline to derive
from. This is not used in Vulkan SC [SCID-8].

The parameters `basePipelineHandle` and `basePipelineIndex` are described in more detail in Pipeline Derivatives.

### Valid Usage

- VUID-VkComputePipelineCreateInfo-basePipelineHandle-05024
  `basePipelineHandle must be VK_NULL_HANDLE`

- VUID-VkComputePipelineCreateInfo-basePipelineIndex-05025
  `basePipelineIndex must be zero`

- VUID-VkComputePipelineCreateInfo-layout-07987
  If a push constant block is declared in a shader, a push constant range in `layout` must match both the shader stage and range

- VUID-VkComputePipelineCreateInfo-layout-07988
  If a `resource variables` is declared in a shader, a descriptor slot in `layout` must match the shader stage

- VUID-VkComputePipelineCreateInfo-layout-07990
  If a `resource variables` is declared in a shader, a descriptor slot in `layout` must match the descriptor type

- VUID-VkComputePipelineCreateInfo-layout-07991
  If a `resource variables` is declared in a shader as an array, a descriptor slot in `layout` must match the descriptor count

- 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-01687
  The number of resources in `layout` accessible to the compute shader stage must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageResources`

### 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 VkPipelineOfflineCreateInfo`

- 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 a VkStructureType value identifying 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. This is not used in Vulkan SC [SCID-8].
- 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.

In Vulkan SC, the pipeline compilation process occurs offline. Accordingly, module must be VK_NULL_HANDLE, and the pName and pSpecializationInfo parameters are not used at runtime and should be ignored by the implementation. If provided, the application must set the pName and pSpecializationInfo parameters to the values that were specified for the offline compilation of this pipeline.

**Valid Usage**

- VUID-VkPipelineShaderStageCreateInfo-stage-00704
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-module-05026
  `module` must be `VK_NULL_HANDLE`.

- VUID-VkPipelineShaderStageCreateInfo-pName-05027
  If `pName` is not `NULL`, it must be the name of an `OpEntryPoint` in the SPIR-V shader module used for offline compilation of this pipeline 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-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`.
Valid Usage (Implicit)

- VUID-VkPipelineShaderStageCreateInfo-sType-sType
  
sType must be VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO

- VUID-VkPipelineShaderStageCreateInfo-pNext-pNext
  
pNext must be NULL

- VUID-VkPipelineShaderStageCreateInfo-flags-zerobitmask
  
flags must be 0

- VUID-VkPipelineShaderStageCreateInfo-stage-parameter
  
stage must be a valid VkShaderStageFlagBits value

- VUID-VkPipelineShaderStageCreateInfo-module-parameter
  
If module is not VK_NULL_HANDLE, module must be a valid VkShaderModule handle

- VUID-VkPipelineShaderStageCreateInfo-pName-parameter
  
If pName is not NULL, pName must be a null-terminated UTF-8 string

- VUID-VkPipelineShaderStageCreateInfo-pSpecializationInfo-parameter
  
If pSpecializationInfo is not NULL, pSpecializationInfo must be a valid pointer to a valid VkSpecializationInfo structure

```c
// 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:
typedef enum VkPipelineShaderStageCreateFlagBits {
} VkPipelineShaderStageCreateFlagBits;

Bits which can be set by commands and structures, specifying one or more shader stages, are:

typedef enum VkShaderStageFlagBits {
    VK_SHADER_STAGE_VERTEX_BIT = 0x00000001,
    VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT = 0x00000002,
    VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT = 0x00000004,
    VK_SHADER_STAGE_GEOMETRY_BIT = 0x00000008,
    VK_SHADER_STAGE_FRAGMENT_BIT = 0x00000010,
    VK_SHADER_STAGE_COMPUTE_BIT = 0x00000020,
    VK_SHADER_STAGE_ALL_GRAPHICS = 0x0000001F,
    VK_SHADER_STAGE_ALL = 0x7FFFFFFF,
} 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.

typedef VkFlags VkShaderStageFlags;

VkShaderStageFlags is a bitmask type for setting a mask of zero or more VkShaderStageFlagBits.

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,
    VkPipelineCache pipelineCache,
    uint32_t createInfoCount,
    const VkGraphicsPipelineCreateInfo* pCreateInfos,
    const VkAllocationCallbacks* pAllocator,
    VkPipeline* pPipelines);
```

- `device` is the logical device that creates the graphics pipelines.
- `pipelineCache` is the handle of a valid pipeline cache object.
- `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.

If a pipeline creation fails due to:

- The identified pipeline not being present in `pipelineCache`
- The `pNext` chain not including a `VkPipelineOfflineCreateInfo` structure

the operation will continue as specified in Multiple Pipeline Creation and the command will return `VK_ERROR_NO_PIPELINE_MATCH`.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkCreateGraphicsPipelines` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

---

**Valid Usage**

- VUID-vkCreateGraphicsPipelines-device-05068
  The number of graphics pipelines currently allocated from `device` plus `createInfoCount` must be less than or equal to the total number of graphics pipelines requested via `VkDeviceObjectReservationCreateInfo::graphicsPipelineRequestCount` specified when `device` was created
Valid Usage (Implicit)

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

- VUID-vkCreateGraphicsPipelines-pipelineCache-parameter
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-null
pAllocator must be NULL

- 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
pipelineCache 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_NO_PIPELINE_MATCH
- VK_ERROR_OUT_OF_POOL_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;
} VkGraphicsPipelineCreateInfo;
```
const VkPipelineMultisampleStateCreateInfo* pMultisampleState;
const VkPipelineDepthStencilStateCreateInfo* pDepthStencilState;
const VkPipelineColorBlendStateCreateInfo* pColorBlendState;
const VkPipelineDynamicStateCreateInfo* pDynamicState;
VkPipelineLayout layout;
VkRenderPass renderPass;
uint32_t subpass;
VkPipeline basePipelineHandle;
uint32_t basePipelineIndex;
} VkGraphicsPipelineCreateInfo;

- **sType** is a VkStructureType value identifying 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. This is not used in Vulkan SC [SCID-8].
• basePipelineIndex is an index into the pCreateInfos parameter to use as a pipeline to derive from. This is not used in Vulkan SC [SCID-8].

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**

Vertex input state is defined by:

- VkPipelineVertexInputStateCreateInfo
- VkPipelineInputAssemblyStateCreateInfo

This state **must** be specified to create a complete graphics pipeline.

**Pre-Rasterization Shader State**

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

This state **must** be specified to create a complete graphics pipeline.

**Fragment Shader State**

Fragment shader state is defined by:

- A VkPipelineShaderStageCreateInfo entry for the fragment shader
- Within the VkPipelineLayout, the full pipeline layout must be specified.
- VkPipelineMultisampleStateCreateInfo
- VkPipelineDepthStencilStateCreateInfo
- VkRenderPass and subpass parameter
If `rasterizerDiscardEnable` is set to `VK_FALSE` this state must be specified to create a complete graphics pipeline.

**Fragment Output State**

Fragment output state is defined by:

- `VkPipelineColorBlendStateCreateInfo`
- `VkRenderPass` and `subpass` parameter
- `VkPipelineMultisampleStateCreateInfo`

If `rasterizerDiscardEnable` is set to `VK_FALSE` this state must be specified to create a complete graphics pipeline.

**Dynamic State**

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.

**Complete Graphics Pipelines**

A complete graphics pipeline always includes pre-rasterization shader state, with other subsets included depending on that state as specified in the above sections.

In Vulkan SC, the pipeline compilation process occurs offline and the `pStages` are not needed at runtime and may be omitted. If omitted, `stageCount` must be set to 0 and `pStages` must be `NULL`. If provided, the values must match the values specified to the offline compiler.

---

### Valid Usage

- **VUID-VkGraphicsPipelineCreateInfo-basePipelineHandle-05024**
  - `basePipelineHandle` must be `VK_NULL_HANDLE`

- **VUID-VkGraphicsPipelineCreateInfo-basePipelineIndex-05025**
  - `basePipelineIndex` must be zero

- **VUID-VkGraphicsPipelineCreateInfo-layout-07987**
  - If a push constant block is declared in a shader, a push constant range in `layout` must match both the shader stage and range

- **VUID-VkGraphicsPipelineCreateInfo-layout-07988**
  - If a `resource variables` is declared in a shader, a descriptor slot in `layout` must match the shader stage

- **VUID-VkGraphicsPipelineCreateInfo-layout-07990**
  - If a `resource variables` is declared in a shader, a descriptor slot in `layout` must match the descriptor type

- **VUID-VkGraphicsPipelineCreateInfo-layout-07991**
  - If a `resource variables` is declared in a shader as an array, a descriptor slot in `layout` must match the descriptor count
If the pipeline requires pre-rasterization shader state, the stage member of one element of pStages must be VK_SHADER_STAGE_VERTEX_BIT.

If the pipeline requires pre-rasterization shader state and pStages includes a tessellation control shader stage, it must include a tessellation evaluation shader stage.

If the pipeline requires pre-rasterization shader state and pStages includes a tessellation evaluation shader stage, it must include a tessellation control shader stage.

If the pipeline requires 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.

If the pipeline requires 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.

If the pipeline requires 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.

If the pipeline requires pre-rasterization shader state and pStages includes tessellation shader stages, and the shader code of both stages contain an OpExecutionMode instruction specifying the output 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 vertex input 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 vertex input state and the topology member of pInputAssembly is VK_PRIMITIVE_TOPOLOGY_PATCH_LIST, then pStages must include tessellation shader stages.

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 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 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 for every vertex emitted.

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 requires 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 requires 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 requires 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.

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.

If the pipeline requires 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.

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 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 for every vertex emitted.

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 requires 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 requires 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 requires 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.

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.

If the pipeline requires 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.

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 requires **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-07609**
  If `renderPass` is not `VK_NULL_HANDLE`, and the pipeline is being created with **fragment output interface state**, and the `pColorBlendState` pointer is not `NULL`, 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 requires **pre-rasterization shader state**, and `pViewportState->pViewports` is not dynamic, then `pViewportState->pViewports` must be a valid pointer to an array of `pViewportState->viewportCount` valid `VkViewport` structures.

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-04131**
  If the pipeline requires **pre-rasterization shader state**, and `pViewportState->pScissors` is not dynamic, then `pViewportState->pScissors` must be a valid pointer to an array of `pViewportState->scissorCount` `VkRect2D` structures.

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-00749**
  If the pipeline requires **pre-rasterization shader state**, and the `wideLines` feature is not enabled, and no element of the `pDynamicStates` member of `pRasterizationState` is `VK_DYNAMIC_STATE_LINE_WIDTH`, the `lineWidth` member of `pRasterizationState` must be `1.0`.

- **VUID-VkGraphicsPipelineCreateInfo-rasterizerDiscardEnable-09024**
  If the pipeline requires **pre-rasterization shader state**, and the `rasterizerDiscardEnable` member of `pRasterizationState` is `VK_FALSE`, `pViewportState` must be a valid pointer to a valid `VkPipelineViewportStateCreateInfo` structure.

- **VUID-VkGraphicsPipelineCreateInfo-pMultisampleState-09026**
  If the pipeline requires **fragment output interface state**, `pMultisampleState` must be a valid pointer to a valid `VkPipelineMultisampleStateCreateInfo` structure.

- **VUID-VkGraphicsPipelineCreateInfo-pMultisampleState-09027**
  If `pMultisampleState` is not `NULL` is must be a valid pointer to a valid `VkPipelineMultisampleStateCreateInfo` structure.

- **VUID-VkGraphicsPipelineCreateInfo-alphaToCoverageEnable-08891**
  If the pipeline is being created with **fragment shader state**, the ` VkPipelineMultisampleStateCreateInfo::alphaToCoverageEnable` is not ignored and is `VK_TRUE`, then the **Fragment Output Interface** must contain a variable for the alpha Component word in Location 0 at Index 0.

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-09028**
  If `renderPass` is not `VK_NULL_HANDLE`, the pipeline is being created with **fragment shader state**, and `subpass` uses a depth/stencil attachment, and `pDepthStencilState` must be a valid.
pointer to a valid \texttt{VkPipelineDepthStencilStateCreateInfo} structure

- **VUID-VkGraphicsPipelineCreateInfo-pDepthStencilState-09029**
  If \texttt{pDepthStencilState} is not null it must be a valid pointer to a valid \texttt{VkPipelineDepthStencilStateCreateInfo} structure

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-09030**
  If \texttt{renderPass} is not \texttt{VK_NULL_HANDLE}, the pipeline is being created with fragment output interface state, and \texttt{subpass} uses color attachments, \texttt{pColorBlendState} must be a valid pointer to a valid \texttt{VkPipelineColorBlendStateCreateInfo} structure

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-00754**
  If the pipeline requires 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

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-02510**
  If the pipeline requires 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

- **VUID-VkGraphicsPipelineCreateInfo-subpass-00758**
  If the pipeline requires fragment output interface state, \texttt{rasterizationSamples} is not dynamic, 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

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06046**
  If \texttt{renderPass} is not \texttt{VK_NULL_HANDLE}, \texttt{subpass} must be a valid subpass within \texttt{renderPass}

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06047**
  If \texttt{renderPass} is not \texttt{VK_NULL_HANDLE}, the pipeline is being created with pre-rasterization shader state, \texttt{subpass} viewMask is not 0, and \texttt{multiviewTessellationShader} is not enabled, then \texttt{pStages} must not include tessellation shaders

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06048**
  If \texttt{renderPass} is not \texttt{VK_NULL_HANDLE}, the pipeline is being created with 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

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06049**
  If \texttt{renderPass} is not \texttt{VK_NULL_HANDLE}, the pipeline is being created with pre-rasterization shader state, and \texttt{subpass} viewMask is not 0, all of the shaders in the pipeline must not write to the Layer built-in output

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-06050**
  If \texttt{renderPass} is not \texttt{VK_NULL_HANDLE} and the pipeline is being created with pre-rasterization shader state, and \texttt{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

- **VUID-VkGraphicsPipelineCreateInfo-renderPass-07717**
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.

- VUID-VkGraphicsPipelineCreateInfo-flags-00764
  `flags` must not contain the `VK_PIPELINE_CREATE_DISPATCH_BASE` flag

- VUID-VkGraphicsPipelineCreateInfo-pStages-01565
  If the pipeline requires 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

- VUID-VkGraphicsPipelineCreateInfo-layout-01688
  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`

- VUID-VkGraphicsPipelineCreateInfo-pStages-02097
  If the pipeline requires vertex input state, and `pVertexInputState` is not dynamic, then `pVertexInputState` must be a valid pointer to a valid `VkPipelineVertexInputStateCreateInfo` structure

- VUID-VkGraphicsPipelineCreateInfo-Input-07904
  If the pipeline is being created with vertex input state and `pVertexInputState` is not dynamic, then all variables with the `Input` storage class decorated with `Location` in the `Vertex` Execution Model `OpEntryPoint` must contain a location in `VkVertexInputAttributeDescription::location`

- VUID-VkGraphicsPipelineCreateInfo-Input-08733
  If the pipeline requires vertex input state and `pVertexInputState` is not dynamic, then the numeric type associated with all `Input` variables of the corresponding `Location` in the `Vertex` Execution Model `OpEntryPoint` must be the same as `VkVertexInputAttributeDescription::format`

- VUID-VkGraphicsPipelineCreateInfo-pVertexInputState-08929
  If the pipeline is being created with vertex input state and `pVertexInputState` is not dynamic, and `VkVertexInputAttributeDescription::format` has a 64-bit component, then the scalar width associated with all `Input` variables of the corresponding `Location` in the `Vertex` Execution Model `OpEntryPoint` must be 64-bit

- VUID-VkGraphicsPipelineCreateInfo-pVertexInputState-08930
  If the pipeline is being created with vertex input state and `pVertexInputState` is not dynamic, and the scalar width associated with a `Location` decorated `Input` variable in the `Vertex` Execution Model `OpEntryPoint` is 64-bit, then the corresponding `VkVertexInputAttributeDescription::format` must have a 64-bit component

- VUID-VkGraphicsPipelineCreateInfo-pVertexInputState-09198
  If the pipeline is being created with vertex input state and `pVertexInputState` is not dynamic, and `VkVertexInputAttributeDescription::format` has a 64-bit component, then all `Input` variables at the corresponding `Location` in the `Vertex` Execution Model `OpEntryPoint` must not use components that are not present in the format

- VUID-VkGraphicsPipelineCreateInfo-dynamicPrimitiveTopologyUnrestricted-09031
  If the pipeline requires vertex input state, `pInputAssemblyState` must be a valid pointer to
a valid VkPipelineInputAssemblyStateCreateInfo structure

• VUID-VkGraphicsPipelineCreateInfo-pInputAssemblyState-09032
  If pInputAssemblyState is not NULL it must be a valid pointer to a valid VkPipelineInputAssemblyStateCreateInfo structure

• VUID-VkGraphicsPipelineCreateInfo-pStages-06600
  If the pipeline requires pre-rasterization shader state or fragment shader state, pStages must be a valid pointer to an array of stageCount valid VkPipelineShaderStageCreateInfo structures

• VUID-VkGraphicsPipelineCreateInfo-pRasterizationState-06601
  If the pipeline requires pre-rasterization shader state, pRasterizationState must be a valid pointer to a valid VkPipelineRasterizationStateCreateInfo structure

• VUID-VkGraphicsPipelineCreateInfo-layout-06602
  If the pipeline requires fragment shader state or pre-rasterization shader state, layout must be a valid VkPipelineLayout handle

• VUID-VkGraphicsPipelineCreateInfo-renderPass-06603
  If pre-rasterization shader state, fragment shader state, or fragment output state, renderPass must be a valid VkRenderPass handle

• VUID-VkGraphicsPipelineCreateInfo-stageCount-06604
  If the pipeline requires pre-rasterization shader state or fragment shader state, stageCount must be greater than 0

• VUID-VkGraphicsPipelineCreateInfo-graphicsPipelineLibrary-06606
  flags must not include VK_PIPELINE_CREATE_LIBRARY_BIT_KHR

• VUID-VkGraphicsPipelineCreateInfo-pStages-06894
  If the pipeline requires pre-rasterization shader state but not fragment shader state, elements of pStages must not have stage set to VK_SHADER_STAGE_FRAGMENT_BIT

• VUID-VkGraphicsPipelineCreateInfo-pStages-06895
  If the pipeline requires fragment shader state but not pre-rasterization shader state, elements of pStages must not have stage set to a shader stage which participates in pre-rasterization

• VUID-VkGraphicsPipelineCreateInfo-pStages-06896
  If the pipeline requires pre-rasterization shader state, all elements of 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 requires fragment shader state and/or pre-rasterization shader state, any value of stage must not be set in more than one element of pStages

• VUID-VkGraphicsPipelineCreateInfo-None-08893
  The pipeline must be created with pre-rasterization shader state

• VUID-VkGraphicsPipelineCreateInfo-pStages-08894
  If pStages includes a vertex shader stage, the pipeline must be created with vertex input state

• VUID-VkGraphicsPipelineCreateInfo-pDynamicState-08896
If \( \text{pRasterizationState->rasterizerDiscardEnable} \) is \( \text{VK_FALSE} \), the pipeline \textbf{must} be created with \textit{fragment shader state} and \textit{fragment output interface state}

- VUID-VkGraphicsPipelineCreateInfo-None-09043
  If the format of any color attachment is \( \text{VK_FORMAT_E5B9G9R9_UFLOAT_PACK32} \), the colorWriteMask member of the corresponding element of \text{pColorBlendState->pAttachments} \textbf{must} either include all of \( \text{VK_COLOR_COMPONENT_R_BIT} \), \( \text{VK_COLOR_COMPONENT_G_BIT} \), and \( \text{VK_COLOR_COMPONENT_B_BIT} \), or none of them

\textbf{Valid Usage (Implicit)}

- VUID-VkGraphicsPipelineCreateInfo-sType-sType
  \textbf{sType} \textbf{must} be \( \text{VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO} \)

- VUID-VkGraphicsPipelineCreateInfo-pNext-pNext
  \textbf{pNext} \textbf{must} be \text{NULL} or a pointer to a valid instance of \text{VkPipelineOfflineCreateInfo}

- VUID-VkGraphicsPipelineCreateInfo-sType-unique
  The \textbf{sType} value of each struct in the \textbf{pNext} chain \textbf{must} be unique

- VUID-VkGraphicsPipelineCreateInfo-flags-parameter
  \textbf{flags} \textbf{must} be a valid combination of \text{VkPipelineCreateFlagBits} values

- VUID-VkGraphicsPipelineCreateInfo-pDynamicState-parameter
  If \text{pDynamicState} is not \text{NULL}, \text{pDynamicState} \textbf{must} be a valid pointer to a valid \text{VkPipelineDynamicStateCreateInfo} structure

- VUID-VkGraphicsPipelineCreateInfo-commonparent
  Each of \text{basePipelineHandle}, \text{layout}, and \text{renderPass} that are valid handles of non-ignored parameters \textbf{must} have been created, allocated, or retrieved from the same \text{VkDevice}

\textbf{Bits which }\textbf{can} \textbf{be set in}

- \text{VkGraphicsPipelineCreateInfo::flags}
- \text{VkComputePipelineCreateInfo::flags}

\textbf{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,
    // Provided by VK_VERSION_1_1
    VK_PIPELINE_CREATE_VIEW_INDEX_FROM_DEVICE_INDEX_BIT = 0x00000008,
    // Provided by VK_VERSION_1_1
    VK_PIPELINE_CREATE_DISPATCH_BASE_BIT = 0x00000010,
    // Provided by VK_VERSION_1_1
    VK_PIPELINE_CREATE_DISPATCH_BASE = VK_PIPELINE_CREATE_DISPATCH_BASE_BIT,
} VkPipelineCreateFlagBits;
```

- \text{VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT} \textbf{specifies that the created pipeline will not be}
optimized. Using this flag may reduce the time taken to create the 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.

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

`VkPipelineCreateFlags` is a bitmask type for setting a mask of zero or more `VkPipelineCreateFlagBits`.

The `VkPipelineDynamicStateCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineDynamicStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineDynamicStateCreateFlags flags;
    uint32_t dynamicStateCount;
    const VkDynamicState* pDynamicStates;
} VkPipelineDynamicStateCreateInfo;
```

- `sType` is a `VkStructureType` value identifying this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `dynamicStateCount` is the number of elements in the `pDynamicStates` array.
- `pDynamicStates` is a pointer to an array of `VkDynamicState` values specifying which pieces of pipeline state will use the values from dynamic state commands rather than from pipeline state creation information.

### Valid Usage

- VUID-VkPipelineDynamicStateCreateInfo-pDynamicStates-01442
  Each element of `pDynamicStates` must be unique

### Valid Usage (Implicit)

- VUID-VkPipelineDynamicStateCreateInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO`

- VUID-VkPipelineDynamicStateCreateInfo-pNext-pNext
  `pNext` must be `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,
} 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 either vkCmdSetDepthBias before any draws are performed with depth
bias enabled.
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.

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 derived from the pipeline cache entry identified by VkPipelineOfflineCreateInfo::pipelineIdentifier.

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

**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-null
  `pAllocator` must be `NULL`

- 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 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.

Pipeline derivatives are not supported in Vulkan SC due to the use of read-only offline generated pipeline caches [SCID-8].

10.6. Pipeline Cache

Pipeline cache objects allow the application to load multiple binary pipeline objects generated by an offline cache creation tool into pipeline cache objects. The cache can then be used during pipeline creation to load offline pipeline data.

Pipeline cache objects are represented by VkPipelineCache handles:

```c
// 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:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreatePipelineCache(
    VkDevice device,
    const VkPipelineCacheCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkPipelineCache* pPipelineCache);
```

- **device** is the logical device that creates the pipeline cache object.
- **pCreateInfo** is a pointer to a VkPipelineCacheCreateInfo structure containing initial parameters for the pipeline cache object.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
- **pPipelineCache** is a pointer to a VkPipelineCache handle in which the resulting pipeline cache
If the pipeline cache data pointed to by `VkPipelineCacheCreateInfo::pInitialData` is not compatible with the device, pipeline cache creation will fail and `VK_ERROR_INVALID_PIPELINE_CACHE_DATA` will be returned.

Once created, a pipeline cache can be passed to the `vkCreateGraphicsPipelines` and `vkCreateComputePipelines` commands. The pipeline cache passed into these commands will be queried by the implementation for matching pipelines on pipeline creation. After the cache is created, its contents cannot be updated. 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 `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkCreatePipelineCache` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

### Valid Usage

- **VUID-vkCreatePipelineCache-pCreateInfo-05045**
  The contents of the structure pointed to by `pCreateInfo` and the data pointed to by `pCreateInfo->pInitialData` must be the same as specified in one of the `VkDeviceObjectReservationCreateInfo::pPipelineCacheCreateInfos` structures when the device was created.

- **VUID-vkCreatePipelineCache-device-05068**
  The number of pipeline caches currently allocated from `device` plus 1 must be less than or equal to the total number of pipeline caches requested via `VkDeviceObjectReservationCreateInfo::pipelineCacheRequestCount` specified when `device` was created.

### 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-pAllocate-NULL**
  `pAllocate` must be NULL.

- **VUID-vkCreatePipelineCache-pPipelineCache-parameter**
  `pPipelineCache` must be a valid pointer to a `VkPipelineCache` handle.

### Return Codes

**Success**

- `VK_SUCCESS`
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 a **VkStructureType** value identifying 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**.
- **pInitialData** is a pointer to pipeline cache data that has been generated offline. If the pipeline cache data is incompatible (as defined below) with the device, **VK_ERROR_INVALID_PIPELINE_CACHE_DATA** is returned.

### Valid Usage

- **VUID-VkPipelineCacheCreateInfo-flags-05043**
  - **flags** must include **VK_PIPELINE_CACHE_CREATE_READ_ONLY_BIT**
- **VUID-VkPipelineCacheCreateInfo-flags-05044**
  - **flags** must include **VK_PIPELINE_CACHE_CREATE_USE_APPLICATION_STORAGE_BIT**
- **VUID-VkPipelineCacheCreateInfo-pInitialData-05139**
  - The pipeline cache data pointed to by **pInitialData** must not contain any pipelines with duplicate pipeline identifiers.

### 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
  pInitialData must be a valid pointer to an array of initialDataSize bytes

* VUID-VkPipelineCacheCreateInfo-initialDataSize-arraylength
  initialDataSize must be greater than 0

```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 VKSC_VERSION_1_0
typedef enum VkPipelineCacheCreateFlagBits {
    // Provided by VKSC_VERSION_1_0
    VK_PIPELINE_CACHE_CREATE_READ_ONLY_BIT = 0x00000002,
    // Provided by VKSC_VERSION_1_0
    VK_PIPELINE_CACHE_CREATE_USE_APPLICATION_STORAGE_BIT = 0x00000004,
} VkPipelineCacheCreateFlagBits;
```

- **VK_PIPELINE_CACHE_CREATE_READ_ONLY_BIT** specifies that the new pipeline cache will be read-only.
- **VK_PIPELINE_CACHE_CREATE_USE_APPLICATION_STORAGE_BIT** specifies that the application will maintain the contents of the memory pointed to by pInitialData for the lifetime of the pipeline cache object created, avoiding the need for the implementation to make a copy of the data. The memory pointed to by pInitialData can be modified or released by the application only after any pipeline cache objects created using it have been destroyed.

### 10.6.2. Pipeline Cache Header

Applications must load data from offline compiled pipeline caches into pipeline cache objects. The results of pipeline compilations may depend on the vendor ID, device ID, driver version, and other details of the target device. To allow detection of pipeline cache data that is incompatible with the device, the pipeline cache data must begin with a valid pipeline cache header.

**Note**

Structures described in this section are not part of the Vulkan API and are only used to describe the representation of data elements in pipeline cache data. Accordingly, the valid usage clauses defined for structures defined in this section do not define valid usage conditions for APIs accepting pipeline cache data as input, as providing invalid pipeline cache data as input to any Vulkan API...
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` 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-05075
  - `headerSize` must be 56
- VUID-VkPipelineCacheHeaderVersionOne-headerVersion-05076
  - `headerVersion` must be `VK_PIPELINE_CACHE_HEADER_VERSION_SAFETY_CRITICAL_ONE`
- VUID-VkPipelineCacheHeaderVersionOne-headerSize-08990
  - `headerSize` must not exceed the size of the pipeline cache

### 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,
    // Provided by VKSC_VERSION_1_0
    VK_PIPELINE_CACHE_HEADER_VERSION_SAFETY_CRITICAL_ONE = 1000298001,
} VkPipelineCacheHeaderVersion;
```

- `VK_PIPELINE_CACHE_HEADER_VERSION_ONE` specifies version one of the pipeline cache, described by `VkPipelineCacheHeaderVersionOne`.
- `VK_PIPELINE_CACHE_HEADER_VERSION_SAFETY_CRITICAL_ONE` specifies version one of the pipeline cache for Vulkan SC, described by `VkPipelineCacheHeaderVersionSafetyCriticalOne`.

Version one of the pipeline cache header for Vulkan SC is defined as:

```c
// Provided by VKSC_VERSION_1_0
typedef struct VkPipelineCacheHeaderVersionSafetyCriticalOne {
    VkPipelineCacheHeaderVersionOne headerVersionOne;
    VkPipelineCacheValidationVersion validationVersion;
    uint32_t implementationData;
    uint32_t pipelineIndexCount;
    uint32_t pipelineIndexStride;
    uint64_t pipelineIndexOffset;
} VkPipelineCacheHeaderVersionSafetyCriticalOne;
```

- `headerVersionOne` is a `VkPipelineCacheHeaderVersionOne` structure.
- `validationVersion` is a `VkPipelineCacheValidationVersion` enum value specifying the version of any validation information that is included in this pipeline cache.
- `implementationData` is 4 bytes of padding to ensure structure members are consistently aligned on all platforms. The contents of this field may be used for implementation-specific information.
- `pipelineIndexCount` is the number of entries contained in the pipeline cache index.
- `pipelineIndexStride` is the number of bytes between consecutive pipeline cache index entries.
- `pipelineIndexOffset` is the offset in bytes from the beginning of the pipeline cache header to the pipeline cache index.

The pipeline cache index consists of `pipelineIndexCount` `VkPipelineCacheSafetyCriticalIndexEntry` structures containing an index of all the pipelines in this cache. The pipeline cache index is located starting at `pipelineIndexOffset` bytes into the cache and the location of pipeline `i` is calculated as: `pipelineIndexOffset + i × pipelineIndexStride`. The `VkPipelineCacheSafetyCriticalIndexEntry` structures may not be tightly packed, enabling additional implementation-specific data to be stored with each entry, or for future extensibility.

\[\text{pipelineIndexOffset} + i \times \text{pipelineIndexStride} \]
Because the pipeline cache index is keyed by pipeline identifier, applications and offline compilers must ensure that there are no pipelines with identical pipeline identifiers in the same pipeline cache.

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 56 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-VkPipelineCacheHeaderVersionSafetyCriticalOne-validationVersion-05077 validationVersion must be VK_PIPELINE_CACHE_VALIDATION_VERSION_SAFETY_CRITICAL_ONE
- VUID-VkPipelineCacheHeaderVersionSafetyCriticalOne-pipelineIndexStride-05078 pipelineIndexStride must be greater than or equal to 56 (the size of the VkPipelineCacheSafetyCriticalIndexEntry structure)
- VUID-VkPipelineCacheHeaderVersionSafetyCriticalOne-pipelineIndexOffset-05079 pipelineIndexOffset + pipelineIndexCount × pipelineIndexStride must not exceed the size of the pipeline cache

Valid Usage (Implicit)

- VUID-VkPipelineCacheHeaderVersionSafetyCriticalOne-headerVersionOne-parameter headerVersionOne must be a valid VkPipelineCacheHeaderVersionOne structure
- VUID-VkPipelineCacheHeaderVersionSafetyCriticalOne-validationVersion-parameter validationVersion must be a valid VkPipelineCacheValidationVersion value

The VkPipelineCacheValidationVersion enumeration determines the contents of the pipeline cache validation information. Possible values are:

```c
// Provided by VKSC_VERSION_1_0
typedef enum VkPipelineCacheValidationVersion {
    VK_PIPELINE_CACHE_VALIDATION_VERSION_SAFETY_CRITICAL_ONE = 1,
} VkPipelineCacheValidationVersion;
```

- VK_PIPELINE_CACHE_VALIDATION_VERSION_SAFETY_CRITICAL_ONE specifies version one of the pipeline cache validation information for Vulkan SC.

Each pipeline cache index entry consists of a VkPipelineCacheSafetyCriticalIndexEntry structure:

```c
// Provided by VKSC_VERSION_1_0
```
typedef struct VkPipelineCacheSafetyCriticalIndexEntry {
    uint8_t  pipelineIdentifier[VK_UUID_SIZE];
    uint64_t pipelineMemorySize;
    uint64_t jsonSize;
    uint64_t jsonOffset;
    uint32_t stageIndexCount;
    uint32_t stageIndexStride;
    uint64_t stageIndexOffset;
} VkPipelineCacheSafetyCriticalIndexEntry;

• pipelineIdentifier is the pipeline identifier indicating which pipeline the information is associated with.

• pipelineMemorySize is the number of bytes of pipeline memory required for this pipeline. This is the minimum value that can be successfully used for VkPipelineOfflineCreateInfo::poolEntrySize when this pipeline is used.

• jsonSize is the size in bytes of the pipeline JSON data representing the pipeline state for this pipeline. This value may be zero, indicating the JSON data is not present in the pipeline cache for this pipeline.

• jsonOffset is the offset in bytes from the beginning of the pipeline cache header to the pipeline JSON data for this pipeline. This value must be zero if the JSON data is not present in the pipeline cache for this pipeline.

• stageIndexCount is the number of entries in the pipeline cache stage validation index for this pipeline. This value may be zero, indicating that no stage validation information is present in the pipeline cache for this pipeline.

• stageIndexStride is the number of bytes between consecutive stage validation index entries.

• stageIndexOffset is the offset in bytes from the beginning of the pipeline cache header to the stage validation index for this pipeline. This value must be zero if no stage validation information is present for this pipeline.

The JSON data and the stage validation index are optionally included in the pipeline cache index entry. They are only intended to be used for validation and debugging. If present they must include both the JSON data and the corresponding SPIR-V modules that were used by the offline compiler to compile the pipeline cache entry.

The data at jsonOffset consists of a byte stream of jsonSize bytes of UTF-8 encoded JSON that was used by the offline pipeline compiler to create this pipeline cache entry.

The stage validation index consists of stageIndexCount VkPipelineCacheStageValidationIndexEntry structures which provide the SPIR-V modules used by this pipeline and these are provided in the same order as provided to the VkPipelineShaderStageCreateInfo structure(s) in the Vk*PipelineCreateInfo structure for this pipeline. The stage validation index is located at stageIndexOffset bytes into the cache and the location of stage i is calculated as: stageIndexOffset + i × stageIndexStride. The VkPipelineCacheStageValidationIndexEntry structures may not be tightly packed, enabling additional implementation-specific data to be stored with each entry, or for future extensibility.
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 56 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-VkPipelineCacheSafetyCriticalIndexEntry-jsonSize-05080**
  If `jsonSize` is 0, `jsonOffset` **must** be 0

- **VUID-VkPipelineCacheSafetyCriticalIndexEntry-jsonSize-05081**
  If `jsonSize` is 0, `stageIndexCount` **must** be 0

- **VUID-VkPipelineCacheSafetyCriticalIndexEntry-jsonSize-08991**
  If `jsonSize` is not 0, `jsonOffset` + `jsonSize` **must** not exceed the size of the pipeline cache

- **VUID-VkPipelineCacheSafetyCriticalIndexEntry-stageIndexCount-05082**
  If `stageIndexCount` is 0, `stageIndexOffset` and `stageIndexStride` **must** be 0

- **VUID-VkPipelineCacheSafetyCriticalIndexEntry-stageIndexCount-05083**
  If `stageIndexCount` is not 0, `stageIndexStride` **must** be greater than or equal to 16 (the size of the `VkPipelineCacheStageValidationIndexEntry` structure)

- **VUID-VkPipelineCacheSafetyCriticalIndexEntry-stageIndexCount-05084**
  If `stageIndexCount` is not 0, `stageIndexOffset` + `stageIndexCount` × `stageIndexStride` **must** not exceed the size of the pipeline cache

Each pipeline cache stage validation index entry consists of a `VkPipelineCacheStageValidationIndexEntry` structure:

```c
// Provided by VKSC_VERSION_1_0
typedef struct VkPipelineCacheStageValidationIndexEntry {
    uint64_t codeSize;
    uint64_t codeOffset;
} VkPipelineCacheStageValidationIndexEntry;
```

- `codeSize` is the size in bytes of the SPIR-V module for this pipeline stage.
- `codeOffset` is the offset in bytes from the beginning of the pipeline cache header to the SPIR-V module for this pipeline stage.

The data at `codeOffset` consists of `codeSize` bytes of SPIR-V module as described in Appendix A that was used by the **offline pipeline compiler** for this shader stage when creating this pipeline cache entry.

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 16 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-VkPipelineCacheStageValidationIndexEntry-codeSize-05085  
  `codeSize` must be greater than 0
- VUID-VkPipelineCacheStageValidationIndexEntry-codeSize-05086  
  `codeSize` must be a multiple of 4
- VUID-VkPipelineCacheStageValidationIndexEntry-codeOffset-05087  
  `codeOffset + codeSize` must not exceed the size of the pipeline cache

### 10.6.3. 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 (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-null  
  `pAllocator` must be `NULL`
- VUID-vkDestroyPipelineCache-pPipelineCache-parent  
  If `pipelineCache` is a valid handle, it must have been created, allocated, or retrieved from `device`
10.7. Offline Pipeline Compilation

In Vulkan SC, the pipeline compilation process occurs offline [SCID-8].

The SPIR-V shader module and pipeline state are supplied to an offline pipeline cache compiler which creates a pipeline cache entry for the pipeline. The set of pipeline cache entries are combined offline into one or more pipeline caches. At application run-time, the offline generated pipeline cache is provided to device creation as part of the VkDeviceObjectReservationCreateInfo structure and then loaded into a VkPipelineCache object by the application. The device, pipeline, and pipeline cache creation functions can extract implementation-specific information from the pipeline cache. The specific pipeline to be loaded from the cache is specified at pipeline creation time using a pipeline identifier. The pipeline state that is provided at runtime to pipeline creation must match the state that was specified to the offline pipeline cache compiler when the pipeline cache entry was created offline (with the exception of the VkPipelineShaderStageCreateInfo structure).

In order to assist with the specification of pipeline state for the offline pipeline cache compiler, Khronos has defined a pipeline JSON schema to represent the pipeline state required to compile a SPIR-V module to device-specific machine code and a set of utilities to help with reading and writing of the JSON files. See https://github.com/KhronosGroup/VulkanSC-Docs/wiki/JSON-schema for more information.

10.8. Pipeline Memory Reservation

Pipeline memory is allocated from a pool that is reserved at device creation time. The offline pipeline cache compiler writes the pipeline memory size requirements for each pipeline into the pipeline's VkPipelineCacheSafetyCriticalIndexEntry::pipelineMemorySize entry in the pipeline cache index. The offline pipeline cache compiler may also report it separately. The elements of VkDeviceObjectReservationCreateInfo::pPipelinePoolSizes are requests for poolEntryCount pool entries each of pool size poolEntrySize, and any pipeline with a VkPipelineCacheSafetyCriticalIndexEntry::pipelineMemorySize less than or equal to VkPipelineOfflineCreateInfo::poolEntrySize can be placed in one of those pool entries. The application should request a set of pool sizes that best suits its anticipated worst-case usage.

On implementations where VkPhysicalDeviceVulkanSC10Properties::recyclePipelineMemory is VK_FALSE, the memory for the pipeline pool is not recycled when a pipeline is destroyed, and once an entry has been used it cannot be reused. On implementations where VkPhysicalDeviceVulkanSC10Properties::recyclePipelineMemory is VK_TRUE, the memory for the pipeline pool is recycled when a pipeline is destroyed, and the entry it was using becomes available to be reused.
10.9. Pipeline Identifier

A pipeline identifier is an identifier that can be used to identify a specific pipeline independently from the pipeline description, shader stages and any relevant fixed-function stages, that were used to create the pipeline object.

The VkPipelineOfflineCreateInfo structure allows an identifier to be specified for the pipeline at pipeline creation via the pNext field of the VkGraphicsPipelineCreateInfo, and VkComputePipelineCreateInfo structures. If a VkPipelineOfflineCreateInfo structure is not included in the pNext chain then pipeline creation will fail and VK_ERROR_NOPIPELINE_MATCH will be returned by the corresponding vkCreate*Pipelines command.

The identifier must be used by the implementation to match against the existing content of the pipeline cache at pipeline creation. This is required for Vulkan SC where pipelines are generated offline and there is no shader code in the pipeline cache to match at runtime.

Note
The identifier values must be specified or generated during the offline pipeline cache generation and embedded in to the pipeline cache blob.

The VkPipelineOfflineCreateInfo structure is defined as:

```c
// Provided by VKSC_VERSION_1_0
typedef struct VkPipelineOfflineCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint8_t pipelineIdentifier[VK_UUID_SIZE];
    VkPipelineMatchControl matchControl;
    VkDeviceSize poolEntrySize;
} VkPipelineOfflineCreateInfo;
```

- **sType** is a VkStructureType value identifying this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **pipelineIdentifier** is an array of VK_UUID_SIZE uint8_t values representing an identifier for the pipeline.
- **matchControl** is a VkPipelineMatchControl value specifying the type of identifier being used and how the match should be performed.
- **poolEntrySize** is the size of the entry in pipeline memory to use for this pipeline. It must be a size that was requested via VkPipelinePoolSize when the device was created.

If a match in the pipeline cache is not found then VK_ERROR_NOPIPELINE_MATCH will be returned to the application.

If poolEntrySize is too small for the pipeline, or the number of entries for the requested pool size exceeds the reserved count for that pool size, pipeline creation will fail and VK_ERROR_OUTOFPOOL_MEMORY will be returned by the corresponding vkCreate*Pipelines command.
Valid Usage

- **VUID-VkPipelineOfflineCreateInfo-poolEntrySize-05028**
  
  poolEntrySize must be one of the sizes requested via VkPipelinePoolSize when the device was created.

- **VUID-VkPipelineOfflineCreateInfo-recyclePipelineMemory-05029**
  
  If VkPhysicalDeviceVulkanSC10Properties::recyclePipelineMemory is VK_TRUE, the number of currently existing pipelines created with this same value of poolEntrySize plus 1 must be less than or equal to the sum of the VkPipelinePoolSize::poolEntryCount values with the same value of poolEntrySize.

- **VUID-VkPipelineOfflineCreateInfo-recyclePipelineMemory-05030**
  
  If VkPhysicalDeviceVulkanSC10Properties::recyclePipelineMemory is VK_FALSE, the total number of pipelines ever created with this same value of poolEntrySize plus 1 must be less than or equal to the sum of the VkPipelinePoolSize::poolEntryCount values with the same value of poolEntrySize.

Valid Usage (Implicit)

- **VUID-VkPipelineOfflineCreateInfo-sType-sType**
  
  sType must be VK_STRUCTURE_TYPE_PIPELINE_OFFLINE_CREATE_INFO.

- **VUID-VkPipelineOfflineCreateInfo-matchControl-parameter**
  
  matchControl must be a valid VkPipelineMatchControl value.

Possible values of the matchControl member of VkPipelineOfflineCreateInfo

```c
// Provided by VKSC_VERSION_1_0
typedef enum VkPipelineMatchControl {
    VK_PIPELINE_MATCH_CONTROL_APPLICATION_UUID_EXACT_MATCH = 0,
} VkPipelineMatchControl;
```

are:

- **VK_PIPELINE_MATCH_CONTROL_APPLICATION_UUID_EXACT_MATCH** specifies that the identifier is a UUID generated by the application and the identifiers must be an exact match.

10.10. 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 compiled offline. This allows a SPIR-V module to have constants that can be modified at compilation time rather than in the SPIR-V source. The pSpecializationInfo parameters are not used at runtime and should be ignored by the implementation. If provided, the application must set the pSpecializationInfo parameters to the values that were specified for the offline compilation of this pipeline.
Note
Specialization constants are useful to allow a compute shader to have its local
workgroup size changed at pipeline compilation time, 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:
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 ; decorate .x component of WorkgroupSize with ID 13
OpDecorate %y SpecId 42 ; decorate .y component of WorkgroupSize with ID 42
OpDecorate %z SpecId 3  ; decorate .z component of WorkgroupSize with ID 3
OpDecorate %wgsize BuiltIn WorkgroupSize ; decorate WorkgroupSize onto constant
%i32 = OpTypeInt 32 0 ; declare an **unsigned** 32-bit type
%uvec3 = OpTypeVector %i32 3 ; declare a 3 element vector type of **unsigned** 32-bit
%x = OpSpecConstant %i32 1 ; declare the .x component of WorkgroupSize
%y = OpSpecConstant %i32 1 ; declare the .y component of WorkgroupSize
%z = OpSpecConstant %i32 1 ; declare the .z component of WorkgroupSize
%wgsize = OpSpecConstantComposite %uvec3 %x %y %z ; declare WorkgroupSize
```

From the above we have three specialization constants, one for each of the x, y & z elements of the WorkgroupSize vector.

Now to specialize the above via the specialization constants mechanism:

```cpp
const VkSpecializationMapEntry entries[] =
{
  
  .constantID = 13,
  .offset = 0 * sizeof(uint32_t),
  .size = sizeof(uint32_t)
}```
const uint32_t data[] = { 16, 8, 4 }; // our workgroup size is 16x8x4

const VkSpecializationInfo info =
{
  .mapEntryCount = 3,
  .pMapEntries = entries,
  .dataSize = 3 * sizeof(uint32_t),
  .pData = data,
};

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
%2 = OpSpecConstant %float 0.5 ; some 32-bit floating-point constant

From the above we have two specialization constants, one is a signed 32-bit integer and the second is a 32-bit floating-point value.

Now to specialize the above via the specialization constants mechanism:

struct SpecializationData {
  int32_t data0;
  float data1;
};
const VkSpecializationMapEntry entries[] =
{
    {
        .constantID = 0,
        .offset = offsetof(SpecializationData, data0),
        .size = sizeof(SpecializationData::data0)
    },
    {
        .constantID = 12,
        .offset = offsetof(SpecializationData, data1),
        .size = sizeof(SpecializationData::data1)
    }
};

SpecializationData data;
data.data0 = -42;  // set the data for the 32-bit integer
data.data1 = 42.0f; // set the data for the 32-bit floating-point

const VkSpecializationInfo info =
{
    .mapEntryCount = 2,
    .pMapEntries = entries,
    .dataSize = sizeof(data),
    .pData = &data,
};

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.

10.11. 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.

### 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.

**10.12. 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.
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.

The Vulkan SC implementation will perform its own host memory allocations. Support for application-provided memory allocation, as supported in Base Vulkan, has been removed in Vulkan SC.

*VkAllocationCallbacks* is not supported and pointers to this type *must* be *NULL* [SCID-2], [SCID-8].

```c
// Provided by VK_VERSION_1_0
typedef struct VkAllocationCallbacks {
    void* pUserData;
    PFN_vkAllocationFunction pfnAllocation;
    PFN_vkReallocationFunction pfnReallocation;
    PFN_vkFreeFunction pfnFree;
    PFN_vkInternalAllocationNotification pfnInternalAllocation;
    PFN_vkInternalFreeNotification pfnInternalFree;
} VkAllocationCallbacks;
```

### 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_PROPERTY_DEVICE_LOCAL_BIT`
- `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT | VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`
- `VK_MEMORYPROPERTY_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_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_PROPERTY_DEVICE_LOCAL_BIT was before
VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, the
list would still be in a valid order.

This ordering requirement enables applications to use a simple search loop to select the desired
memory type along the lines of:

```c
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);
    }
    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,
                      VK_MEMORY_PROPERTY_HOST_COHERENT_BIT);
```
VK_MAX_MEMORY_TYPES is the length of an array of VkMemoryType structures describing memory types, as returned in VkPhysicalDeviceMemoryProperties::memoryTypes.

```c
#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.

```c
#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 a `VkStructureType` value identifying 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:

```c
// Provided by VK_VERSION_1_0
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:

```c
// Provided by VK_VERSION_1_0
typedef enum VkMemoryHeapFlagBits {
    VK_MEMORY_HEAP_DEVICE_LOCAL_BIT = 0x00000001,
    // Provided by VK_VERSION_1_1
    VK_MEMORY_HEAP_MULTI_INSTANCE_BIT = 0x00000002,
    // Provided by VKSC_VERSION_1_0
    VK_MEMORY_HEAP_SEU_SAFE_BIT = 0x00000004,
} 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.

- **VK_MEMORY_HEAP_SEU_SAFE_BIT** specifies that the heap is protected against single event upsets.

---

**Note**
Many safety critical environments are required to contend with single event upsets (SEUs). It is typical for host memory to include automatic error detection (EDC) or correction (ECC) on platforms where this a concern. `VK_MEMORY_HEAP_SEU_SAFE_BIT` is used to denote device memory heaps that have this protection.

SEU-safe memory may have different performance characteristics than SEU-unsafe memory.

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

`VkMemoryHeapFlags` is a bitmask type for setting a mask of zero or more `VkMemoryHeapFlagBits`.

The `VkMemoryType` structure is defined as:

```c
// Provided by VK_VERSION_1_0
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:

```c
// Provided by VK_VERSION_1_0
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,
    // Provided by VK_VERSION_1_1
    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`.  

---

368
• **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.

```c
// 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:

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

### 11.2.3. Device Memory Allocation

To allocate memory objects, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkAllocateMemory(
    VkDevice device,       // Logical device
    const VkMemoryAllocateInfo* pAllocateInfo,       // Memory allocation info
    const VkAllocationCallbacks* pAllocator,       // Callbacks
    VkDeviceMemory* pMemory);       // Pointer to allocated memory
```

- **device** is the logical device that owns the memory.
- **pAllocateInfo** is a pointer to a **VkMemoryAllocateInfo** structure describing parameters of the
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**

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 `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 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 may have a limit on the maximum size of a single allocation. For example, certain systems may fail to create allocations with a size greater than or equal to 4GB. Such a limit is implementation-dependent, and if such a failure occurs then the error `VK_ERROR_OUT_OF_DEVICE_MEMORY` must be returned.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`,...
vkAllocateMemory must not return VK_ERROR_OUT_OF_HOST_MEMORY.

valid usage

- VUID-vkAllocateMemory-pAllocateInfo-01713
  pAllocateInfo->allocationSize must be less than or equal to
  VkPhysicalDeviceMemoryProperties::memoryHeaps[memindex].size where memindex =
  VkPhysicalDeviceMemoryProperties::memoryTypes[pAllocateInfo->memoryTypeIndex]
  .heapIndex as returned by vkGetPhysicalDeviceMemoryProperties
  for the VkPhysicalDevice that device was created from

- VUID-vkAllocateMemory-pAllocateInfo-01714
  pAllocateInfo->memoryTypeIndex must be less than
  VkPhysicalDeviceMemoryProperties::memoryTypeCount
  as returned by VkPhysicalDeviceMemoryProperties

- VUID-vkAllocateMemory-maxMemoryAllocationCount-04101
  There must be less than
  VkPhysicalDeviceLimits::maxMemoryAllocationCount
  device memory allocations currently allocated on the device

- VUID-vkAllocateMemory-device-05068
  The number of device memory objects currently allocated from
  device plus 1 must be less than or equal to
  the total number of device memory objects requested via
  VkDeviceObjectReservationCreateInfo::deviceMemoryRequestCount
  specified when device was created

valid usage (implicit)

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

- VUID-vkAllocateMemory-pAllocateInfo-parameter
  pAllocateInfo must be a valid pointer to a valid VkMemoryAllocateInfo structure

- VUID-vkAllocateMemory-pAllocator-null
  pAllocator must be NULL

- VUID-vkAllocateMemory-pMemory-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
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 a `VkStructureType` value identifying 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-allocationSize-07897
  `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 `VK_MEMORY_PROPERTY_PROTECTED_BIT`

- VUID-VkMemoryAllocateInfo-opaqueCaptureAddress-03329
  If `VkMemoryOpaqueCaptureAddressAllocateInfo::opaqueCaptureAddress` is not zero, `VkMemoryAllocateFlagsInfo::flags` must include `VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT`

- VUID-VkMemoryAllocateInfo-flags-03330
  If `VkMemoryAllocateFlagsInfo::flags` includes `VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT`, the `bufferDeviceAddressCaptureReplay` feature must be enabled

- VUID-VkMemoryAllocateInfo-flags-03331
  If `VkMemoryAllocateFlagsInfo::flags` includes `VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT`, the `bufferDeviceAddress` feature must be enabled

- VUID-VkMemoryAllocateInfo-opaqueCaptureAddress-03333
  If the parameters define an import operation, `VkMemoryOpaqueCaptureAddressAllocateInfo::opaqueCaptureAddress` must be zero
Valid Usage (Implicit)

- VUID-VkMemoryAllocateInfo-sType-sType
  
  **sType** must be **VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO**

- VUID-VkMemoryAllocateInfo-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 **VkExportMemoryAllocateInfo**, **VkMemoryAllocateFlagsInfo**, **VkMemoryDedicatedAllocateInfo**, or **VkMemoryOpaqueCaptureAddressAllocateInfo**

- VUID-VkMemoryAllocateInfo-sType-unique
  
  The **sType** value of each struct in the **pNext** chain **must** be unique

If the **pNext** chain includes a **VkMemoryDedicatedAllocateInfo** structure, then that structure includes a handle of the sole buffer or image resource that the memory can be bound to.

The **VkMemoryDedicatedAllocateInfo** structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkMemoryDedicatedAllocateInfo {
    VkStructureType sType;
    const void* pNext;
    VkImage image;
    VkBuffer buffer;
} VkMemoryDedicatedAllocateInfo;
```

- **sType** is a **VkStructureType** value identifying this structure.
- **pNext** is **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-02964
  
  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-02965
  
  If **buffer** is not **VK_NULL_HANDLE**, **VkMemoryAllocateInfo::allocationSize** must equal the **VkMemoryRequirements::size** of the buffer
If `buffer` is not `VK_NULL_HANDLE`, `buffer` must have been created without `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` set in `VkBufferCreateInfo::flags`.

### 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;
} VkExportMemoryAllocateInfo;
```

- `sType` is a `VkStructureType` value identifying 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`

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 a `VkStructureType` value identifying 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 `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT` is not set, the number of instances allocated depends on whether `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` is set in the memory heap. If `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` is set, then memory is allocated for every physical device in the logical device (as if `deviceMask` has bits set for all device indices). If `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` is not set, then a single instance of memory is allocated (as if `deviceMask` is set to one).

On some implementations, allocations from a multi-instance heap **may** consume memory on all physical devices even if the `deviceMask` excludes some devices. If `VkPhysicalDeviceGroupProperties::subsetAllocation` is `VK_TRUE`, then memory is only consumed for the devices in the device mask.

```
// Provided by VK_VERSION_1_1
typedef struct VkMemoryAllocateFlagsInfo {
    VkStructureType sType;
    const void* pNext;
    VkMemoryAllocateFlags flags;
    uint32_t deviceMask;
} VkMemoryAllocateFlagsInfo;
```

- **sType** is a `VkStructureType` value identifying 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 `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT` is not set, the number of instances allocated depends on whether `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` is set in the memory heap. If `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` is set, then memory is allocated for every physical device in the logical device (as if `deviceMask` has bits set for all device indices). If `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` is not set, then a single instance of memory is allocated (as if `deviceMask` is set to one).

On some implementations, allocations from a multi-instance heap **may** consume memory on all physical devices even if the `deviceMask` excludes some devices. If `VkPhysicalDeviceGroupProperties::subsetAllocation` is `VK_TRUE`, then memory is only consumed for the devices in the device mask.

*Note*

In practice, most allocations on a multi-instance heap will be allocated across all physical devices. Unicast allocation support is an optional optimization for a
Bits which **can** be set in `VkMemoryAllocateFlagsInfo::flags`, controlling device memory allocation, are:

```c
// 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;
```

- **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.

```c
// 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 a `VkStructureType` value identifying 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

Device memory **cannot** be freed [SCID-4]. If `VkPhysicalDeviceVulkanSC10Properties::deviceDestroyFreesMemory` is `VK_TRUE`, the memory is returned to the system when the device is destroyed.
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,       // Logical device that owns the memory.
    VkDeviceMemory memory, // Memory object to be mapped.
    VkDeviceSize offset,   // Zero-based byte offset from the beginning of the memory object.
    VkDeviceSize size,     // Size of the memory range to map, or VK_WHOLE_SIZE to map from offset to the end of the allocation.
    VkMemoryMapFlags flags, // Reserverd for future use.
    void** ppData);        // 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.
```

- `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*.

---

**Note**

It is an application error to call `vkMapMemory` on a memory object that is already *host mapped*.

**Note**

`vkMapMemory` will fail if the implementation is unable to allocate an appropriately sized contiguous virtual address range, e.g. due to virtual address space fragmentation or platform limits. In such cases, `vkMapMemory` must return `VK_ERROR_MEMORY_MAP_FAILED`. The application can improve the likelihood of success by reducing the size of the mapped range and/or removing unneeded mappings using `vkUnmapMemory`.

---

`vkMapMemory` does not check whether the device memory is currently in use before returning the
host-accessible pointer. The application must guarantee that any previously submitted command that writes to this range has completed before the host reads from or writes to that range, and that any previously submitted command that reads from that range has completed before the host writes to that region (see here for details on fulfilling such a guarantee). If the device memory was allocated without the VK_MEMORY_PROPERTY_HOST_COHERENT_BIT set, these guarantees must be made for an extended range: the application must round down the start of the range to the nearest multiple of VkPhysicalDeviceLimits::nonCoherentAtomSize, and round the end of the range up to the nearest multiple of VkPhysicalDeviceLimits::nonCoherentAtomSize.

While a range of device memory is host mapped, the application is responsible for synchronizing both device and host access to that memory range.

Note

It is important for the application developer to become meticulously familiar with all of the mechanisms described in the chapter on Synchronization and Cache Control as they are crucial to maintaining memory access ordering.

If VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations is VK_TRUE, vkMapMemory must not return VK_ERROR_OUT_OF_HOST_MEMORY.

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
Host Synchronization

- Host access to memory must be externally synchronized

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_MEMORY_MAP_FAILED

Two commands are provided to enable applications to work with non-coherent memory allocations: vkFlushMappedMemoryRanges and vkInvalidateMappedMemoryRanges.

Note
If the memory object was created with the VK_MEMORY_PROPERTY_HOST_COHERENT_BIT set, vkFlushMappedMemoryRanges and vkInvalidateMappedMemoryRanges are unnecessary and may have a performance cost. However, availability and visibility operations still need to be managed on the device. See the description of host access types for more information.

After a successful call to vkMapMemory the memory object memory is considered to be currently host mapped.

To flush ranges of non-coherent memory from the host caches, call:

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

VkResult vkFlushMappedMemoryRanges(
    VkDevice device,
    uint32_t memoryRangeCount,
    uint32_t offset,
    uint32_t rangeCount,
    VkMemoryMapFlags flags);
```
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).

If VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations is VK_TRUE, vkFlushMappedMemoryRanges must not return VK_ERROR_OUT_OF_HOST_MEMORY.

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.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkInvalidateMappedMemoryRanges` must not return `VK_ERROR_OUT_OF_HOST_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 a `VkStructureType` value identifying 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-00684
  `memory` must be currently host mapped

- VUID-VkMappedMemoryRange-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
If \( \text{size} \) is not equal to \( \text{VK\_WHOLE\_SIZE} \), \( \text{size} \) must either be a multiple of \( \text{VkPhysicalDeviceLimits::nonCoherentAtomSize} \), or \( \text{offset} \) plus \( \text{size} \) must equal the size of memory.

**Valid Usage (Implicit)**

- VUID-VkMappedMemoryRange-sType-sType
  - \( \text{sType} \) must be \( \text{VK\_STRUCTURE\_TYPE\_MAPPED\_MEMORY\_RANGE} \)
- VUID-VkMappedMemoryRange-pNext-pNext
  - \( \text{pNext} \) must be \( \text{NULL} \)
- VUID-VkMappedMemoryRange-memory-parameter
  - \( \text{memory} \) must be a valid \( \text{VkDeviceMemory} \) 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 \text{vkUnmapMemory}( 
    \text{VkDevice} \hspace{1em} \text{device}, 
    \text{VkDeviceMemory} \hspace{1em} \text{memory});
```

- \( \text{device} \) is the logical device that owns the memory.
- \( \text{memory} \) is the memory object to be unmapped.

**Valid Usage**

- VUID-vkUnmapMemory-memory-00689
  - \( \text{memory} \) must be currently host mapped

**Valid Usage (Implicit)**

- VUID-vkUnmapMemory-device-parameter
  - \( \text{device} \) must be a valid \( \text{VkDevice} \) handle
- VUID-vkUnmapMemory-memory-parameter
  - \( \text{memory} \) must be a valid \( \text{VkDeviceMemory} \) handle
- VUID-vkUnmapMemory-memory-parent
  - \( \text{memory} \) must have been created, allocated, or retrieved from \( \text{device} \)

**Host Synchronization**

- Host access to \( \text{memory} \) 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 may be out of date.

The implementation guarantees to allocate any committed memory from the `heapIndex` indicated by the memory type that the memory object was created with.

**Valid Usage**

- VUID-vkGetDeviceMemoryCommitment-mem0-00690

  `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
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 \( \text{VkPhysicalDeviceProtectedMemoryProperties}::\text{protectedNoFault} \) is \( \text{VK_FALSE} \), 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 \( \text{VkPhysicalDeviceProtectedMemoryProperties}::\text{protectedNoFault} \) is \( \text{VK_TRUE} \), 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:

```c
// Provided by VK_VERSION_1_1
void vkGetDeviceGroupPeerMemoryFeatures(
    VkDevice device,
    uint32_t heapIndex,
    uint32_t localDeviceIndex,
    uint32_t remoteDeviceIndex,
    VkPeerMemoryFeatureFlags* pPeerMemoryFeatures);
```

- \( \text{device} \) is the logical device that owns the memory.
- \( \text{heapIndex} \) is the index of the memory heap from which the memory is allocated.
- \( \text{localDeviceIndex} \) is the device index of the physical device that performs the memory access.
- \( \text{remoteDeviceIndex} \) is the device index of the physical device that the memory is allocated for.
- \( \text{pPeerMemoryFeatures} \) is a pointer to a \( \text{VkPeerMemoryFeatureFlags} \) 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:

```c
// 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
  - pInfo **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 a VkStructureType value identifying 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 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.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkCreateBuffer` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

### Valid Usage

- **VUID-vkCreateBuffer-device-05068**
  The number of buffers currently allocated from **device** plus 1 must be less than or equal to the total number of buffers requested via `VkDeviceObjectReservationCreateInfo ::bufferRequestCount` specified when **device** was created.
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-null**
  - `pAllocator` must be `NULL`

- **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 a `VkStructureType` value identifying 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 `vkGetPhysicalDeviceQueueFamilyProperties2` or `vkGetPhysicalDeviceQueueFamilyProperties` for the `physicalDevice` that was used to create `device`

- **VUID-VkBufferCreateInfo-flags-00915**
  - `flags` **must** not contain `VK_BUFFER_CREATE_SPARSE_BINDING_BIT`

- **VUID-VkBufferCreateInfo-flags-00916**
  - `flags` **must** not contain `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT`

- **VUID-VkBufferCreateInfo-flags-00917**
  - `flags` **must** not contain `VK_BUFFER_CREATE_SPARSE_ALIASED_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-opaqueCaptureAddress-03337**
  - If `VkBufferOpaqueCaptureAddressCreateInfo::opaqueCaptureAddress` is not zero, `flags` **must** include `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT`

- **VUID-VkBufferCreateInfo-flags-03338**
  - If `flags` includes `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT`, the `bufferDeviceAddressCaptureReplay` feature **must** be enabled

- **VUID-VkBufferCreateInfo-None-09205**
  - `usage` **must** be a valid combination of `VkBufferUsageFlagBits` values
Valid Usage (Implicit)

- **VUID-VkBufferCreateInfo-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO`

- **VUID-VkBufferCreateInfo-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 `VkBufferOpaqueCaptureAddressCreateInfo` or `VkExternalMemoryBufferCreateInfo`

- **VUID-VkBufferCreateInfo-sType-unique**
  - The `sType` value of each struct in the `pNext` chain must be unique

- **VUID-VkBufferCreateInfo-flags-parameter**
  - `flags` must be a valid combination of `VkBufferCreateFlagBits` values

- **VUID-VkBufferCreateInfo-sharingMode-parameter**
  - `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,

    // Provided by VK_VERSION_1_2
    VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT = 0x00020000,
}
VkBufferUsageFlagBits;
```

- **VK_BUFFER_USAGE_TRANSFER_SRC_BIT** specifies that the buffer can be used as the source of a transfer command (see the definition of `VK_PIPELINE_STAGE_TRANSFER_BIT`).

- **VK_BUFFER_USAGE_TRANSFER_DST_BIT** specifies that the buffer can be used as the destination of a transfer command.

- **VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT** specifies that the buffer can be used to create a `VkBufferView` suitable for occupying a `VkDescriptorSet` slot of type `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER`.

- **VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT** specifies that the buffer can be used to create a
VkBufferView suitable for occupying a VkDescriptorSet slot of type 
VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER.

- **VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT** specifies that the buffer can be used in a 
  VkDescriptorBufferInfo suitable for occupying a VkDescriptorSet slot either of type 
  VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC.

- **VK_BUFFER_USAGE_STORAGE_BUFFER_BIT** specifies that the buffer can be used in a 
  VkDescriptorBufferInfo suitable for occupying a VkDescriptorSet slot either of type 
  VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC.

- **VK_BUFFER_USAGE_INDEX_BUFFER_BIT** specifies that the buffer is suitable for passing as the buffer 
  parameter to vkCmdBindIndexBuffer.

- **VK_BUFFER_USAGE_VERTEX_BUFFER_BIT** specifies that the buffer is suitable for passing as an element 
  of the pBuffers array to vkCmdBindVertexBuffers.

- **VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT** specifies that the buffer is suitable for passing as the buffer 
  parameter to vkCmdDrawIndirect, vkCmdDrawIndexedIndirect, or vkCmdDispatchIndirect.

- **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.

```c
// 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:

```c
// 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,
    // Provided by VK_VERSION_1_1
    VK_BUFFER_CREATE.Protected_BIT = 0x00000008,
    // Provided by VK_VERSION_1_2
    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. This flag is not supported in Vulkan SC [SCID-8].

- **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. This flag is not supported in Vulkan SC [SCID-8].

- **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. This flag is not supported in Vulkan SC [SCID-8].

- 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.

```c
// Provided by VK_VERSION_1_0
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:

```c
// Provided by VK_VERSION_1_1
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 a VkStructureType value identifying 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:

```
// Provided by VK_VERSION_1_2
typedef struct VkBufferOpaqueCaptureAddressCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint64_t opaqueCaptureAddress;
} VkBufferOpaqueCaptureAddressCreateInfo;
```

- `sType` is a `VkStructureType` value identifying 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`
// 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

**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-null
  **pAllocator** must be NULL
- 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.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkCreateBufferView` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

### Valid Usage

- **VUID-vkCreateBufferView-device-05068**
  The number of buffer views currently allocated from `device` plus 1 must be less than or equal to the total number of buffer views requested via `VkDeviceObjectReservationCreateInfo::bufferViewRequestCount` specified when `device` was created.

### 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-null**
  `pAllocator` must be `NULL`

- **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 a VkStructureType value identifying 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.

The buffer view has a buffer view usage identifying which descriptor types can be created from it. This usage is equal to the VkBufferCreateInfo::usage value used to create `buffer`.

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 \( \lceil \frac{\text{range}}{\text{texel block size}} \rceil \times (\text{texels per block}) \) where 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 \( \lceil \frac{(\text{size} - \text{offset})}{\text{texel block size}} \rceil \times (\text{texels per block}) \) 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-format-08778
  If the buffer view usage contains VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT, then format features of format must contain VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT

- VUID-VkBufferViewCreateInfo-format-08779
  If the buffer view usage contains VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT, then format features of format must contain VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT

- 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
  offset must be a multiple of VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment

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-zero bitmask
  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
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:

```c
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

### 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-null`
  `pAllocator` must be `NULL`

- `VUID-vkDestroyBufferView-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
12.2.1. Buffer View Format Features

Valid uses of a VkBufferView may depend on the buffer view's format features, defined below. Such constraints are documented in the affected valid usage statement.

- The buffer view's set of format features is the value of VkFormatProperties::bufferFeatures found by calling vkGetPhysicalDeviceFormatProperties on the same format as VkBufferViewCreateInfo::format.

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:

// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkImage)

To create images, call:

// Provided by VK_VERSION_1_0
VkResult vkCreateImage(
    VkDevice device,           // Provided by VK_VERSION_1_0
    const VkImageCreateInfo* pCreateInfo, // Provided by VK_VERSION_1_0
    const VkAllocationCallbacks* pAllocator, // Provided by VK_VERSION_1_0
    VkImage* pImage); // Provided by VK_VERSION_1_0

- 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.

If VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations is VK_TRUE, vkCreateImage must not return VK_ERROR_OUT_OF_HOST_MEMORY.

Valid Usage

- VUID-vkCreateImage-device-05068
  The number of images currently allocated from device plus 1 must be less than or equal to the total number of images requested via VkDeviceObjectReservationCreateInfo::imageRequestCount specified when device was created.
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-null**
  pAllocator** must** be **NULL**

- **VUID-vkCreateImage-pImage-parameter**
  pImage** must** be a valid pointer to a **VkImage** handle

Return Codes

**Success**
- **VK_SUCCESS**

**Failure**
- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**

The **VkImageCreateInfo** structure is defined as:

```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 a **VkStructureType** value identifying 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 `vkGetPhysicalDeviceExternalImageFormatProperties2` 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-08865
  If flags contains VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT, extent.width and extent.height must be equal

- VUID-VkImageCreateInfo-flags-08866
  If flags contains VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT, arrayLayers must be greater than or equal to 6

- VUID-VkImageCreateInfo-flags-00950
  If flags contains VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT, imageType must be VK_IMAGE_TYPE_3D

- VUID-VkImageCreateInfo-flags-09403
  If flags contains VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT, flags must not include VK_IMAGE_CREATE_SPARSE_ALIASED_BIT, VK_IMAGE_CREATE_SPARSE_BINDING_BIT, or VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT

- 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-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-05062
  flags must not contain VK_IMAGE_CREATE_SPARSE_BINDING_BIT,
VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT, VK_IMAGE_CREATE_SPARSE_ALIASED_BIT, or VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT

- VUID-VkImageCreateInfo-flags-01890
  If the protectedMemory feature is not enabled, flags must not contain VK_IMAGE_CREATE_PROTECTED_BIT

- VUID-VkImageCreateInfo-pNext-00990
  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

- 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-imageCreateInfoFormatFeatures-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_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT, and the pNext chain includes a VkImageStencilUsageCreateInfo structure, then its VkImageStencilUsageCreateInfo ::stencilUsage member must also not include VK_IMAGE_USAGE_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_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`, `VkImageFormatListCreateInfo`, or `VkImageStencilUsageCreateInfo`

- VUID-VkImageCreateInfo-sType-unique
  The `sType` value of each struct in the `pNext` chain must be unique

- VUID-VkImageCreateInfo-flags-parameter
  `flags` must be a valid combination of `VkImageCreateFlagBits` values

- VUID-VkImageCreateInfo-imageType-parameter
  `imageType` must be a valid `VkImageType` value

- VUID-VkImageCreateInfo-format-parameter
  `format` must be a valid `VkFormat` value

- VUID-VkImageCreateInfo-samples-parameter
  `samples` must be a valid `VkSampleCountFlagBits` value

- VUID-VkImageCreateInfo-tiling-parameter
  `tiling` must be a valid `VkImageTiling` value

- VUID-VkImageCreateInfo-usage-parameter
  `usage` must be a valid combination of `VkImageUsageFlagBits` values

- VUID-VkImageCreateInfo-usage-requiredbitmask
  `usage` must not be 0

- VUID-VkImageCreateInfo-sharingMode-parameter
  `sharingMode` must be a valid `VkSharingMode` value

- VUID-VkImageCreateInfo-initialLayout-parameter
  `initialLayout` must be a valid `VkImageLayout` value

The `VkImageStencilUsageCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkImageStencilUsageCreateInfo {
    VkStructureType     sType;
    ...  // Other members defined
};
```
const void* pNext;
VkImageUsageFlags stencilUsage;
} VkImageStencilUsageCreateInfo;

• **sType** is a `VkStructureType` value identifying 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-requiredbitmapmask**
  **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 a `VkStructureType` value identifying 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 a `VkStructureType` value identifying 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.

// 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_STORAGE_BIT**
• **VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT**
• **VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_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:

// 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_0
};
• **VK_IMAGE_CREATE_SPARSE_BINDING_BIT** specifies that the image will be backed using sparse memory binding. This flag is not supported in Vulkan SC [SCID-8].

• **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. This flag is not supported in Vulkan SC [SCID-8].

• **VK_IMAGE_CREATE_SPARSE_ALIASED_BIT** specifies that the image will be backed using sparse memory binding with memory ranges that might also simultaneously be backing another image (or another portion of the same image). Images created with this flag must also be created with the **VK_IMAGE_CREATE_SPARSE_BINDING_BIT** flag. This flag is not supported in Vulkan SC [SCID-8].

• **VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT** specifies that the image can be used to create a **VkImageView** with a different format from the image. For multi-planar formats, **VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT** specifies that a **VkImageView** can be created of a plane of the image.

• **VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT** specifies that the image can be used to create a **VkImageView** of type **VK_IMAGE_VIEW_TYPE_CUBE** or **VK_IMAGE_VIEW_TYPE_CUBE_ARRAY**.

• **VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT** specifies that the image can be used to create a **VkImageView** of type **VK_IMAGE_VIEW_TYPE_2D** or **VK_IMAGE_VIEW_TYPE_2D_ARRAY**.

• **VK_IMAGE_CREATE_PROTECTED_BIT** specifies that the image is a protected image.

• **VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT** specifies that the image can be used with a non-zero value of the **splitInstanceBindRegionCount** member of a **VkBindImageMemoryDeviceGroupInfo** structure passed into **vkBindImageMemory2**. This flag also has the effect of making the image use the standard sparse image block dimensions. This flag is not supported in Vulkan SC [SCID-8].

• **VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT** specifies that the image having a compressed format can be used to create a **VkImageView** with an uncompressed format where each texel in the image view corresponds to a compressed texel block of the image.

• **VK_IMAGE_CREATE_EXTENDED_USAGE_BIT** specifies that the image can be created with usage flags that are not supported for the format the image is created with but are supported for at least one format a **VkImageView** created from the image can have.

• **VK_IMAGE_CREATE_DISJOINT_BIT** specifies that an image with a multi-planar format must have each plane separately bound to memory, rather than having a single memory binding for the whole image; the presence of this bit distinguishes a disjoint image from an image without this bit set.

• **VK_IMAGE_CREATE_ALIAS_BIT** specifies that two images created with the same creation parameters and aliased to the same memory can interpret the contents of the memory consistently with each other, subject to the rules described in the Memory Aliasing section. This flag further specifies that each plane of a disjoint image can share an in-memory non-linear representation
with single-plane images, and that a single-plane image can share an in-memory non-linear representation with a plane of a multi-planar disjoint image, according to the rules in Compatible Formats of Planes of Multi-Planar Formats. If the pNext chain includes a VkExternalMemoryImageCreateInfo structure whose handleTypes member is not 0, it is as if VK_IMAGE_CREATE_ALIAS_BIT is set.

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

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

VkImageCreateFlags is a bitmask type for setting a mask of zero or more VkImageCreateFlagBits.

Possible values of VkImageCreateInfo::imageType, specifying the basic dimensionality of an image, are:

```c
// Provided by VK_VERSION_1_0
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:

```c
// Provided by VK_VERSION_1_0
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:

```c
// Provided by VK_VERSION_1_0
void vkGetImageSubresourceLayout(
```
VkDevice
VkImage
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-07789
  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 image

• VUID-vkGetImageSubresourceLayout-arrayLayer-01717
  The arrayLayer member of pSubresource must be less than the arrayLayers specified in image

• VUID-vkGetImageSubresourceLayout-format-08886
  If format of the image is a color format that is not a multi-planar image format, and tiling of the image is VK_IMAGE_TILING_LINEAR or VK_IMAGE_TILING_OPTIMAL, the aspectMask member of pSubresource must be VK_IMAGE_ASPECT_COLOR_BIT

• VUID-vkGetImageSubresourceLayout-format-04462
  If format of the image has a depth component, the aspectMask member of pSubresource must contain VK_IMAGE_ASPECT_DEPTH_BIT

• VUID-vkGetImageSubresourceLayout-format-04463
  If format of the image has a stencil component, the aspectMask member of pSubresource must contain VK_IMAGE_ASPECT_STENCIL_BIT

• VUID-vkGetImageSubresourceLayout-format-04464
  If format of the image 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
If the **tiling** of the image is **VK_IMAGE_TILING_LINEAR** and has a multi-planar image format, then the **aspectMask** member of **pSubresource** must be a single valid multi-planar aspect mask 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**
  
  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:

```c
// Provided by VK_VERSION_1_0
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

### 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-null
  pAllocator must be NULL

- 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 Mip Level Sizing

A complete mipmap chain is the full set of mip levels, from the largest mip level provided, down to the minimum mip level size.

Conventional Images

For conventional images, the dimensions of each successive mip level, n+1, are:

\[
\text{width}_{n+1} = \max(\lceil \text{width}_n/2 \rceil, 1)
\]

\[
\text{height}_{n+1} = \max(\lceil \text{height}_n/2 \rceil, 1)
\]

\[
\text{depth}_{n+1} = \max(\lceil \text{depth}_n/2 \rceil, 1)
\]

where \(\text{width}_n, \text{height}_n,\) and \(\text{depth}_n\) are the dimensions of the next larger mip level, \(n\).

The minimum mip level 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) mip level, 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 \( \text{vkCmdPipelineBarrier} \) or a \( \text{vkCmdWaitEvents} \) command buffer command (see Image Memory Barriers), or as part of a subpass dependency within a render pass (see \( \text{VkSubpassDependency} \)).

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 \( \text{separateDepthStencilLayouts} \) 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 \( \text{VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL} \) 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 \( \text{VkImageCreateInfo}::\text{initialLayout} \) member. The initialLayout must be either \( \text{VK_IMAGE_LAYOUT_UNDEFINED} \) or \( \text{VK_IMAGE_LAYOUT_PREINITIALIZED} \). If it is \( \text{VK_IMAGE_LAYOUT_PREINITIALIZED} \), 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 \( \text{VK_IMAGE_LAYOUT_UNDEFINED} \), 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 \( \text{VK_IMAGE_LAYOUT_PREINITIALIZED} \) or \( \text{VK_IMAGE_LAYOUT_GENERAL} \) layout. Calling \( \text{vkGetImageSubresourceLayout} \) 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:

```
// Provided by VK_VERSION_1_0
```
```c
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
    VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL = 1000117000,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL = 1000117001,
    // Provided by VK_VERSION_1_2
    VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL = 1000241000,
    // Provided by VK_VERSION_1_2
    VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL = 1000241001,
    // Provided by VK_VERSION_1_2
    VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL = 1000241002,
    // Provided by VK_VERSION_1_2
    VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL = 1000241003,
} VkImageLayout;
```

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_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. It is equivalent to `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL` and `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`.

- `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 `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL` and `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL`.

- `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 `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL` and `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`.

- `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL` specifies a layout for the depth aspect of a depth/stencil format image allowing read and write access as a depth attachment.

- `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL` 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.

- `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.

- `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.

- `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 `VK_IMAGE_USAGE_SAMPLED_BIT` or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT` usage bits enabled.

- `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` must only be used as a source image of a transfer command (see the definition of `VK_PIPELINE_STAGE_TRANSFER_BIT`). This layout is valid only for image subresources of images created with the `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage bit enabled.

- `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` must only be used as a destination image of a transfer command. This layout is valid only for image subresources of images created with the `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage bit enabled.

The layout of each image subresource is not a state of the image subresource itself, but is rather a property of how the data in memory is organized, and thus for each mechanism of accessing an image in the API the application must specify a parameter or structure member that indicates which image layout the image subresource(s) are considered to be in when the image will be accessed. For transfer commands, this is a parameter to the command (see Clear Commands and Copy Commands). For use as a framebuffer attachment, this is a member in the substructures of the VkRenderPassCreateInfo (see Render Pass). For use in a descriptor set, this is a member in the VkDescriptorImageInfo structure (see Descriptor Set Updates).
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
// Provided by VK_VERSION_1_0
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.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkCreateImageView` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

**Valid Usage**

- **VUID-vkCreateImageView-image-09179**
  `VkImageViewCreateInfo::image` must have been created from `device`

- **VUID-vkCreateImageView-device-05068**
  The number of image views currently allocated from `device` plus 1 must be less than or equal to the total number of image views requested via `VkDeviceObjectReservationCreateInfo::imageViewRequestCount` specified when `device` was created.

- **VUID-vkCreateImageView-subresourceRange-05063**
  If `VkImageViewCreateInfo::subresourceRange.layerCount` is not `VK_REMAINING_ARRAY_LAYERS` and is greater than 1, or if `VkImageViewCreateInfo::subresourceRange.layerCount` is `VK_REMAINING_ARRAY_LAYERS` and the remaining number of layers in `VkImageViewCreateInfo::image` is greater than 1, the number of image views with more
than one array layer currently allocated from device plus 1 must be less than or equal to
the total number of image views requested via VkDeviceObjectReservationCreateInfo::layeredImageViewRequestCount specified when device was created

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-null
pAllocator must be NULL
- 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 a VkStructureType value identifying 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.

- 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.

If **image** was created with a multi-planar format, and the image view’s **aspectMask** is one of **VK_IMAGE_ASPECT_PLANE_0_BIT**, **VK_IMAGE_ASPECT_PLANE_1_BIT** or **VK_IMAGE_ASPECT_PLANE_2_BIT**, the view’s aspect mask is considered to be equivalent to **VK_IMAGE_ASPECT_COLOR_BIT** when used as a framebuffer attachment.

**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’C_bC_r 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, subresourceRange.aspectMask is VK_IMAGE_ASPECT_COLOR_BIT, and usage includes VK_IMAGE_USAGE_SAMPLED_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_plane, v_plane) unnormalized coordinates of the reduced-resolution plane, and these coordinates access the same memory locations as the (u_color, v_color) unnormalized coordinates of the color aspect for which chroma reconstruction operations operate on the same (u_plane, v_plane) or (i_plane, j_plane) coordinates.

Table 8. 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_1D</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>Image View Type</td>
<td>Compatible Image Types</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>------------------------</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 `image` was not created with `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT` then `viewType` must not be `VK_IMAGE_VIEW_TYPE_CUBE` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`

- VUID-VkImageViewCreateInfo-viewType-01004
  If the `imageCubeArray` feature is not enabled, `viewType` must not be `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`

- VUID-VkImageViewCreateInfo-image-06723
  If `image` was created with `VK_IMAGE_TYPE_3D` but without `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT` set then `viewType` must not be `VK_IMAGE_VIEW_TYPE_2D_ARRAY`

- VUID-VkImageViewCreateInfo-image-06727
  If `image` was created with `VK_IMAGE_TYPE_3D` but without `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT` set then `viewType` must not be `VK_IMAGE_VIEW_TYPE_2D`

- VUID-VkImageViewCreateInfo-image-04970
  If `image` was created with `VK_IMAGE_TYPE_3D` and `viewType` is `VK_IMAGE_VIEW_TYPE_2D` or `VK_IMAGE_VIEW_TYPE_2D_ARRAY` then `subresourceRange.levelCount` must be 1

- VUID-VkImageViewCreateInfo-image-04972
  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`

- VUID-VkImageViewCreateInfo-image-04441
  `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

- VUID-VkImageViewCreateInfo-None-02273
  The `format features` of the resultant image view must contain at least one bit

- VUID-VkImageViewCreateInfo-usage-02274
  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`

- VUID-VkImageViewCreateInfo-usage-02275
  If `usage` contains `VK_IMAGE_USAGE_STORAGE_BIT`, then the image view's `format features` must contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT`

- VUID-VkImageViewCreateInfo-usage-02276
  If `usage` contains `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`, then the image view's `format features` must contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`

- VUID-VkImageViewCreateInfo-usage-02277
  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`
• VUID-VkImageViewCreateInfo-usage-08932
  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

• VUID-VkImageViewCreateInfo-subresourceRange-01478
  subresourceRange.baseMipLevel must be less than the mipLevels specified in VkImageCreateInfo when image was created

• VUID-VkImageViewCreateInfo-subresourceRange-01718
  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-VkImageViewCreateInfo-image-01482
  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

• VUID-VkImageViewCreateInfo-subresourceRange-01483
  If subresourceRange.layerCount is not VK_REMAINING_ARRAY_LAYERS, image is not a 3D image created with VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT set, or viewType is not VK_IMAGE_VIEW_TYPE_2D or VK_IMAGE_VIEW_TYPE_2D_ARRAY, subresourceRange.layerCount must be non-zero and subresourceRange.baseArrayLayer + subresourceRange.layerCount must be less than or equal to the arrayLayers specified in VkImageCreateInfo when image was created

• 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 Mip Level 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 Mip Level 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 the multi-planar aspect mask bits, 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-subresourceRange-07818
  subresourceRange.aspectMask must only have at most 1 valid multi-planar aspect mask bit

- 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-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 `VkImageStencilUsageCreateInfo` 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_ARRAY` 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

- VUID-VkImageViewCreateInfo-subresourceRange-05064
  If subresourceRange.levelCount is not VK_REMAINING_MIP_LEVELS, subresourceRange.levelCount must be less than or equal to VkDeviceObjectReservationCreateInfo::maxImageViewMipLevels

- VUID-VkImageViewCreateInfo-subresourceRange-05200
  If subresourceRange.levelCount is VK_REMAINING_MIP_LEVELS, the remaining number of mip levels must be less than or equal to VkDeviceObjectReservationCreateInfo::maxImageViewMipLevels

- VUID-VkImageViewCreateInfo-subresourceRange-05065
  If subresourceRange.layerCount is not VK_REMAINING_ARRAY_LAYERS, subresourceRange.layerCount must be less than or equal to VkDeviceObjectReservationCreateInfo::maxImageViewArrayLayers

- VUID-VkImageViewCreateInfo-subresourceRange-05201
  If subresourceRange.layerCount is VK_REMAINING_ARRAY_LAYERS, the remaining number of layers must be less than or equal to VkDeviceObjectReservationCreateInfo::maxImageViewArrayLayers

- VUID-VkImageViewCreateInfo-subresourceRange-05066
  If subresourceRange.layerCount is not VK_REMAINING_ARRAY_LAYERS and is greater than 1, or if subresourceRange.layerCount is VK_REMAINING_ARRAY_LAYERS and the remaining number of layers is greater than 1, then if subresourceRange.levelCount is not VK_REMAINING_MIP_LEVELS, subresourceRange.levelCount must be less than or equal to VkDeviceObjectReservationCreateInfo::maxLayeredImageViewMipLevels

- VUID-VkImageViewCreateInfo-subresourceRange-05202
  If subresourceRange.layerCount is not VK_REMAINING_ARRAY_LAYERS and is greater than 1, or if subresourceRange.layerCount is VK_REMAINING_ARRAY_LAYERS and the remaining number of layers is greater than 1, then if subresourceRange.levelCount is VK_REMAINING_MIP_LEVELS, the remaining number of mip levels must be less than or equal to VkDeviceObjectReservationCreateInfo::maxLayeredImageViewMipLevels

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 a VkStructureType value identifying 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-requiredbitmask**
  
  `usage` **must** not be 0

The `VkImageSubresourceRange` structure is defined as:

```c
// Provided by VK_VERSION_1_0
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-required bitmask**
  `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:
// 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,
} VkImageAspectFlagBits;

• VK_IMAGE_ASPECT_COLOR_BIT specifies the color aspect.
• VK_IMAGE_ASPECT_DEPTH_BIT specifies the depth aspect.
• VK_IMAGE_ASPECT_STENCIL_BIT specifies the stencil aspect.
• VK_IMAGE_ASPECT_METADATA_BIT specifies the metadata aspect used for sparse 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.

// 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:

// 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`  
- **VUID-VkComponentMapping-b-parameter**  
  `b must be a valid VkComponentSwizzle value`  
- **VUID-VkComponentMapping-a-parameter**  
  `a must be a valid VkComponentSwizzle value`

Possible values of the members of VkComponentMapping, specifying the component values placed in each component of the output vector, are:

```c
// 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>
<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

**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-null
  `pAllocator` must be `NULL`
- 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 `VkImageViewCreateInfo::image` was created with `VK_IMAGE_TILING_LINEAR`, then the image view’s set of `format features` is the value of `VkFormatProperties::linearTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` on the same `format` as `VkImageViewCreateInfo::format`.

- If `VkImageViewCreateInfo::image` was created with `VK_IMAGE_TILING_OPTIMAL`, then the image view’s set of `format features` is the value of `VkFormatProperties::optimalTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` on the same `format` as `VkImageViewCreateInfo::format`.

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,
    VkBuffer buffer,
    VkMemoryRequirements* pMemoryRequirements);
```

- `device` is the logical device that owns the buffer.
- `buffer` is the buffer to query.
- `pMemoryRequirements` is a pointer to a `VkMemoryRequirements` structure in which the memory requirements of the buffer object are returned.

**Valid Usage (Implicit)**

- 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 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 and the VK_MEMORY_PROPERTY_HOST_COHERENT_BIT bit set. In other words, mappable coherent memory can always be attached to these objects.
- If buffer was created with VkExternalMemoryBufferCreateInfo::handleTypes set to 0 or image was created with VkExternalMemoryImageCreateInfo::handleTypes set to 0, the memoryTypeBits member always contains at least one bit set corresponding to a VkMemoryType with a propertyFlags that has the VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT bit set.
- The memoryTypeBits member is identical for all VkBuffer objects created with the same value for the flags and usage members in the VkBufferCreateInfo structure and the handleTypes member of the VkExternalMemoryBufferCreateInfo structure passed to vkCreateBuffer. Further, if
usage1 and usage2 of type VkBufferUsageFlags are such that the bits set in usage2 are a subset of the bits set in usage1, and they have the same flags and VkExternalMemoryBufferCreateInfo::handleTypes, then the bits set in memoryTypeBits returned for usage1 must be a subset of the bits set in memoryTypeBits returned for usage2, for all values of flags.

- The alignment member is a power of two.
- The alignment member is identical for all VkBuffer objects created with the same combination of values for the usage and flags members in the VkBufferCreateInfo structure passed to vkCreateBuffer.
- The alignment member satisfies the buffer descriptor offset alignment requirements associated with the VkBuffer’s usage:
  - If usage included VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT or VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT, alignment must be an integer multiple of VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment.
  - If usage included VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT, alignment must be an integer multiple of VkPhysicalDeviceLimits::minUniformBufferOffsetAlignment.
  - If usage included VK_BUFFER_USAGE_STORAGE_BUFFER_BIT, alignment must be an integer multiple of VkPhysicalDeviceLimits::minStorageBufferOffsetAlignment.
- For images created with a color format, the memoryTypeBits member is identical for all VkImage objects created with the same combination of values for the tiling member, the VK_IMAGE_CREATE_SPARSE_BINDING_BIT bit of the flags member, the VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT bit of the flags member, handleTypes member of VkExternalMemoryImageCreateInfo, and the VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT of the usage member in the VkImageCreateInfo structure passed to vkCreateImage.
- For images created with a depth/stencil format, the memoryTypeBits member is identical for all VkImage objects created with the same combination of values for the format member, the tiling member, the VK_IMAGE_CREATE_SPARSE_BINDING_BIT bit of the flags member, the VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT bit of the flags member, handleTypes member of VkExternalMemoryImageCreateInfo, and the VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT of the usage member in the VkImageCreateInfo structure passed to vkCreateImage.
- If the memory requirements are for a VkImage, the memoryTypeBits member must not refer to a VkMemoryType with a propertyFlags that has the VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT bit set if the image did not have VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT bit set in the usage member of the VkImageCreateInfo structure passed to vkCreateImage.
- If the memory requirements are for a VkBuffer, the memoryTypeBits member must not refer to a VkMemoryType with a propertyFlags that has the VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT bit set.

Note
The implication of this requirement is that lazily allocated memory is disallowed for buffers in all cases.

- The size member is identical for all VkBuffer objects created with the same combination of creation parameters specified in VkBufferCreateInfo and its pNext chain.
- The size member is identical for all VkImage objects created with the same combination of
creation parameters specified in VkImageCreateInfo and its pNext chain.

Note
This, however, does not imply that they interpret the contents of the bound memory identically with each other. That additional guarantee, however, can be explicitly requested using VK_IMAGE_CREATE_ALIAS_BIT.

To determine the memory requirements for a buffer resource, call:

```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

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 a VkStructureType value identifying 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

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

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 a VkStructureType value identifying 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**
  The `sType` must be `VK_STRUCTURE_TYPE_IMAGE_MEMORY_REQUIREMENTS_INFO_2`.

- **VUID-VkImageMemoryRequirementsInfo2-pNext-pNext**
  The `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**
  The `image` must be a valid `VkImage` handle.

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 a `VkStructureType` value identifying 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 multi-planar aspect mask 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 a `VkStructureType` value identifying 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:
typedef struct VkMemoryDedicatedRequirements {
    VkStructureType sType;
    void* pNext;
    VkBool32 prefersDedicatedAllocation;
    VkBool32 requiresDedicatedAllocation;
} VkMemoryDedicatedRequirements;

• sType is a VkStructureType value identifying 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 VkExternalImageFormatProperties2::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

To attach memory to a buffer object, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkBindBufferMemory(
    VkDevice device,            // device is the logical device that owns the buffer and memory.
    VkBuffer buffer,           // buffer is the buffer to be attached to memory.
    VkDeviceMemory memory,     // memory is a VkDeviceMemory object describing the device memory to attach.
    VkDeviceSize memoryOffset  // 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.

If VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations is VK_TRUE, vkBindBufferMemory must not return VK_ERROR_OUT_OF_HOST_MEMORY.

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
The size member of the VkMemoryRequirements structure returned from a call to
vkGetBufferMemoryRequirements with buffer must be less than or equal to the size of memory
minus memoryOffset

If buffer requires a dedicated allocation (as reported by
vkGetBufferMemoryRequirements2 in VkMemoryDedicatedRequirements::requiresDedicatedAllocation for buffer), memory must have been allocated with
VkMemoryDedicatedAllocateInfo::buffer equal to buffer

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

If buffer was created with the VK_BUFFER_CREATE_PROTECTED_BIT bit set, the buffer must be bound to a memory object allocated with a memory type that reports
VK_MEMORY_PROPERTY_PROTECTED_BIT

If buffer was created with the VK_BUFFER_CREATE_PROTECTED_BIT bit not set, the buffer must not be bound to a memory object allocated 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 VkExternalMemoryBufferCreateInfo::handleTypes when buffer was created

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

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

If the VkPhysicalDeviceBufferDeviceAddressFeatures::bufferDeviceAddressCaptureReplay feature is enabled and buffer was created with the
VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT bit set, memory must have been allocated with the VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_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

Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

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

```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.
Note

If `vkBindBufferMemory2` fails, and `bindInfoCount` was greater than one, then the buffers referenced by `pBindInfos` will be in an indeterminate state, and must not be used. Applications should destroy these buffers.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkBindBufferMemory2` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

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

Return Codes

Success

- `VK_SUCCESS`

Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

`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 a `VkStructureType` value identifying 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-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-buf-07459**
  - **buffer** must not have been bound to a memory object

- **VUID-VkBindBufferMemoryInfo-buf-01030**
  - **buffer** must not have been created with any sparse memory binding flags

- **VUID-VkBindBufferMemoryInfo-buf-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-VkBindBufferMemoryInfo-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-VkBindBufferMemoryInfo-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-VkBindBufferMemoryInfo-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`
VK_MEMORY_PROPERTY_PROTECTED_BIT

• VUID-VkBindBufferMemoryInfo-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-VkBindBufferMemoryInfo-memory-02985
  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-VkBindBufferMemoryInfo-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

• VUID-VkBindBufferMemoryInfo-bufferDeviceAddressCaptureReplay-09200
  If the VkPhysicalDeviceBufferDeviceAddressFeatures::bufferDeviceAddressCaptureReplay feature is enabled and buffer was created with the VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT bit set, memory must have been allocated with the VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT bit set

• VUID-VkBindBufferMemoryInfo-pNext-01605
  If the pNext chain includes a VkBindBufferMemoryDeviceGroupInfo structure, all instances of memory specified by VkBindBufferMemoryDeviceGroupInfo::pDeviceIndices must have been allocated

Valid Usage (Implicit)

• VUID-VkBindBufferMemoryInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO

• VUID-VkBindBufferMemoryInfo-pNext-pNext
  pNext must be NULL or a pointer to a valid instance of 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:
typedef struct VkBindBufferMemoryDeviceGroupInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t deviceIndexCount;
    const uint32_t* pDeviceIndices;
} VkBindBufferMemoryDeviceGroupInfo;

- **sType** is a VkStructureType value identifying 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

- VUID-VkBindBufferMemoryDeviceGroupInfo-pDeviceIndices-01607
  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:

```c
// 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.

If VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations is VK_TRUE, vkBindImageMemory must not return VK_ERROR_OUT_OF_HOST_MEMORY.

### 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

- VUID-vkBindImageMemory-memory-02628
  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-vkBindImageMemory-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-vkBindImageMemory-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-vkBindImageMemory-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-vkBindImageMemory-memory-02989
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-vkBindImageMemory-image-01608
image must not have been created with the VK_IMAGE_CREATE_DISJOINT_BIT set

• VUID-vkBindImageMemory-memory-01047
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

• VUID-vkBindImageMemory-memoryOffset-01048
memoryOffset must be an integer multiple of the alignment member of the VkMemoryRequirements structure returned from a call to vkGetImageMemoryRequirements with image

• VUID-vkBindImageMemory-size-01049
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)

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

• VUID-vkBindImageMemory-image-parameter
image must be a valid VkImage handle

• VUID-vkBindImageMemory-memory-parameter
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:

```c
// 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.

**Note**
If vkBindImageMemory2 fails, and bindInfoCount was greater than one, then the images referenced by pBindInfos will be in an indeterminate state, and must not be used. Applications should destroy these images.

If **VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations** is VK_TRUE, **vkBindImageMemory2** must not return VK_ERROR_OUT_OF_HOST_MEMORY.

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
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 a VkStructureType value identifying 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
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-02628
  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_MEMORYPROPERTY_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_MEMORYPROPERTY_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-02989
  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-pNext-01619
  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-01620
  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-01621
  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-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.
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 a **VkStructureType** value identifying 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]**.
In Vulkan SC, `splitInstanceBindRegionCount` must be zero because sparse allocations are not supported [SCID-8].

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-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-05067**
  - `splitInstanceBindRegionCount` must be zero

### 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 a `VkStructureType` value identifying 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 multi-planar aspect mask 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:

```plaintext
resourceA.end = resourceA.memoryOffset + resourceA.size - 1
resourceA.endPage = resourceA.end & ~(bufferImageGranularity-1)
resourceB.start = resourceB.memoryOffset
resourceB.startPage = resourceB.start & ~(bufferImageGranularity-1)
```

The following property must hold:
resourceA.endPage < resourceB.startPage

That is, the end of the first resource (A) and the beginning of the second resource (B) must be on separate “pages” of size \texttt{bufferImageGranularity}. \texttt{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 \texttt{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 \texttt{bufferImageGranularity}. Therefore, memory bound to sparse resources naturally satisfies the \texttt{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:

```plaintext
// Provided by VK_VERSION_1_0
typedef enum VkSharingMode {
    VK_SHARING_MODE_EXCLUSIVE = 0,
    VK_SHARING_MODE_CONCURRENT = 1,
} VkSharingMode;
```

- \texttt{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.
- \texttt{VK_SHARING_MODE_CONCURRENT} specifies that concurrent access to any range or image subresource of the object from multiple queue families is supported.

\textbf{Note} \texttt{VK_SHARING_MODE_CONCURRENT} may result in lower performance access to the buffer or image than \texttt{VK_SHARING_MODE_EXCLUSIVE}.

Ranges of buffers and image subresources of image objects created using \texttt{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 \texttt{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.

\textbf{Note} Images still require a layout transition from \texttt{VK_IMAGE_LAYOUT_UNDEFINED} or \texttt{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`, 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::pSplitInstanceBindRegions, then they interpret the contents of the memory in consistent ways, and data written to one alias can be read by the other alias.

Additionally, if an individual plane of a multi-planar image and a single-plane image alias the same memory, then they also interpret the contents of the memory in consistent ways under the same conditions, but with the following modifications:

- Both must have been created with the VK_IMAGE_CREATE_DISJOINT_BIT flag.
- The single-plane image must have a VkFormat that is equivalent to that of the multi-planar image's individual plane.
- The single-plane image and the individual plane of the multi-planar image must be bound identically to memory except for VkBindImageMemoryDeviceGroupInfo::pDeviceIndices and VkBindImageMemoryDeviceGroupInfo::pSplitInstanceBindRegions.
- The width and height of the single-plane image are derived from the multi-planar image's dimensions in the manner listed for plane compatibility for the aliased plane.
- 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.

If **VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations** is **VK_TRUE**, **vkCreateSampler** must not return **VK_ERROR_OUT_OF_HOST_MEMORY**.

### Valid Usage

- **VUID-vkCreateSampler-maxSamplerAllocationCount-04110**
  There must be less than **VkPhysicalDeviceLimits::maxSamplerAllocationCount** **VkSampler** objects currently created on the device.

- **VUID-vkCreateSampler-device-05068**
  The number of samplers currently allocated from **device** plus 1 must be less than or equal to the total number of samplers requested via **VkDeviceObjectReservationCreateInfo::samplerRequestCount** specified when **device** was created.

### Valid Usage (Implicit)

- **VUID-vkCreateSampler-device-parameter**
  **device** must be a valid **VkDevice** handle

473
• **VUID-vkCreateSampler-pCreateInfo-parameter**
  
  `pCreateInfo` must be a valid pointer to a valid `VkSamplerCreateInfo` structure.

• **VUID-vkCreateSampler-pAllocator-null**
  
  `pAllocator` must be NULL.

• **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 a `VkStructureType` value identifying 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).

• **mipLodBias** is the bias to be added to mipmap LOD calculation and bias provided by image sampling functions in SPIR-V, as described in the LOD 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_MIPMAP_mipmapMode (e.g. minFilter of VK_FILTER_LINEAR and mipmapMode of VK_SAMPLER_MIPMAP_MODE_NEAREST correspond to GL_LINEAR_MIPMAP_NEAREST).

There are no Vulkan filter modes that directly correspond to OpenGL minification filters of GL_LINEAR or GL_NEAREST, but they can be emulated using VK_SAMPLER_MIPMAP_MODE_NEAREST, minLod = 0, and maxLod = 0.25, and using minFilter = VK_FILTER_LINEAR or minFilter = VK_FILTER_NEAREST, respectively.

Note that using a maxLod of zero would cause magnification to always be performed, and the magFilter to always be used. This is valid, just not an exact match for OpenGL behavior. Clamping the maximum LOD to 0.25 allows the \( \lambda \) value to be non-zero and minification to be performed, while still always rounding down to the base level. If the minFilter and magFilter are equal, then using a maxLod of zero also works.

The maximum number of sampler objects which can be simultaneously created on a device is implementation-dependent and specified by the maxSamplerAllocationCount member of the VkPhysicalDeviceLimits structure.

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,
\( \text{minFilter} \) and \( \text{magFilter} \) must be equal to the sampler \( Y'\text{C}_{B}\text{C}_{R} \) conversion’s \text{chromaFilter}.

- VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01072
  If \( \text{unnormalizedCoordinates} \) is \( \text{VK_TRUE} \), \( \text{minFilter} \) and \( \text{magFilter} \) must be equal

- VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01073
  If \( \text{unnormalizedCoordinates} \) is \( \text{VK_TRUE} \), \( \text{mipmapMode} \) must be \( \text{VK_SAMPLER_MIPMAP_MODE_NEAREST} \)

- VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01074
  If \( \text{unnormalizedCoordinates} \) is \( \text{VK_TRUE} \), \( \text{minLod} \) and \( \text{maxLod} \) must be zero

- VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01075
  If \( \text{unnormalizedCoordinates} \) is \( \text{VK_TRUE} \), \( \text{addressModeU} \) and \( \text{addressModeV} \) must each be either
  \( \text{VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE} \) or \( \text{VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER} \)

- VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01076
  If \( \text{unnormalizedCoordinates} \) is \( \text{VK_TRUE} \), \( \text{anisotropyEnable} \) must be \( \text{VK_FALSE} \)

- VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01077
  If \( \text{unnormalizedCoordinates} \) is \( \text{VK_TRUE} \), \( \text{compareEnable} \) must be \( \text{VK_FALSE} \)

- VUID-VkSamplerCreateInfo-addressModeU-01078
  If any of \( \text{addressModeU} \), \( \text{addressModeV} \) or \( \text{addressModeW} \) are
  \( \text{VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER} \), \( \text{borderColor} \) must be a valid \( \text{VkBorderColor} \) value

- VUID-VkSamplerCreateInfo-addressModeU-01646
  If \( \text{sampler} \ Y'\text{C}_{B}\text{C}_{R} \) conversion is enabled, \( \text{addressModeU} \), \( \text{addressModeV} \), and \( \text{addressModeW} \) must be
  \( \text{VK_SAMPLER_ADDRESS_MODE.Clamp_TO_EDGE} \), \( \text{anisotropyEnable} \) must be \( \text{VK_FALSE} \), and
  \( \text{unnormalizedCoordinates} \) must be \( \text{VK_FALSE} \)

- VUID-VkSamplerCreateInfo-None-01647
  If \( \text{sampler} \ Y'\text{C}_{B}\text{C}_{R} \) conversion is enabled and the \text{pNext} chain includes a
  \text{VkSamplerReductionModeCreateInfo} structure, then the sampler reduction mode must
  be set to \( \text{VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE} \)

- VUID-VkSamplerCreateInfo-pNext-06726
  If \( \text{samplerFilterMinmax} \) is not enabled and the \text{pNext} chain includes a
  \text{VkSamplerReductionModeCreateInfo} structure, then the sampler reduction mode must
  be set to \( \text{VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE} \)

- VUID-VkSamplerCreateInfo-addressModeU-01079
  If \( \text{samplerMirrorClampToEdge} \) is not enabled, and if the \text{VK_KHR_sampler_mirror_clamp_to_edge}
  extension is not enabled, \( \text{addressModeU} \), \( \text{addressModeV} \) and \( \text{addressModeW} \) must not be
  \( \text{VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE} \)

- VUID-VkSamplerCreateInfo-compareEnable-01080
  If \( \text{compareEnable} \) is \( \text{VK_TRUE} \), \( \text{compareOp} \) must be a valid \( \text{VkCompareOp} \) value

- VUID-VkSamplerCreateInfo-compareEnable-01423
  If \( \text{compareEnable} \) is \( \text{VK_TRUE} \), the \text{reductionMode} member of
  \text{VkSamplerReductionModeCreateInfo} must be
  \( \text{VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE} \)
Valid Usage (Implicit)

- **VUID-VkSamplerCreateInfo-sType-sType**
  
  *sType must be* `VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO`

- **VUID-VkSamplerCreateInfo-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 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;
```

- **sType** is a **VkStructureType** value identifying 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,
} VkFilter;
```

- `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 `samplerMirrorClampToEdge` 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:

```c
// 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 `VkStencilOpState::compareOp`.
- The Depth Comparison operator for the depth test, specified by `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:
typedef enum VkBorderColor {
  VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK = 0,
  VK_BORDER_COLOR_INT_TRANSPARENT_BLACK = 1,
  VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK = 2,
  VK_BORDER_COLOR_INT_OPAQUE_BLACK = 3,
  VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE = 4,
  VK_BORDER_COLOR_INT_OPAQUE_WHITE = 5,
} VkBorderColor;

• 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

**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.

- VUID-vkDestroySampler-pAllocator-null
  `pAllocator` must be `NULL`.

- VUID-vkDestroySampler-sampler-parent
  If `sampler` is a valid handle, it must have been created, allocated, or retrieved from `device`.

---

### Host Synchronization

- Host access to `sampler` must be externally synchronized.

---

### 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 a `VkStructureType` value identifying 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’C_bC_R conversion is an opaque representation of a device-specific sampler Y’C_bC_R conversion description, represented as a `VkSamplerYcbcrConversion` handle:

```cpp
// Provided by VK_VERSION_1_1
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSamplerYcbcrConversion)
```

To create a `VkSamplerYcbcrConversion`, call:

```cpp
// 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’C_bC_R conversion.
- `pCreateInfo` is a pointer to a `VkSamplerYcbcrConversionCreateInfo` structure specifying the requested sampler Y’C_bC_R 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’C_bC_R conversion is returned.

The interpretation of the configured sampler Y’C_bC_R conversion is described in more detail in the description of sampler Y’C_bC_R conversion in the Image Operations chapter.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkCreateSamplerYcbcrConversion` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

**Valid Usage**

- VUID-vkCreateSamplerYcbcrConversion-None-01648
  The `samplerYcbcrConversion` feature must be enabled

- VUID-vkCreateSamplerYcbcrConversion-device-05068
  The number of sampler conversions currently allocated from `device` plus 1 must be less than or equal to the total number of sampler conversions requested via `VkDeviceObjectReservationCreateInfo::samplerYcbcrConversionRequestCount` specified when `device` was created

**Valid Usage (Implicit)**

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

- VUID-vkCreateSamplerYcbcrConversion-pCreateInfo-parameter
The `VkSamplerYcbcrConversionCreateInfo` structure is defined as:

```
// 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 a `VkStructureType` value identifying 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’C₈₆Cᵦ conversion objects do not support *external format conversion* without additional extensions defining *external formats*.

---

**Valid Usage**

- **VUID-VkSamplerYcbcrConversionCreateInfo-format-04061**
  
  If **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’C₈₆Cᵦ 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’C₈₆Cᵦ 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 downsampled

- **VUID-VkSamplerYcbcrConversionCreateInfo-xChromaOffset-01652**
  
  If the potential format features of the sampler Y’C₈₆Cᵦ 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 downsampled

- **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_SWIZZLEZERO**

- **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**
If the format has a \_422 or \_420 suffix, then components.b must be the identity swizzle or \_R

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

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

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

If the potential format features of the sampler Y’C_bC_r conversion do not support VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT, forceExplicitReconstruction must be VK_FALSE

If the potential format features of the sampler Y’C_bC_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)

sType must be VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_CREATE_INFO

pNext must be NULL

format must be a valid VkFormat value

ycbcrModel must be a valid VkSamplerYcbcrModelConversion value

ycbcrRange must be a valid VkSamplerYcbcrRange value

components must be a valid VkComponentMapping structure

xChromaOffset must be a valid VkChromaLocation value
The **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_B’C_R.
- **VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_709** specifies the color model conversion from Y’C_B’C_R to R’G’B’ defined in BT.709 and described in the “BT.709 Y’C_B’C_R conversion” section of the Khronos Data Format Specification.
- **VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_601** specifies the color model conversion from Y’C_B’C_R to R’G’B’ defined in BT.601 and described in the “BT.601 Y’C_B’C_R conversion” section of the Khronos Data Format Specification.
- **VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_2020** specifies the color model conversion from Y’C_B’C_R to R’G’B’ defined in BT.2020 and described in the “BT.2020 Y’C_B’C_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_B’C_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:

- `components.a = VK_COMPONENT_SWIZZLE_IDENTITY`
- `components.r = VK_COMPONENT_SWIZZLE_B`
- `components.g = VK_COMPONENT_SWIZZLE_R`
- `components.b = VK_COMPONENT_SWIZZLE_G`

The `VkSamplerYcbcrRange` enum describes whether color components are encoded using the full range of numerical values or whether values are reserved for headroom and foot room. `VkSamplerYcbcrRange` is defined as:

```cpp
// 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’C₉B₉ 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:

```cpp
// 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’CₘCₜ conversion, call:

```c
// Provided by VK_VERSION_1_1
void vkDestroySamplerYcbcrConversion(
    VkDevice device,
    VkSamplerYcbcrConversion ycbcrConversion,
    const VkAllocationCallbacks* pAllocator);
```

• `device` is the logical device that destroys the Y’CₘCₜ 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-null
  
  `pAllocator` must be `NULL`

- 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_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL
- VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_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 Y′C′bC′ 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_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL
- VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL
- VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL
- VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL

492
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 buffer view formats which report format features support for VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT

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 view formats which report format features support for VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT

Stores to storage texel buffers are supported in compute shaders for texel buffer formats which report format features support for VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT

Atomic operations on storage texel buffers are supported in compute shaders for texel buffer formats which report format features support for VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT

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. 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:

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

To create descriptor set layout objects, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateDescriptorSetLayout(
    VkDevice device,
    const VkDescriptorSetLayoutCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkDescriptorSetLayout* pSetLayout);
```

- `device` is the logical device that creates the descriptor set layout.
- `pCreateInfo` is a pointer to a VkDescriptorSetLayoutCreateInfo structure specifying the state of
  the descriptor set layout object.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pSetLayout` is a pointer to a VkDescriptorSetLayout handle in which the resulting descriptor set
  layout object is returned.

If VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations is VK_TRUE,
vkCreateDescriptorSetLayout must not return VK_ERROR_OUT_OF_HOST_MEMORY.

### Valid Usage

- **VUID-vkCreateDescriptorSetLayout-device-05068**
The number of descriptor set layouts currently allocated from `device` plus 1 must be less
than or equal to the total number of descriptor set layouts requested via
VkDeviceObjectReservationCreateInfo::descriptorSetLayoutRequestCount specified when
`device` was created.

- **VUID-vkCreateDescriptorSetLayout-layoutbindings-device-05089**
The number of descriptor set layout bindings currently allocated from `device` across all
VkDescriptorSetLayout objects plus `pCreateInfo->bindingCount` must be less than or equal
to the total number of descriptor set layout bindings requested via
VkDeviceObjectReservationCreateInfo::descriptorSetLayoutBindingRequestCount specified
when device was created

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-null
pAllocator must be NULL

- 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:

```
// 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 a VkStructureType value identifying 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

- VUID-VkDescriptorsetLayoutCreateInfo-bindingCount-05011
  bindingCount must be less than or equal to maxDescriptorSetLayoutBindings

- VUID-VkDescriptorsetLayoutCreateInfo-descriptorCount-05071
  The sum of descriptorCount over all bindings in pBindings that have descriptorType of VK_DESCRIPTOR_TYPE_SAMPLER or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER and pImmutableSamplers not equal to NULL must be less than or equal to VkDeviceObjectReservationCreateInfo::maxImmutableSamplersPerDescriptorsetLayout

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,
    // Provided by VK_VERSION_1_3
    ...}
```
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. 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-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.

- **VUID-VkDescriptorSetLayoutBinding-binding-05012**
  **binding** must be less than the value of VkDeviceObjectReservationCreateInfo::descriptorSetLayoutBindingLimit provided when the device was created.
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 a VkStructureType value identifying 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
VkDescriptorBindingFlags 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-pBindingFlags-09379
  If an element of pBindingFlags includes VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT, then it must be the element with
  the the highest binding number

- VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-
If `VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingUniformBufferUpdateAfterBind` is not enabled, all bindings with descriptor type `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` must not use `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT`.

- VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingSampledImageUpdateAfterBind-03006
  If `VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingSampledImageUpdateAfterBind` is not enabled, all bindings with descriptor type `VK_DESCRIPTOR_TYPE_SAMPLER`, `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, or `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE` must not use `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT`.

- VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingStorageImageUpdateAfterBind-03007
  If `VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingStorageImageUpdateAfterBind` is not enabled, all bindings with descriptor type `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE` must not use `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT`.

- VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingStorageBufferUpdateAfterBind-03008
  If `VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingStorageBufferUpdateAfterBind` is not enabled, all bindings with descriptor type `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` must not use `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT`.

- VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingUniformTexelBufferUpdateAfterBind-03009
  If `VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingUniformTexelBufferUpdateAfterBind` is not enabled, all bindings with descriptor type `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` must not use `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT`.

- VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingStorageTexelBufferUpdateAfterBind-03010
  If `VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingStorageTexelBufferUpdateAfterBind` is not enabled, all bindings with descriptor type `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` must not use `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT`.

- VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-None-03011
  All bindings with descriptor type `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`, `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`, or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` must not use `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT`.

- 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`.  


If `VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingPartiallyBound` is not enabled, all elements of `pBindingFlags` must not include `VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT`.

If `VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingVariableDescriptorCount` is not enabled, all elements of `pBindingFlags` must not include `VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT`.

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`.

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.

```
// Provided by VK_VERSION_1_2
typedef VkFlags VkDescriptorBindingFlags;
```

`VkDescriptorBindingFlags` is a bitmask type for setting a mask of zero or more `VkDescriptorBindingFlagBits`.

To query information about whether a descriptor set layout can be created, call:

```
// Provided by VK_VERSION_1_1
void vkGetDescriptorSetLayoutSupport(
    VkDevice device,
    const VkDescriptorSetLayoutCreateInfo* pCreateInfo,
    const VkDescriptorSetLayoutSupportInfo* pLayoutSupport,
    void* pWrittenSupport);
```
VkDescriptorSetLayoutSupport

• **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:

```c
// Provided by VK_VERSION_1_1
typedef struct VkDescriptorSetLayoutSupport {
    VkStructureType sType;
    void* pNext;
    VkBool32 supported;
} VkDescriptorSetLayoutSupport;
```
• **sType** is a `VkStructureType` value identifying 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.

```c
// Provided by VK_VERSION_1_2
typedef struct VkDescriptorSetVariableDescriptorCountLayoutSupport {
    VkStructureType sType;
    void* pNext;
    uint32_t maxVariableDescriptorCount;
} VkDescriptorSetVariableDescriptorCountLayoutSupport;
```

• **sType** is a `VkStructureType` value identifying 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 `VkDescriptorSetLayoutCreateInfo` structure specified in `vkGetDescriptorSetLayoutSupport` includes a variable-sized descriptor, then **supported** is determined assuming the requested size of the variable-sized descriptor, and **maxVariableDescriptorCount** is set to the maximum size of that descriptor that can be successfully created (which is greater than or equal to the requested size passed in). If the `VkDescriptorSetLayoutCreateInfo` structure does not include a variable-sized descriptor, or if the `VkPhysicalDeviceDescriptorIndexingFeatures`::descriptorBindingVariableDescriptorCount feature is not enabled, then **maxVariableDescriptorCount** is set to zero. For the purposes of this command, a variable-sized descriptor binding with a **descriptorCount** of zero is treated as having a **descriptorCount** of one, and thus the binding is not ignored and the maximum descriptor count will be returned. If the layout is not supported, then the value written to **maxVariableDescriptorCount** is undefined.
Valid Usage (Implicit)

- VUID-VkDescriptorSetVariableDescriptorCountLayoutSupport-sType-sType
  - sType must be VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_LAYOUT_SUPPORT

The following examples show a shader snippet using two descriptor sets, and application code that creates corresponding descriptor set layouts.

**GLSL example**

```glsl
//
// binding to a single sampled image descriptor in set 0
//
layout (set=0, binding=0) uniform texture2D mySampledImage;

//
// binding to an array of sampled image descriptors in set 0
//
layout (set=0, binding=1) uniform texture2D myArrayOfSampledImages[12];

//
// binding to a single uniform buffer descriptor in set 1
//
layout (set=1, binding=0) uniform myUniformBuffer
{
  vec4 myElement[32];
};
```

**SPIR-V example**

```spirv
...%1 = OpExtInstImport "GLSL.std.450"
...
OpName %9 "mySampledImage"
OpName %14 "myArrayOfSampledImages"
OpName %18 "myUniformBuffer"
OpMemberName %18 0 "myElement"
OpName %20 ""
OpDecorate %9 DescriptorSet 0
OpDecorate %9 Binding 0
OpDecorate %14 DescriptorSet 0
OpDecorate %14 Binding 1
OpDecorate %17 ArrayStride 16
OpMemberDecorate %18 0 Offset 0
OpDecorate %18 Block
OpDecorate %20 DescriptorSet 1
OpDecorate %20 Binding 0
%2 = OpTypeVoid
...```
API example

```cpp
VkResult myResult;

const VkDescriptorSetLayoutBinding myDescriptorSetLayoutBinding[] = {
    // binding to a single image descriptor
    {
        .binding = 0,
        .descriptorType = VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE,
        .descriptorCount = 1,
        .stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT,
        .pImmutableSamplers = NULL
    },

    // binding to an array of image descriptors
    {
        .binding = 1,
        .descriptorType = VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE,
        .descriptorCount = 12,
        .stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT,
        .pImmutableSamplers = NULL
    },

    // binding to a single uniform buffer descriptor
    {
        .binding = 0,
        .descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
        .descriptorCount = 1,
        .stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT,
        .pImmutableSamplers = NULL
    }
};
```
const VkDescriptorSetLayoutCreateInfo myDescriptorSetLayoutCreateInfo[] = {
    // Information for first descriptor set with two descriptor bindings
    { .sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO,
      .pNext = NULL,
      .flags = 0,
      .bindingCount = 2,
      .pBindings = &myDescriptorSetLayoutBinding[0] },

    // Information for second descriptor set with one descriptor binding
    { .sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO,
      .pNext = NULL,
      .flags = 0,
      .bindingCount = 1,
};

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:

// 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 (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-null
  - `pAllocator` must be `NULL`
- 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:

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

To create a pipeline layout, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreatePipelineLayout(
    VkDevice device,
    const VkPipelineLayoutCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator)
```
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.

If VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations is VK_TRUE, vkCreatePipelineLayout must not return VK_ERROR_OUT_OF_HOST_MEMORY.

### Valid Usage

- VUID-vkCreatePipelineLayout-device-05068
  The number of pipeline layouts currently allocated from device plus 1 must be less than or equal to the total number of pipeline layouts requested via VkDeviceObjectReservationCreateInfo::pipelineLayoutRequestCount specified when device was created

### 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-null
  pAllocator must be NULL
- 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:
typedef struct VkPipelineLayoutCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineLayoutCreateFlags flags;
    uint32_t setLayoutCount;
    const VkDescriptorSetLayout* pSetLayouts;
    uint32_t pushConstantRangeCount;
    const VkPushConstantRange* pPushConstantRanges;
} VkPipelineLayoutCreateInfo;

- `sType` is a `VkStructureType` value identifying 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.

In Vulkan SC, the pipeline compilation process occurs offline, but the application must still provide values to `VkPipelineLayoutCreateInfo` that match the values used for offline compilation of pipelines using this `VkPipelineLayout`.

**Valid Usage**

- VUID-VkPipelineLayoutCreateInfo-setLayoutCount-00286
  
  `setLayoutCount` must be less than or equal to `VkPhysicalDeviceLimits::maxBoundDescriptorSets`

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03016
  
  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`

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03017
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`.

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-03018**
  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_STORAGEBuffer_DYNAMIC` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorStorageBuffers`.

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-06939**
  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`.

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-03020**
  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`.

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-03021**
  The total number of descriptors with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceDescriptorIndexingProperties::maxPerStageDescriptorInputAttachments`.

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-03022**
  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::maxPerStageDescriptorUpdateAfterBindUniformBuffers`.
accessible to any given shader stage across all elements of \texttt{pSetLayouts} must be less than or equal to
\texttt{VkPhysicalDeviceDescriptorIndexingProperties::maxPerStageDescriptorUpdateAfterBindStorageBuffers}

\textbullet\ VUID-VkPipelineLayoutCreateInfo-descriptorType-03025

The total number of descriptors with a \texttt{descriptorType} of
\texttt{VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER}, \texttt{VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE}, and
\texttt{VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER} accessible to any given shader stage across all
elements of \texttt{pSetLayouts} must be less than or equal to
\texttt{VkPhysicalDeviceDescriptorIndexingProperties::maxPerStageDescriptorUpdateAfterBindSampledImages}

\textbullet\ VUID-VkPipelineLayoutCreateInfo-descriptorType-03026

The total number of descriptors with a \texttt{descriptorType} of
\texttt{VK_DESCRIPTOR_TYPE_STORAGE_IMAGE}, and \texttt{VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER} accessible to any given shader stage across all elements of \texttt{pSetLayouts} must be less than or equal to
\texttt{VkPhysicalDeviceDescriptorIndexingProperties::maxPerStageDescriptorUpdateAfterBindStorageImages}

\textbullet\ VUID-VkPipelineLayoutCreateInfo-descriptorType-03027

The total number of descriptors with a \texttt{descriptorType} of
\texttt{VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT} accessible to any given shader stage across all
elements of \texttt{pSetLayouts} must be less than or equal to
\texttt{VkPhysicalDeviceDescriptorIndexingProperties::maxPerStageDescriptorUpdateAfterBindInputAttachments}

\textbullet\ VUID-VkPipelineLayoutCreateInfo-descriptorType-03028

The total number of descriptors in descriptor set layouts created
without the \texttt{VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT} bit set with a \texttt{descriptorType}
of \texttt{VK_DESCRIPTOR_TYPE_SAMPLER} and \texttt{VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER} accessible
across all shader stages and across all elements of \texttt{pSetLayouts} must be less than or equal
to \texttt{VkPhysicalDeviceLimits::maxDescriptorSetSamplers}

\textbullet\ VUID-VkPipelineLayoutCreateInfo-descriptorType-03029

The total number of descriptors in descriptor set layouts created
without the \texttt{VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT} bit set with a \texttt{descriptorType}
of \texttt{VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER} accessible across all shader stages and across all
elements of \texttt{pSetLayouts} must be less than or equal to \texttt{VkPhysicalDeviceLimits::maxDescriptorSetUniformBuffers}

\textbullet\ VUID-VkPipelineLayoutCreateInfo-descriptorType-03030

The total number of descriptors in descriptor set layouts created
without the \texttt{VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT} bit set with a \texttt{descriptorType}
of \texttt{VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC} accessible across all shader stages and
across all elements of \texttt{pSetLayouts} must be less than or equal to \texttt{VkPhysicalDeviceLimits::maxDescriptorSetUniformBuffersDynamic}

\textbullet\ VUID-VkPipelineLayoutCreateInfo-descriptorType-03031

The total number of descriptors in descriptor set layouts created
without the \texttt{VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT} bit set with a \texttt{descriptorType}
of \texttt{VK_DESCRIPTOR_TYPE_STORAGE_BUFFER} accessible across all shader stages and across all
elements of \texttt{pSetLayouts} must be less than or equal to \texttt{VkPhysicalDeviceLimits::maxDescriptorSetUniformBuffersDynamic}
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`.

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`.

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`.

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 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`.
maxDescriptorSetUpdateAfterBindStorageBuffers

- **VUID-VkPipelineLayoutCreateInfo-pSetLayouts-03040**
  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.

- **VUID-VkPipelineLayoutCreateInfo-pSetLayouts-03041**
  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.

- **VUID-VkPipelineLayoutCreateInfo-pSetLayouts-03042**
  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.

- **VUID-VkPipelineLayoutCreateInfo-pSetLayouts-03043**
  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-pPushConstantRanges-00292**
  Any two elements of pPushConstantRanges must not include the same stage in stageFlags.

- **VUID-VkPipelineLayoutCreateInfo-graphicsPipelineLibrary-06753**
  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.
typedef enum VkPipelineLayoutCreateFlagBits {
} VkPipelineLayoutCreateFlagBits;

All values for this enum are defined by extensions.

// 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:

// Provided by VK_VERSION_1_0
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
  \texttt{stageFlags} \textbf{must} be a valid combination of \texttt{VkShaderStageFlagBits} values

- VUID-VkPushConstantRange-stageFlags-requiredbitsetmask
  \texttt{stageFlags} \textbf{must} not be 0

Once created, pipeline layouts are used as part of pipeline creation (see \texttt{Pipelines}), as part of binding descriptor sets (see \texttt{Descriptor Set Binding}), and as part of setting push constants (see \texttt{Push Constant Updates}). Pipeline creation accepts a pipeline layout as input, and the layout \textbf{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 \textit{statically used} in all shaders in a pipeline \textbf{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 \texttt{stageFlags}. The pipeline layout \textbf{can} include entries that are not used by a particular pipeline. 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 \textbf{may} cheaply switch pipelines without reprogramming the bindings.

Similarly, the push constant block declared in each shader (if present) \textbf{must} only place variables at offsets that are each included in a push constant range with \texttt{stageFlags} including the bit corresponding to the shader stage that uses it. The pipeline layout \textbf{can} include ranges or portions of ranges that are not used by a particular pipeline.

There is a limit on the total number of resources of each type that \textbf{can} be included in bindings in all descriptor set layouts in a pipeline layout as shown in \texttt{Pipeline Layout Resource Limits}. The “Total Resources Available” column gives the limit on the number of each type of resource that \textbf{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 \textbf{can} be used in any pipeline stage as described in \texttt{Shader Resource Limits}.

\textit{Table 10. Pipeline Layout Resource Limits}

<table>
<thead>
<tr>
<th>Total Resources Available</th>
<th>Resource Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxDescriptorSetSamplers or maxDescriptorSetUpdateAfterBindSamplers</td>
<td>sampler</td>
</tr>
<tr>
<td>maxDescriptorSetSampledImages or maxDescriptorSetUpdateAfterBindSampledImages</td>
<td>combined image sampler</td>
</tr>
<tr>
<td>maxDescriptorSetStorageImages or maxDescriptorSetUpdateAfterBindStorageImages</td>
<td>sampled image</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindSampledImages</td>
<td>combined image sampler</td>
</tr>
<tr>
<td>maxDescriptorSetStorageImages or maxDescriptorSetUpdateAfterBindStorageImages</td>
<td>uniform texel buffer</td>
</tr>
<tr>
<td></td>
<td>storage image</td>
</tr>
<tr>
<td></td>
<td>storage texel buffer</td>
</tr>
</tbody>
</table>
To destroy a pipeline layout, call:

```c
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-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-null
  pAllocator must be NULL
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, a previously bound descriptor set bound with lower index M than N is disturbed if the pipeline layouts for set M and N are not compatible for set M. Otherwise, the bound descriptor set in M is not disturbed.

If, additionally, the previously bound descriptor set for set N was bound using a pipeline layout not compatible for set N, then all bindings in sets numbered greater than N are disturbed.

When binding a pipeline, the pipeline can correctly access any previously bound descriptor set N if it was bound with compatible pipeline layout for set N, and it was not disturbed.

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.

When a descriptor set is disturbed by binding descriptor sets, the disturbed set is considered to contain undefined descriptors bound with the same pipeline layout as the disturbing descriptor set.

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 }; const VkPushConstantRange ranges[] = {
    { .stageFlags = VK_SHADER_STAGE_VERTEX_BIT, .offset = 0, .size = 4 },
    { .stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT, .offset = 4, .size = 4 }
};
const VkPipelineLayoutCreateInfo createInfo = {
    .sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO, .pNext = NULL, .flags = 0, .setLayoutCount = 2, .pSetLayouts = layouts, .pushConstantRangeCount = 2, .pPushConstantRanges = ranges
};
VkPipelineLayout myPipelineLayout;
myResult = vkCreatePipelineLayout(
    myDevice, &createInfo, NULL, &myPipelineLayout);
```

### 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)
```

520
To create a descriptor pool object, call:

```c
// 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`.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkCreateDescriptorPool` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

### Valid Usage

- **VUID-vkCreateDescriptorPool-device-05068**
  The number of descriptor pools currently allocated from `device` plus 1 must be less than or equal to the total number of descriptor pools requested via `VkDeviceObjectReservationCreateInfo::descriptorPoolRequestCount` specified when `device` was created.

### 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-null**
  `pAllocator must be NULL`

- **VUID-vkCreateDescriptorPool-pDescriptorPool-parameter**
  `pDescriptorPool must be a valid pointer to a VkDescriptorPool handle`
\textbf{Return Codes}

\textbf{Success}
- \texttt{VK_SUCCESS}

\textbf{Failure}
- \texttt{VK_ERROR_OUT_OF_HOST_MEMORY}
- \texttt{VK_ERROR_OUT_OF_DEVICE_MEMORY}

Additional information about the pool is passed in a \texttt{VkDescriptorPoolCreateInfo} structure:

\begin{verbatim}
// Provided by VK_VERSION_1_0
typedef struct VkDescriptorPoolCreateInfo {
    VkStructureType                   sType;
    const void*                        pNext;
    VkDescriptorPoolCreateFlags       flags;
    uint32_t                           maxSets;
    uint32_t                           poolSizeCount;
    const VkDescriptorPoolSize*        pPoolSizes;
} VkDescriptorPoolCreateInfo;
\end{verbatim}

- \texttt{sType} is a \texttt{VkStructureType} value identifying this structure.
- \texttt{pNext} is \texttt{NULL} or a pointer to a structure extending this structure.
- \texttt{flags} is a bitmask of \texttt{VkDescriptorPoolCreateFlagBits} specifying certain supported operations on the pool.
- \texttt{maxSets} is the maximum number of descriptor sets that \texttt{can} be allocated from the pool.
- \texttt{poolSizeCount} is the number of elements in \texttt{pPoolSizes}.
- \texttt{pPoolSizes} is a pointer to an array of \texttt{VkDescriptorPoolSize} structures, each containing a descriptor type and number of descriptors of that type to be allocated in the pool.

If multiple \texttt{VkDescriptorPoolSize} structures containing the same descriptor type appear in the \texttt{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 \texttt{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 \texttt{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 \texttt{must} not cause an allocation failure (note that this is always the case for a pool created without the \texttt{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-descriptorPoolOverallocation-09227
  maxSets must be greater than 0

**Valid Usage (Implicit)**

- VUID-VkDescriptorPoolCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO

- VUID-VkDescriptorPoolCreateInfo-pNext-pNext
  pNext must be NULL

- 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

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 {
    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.

```c
// 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:

```c
// 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.

### Valid Usage

- VUID-VkDescriptorPoolSize-descriptorCount-00302
descriptorCount must be greater than 0

### Valid Usage (Implicit)

- VUID-VkDescriptorPoolSize-type-parameter
type must be a valid VkDescriptorType value

Descriptor pools cannot be destroyed [SCID-4]. If VkPhysicalDeviceVulkanSC10Properties::deviceDestroyFreesMemory is VK_TRUE, the memory is returned to the system when the device is destroyed.

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:
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.
- 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.

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.

If VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations is VK_TRUE, vkAllocateDescriptorSets must not return VK_ERROR_OUT_OF_HOST_MEMORY.
Valid Usage

- VUID-vkAllocateDescriptorSets-device-05068
  The number of descriptor sets currently allocated from device plus 
  VkDescriptorSetAllocateInfo::descriptorSetCount must be less than or equal to the total 
  number of descriptor sets requested via VkDeviceObjectReservationCreateInfo::descriptorSetRequestCount specified when device was created

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

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_FRAGMENTED_POOL
- VK_ERROR_OUT_OF_POOL_MEMORY

The VkDescriptorSetAllocateInfo structure is defined as:

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

tuint32_t

const VkDescriptorSetLayout*

} VkDescriptorSetAllocateInfo;

- **sType** is a `VkStructureType` value identifying 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.

- **VUID-VkDescriptorSetAllocateInfo-pSetLayouts-09380**
  If `pSetLayouts[i]` was created with an element of `pBindingFlags` that includes `VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT`, and `VkDescriptorSetVariableDescriptorCountAllocateInfo` is included in the `pNext` chain, and `VkDescriptorSetVariableDescriptorCountAllocateInfo::descriptorSetCount` is not zero, then `VkDescriptorSetVariableDescriptorCountAllocateInfo::pDescriptorCounts[i]` must be less than or equal to `VkDescriptorSetLayoutBinding::descriptorCount` for the corresponding binding used to create `pSetLayouts[i]`.

### 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**
  `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 a **VkStructureType** value identifying 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 **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 **VkDescriptorSetAllocateInfo::descriptorSetCount**

- VUID-VkDescriptorSetVariableDescriptorCountAllocateInfo-pSetLayouts-03046
  If **VkDescriptorSetAllocateInfo::pSetLayouts[i]** has a variable-sized descriptor binding, then **pDescriptorCounts[i]** must be less than or equal to the descriptor count specified for that binding when the descriptor set layout was created
Valid Usage (Implicit)

- VUID-VkDescriptorSetVariableDescriptorCountAllocateInfo-sType-sType must be VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_ALLOCATE_INFO

- VUID-VkDescriptorSetVariableDescriptorCountAllocateInfo-pDescriptorCounts-parameter If descriptorSetCount is not 0, pDescriptorCounts must be a valid pointer to an array of descriptorSetCount uint32_t values

To free allocated descriptor sets, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkFreeDescriptorSets(
    VkDevice device,
    VkDescriptorPool descriptorPool,
    uint32_t descriptorSetCount,
    const VkDescriptorSet* pDescriptorSets);
```

- `device` is the logical device that owns the descriptor pool.
- `descriptorPool` is the descriptor pool from which the descriptor sets were allocated.
- `descriptorSetCount` is the number of elements in the `pDescriptorSets` array.
- `pDescriptorSets` is a pointer to an array of handles to `VkDescriptorSet` objects.

After calling `vkFreeDescriptorSets`, all descriptor sets in `pDescriptorSets` are invalid.

If `recycleDescriptorSetMemory` is VK_FALSE, then freeing a descriptor set does not make the pool memory it used available to be reallocated until the descriptor pool is reset. If `recycleDescriptorSetMemory` is VK_TRUE, then the memory is available to be reallocated immediately after freeing the descriptor set.

Valid Usage

- VUID-vkFreeDescriptorSets-pDescriptorSets-00309 All submitted commands that refer to any element of `pDescriptorSets` must have completed execution

- VUID-vkFreeDescriptorSets-pDescriptorSets-00310 `pDescriptorSets` must be a valid pointer to an array of `descriptorSetCount` `VkDescriptorSet` handles, each element of which must either be a valid handle or VK_NULL_HANDLE

- VUID-vkFreeDescriptorSets-descriptorPool-00312 `descriptorPool` must have been created with the VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT 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 `pDescriptorWrites` are performed first, followed by the operations described by `pDescriptorCopies`. Within each array, the operations are performed in the order they appear in the array.

Each element in the `pDescriptorWrites` array describes an operation updating the descriptor set using descriptors for resources specified in the structure.

Each element in the `pDescriptorCopies` array is a `VkCopyDescriptorSet` structure describing an operation copying descriptors between sets.

If the `dstSet` member of any element of `pDescriptorWrites` or `pDescriptorCopies` is bound, accessed, or modified by any command that was recorded to a command buffer which is currently in the recording or executable state, and any of the descriptor bindings that are updated were not created with the `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` 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 $p_{	ext{DescriptorWrites}}[i].\text{descriptorType}$ is
\begin{itemize}
  \item VK_DESCRIPTOR_TYPE_SAMPLER or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,
  \end{itemize}
and $\text{dstSet}$ was not allocated with a layout that included immutable samplers for $\text{dstBinding}$ with
$\text{descriptorType}$, the sampler member of any element of the $p_{\text{ImageInfo}}$ member of
$p_{\text{DescriptorWrites}}[i]$ must have been created on device

- VUID-vkUpdateDescriptorSets-pDescriptorWrites-06239
  
  For each element $i$ where $p_{\text{DescriptorWrites}}[i].\text{descriptorType}$ is
  \begin{itemize}
    \item VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE,
    \item VK_DESCRIPTOR_TYPE_STORAGE_IMAGE,
    \item VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT,
    \item or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER
  \end{itemize}
  the imageView member of any element of $p_{\text{DescriptorWrites}}[i]$ must have been created on device

- VUID-vkUpdateDescriptorSets-pDescriptorWrites-06493
  
  For each element $i$ where $p_{\text{DescriptorWrites}}[i].\text{descriptorType}$ is
  \begin{itemize}
    \item VK_DESCRIPTOR_TYPE_SAMPLER,
    \item VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,
    \item VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE,
    \item VK_DESCRIPTOR_TYPE_STORAGE_IMAGE,
    \item or VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT
  \end{itemize}
  the $p_{\text{ImageInfo}}$ member of any element of $p_{\text{DescriptorWrites}}[i]$ must be a valid
  pointer to an array of $p_{\text{DescriptorWrites}}[i].\text{descriptorCount}$ valid VkDescriptorImageInfo
  structures

- VUID-vkUpdateDescriptorSets-None-03047
  
  The $\text{dstSet}$ member of each element of $p_{\text{DescriptorWrites}}$ or $p_{\text{DescriptorCopies}}$ for
  bindings which were created without the VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT or
  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 $p_{\text{DescriptorWrites}}[i].\text{dstSet}$ and $p_{\text{DescriptorCopies}}[i].\text{dstSet}$ must be
  externally synchronized unless explicitly denoted otherwise for specific flags

**Valid Usage (Implicit)**

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

- VUID-vkUpdateDescriptorSets-pDescriptorWrites-parameter
  
  If $\text{descriptorWriteCount}$ is not 0, $p_{\text{DescriptorWrites}}$ must be a valid pointer to an array of
  $\text{descriptorWriteCount}$ valid VkWriteDescriptorSet structures

- VUID-vkUpdateDescriptorSets-pDescriptorCopies-parameter
  
  If $\text{descriptorCopyCount}$ is not 0, $p_{\text{DescriptorCopies}}$ must be a valid pointer to an array of
  $\text{descriptorCopyCount}$ valid VkCopyDescriptorSet structures

The VkWriteDescriptorSet structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkWriteDescriptorSet {
    VkStructureType sType;
    const void* pNext;
};
```
- `sType` is a `VkStructureType` value identifying 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.
- `descriptorCount` is the number of descriptors to update. `descriptorCount` is one of
  - the number of elements in `pImageInfo`
  - the number of elements in `pBufferInfo`
  - the number of elements in `pTexelBufferView`
- `descriptorType` is a `VkDescriptorType` specifying the type of each descriptor in `pImageInfo`, `pBufferInfo`, or `pTexelBufferView`, as described below. It must be the same type as 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, as specified below.

If the `dstBinding` has fewer than `descriptorCount` array elements remaining starting from `dstArrayElement`, then the remainder will be used to update the subsequent binding - `dstBinding`+1 starting at array element zero. If a binding has a `descriptorCount` of zero, it is skipped. This behavior applies recursively, with the update affecting consecutive bindings as needed to update all `descriptorCount` descriptors. Consecutive bindings must have identical `VkDescriptorType`, `VkShaderStageFlags`, `VkDescriptorBindingFlagBits`, and immutable samplers references.
Valid Usage

- **VUID-VkWriteDescriptorSet-dstBinding-00315**
  
  \textit{dstBinding must} be less than or equal to the maximum value of \textit{binding} of all \texttt{VkDescriptorSetLayoutBinding} structures specified when \textit{dstSet}'s descriptor set layout was created.

- **VUID-VkWriteDescriptorSet-dstBinding-00316**
  
  \textit{dstBinding must} be a binding with a non-zero \textit{descriptorCount}.

- **VUID-VkWriteDescriptorSet-descriptorCount-00317**
  
  All consecutive bindings updated via a single \texttt{VkWriteDescriptorSet} structure, except those with a \textit{descriptorCount} of zero, \textbf{must} have identical \textit{descriptorType} and \textit{stageFlags}.

- **VUID-VkWriteDescriptorSet-descriptorCount-00318**
  
  All consecutive bindings updated via a single \texttt{VkWriteDescriptorSet} structure, except those with a \textit{descriptorCount} of zero, \textbf{must} all either use immutable samplers or \textbf{must} all not use immutable samplers.

- **VUID-VkWriteDescriptorSet-descriptorType-00319**
  
  \textit{descriptorType} \textbf{must} match the type of \textit{dstBinding} within \textit{dstSet}.

- **VUID-VkWriteDescriptorSet-dstSet-00320**
  
  \textit{dstSet must} be a valid \texttt{VkDescriptorSet} handle.

- **VUID-VkWriteDescriptorSet-dstArrayElement-00321**
  
  The sum of \textit{dstArrayElement} and \textit{descriptorCount} \textbf{must} be less than or equal to the number of array elements in the descriptor set binding specified by \textit{dstBinding}, and all applicable consecutive bindings, as described by \textit{consecutive binding updates}.

- **VUID-VkWriteDescriptorSet-descriptorType-02994**
  
  If \textit{descriptorType} is \texttt{VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER} or \texttt{VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER}, each element of \textit{pTexelBufferView} \textbf{must} be either a valid \texttt{VkBufferView} handle or \texttt{VK_NULL_HANDLE}.

- **VUID-VkWriteDescriptorSet-descriptorType-02995**
  
  If \textit{descriptorType} is \texttt{VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER} or \texttt{VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER} and the \textit{nullDescriptor} feature is not enabled, each element of \textit{pTexelBufferView} \textbf{must} not be \texttt{VK_NULL_HANDLE}.

- **VUID-VkWriteDescriptorSet-descriptorType-00324**
  
  If \textit{descriptorType} is \texttt{VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER}, \texttt{VK_DESCRIPTOR_TYPE_STORAGE_BUFFER}, \texttt{VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC}, or \texttt{VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC}, \textit{pBufferInfo} \textbf{must} be a valid pointer to an array of \textit{descriptorCount} valid \texttt{VkDescriptorBufferInfo} structures.

- **VUID-VkWriteDescriptorSet-descriptorType-00325**
  
  If \textit{descriptorType} is \texttt{VK_DESCRIPTOR_TYPE_SAMPLER} or \texttt{VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER}, and \textit{dstSet} was not allocated with a layout that included immutable samplers for \textit{dstBinding} with \textit{descriptorType}, the \textit{_sampler} member of each element of \textit{pImageInfo} \textbf{must} be a valid \texttt{VkSampler} object.

- **VUID-VkWriteDescriptorSet-descriptorType-02996**
  
  If \textit{descriptorType} is \texttt{VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER},
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`.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`, the `imageView` member of each element of `pImageInfo` must not be `VK_NULL_HANDLE`.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`, the `offset` member of each element of `pBufferInfo` must be a multiple of `VkPhysicalDeviceLimits::minUniformBufferOffsetAlignment`.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`, the `offset` member of each element of `pBufferInfo` must be a multiple of `VkPhysicalDeviceLimits::minStorageBufferOffsetAlignment`.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` and the `buffer` member of any element of `pBufferInfo` is the handle of a non-sparse buffer, then that buffer must be bound completely and contiguously to a single `VkDeviceMemory` object.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`, the `buffer` member of each element of `pBufferInfo` must have been created with `VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT` set.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`, the `buffer` member of each element of `pBufferInfo` must have been created with `VK_BUFFER_USAGE_STORAGE_BUFFER_BIT` set.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`, the `range` member of each element of `pBufferInfo`, or the effective range if `range` is `VK_WHOLE_SIZE`, must be less than or equal to `VkPhysicalDeviceLimits::maxUniformBufferRange`.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`, the `range` member of each element of `pBufferInfo`, or the effective range if `range` is `VK_WHOLE_SIZE`, must be less than or equal to `536`.
VkPhysicalDeviceLimits::maxStorageBufferRange

- VUID-VkWriteDescriptorSet-descriptorType-08765
  If `descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER`, the `pTexelBufferView` buffer view usage must include `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT`

- VUID-VkWriteDescriptorSet-descriptorType-08766
  If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER`, the `pTexelBufferView` buffer view usage must include `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT`

- 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

- 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,
} 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.

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.

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.

**Note**

When setting range to VK_WHOLE_SIZE, the effective range must not be larger than the maximum range for the descriptor type (maxUniformBufferRange or maxStorageBufferRange). This means that VK_WHOLE_SIZE is not typically useful in the common case where uniform buffer descriptors are suballocated from a buffer that is much larger than maxUniformBufferRange.

For VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC and VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC descriptor types, offset is the base offset from which the dynamic offset is applied and range is the static size used for all dynamic offsets.

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

- VUID-VkDescriptorBufferInfo-range-00341
  If range is not equal to VK_WHOLE_SIZE, range must be greater than 0

- VUID-VkDescriptorBufferInfo-range-00342
  If range is not equal to VK_WHOLE_SIZE, range must be less than or equal to the size of buffer minus offset

- VUID-VkDescriptorBufferInfo-buffer-02998
  If the nullDescriptor feature is not enabled, buffer must not be VK_NULL_HANDLE

Valid Usage (Implicit)

- VUID-VkDescriptorBufferInfo-buffer-parameter
  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

- VUID-VkDescriptorImageInfo-imageView-06712
  imageView must not be a 2D array image view created from a 3D image
**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`

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;
} VkCopyDescriptorSet;
```
VkCopyDescriptorSet
{
    VkDescriptorSet  dstSet;
    uint32_t          dstBinding;
    uint32_t          dstArrayElement;
    uint32_t          descriptorCount;
} VkCopyDescriptorSet;

• sType is a VkStructureType value identifying 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.
• dstSet, dstBinding, and dstArrayElement are the destination set, binding, and array element, respectively.
• descriptorCount is the number of descriptors to copy from the source to destination. If descriptorCount is greater than the number of remaining array elements in the source or destination binding, those affect consecutive bindings in a manner similar to VkWriteDescriptorSet above.

**Valid Usage**

• 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-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-04885
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 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-04887
  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 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 Set Binding

To bind one or more descriptor sets to a command buffer, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdBindDescriptorSets(  
    VkCommandBuffer commandBuffer,  
    VkPipelineBindPoint pipelineBindPoint,  
    VkPipelineLayout layout,  
    uint32_t firstSet,  
    uint32_t setCount,  
    const VkDescriptorSet* pDescriptorSets,  
    uint32_t stageMask)
```
uint32_t descriptorSetCount,
const VkDescriptorSet* pDescriptorSets,
uint32_t dynamicOffsetCount,
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.

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-graphicsPipelineLibrary-06754**
  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

- **VUID-vkCmdBindDescriptorSets-commonparent**
  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
**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>

### 14.2.6. 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.
• \textit{pValues} is a pointer to an array of \textit{size} bytes containing the new push constant values.

When a command buffer begins recording, all push constant values are undefined.

Push constant values \textbf{can} be updated incrementally, causing shader stages in \textit{stageFlags} to read the new data from \textit{pValues} for push constants modified by this command, while still reading the previous data for push constants not modified by this command. When a \textbf{bound pipeline command} is issued, the bound pipeline's layout \textbf{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 \textbf{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.

\textbf{Valid Usage}

\begin{itemize}
  \item VUID-vkCmdPushConstants-offset-01795
  For each byte in the range specified by \textit{offset} and \textit{size} and for each shader stage in \textit{stageFlags}, there \textbf{must} be a push constant range in \textit{layout} that includes that byte and that stage.
  \item VUID-vkCmdPushConstants-offset-01796
  For each byte in the range specified by \textit{offset} and \textit{size} and for each push constant range that overlaps that byte, \textit{stageFlags} \textbf{must} include all stages in that push constant range's \texttt{VkPushConstantRange::stageFlags}.
  \item VUID-vkCmdPushConstants-offset-00368
  \textit{offset} \textbf{must} be a multiple of 4.
  \item VUID-vkCmdPushConstants-size-00369
  \textit{size} \textbf{must} be a multiple of 4.
  \item VUID-vkCmdPushConstants-offset-00370
  \textit{offset} \textbf{must} be less than \texttt{VkPhysicalDeviceLimits::maxPushConstantsSize}.
  \item VUID-vkCmdPushConstants-size-00371
  \textit{size} \textbf{must} be less than or equal to \texttt{VkPhysicalDeviceLimits::maxPushConstantsSize} minus \textit{offset}.
\end{itemize}

\textbf{Valid Usage (Implicit)}

\begin{itemize}
  \item VUID-vkCmdPushConstants-commandBuffer-parameter
  \textit{commandBuffer} \textbf{must} be a valid \texttt{VkCommandBuffer} handle.
  \item VUID-vkCmdPushConstants-layout-parameter
  \textit{layout} \textbf{must} be a valid \texttt{VkPipelineLayout} handle.
\end{itemize}

\textbf{Note}

As \textit{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 \texttt{VkCommandPool} used to allocate \texttt{commandBuffer} was created on are ignored.
• VUID-vkCmdPushConstants-stageFlags-parameter
  stageFlags must be a valid combination of VkShaderStageFlagBits values

• VUID-vkCmdPushConstants-stageFlags-required bitmask
  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(
  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 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-bufferDeviceAddress-03324**
  - The `bufferDeviceAddress` feature must be enabled

- **VUID-vkGetBufferDeviceAddress-device-03325**
  - 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:

```c
// Provided by VK_VERSION_1_2
typedef struct VkBufferDeviceAddressInfo {
    VkStructureType     sType;
    VkDeviceAddress     bufferDeviceAddress;
    VkBufferDeviceAddressInfo::alignment alignment;
} VkBufferDeviceAddressInfo;
```
const void* pNext;
VkBuffer buffer;
} VkBufferDeviceAddressInfo;

• **sType** is a **VkStructureType** value identifying 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 `VkPipelineCreateInfo` 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

In Vulkan SC, the pipeline compilation process occurs offline using the implementation-provided pipeline cache compiler. The set of shaders being used to create a pipeline can be specified using the pipeline JSON schema.

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 numeric type or, recursively, composite types of such types. If an implementation supports storageInputOutput16, components can 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 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 Location slots 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 Location slots the type consumes.

The Location value specifies an interface slot comprised of a 32-bit four-component vector conveyed between stages. The Component specifies word components within these vector Location slots. 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 Location slots.

If a declared input or output is an array of size \( n \) and each element takes \( m \) Location slots, it will be assigned \( m \times n \) consecutive Location slots starting with the specified Location.

If the declared input or output is an \( n \times m \) 16-, 32- or 64-bit matrix, it will be assigned multiple Location slots starting with the specified Location. The number of Location slots 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 Location slots 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 Location slots 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 slot and Component word, either explicitly or implicitly. Any two outputs listed as operands on the same OpEntryPoint must not be assigned the same Location slot and Component word, either explicitly or implicitly.
The number of input and output Location slots available for a shader input or output interface is 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 Location slots available for the given interface, as specified in the “Locations Available” column in Shader Input and Output Locations.

Table 11. 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 Component word within a Location slot that are consumed. The Component word within a Location are 0, 1, 2, and 3. A variable or block member starting at Component N will consume Component words N, N+1, N+2, ... up through its size. For 16-, and 32-bit types, it is invalid if this sequence of Component words gets larger than 3. A scalar 64-bit type will consume two of these Component words in sequence, and a two-component 64-bit vector type will consume all four Component words available within a Location. A three- or four-component 64-bit vector type must not specify a non-zero Component decoration. A three-component 64-bit vector type will consume all four Component words of the first Location and Component 0 and 1 of the second Location. This leaves Component 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.

A four-component 64-bit data type will consume all four Component words of the first Location and all four Component words of the second Location.

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 Location slots 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 color attachment indicated by VkSubpassDescription::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 Location slots is given by the maxFragmentOutputAttachments member of the VkPhysicalDeviceLimits structure.

When an active fragment shader invocation finishes, the values of all fragment shader outputs are copied out and used as blend inputs or color attachments writes. If the invocation does not set a value for them, the input values to those blending or color attachment writes are undefined.

Components of the output variables are assigned as described in Component Assignment. Output Component words 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). Component words which do not correspond to any fragment shader output will also result in undefined values for blending or color attachment writes.

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.

There must be no output variable which has the same Location, Component, and Index as any other, 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 per image aspect. A multi-aspect image (e.g. a depth/stencil format) can use the same input variable. 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 numeric format of the subpass input must match the 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 numeric format 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:

1. 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.

2. 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.

3. 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:

1. typed as OpTypeStruct, or an array of this type,
2. identified with a Block or BufferBlock decoration, and
3. 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:

1. typed as OpTypeStruct, or an array of this type,
2. identified with a Block decoration, and
3. laid out explicitly using the Offset, ArrayStride, and MatrixStride decorations as specified in Offset and Stride Assignment.

The Offset decoration for any member of a Block-decorated variable in the Uniform storage class must not cause the space required for that variable to extend outside the range \([0, \text{maxUniformBufferRange})\). The Offset decoration for any member of a Block-decorated variable in the StorageBuffer storage class must not cause the space required for that variable to extend outside the range \([0, \text{maxStorageBufferRange})\).

Variables identified with 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 $Y' C_b C_r$ conversion, it must be indexed only by constant integral expressions when aggregated into arrays in shader code, irrespective of the shaderSampledImageArrayDynamicIndexing feature.

Table 12. Shader Resource and Descriptor Type Correspondence

<table>
<thead>
<tr>
<th>Resource type</th>
<th>Descriptor Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sampler</td>
<td>VK_DESCRIPTOR_TYPE_SAMPLER or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER</td>
</tr>
<tr>
<td>sampled image</td>
<td>VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER</td>
</tr>
<tr>
<td>storage image</td>
<td>VK_DESCRIPTOR_TYPE_STORAGE_IMAGE</td>
</tr>
<tr>
<td>combined image sampler</td>
<td>VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER</td>
</tr>
<tr>
<td>uniform texel buffer</td>
<td>VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER</td>
</tr>
<tr>
<td>storage texel buffer</td>
<td>VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER</td>
</tr>
<tr>
<td>uniform buffer</td>
<td>VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC</td>
</tr>
<tr>
<td>storage buffer</td>
<td>VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC</td>
</tr>
<tr>
<td>input attachment</td>
<td>VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT</td>
</tr>
</tbody>
</table>

Table 13. Shader Resource and Storage Class Correspondence

<table>
<thead>
<tr>
<th>Resource type</th>
<th>Storage Class</th>
<th>Type$^1$</th>
<th>Decoration(s)$^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sampler</td>
<td>UniformConstant</td>
<td>OpTypeSampler</td>
<td></td>
</tr>
<tr>
<td>sampled image</td>
<td>UniformConstant</td>
<td>OpTypeImage (Sampled=1)</td>
<td></td>
</tr>
<tr>
<td>storage image</td>
<td>UniformConstant</td>
<td>OpTypeImage (Sampled=2)</td>
<td></td>
</tr>
<tr>
<td>combined image sampler</td>
<td>UniformConstant</td>
<td>OpTypeSampledImage, OpTypeImage (Sampled=1), OpTypeSampler</td>
<td></td>
</tr>
<tr>
<td>uniform texel buffer</td>
<td>UniformConstant</td>
<td>OpTypeImage (Dim=Buffer, Sampled=1)</td>
<td></td>
</tr>
<tr>
<td>storage texel buffer</td>
<td>UniformConstant</td>
<td>OpTypeImage (Dim=Buffer, Sampled=2)</td>
<td></td>
</tr>
<tr>
<td>uniform buffer</td>
<td>Uniform</td>
<td>OpTypeStruct</td>
<td>Block, Offset, (ArrayStride), (MatrixStride)</td>
</tr>
<tr>
<td>Resource type</td>
<td>Storage Class</td>
<td>Type(^1)</td>
<td>Decoration(s)(^2)</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------</td>
<td>----------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>storage buffer</td>
<td>Uniform</td>
<td>OpTypeStruct</td>
<td>BufferBlock, Offset,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(ArrayStride), (MatrixStride)</td>
</tr>
<tr>
<td></td>
<td>StorageBuffer</td>
<td></td>
<td>Block, Offset, (ArrayStride), (MatrixStride)</td>
</tr>
<tr>
<td>input attachment</td>
<td>UniformConstant</td>
<td>OpTypeImage (Dim =SubpassData, Sampled=2)</td>
<td>InputAttachmentIndex</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 14. 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>maxPerStageDescriptorSampledImages or maxPerStageDescriptorUpdateAfterBindSampledImages</td>
<td>combined image sampler</td>
</tr>
<tr>
<td>maxPerStageDescriptorStorageImages or maxPerStageDescriptorUpdateAfterBindStorageImages</td>
<td>sampled image</td>
</tr>
<tr>
<td>maxPerStageDescriptorStorageImages or maxPerStageDescriptorUpdateAfterBindStorageImages</td>
<td>combined image sampler</td>
</tr>
<tr>
<td>maxPerStageDescriptorUniformBuffers or maxPerStageDescriptorUpdateAfterBindUniformBuffers</td>
<td>uniform texel buffer</td>
</tr>
<tr>
<td>maxPerStageDescriptorStorageBuffers or maxPerStageDescriptorUpdateAfterBindStorageBuffers</td>
<td>storage image</td>
</tr>
<tr>
<td>maxPerStageDescriptorStorageBuffers or maxPerStageDescriptorUpdateAfterBindStorageBuffers</td>
<td>storage texel buffer</td>
</tr>
<tr>
<td>maxPerStageDescriptorUniformBuffers or maxPerStageDescriptorUpdateAfterBindUniformBuffers</td>
<td>uniform buffer</td>
</tr>
<tr>
<td>maxPerStageDescriptorStorageBuffers or maxPerStageDescriptorUpdateAfterBindStorageBuffers</td>
<td>uniform buffer dynamic</td>
</tr>
<tr>
<td>maxPerStageDescriptorUniformBuffers or maxPerStageDescriptorUpdateAfterBindUniformBuffers</td>
<td>storage buffer</td>
</tr>
<tr>
<td>maxPerStageDescriptorStorageBuffers or maxPerStageDescriptorUpdateAfterBindStorageBuffers</td>
<td>storage buffer dynamic</td>
</tr>
</tbody>
</table>
### Resources per Stage

<table>
<thead>
<tr>
<th>Resources per Stage</th>
<th>Resource Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxPerStageDescriptorInputAttachments *maxPerStageDescriptorUpdateAfterBindInputAttachments</td>
<td>input attachment$^1$</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.
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 command.

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-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
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`.

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

- The `CullDistance` decoration must be used only within the `MeshEXT`, `MeshNV`, `Vertex`, `Fragment`, `TessellationControl`, `TessellationEvaluation`, or `Geometry Execution Model`.

- The variable decorated with `CullDistance` within the `MeshEXT`, `MeshNV` or `Vertex Execution Model` must be declared using the `Output Storage Class`.

- The variable decorated with `CullDistance` within the `Fragment Execution Model` must be declared using the `Input Storage Class`.

- 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`.

- 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 draw that invoked the current vertex shader invocation. For *indirect drawing commands*, `DrawIndex` begins at zero and increments by one for each draw executed. The number of draws 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**
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 `SampleMask` fragment shader input value that is zero.

Valid Usage

- VUID-HelperInvocation-HelperInvocation-04239
  The `HelperInvocation` decoration must be used only within the Fragment Execution Model
- VUID-HelperInvocation-HelperInvocation-04240
  The variable decorated with `HelperInvocation` must be declared using the `Input Storage` Class
- VUID-HelperInvocation-HelperInvocation-04241
  The variable decorated with `HelperInvocation` must be declared as a boolean value
**InvocationId**

Decorating a variable with the `InvocationId` built-in decoration will make that variable contain the index of the current shader invocation in a geometry shader, or the index of the output patch vertex in a tessellation control shader.

In a geometry shader, the index of the current shader invocation ranges from zero to the number of instances declared in the shader minus one. If the instance count of the geometry shader is one or is not specified, then `InvocationId` will be zero.

### Valid Usage

- VUID-InvocationId-InvocationId-04257
  The `InvocationId` decoration **must** be used only within the **TessellationControl** or **Geometry Execution Model**

- VUID-InvocationId-InvocationId-04258
  The variable decorated with `InvocationId` **must** be declared using the **Input Storage Class**

- VUID-InvocationId-InvocationId-04259
  The variable decorated with `InvocationId` **must** be declared as a scalar 32-bit integer value

**InstanceIndex**

Decorating a variable in a vertex shader with the `InstanceIndex` built-in decoration will make that variable contain the index of the instance that is being processed by the current vertex shader invocation. `InstanceIndex` begins at the `firstInstance` parameter to `vkCmdDraw` or `vkCmdDrawIndexed` or at the `firstInstance` member of a structure consumed by `vkCmdDrawIndirect` or `vkCmdDrawIndexedIndirect`.

### Valid Usage

- VUID-InstanceIndex-InstanceIndex-04263
  The `InstanceIndex` decoration **must** be used only within the **Vertex Execution Model**

- VUID-InstanceIndex-InstanceIndex-04264
  The variable decorated with `InstanceIndex` **must** be declared using the **Input Storage Class**

- VUID-InstanceIndex-InstanceIndex-04265
  The variable decorated with `InstanceIndex` **must** be declared as a scalar 32-bit integer value

**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

- **VUID-Layer-Layer-07039**
  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 TessellationEvaluationExecution 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`.

- **VUID-Position-Position-04321**
  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

- **VUID-PrimitiveId-Fragment-04332**
  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

- **VUID-PrimitiveId-Fragment-04333**
  If `Fragment Execution Model` contains a variable decorated with `PrimitiveId`, then either the `MeshShadingEXT, MeshShadingNV, Geometry` or `Tessellation` capability **must** also be declared

- **VUID-PrimitiveId-PrimitiveId-04334**
  The variable decorated with `PrimitiveId` within the `TessellationControl, TessellationEvaluation, Fragment, IntersectionKHR, AnyHitKHR, or ClosestHitKHR Execution Model` **must** be declared using the `Input Storage Class`

- **VUID-PrimitiveId-PrimitiveId-04335**
  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

- **VUID-SampleId-SampleId-04354**
  The `SampleId` decoration must be used only within the Fragment Execution Model.

- **VUID-SampleId-SampleId-04355**
  The variable decorated with `SampleId` must be declared using the Input Storage Class.

- **VUID-SampleId-SampleId-04356**
  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 × 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 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
The variable decorated with `SubgroupId` must be declared using the Input Storage Class

The variable decorated with `SubgroupId` must be declared as a scalar 32-bit integer 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].

Note
There is no direct relationship between SubgroupLocalInvocationId and LocalInvocationId or LocalInvocationIndex.

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.

The variable decorated with SubgroupSize will match subgroupSize.

The maximum number of invocations that an implementation can support per subgroup is 128.

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`.

- **VUID-VertexIndex-VertexIndex-04400**
  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 ViewIndex 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 ViewIndex 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 ViewIndex that is used. Outputs in previous shader stages are not used, even if the last stage fails to write the ViewIndex.

If the last active pre-rasterization shader stage shader entry point's interface does not include a variable decorated with ViewIndex then the first viewport is used. If a pre-rasterization shader stage shader entry point's interface includes a variable decorated with ViewIndex, it must write the same value to ViewIndex for all output vertices of a given primitive.

In a fragment shader, the variable decorated with ViewIndex contains the viewport index of the primitive that the fragment invocation belongs to.

Valid Usage

- VUID-ViewportIndex-ViewportIndex-04404
  The ViewIndex 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 ViewIndex decoration must be used only within the Geometry or Fragment Execution Model

- VUID-ViewportIndex-ViewportIndex-04406
  The variable decorated with ViewIndex 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**

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` 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.
- `OpImageGather` and `OpImageSparseGather` 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)


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 \text{Dim} of the image, and \(l\) conditionally present based on the \text{Arrayed} property of the image. \(n\) is conditionally present and is taken from the \text{Sample} image operand.

For all coordinate types, unused coordinates are assigned a value of zero.

\[
\begin{array}{cccccccc}
\textbf{i} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\textbf{j} & & & & & & & & \\
\textbf{k} & & & & & & & & \\
\textbf{l} & & & & & & & & \\
\textbf{n} & & & & & & & & \\
\end{array}
\]

\(0.0\) \(1.0\) \(2.0\) \(3.0\) \(4.0\) \(5.0\) \(6.0\) \(7.0\)

For the example shown of an \(8 \times 4\) texel two dimensional image.

- **Normalized texel coordinates:**
  - The \(s\) coordinate goes from 0.0 to 1.0.
  - The \(t\) coordinate goes from 0.0 to 1.0.

- **Unnormalized texel coordinates:**
  - The \(u\) coordinate within the range 0.0 to 8.0 is within the image, otherwise it is outside the image.
  - The \(v\) coordinate within the range 0.0 to 4.0 is within the image, otherwise it is outside the image.

- **Integer texel coordinates:**
  - The \(i\) coordinate within the range 0 to 7 addresses texels within the image, otherwise it is outside the image.
• The \( j \) coordinate within the range 0 to 3 addresses texels within the image, otherwise it is outside the image.

• Also shown for linear filtering:
  • Given the unnormalized coordinates \((u,v)\), the four texels selected are \(i_{0j_0}, i_{1j_0}, i_{0j_1}, \) and \(i_{1j_1}\).
  • The fractions \( \alpha \) and \( \beta \).
  • Given the offset \( \Delta_i \) and \( \Delta_j \), the four texels selected by the offset are \(i_{0j_0'}, i_{1j_0'}, i_{0j_1'}, \) and \(i_{1j_1'}\).

Note
For formats with reduced-resolution 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.

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:

- \(N = 9\)
- \(B = 15\)
- \(E_{\text{max}} = 31\)
- \(\text{sharedexp}_{\text{max}} = \frac{(2^N - 1)}{2^N} \times 2^{(E_{\text{max}} - B)}\)

**Note**

NaN, if supported, is handled as in IEEE 754-2008 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:

\[
\text{max}_{\text{shared}} = \left\lfloor \frac{\text{max}_{\text{clamped}}}{2^{(\text{exp}' - B - N)}} + \frac{1}{2} \right\rfloor
\]

\[
\text{exp}_{\text{shared}} = \begin{cases} 
\text{exp}' & \text{for } 0 \leq \text{max}_{\text{shared}} < 2^N \\
\text{exp}' + 1 & \text{for } \text{max}_{\text{shared}} = 2^N
\end{cases}
\]

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:

\[
\text{red}_{\text{shared}} = \left( \frac{\text{red}_{\text{clamped}}}{2^{(\text{exp}_{\text{shared}} - B - N)}} + \frac{1}{2} \right)
\]
\[
\text{green}_{\text{shared}} = \left( \frac{\text{green}_{\text{clamped}}}{2^{(\text{exp}_{\text{shared}} - B - N)}} + \frac{1}{2} \right)
\]
\[
\text{blue}_{\text{shared}} = \left( \frac{\text{blue}_{\text{clamped}}}{2^{(\text{exp}_{\text{shared}} - B - N)}} + \frac{1}{2} \right)
\]

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’C_bC_r conversion

For texel input instructions involving multiple texels (for sampling or gathering), these steps are applied for each texel that is used in the instruction. Depending on the type of image instruction, other steps are conditionally performed between these steps or involving multiple coordinate or texel values.

If Chroma Reconstruction is implicit, Texel Filtering instead takes place during chroma reconstruction, before sampler Y’C_bC_r 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 and the image view format is not one of the depth/stencil formats with a depth component, or the image view aspect is not...
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 SPIR-V Type.
- 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 YC6C4 conversion.

`OpImageFetch, OpImageSparseFetch, OpImage*Gather, and OpImageSparse*Gather` must not be used with a sampler or image view that enables sampler YC6C4 conversion.

The `ConstOffset` and `Offset` operands must not be used with a sampler or image view that enables sampler YC6C4 conversion.

**Integer Texel Coordinate Validation**

Integer texel coordinates are validated against the size of the image level, and the number of layers and number of samples in the image. For SPIR-V instructions that use integer texel coordinates, this is performed directly on the integer coordinates. For instructions that use normalized or unnormalized texel coordinates, this is performed on the coordinates that result after conversion to integer texel coordinates.

If the integer texel coordinates do not satisfy all of the conditions

\[ 0 \leq i < w_s \]
\[ 0 \leq j < h_s \]
\[ 0 \leq k < d_s \]
\[ 0 \leq l < \text{layers} \]
\[ 0 \leq n < \text{samples} \]

where:

\[ w_s = \text{width of the image level} \]
\[ h_s = \text{height of the image level} \]
\[ d_s = \text{depth of the image level} \]

\[ \text{layers} = \text{number of layers in the image} \]
\[ \text{samples} = \text{number of samples per texel in the image} \]

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 **texel replacement** is performed.

3. **Invalid Texel**
   
   ◦ If the texel coordinates fail validation, and
   
   ◦ If the read is the result of an image fetch instruction, image read instruction, or atomic
     instruction,
   
   then the texel is an invalid texel and **texel replacement** is performed.
4. Cube Map Edge or Corner

Otherwise the texel coordinates lie beyond the edges or corners of the selected cube map face, and Cube map edge handling is performed.

Cube Map Edge Handling

If the texel coordinates lie beyond the edges or corners of the selected cube map face (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 Y’C_yC_b 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>
<thead>
<tr>
<th>Sampler</th>
<th>Corresponding Border Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK</td>
<td>([0.0, 0.0, 0.0, 0.0])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK</td>
<td>([0.0, 0.0, 0.0, 1.0])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE</td>
<td>([1.0, 1.0, 1.0, 1.0])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_INT_TRANSPARENT_BLACK</td>
<td>([0, 0, 0, 0])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_INT_OPAQUE_BLACK</td>
<td>([0, 0, 0, 1])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_INT_OPAQUE_WHITE</td>
<td>([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>
<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>
The value returned by a read of an invalid texel is undefined, unless that read operation is from a buffer resource and the `robustBufferAccess` feature is enabled. In that case, an invalid texel is replaced as described by the `robustBufferAccess` feature.

If the `VkPhysicalDeviceSparseProperties::residencyNonResidentStrict` property is `VK_TRUE`, a sparse unbound texel is replaced with 0 or 0.0 values for integer and floating-point components of the image format, respectively.

If `residencyNonResidentStrict` is `VK_FALSE`, the value of the sparse unbound texel is undefined.

### 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 \( D_{\text{ref}} \) and the texel depth value \( D_{\text{tex}} \) 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 \( D_{\text{ref}} \) 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>
<thead>
<tr>
<th>Texel Aspect or Format</th>
<th>RGBA Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth aspect</td>
<td>([\text{Color}_r, \text{Color}_g, \text{Color}_b, \text{Color}_a] = [D,0,0,one])</td>
</tr>
<tr>
<td>Stencil aspect</td>
<td>([\text{Color}_r, \text{Color}_g, \text{Color}_b, \text{Color}_a] = [S,0,0,one])</td>
</tr>
<tr>
<td>One component color format</td>
<td>([\text{Color}_r, \text{Color}_g, \text{Color}_b, \text{Color}_a] = [\text{Color}_r,0,0,one])</td>
</tr>
<tr>
<td>Two component color format</td>
<td>([\text{Color}_r, \text{Color}_g, \text{Color}_b, \text{Color}_a] = [\text{Color}_r,\text{Color}_g,0,one])</td>
</tr>
<tr>
<td>Three component color format</td>
<td>([\text{Color}_r, \text{Color}_g, \text{Color}_b, \text{Color}_a] = [\text{Color}_r,\text{Color}_g,\text{Color}_b,one])</td>
</tr>
<tr>
<td>Four component color format</td>
<td>([\text{Color}_r, \text{Color}_g, \text{Color}_b, \text{Color}_a] = [\text{Color}_r,\text{Color}_g,\text{Color}_b,\text{Color}_a])</td>
</tr>
</tbody>
</table>

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’C_b conversion is not enabled, and
- the VkComponentSwizzle enums in the components member of the VkSamplerYcbcrConversionCreateInfo structure for the sampler Y’C_b conversion if sampler Y’C_b 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.

If the image view has a depth/stencil format and the VkComponentSwizzle is VK_COMPONENT_SWIZZLE_ONE, 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.

- “\texttt{444}” formats do not spatially downsample chroma values compared with luma: there are unique chroma samples for each texel.
- “\texttt{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.
- “\texttt{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 $x_{\text{ChromaOffset}}$ and $y_{\text{ChromaOffset}}$ members of the VkSamplerYcbcrConversionCreateInfo structure used to create the sampler Y’C$_b$C$_r$ 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. 420 downsampling, xChromaOffset=COSITED_EVEN, yChromaOffset=COSITED_EVEN

Figure 8. 420 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_SAMPL...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_SAMPL...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 $\tau_{ij}[level]$ values accessed by texel filtering are reconstructed as follows:
    \[
    \tau_R'(i, j) = \tau_R([i \times 0.5], j)[level] \\
    \tau_B'(i, j) = \tau_B([i \times 0.5], j)[level]
    \]
  - 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 $\tau_{ij}[level]$ values accessed by texel filtering are reconstructed as follows:
    \[
    \tau_R'(i, j) = \tau_R([i \times 0.5], [j \times 0.5])[level] \\
    \tau_B'(i, j) = \tau_B([i \times 0.5], [j \times 0.5])[level]
    \]

  *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:
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 \(\text{chromaFilter} = \text{VK_FILTER_LINEAR}\), 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 \(\text{VK_FILTER_NEAREST}\), an implementation may behave as if \(\text{xChromaOffset}\) and \(\text{yChromaOffset}\) were both \(\text{VK_CHROMA_LOCATION_MIDPOINT}\), irrespective of the values set.

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):

\[
\begin{align*}
i_{RB} &= \begin{cases} 
0.5 \times (i) & \text{xChromaOffset} = \text{COSITED_EVEN} \\
0.5 \times (i - 0.5) & \text{xChromaOffset} = \text{MIDPOINT}
\end{cases} \\
j_{RB} &= \begin{cases} 
0.5 \times (j) & \text{yChromaOffset} = \text{COSITED_EVEN} \\
0.5 \times (j - 0.5) & \text{yChromaOffset} = \text{MIDPOINT}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
i_{floor} &= \lfloor i_{RB} \rfloor \\
j_{floor} &= \lfloor j_{RB} \rfloor \\
i_{frac} &= i_{RB} - i_{floor} \\
j_{frac} &= j_{RB} - j_{floor}
\end{align*}
\]

\[
\tau_{RB}(i, j) = \tau_{RB}(i_{floor}, j_{floor})(level) \times (1 - i_{frac}) \times (1 - j_{frac}) + \\
\tau_{RB}(1 + i_{floor}, j_{floor})(level) \times (i_{frac}) \times (1 - j_{frac}) + \\
\tau_{RB}(i_{floor}, 1 + j_{floor})(level) \times (1 - i_{frac}) \times (j_{frac}) + \\
\tau_{RB}(1 + i_{floor}, 1 + j_{floor})(level) \times (i_{frac}) \times (j_{frac})
\]

\[
\begin{align*}
u_{RB}'(422 / 420) &= \begin{cases} 
0.5 \times (u + 0.5), & \text{xChromaOffset} = \text{COSITED_EVEN} \\
0.5 \times u, & \text{xChromaOffset} = \text{MIDPOINT}
\end{cases} \\
v_{RB}'(420) &= \begin{cases} 
0.5 \times (v + 0.5), & \text{yChromaOffset} = \text{COSITED_EVEN} \\
0.5 \times v, & \text{yChromaOffset} = \text{MIDPOINT}
\end{cases}
\end{align*}
\]
16.3.9. Sampler Y’C₇B₇C₇R Conversion

Sampler Y’C₇B₇C₇R conversion performs the following operations, which an implementation may combine into a single mathematical operation:

- Sampler Y’C₇B₇C₇R Range Expansion
- Sampler Y’C₇B₇C₇R Model Conversion

Sampler Y’C₇B₇C₇R Range Expansion

Sampler Y’C₇B₇C₇R range expansion is applied to color component values after all texel input operations which are not specific to sampler Y’C₇B₇C₇R conversion. For example, the input values to this stage have been converted using the normal format conversion rules.

Sampler Y’C₇B₇C₇R 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’C₇B₇C₇R 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’C₇B₇C₇R range expansion.

The range expansion to be applied is defined by the ycbcrRange member of the VkSamplerYcbcrConversionCreateInfo structure:

- If ycbcrRange is VK_SAMPLER_YCBCR_RANGE_ITU_FULL, the following transformations are applied:

\[
\begin{align*}
Y' &= C'rgba[G] \\
C_B &= C'rgba[B] - \frac{2^{(n-1)}}{(2^n) - 1} \\
C_R &= C'rgba[R] - \frac{2^{(n-1)}}{(2^n) - 1}
\end{align*}
\]

**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.

\[
Y' = \frac{C'_{rgb} [G] \times (2^n - 1) - 16 \times 2^n - 8}{219 \times 2^n - 8}
\]
\[
C_B = \frac{C'_{rgb} [B] \times (2^n - 1) - 128 \times 2^n - 8}{224 \times 2^n - 8}
\]
\[
C_R = \frac{C'_{rgb} [R] \times (2^n - 1) - 128 \times 2^n - 8}{224 \times 2^n - 8}
\]

• \( n \) 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 \( C_B \) and \( C_R \) fall in the range \([-0.5,0.5]\).

**Sampler Y’C_bC_R Model Conversion**

The range-expanded values are converted between color models, according to the color model conversion specified in the \texttt{ycbcrModel} member:

**\texttt{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; \( Y'C_bC_R \) range expansion is also ignored.

**\texttt{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 \( Y'C_bC_R \) form both in memory and in the shader; \( Y'C_bC_R \) range expansion is applied to the components as for other \( Y'C_bC_R \) models, with the vector \((C_R,Y',C_B,A)\) provided to the shader.

**\texttt{VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_709}**

The color components are transformed from a \( Y'C_bC_R \) representation to an \( R'G'B' \) representation as described in the “BT.709 \ Y'C_bC_R conversion” section of the Khronos Data Format Specification.

**\texttt{VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_601}**

The color components are transformed from a \( Y'C_bC_R \) representation to an \( R'G'B' \) representation as described in the “BT.601 \ Y'C_bC_R conversion” section of the Khronos Data Format Specification.

**\texttt{VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_2020}**

The color components are transformed from a \( Y'C_bC_R \) representation to an \( R'G'B' \) representation as described in the “BT.2020 \ Y'C_bC_R conversion” section of the Khronos Data Format Specification.
In this operation, each output 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'CbCr conversion at this location.

- For a minFilter or magFilter of VK_FILTER_LINEAR:
  1. Calculate \((i_0,j_0)\) for the sample location as described under the “linear filtering” formulae in \((u,v,w,a)\) to \((i,j,k,l,n)\) Transformation and Array Layer Selection
  2. Calculate the normalized texel coordinates corresponding to these integer coordinates.
  3. Sample using sampler Y'CbCr conversion at each of these locations.
  4. Convert the non-linear A'R'G'B' outputs of the Y'CbCr conversions to linear ARGB values as described in the “Transfer Functions” chapter of the Khronos Data Format Specification.
  5. Interpolate the linear ARGB values using the \(\alpha\) and \(\beta\) values described in the “linear filtering” section of \((u,v,w,a)\) to \((i,j,k,l,n)\) Transformation and Array Layer Selection and the equations in Texel Filtering.

The additional calculations and, especially, additional number of sampling operations in the VK_FILTER_LINEAR case can be expected to have a performance impact compared with using the outputs directly. Since the 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’C_bC_r 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 *SPIR-V Type*, 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 \(D_{ref}\) coordinate are transformed as follows:

\[
\begin{align*}
  s &= \frac{s}{q}, & \text{for 1D, 2D, or 3D image} \\
  t &= \frac{t}{q}, & \text{for 2D or 3D image} \\
  r &= \frac{r}{q}, & \text{for 3D image} \\
  D_{ref} &= \frac{D_{ref}}{q}, & \text{if provided}
\end{align*}
\]

#### 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:
\[ \frac{\partial s}{\partial x} = dPdx(s), \quad \frac{\partial s}{\partial y} = dPdy(s), \quad \text{for 1D, 2D, Cube, or 3D image} \]
\[ \frac{\partial t}{\partial x} = dPdx(t), \quad \frac{\partial t}{\partial y} = dPdy(t), \quad \text{for 2D, Cube, or 3D image} \]
\[ \frac{\partial r}{\partial x} = dPdx(r), \quad \frac{\partial r}{\partial y} = dPdy(r), \quad \text{for Cube or 3D image} \]

Partial derivatives not defined above for certain image dimensionalities are set to zero.

For explicit LOD image instructions, if the optional SPIR-V operand \texttt{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 \texttt{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 LOD Operation.

If the image or sampler object used by an implicit derivative image instruction is not uniform across the quad and \texttt{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 \texttt{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 \( s, t, r \), and the selection of derivatives, are determined by the major axis direction as specified in the following two tables.

\begin{table}[h]
\centering
\caption{Cube map face and coordinate selection}
\end{table}
### Table 19. Cube map derivative selection

<table>
<thead>
<tr>
<th>Major Axis Direction</th>
<th>Layer Number</th>
<th>Cube Map Face</th>
<th>$\partial s_c / \partial x$</th>
<th>$\partial s_c / \partial y$</th>
<th>$\partial t_c / \partial x$</th>
<th>$\partial t_c / \partial y$</th>
<th>$\partial r_c / \partial x$</th>
<th>$\partial r_c / \partial y$</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>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$+r_y$</td>
<td>2</td>
<td>Positive Y</td>
<td>$+r_x$</td>
<td>$+r_z$</td>
<td></td>
<td></td>
<td>$r_y$</td>
<td></td>
</tr>
<tr>
<td>$-r_y$</td>
<td>3</td>
<td>Negative Y</td>
<td>$+r_x$</td>
<td>$-r_z$</td>
<td></td>
<td></td>
<td>$r_y$</td>
<td></td>
</tr>
<tr>
<td>$+r_z$</td>
<td>4</td>
<td>Positive Z</td>
<td>$+r_x$</td>
<td>$-r_y$</td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$$s_{\text{face}} = \frac{1}{2} \times \frac{s_c}{|r_c|} + \frac{1}{2}$$

$$t_{\text{face}} = \frac{1}{2} \times \frac{t_c}{|r_c|} + \frac{1}{2}$$

#### 16.5.5. Cube Map Coordinate Transformation

#### 16.5.6. Cube Map Derivative Transformation
16.5.7. Scale Factor Operation, LOD 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 \( \text{mipmapPrecisionBits} \) of accuracy.

**Scale Factor Operation**

The magnitude of the derivatives are calculated by:

\[
\begin{align*}
\frac{\partial s_{\text{face}}}{\partial y} &= \frac{1}{2} \times \left( \frac{|r_c| \times \partial s_c / \partial y - s_c \times \partial r_c / \partial y}{(r_c)^2} \right) \\
\frac{\partial t_{\text{face}}}{\partial x} &= \frac{1}{2} \times \left( \frac{|r_c| \times \partial t_c / \partial x - t_c \times \partial r_c / \partial x}{(r_c)^2} \right) \\
\frac{\partial t_{\text{face}}}{\partial y} &= \frac{1}{2} \times \left( \frac{|r_c| \times \partial t_c / \partial y - t_c \times \partial r_c / \partial y}{(r_c)^2} \right)
\end{align*}
\]

where:

\( \partial t/\partial x = \partial t/\partial y = 0 \) (for 1D images)

\( \partial r/\partial x = \partial r/\partial y = 0 \) (for 1D, 2D or Cube images)

and:

\( w_{\text{base}} = \text{image.w} \)
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:

- $f_x$ is continuous and monotonically increasing in each of $m_{ux}$, $m_{vx}$, and $m_{wx}$
- $f_y$ is continuous and monotonically increasing in each of $m_{uy}$, $m_{vy}$, and $m_{wy}$

$$\max(|m_{ux}|, |m_{vx}|, |m_{wx}|) \leq f_x \leq \sqrt{2(|m_{ux}| + |m_{vx}| + |m_{wx}|)}$$

$$\max(|m_{uy}|, |m_{vy}|, |m_{wy}|) \leq f_y \leq \sqrt{2(|m_{uy}| + |m_{vy}| + |m_{wy}|)}$$

The minimum and maximum scale factors ($\rho_{\text{min}}, \rho_{\text{max}}$) are determined by:

$$\rho_{\text{max}} = \max(\rho_x, \rho_y)$$

$$\rho_{\text{min}} = \min(\rho_x, \rho_y)$$

The ratio of anisotropy is determined by:

$$\eta = \min(\rho_{\text{max}}/\rho_{\text{min}}, \text{maxAniso})$$

where:

- sampler.maxAniso = maxAnisotropy (from sampler descriptor)
- limits.maxAniso = maxSamplerAnisotropy (from physical device limits)

$$\text{maxAniso} = \min(\text{sampler.maxAniso}, \text{limits.maxAniso})$$

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 \( \eta \), even when \( \rho_{\min} \) is zero or close to zero. If either \texttt{VkPhysicalDeviceFeatures::samplerAnisotropy} or \texttt{VkSamplerCreateInfo::anisotropyEnable} are \texttt{VK_FALSE}, \( \text{maxAniso} \) 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 \( \eta \).

### LOD Operation

The LOD parameter \( \lambda \) is computed as follows:

\[
\lambda_{\text{base}}(x, y) = \begin{cases} 
\text{shaderOp.Lod} & \text{(from optional SPIR-V operand)} \\
\log_{\eta} \left( \frac{\rho_{\max}}{\rho} \right) & \text{otherwise}
\end{cases}
\]

\[
\lambda'(x, y) = \lambda_{\text{base}} + \text{clamp} \left( \text{sampler.bias} + \text{shaderOp.bias}, -\text{maxSamplerLodBias}, \text{maxSamplerLodBias} \right)
\]

\[
\lambda = \begin{cases} 
lod_{\max}, & \lambda' > lod_{\max} \\
\lambda', & lod_{\min} \leq \lambda' \leq lod_{\max} \\
\text{undefined}, & lod_{\min} > lod_{\max}
\end{cases}
\]

where:

\[
\text{sampler.bias} = \text{mipLodBias}
\]

\[
\text{shaderOp.bias} = \begin{cases} 
\text{Bias} & \text{(from optional SPIR-V operand)} \\
0 & \text{otherwise}
\end{cases}
\]

\[
\text{shaderOp.lod}_{\min} = \text{minLod}
\]

\[
\text{lod}_{\min} = \max(\text{sampler.lod}_{\min}, \text{shaderOp.lod}_{\min})
\]

\[
\text{lod}_{\max} = \maxLod
\]

and \text{maxSamplerLodBias} is the value of the \texttt{VkPhysicalDeviceLimits} feature \texttt{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:

\[
d_{l} = \begin{cases} 
\text{nearest}(d'), & \text{mipmapMode is VK_SAMPLER_MIPMAP_MODE_NEAREST} \\
d', & \text{otherwise}
\end{cases}
\]

where:

\[
d' = \text{level}_{\text{base}} + \text{clamp}(\lambda, 0, q)
\]
and:

\[
\text{nearest}(d^{'}) = \begin{cases} 
[d^{'} + 0.5] - 1, & \text{preferred} \\
|d^{'} + 0.5|, & \text{alternative}
\end{cases}
\]

\[
\text{level}_{\text{base}} = \text{baseMipLevel}
\]
\[
q = \text{levelCount} - 1
\]

\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 \(d = d_{l}\).

If the sampler’s \text{mipmapMode} is \text{VK_SAMPLER_MIPMAP_MODE_LINEAR}, two neighboring levels are selected:

\[
\begin{align*}
\delta &= d_{l} - d_{hi} \\
\delta &= \text{min}(d_{hi} + 1, \text{level}_{\text{base}} + q)
\end{align*}
\]

\(\delta\) is the fractional value, quantized to the number of \text{mipmap precision bits}, used for \text{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 \(d_{lo}\)).

\[
\begin{align*}
\text{u}(x, \ y) &= s(x, \ y) \times \text{width}_{\text{scale}} + \Delta_{i} \\
\text{v}(x, \ y) &= \begin{cases} 
0 & \text{for 1D images} \\
t(x, \ y) \times \text{height}_{\text{scale}} + \Delta_{j} & \text{otherwise}
\end{cases} \\
\text{w}(x, \ y) &= \begin{cases} 
0 & \text{for 2D or Cube images} \\
r(x, \ y) \times \text{depth}_{\text{scale}} + \Delta_{k} & \text{otherwise}
\end{cases} \\
\text{a}(x, \ y) &= \begin{cases} 
\text{a}(x, \ y) & \text{for array images} \\
0 & \text{otherwise}
\end{cases}
\end{align*}
\]

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 `layerCount` is the number of layers in the image subresource range of the image view, `baseArrayLayer` is the first layer from the subresource range, and where:

\[
\text{RNE}(a) = \begin{cases} 
\text{roundTiesToEven}(a) & \text{preferred, from IEEE Std 754-2008 Floating-Point Arithmetic} \\
[a + 0.5] & \text{alternative}
\end{cases}
\]

The sample index \(n\) is assigned the value 0.

Nearest filtering (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 (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 \(\gamma\).

\[
i_0 = \lfloor u - \text{shift} \rfloor \\
i_1 = i_0 + 1 \\
j_0 = \lfloor v - \text{shift} \rfloor \\
j_1 = j_0 + 1 \\
k_0 = \lfloor w - \text{shift} \rfloor \\
k_1 = k_0 + 1
\]
where:

\[ \alpha = \text{frac}(u - \text{shift}) \]
\[ \beta = \text{frac}(v - \text{shift}) \]
\[ \gamma = \text{frac}(w - \text{shift}) \]

where:

\[ \text{shift} = 0.5 \]

and where:

\[ \text{frac}(x) = x - \lfloor x \rfloor \]

where the number of fraction bits retained is specified by \( \text{VkPhysicalDeviceLimits}::\text{subTexelPrecisionBits} \).

16.7. Integer Texel Coordinate Operations

The \text{OpImageFetch} and \text{OpImageFetchSparse} SPIR-V instructions \textit{may} supply a LOD from which texels are to be fetched using the optional SPIR-V operand \textit{Lod}. Other integer-coordinate operations \textbf{must} not. If the \textit{Lod} is provided then it \textbf{must} be an integer.

The image level selected is:

\[ d = \text{level}_{\text{base}} + \begin{cases} 
\text{Lod} & \text{(from optional SPIR-V operand)} \\
0 & \text{otherwise} 
\end{cases} \]

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 \textit{VK_FILTER_NEAREST} is used within a mip level then \textit{VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE} is used, and if \textit{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 \textit{addressModeU} parameter of the sampler.
where:

\[
\text{mirror}(n) = \begin{cases} 
  n & \text{for } n \geq 0 \\
  -(1 + n) & \text{otherwise}
\end{cases}
\]

\(j\) (for 2D and Cube image) and \(k\) (for 3D image) are similarly transformed based on the \(\text{addressModeV}\) and \(\text{addressModeW}\) parameters of the sampler, respectively.

### 16.8.2. Texel Gathering

SPIR-V instructions with **Gather** in the name return a vector derived from 4 texels in the base level of the image view. The rules for the **VK_FILTER_LINEAR** minification filter are applied to identify the four selected texels. Each texel is then converted to an RGBA value according to **conversion to RGBA** and then **swizzled**. A four-component vector is then assembled by taking the component indicated by the **Component** value in the instruction from the swizzled color value of the four texels. If the operation does not use the **ConstOffsets** image operand then the four texels form the \(2 \times 2\) rectangle used for texture filtering:

\[
\begin{align*}
\tau[R] &= \tau_{i0j1}[\text{level}_\text{base}][\text{comp}] \\
\tau[G] &= \tau_{i1j0}[\text{level}_\text{base}][\text{comp}] \\
\tau[B] &= \tau_{i1j0}[\text{level}_\text{base}][\text{comp}] \\
\tau[A] &= \tau_{i0j0}[\text{level}_\text{base}][\text{comp}]
\end{align*}
\]

If the operation does use the **ConstOffsets** image operand then the offsets allow a custom filter to be defined:

\[
\begin{align*}
\tau[R] &= \tau_{i0j0} + \Delta_0[\text{level}_\text{base}][\text{comp}] \\
\tau[G] &= \tau_{i0j0} + \Delta_1[\text{level}_\text{base}][\text{comp}] \\
\tau[B] &= \tau_{i0j0} + \Delta_2[\text{level}_\text{base}][\text{comp}] \\
\tau[A] &= \tau_{i0j0} + \Delta_3[\text{level}_\text{base}][\text{comp}]
\end{align*}
\]

where:

\[
\tau[\text{level}_\text{base}][\text{comp}] = \begin{cases} 
  \tau[\text{level}_\text{base}][R], & \text{for } \text{comp} = 0 \\
  \tau[\text{level}_\text{base}][G], & \text{for } \text{comp} = 1 \\
  \tau[\text{level}_\text{base}][B], & \text{for } \text{comp} = 2 \\
  \tau[\text{level}_\text{base}][A], & \text{for } \text{comp} = 3
\end{cases}
\]

\(\text{comp}\) 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.

\[
\tau[level] = \begin{cases} 
\tau_{ijk}[level], & \text{for 3D image} \\
\tau_{ij}[level], & \text{for 2D or Cube image} \\
\tau_i[level], & \text{for 1D image}
\end{cases}
\]

**Texel Linear Filtering**

Within a mip level, VK_FILTER_LINEAR filtering combines \( 8 \) (for 3D), \( 4 \) (for 2D or Cube), or \( 2 \) (for 1D) texel values, together with their linear weights. The linear weights are derived from the fractions computed earlier:

\[
\begin{align*}
  w_{i_0} &= (1 - \alpha) \\
  w_{i_1} &= \alpha \\
  w_{j_0} &= (1 - \beta) \\
  w_{j_1} &= \beta \\
  w_{k_0} &= (1 - \gamma) \\
  w_{k_1} &= \gamma
\end{align*}
\]

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 values, together with their weights, computing a component-wise minimum or maximum, respectively, of the components of the values with non-zero weights.

**Texel Mipmap Filtering**

`VK_SAMPLER_MIPMAP_MODE_NEAREST` filtering returns the value of a single mipmap level,

\[ \tau = \tau[d] \]

`VK_SAMPLER_MIPMAP_MODE_LINEAR` filtering combines the values of multiple mipmap levels (\( \tau[hi] \) and \( \tau[lo] \)), together with their linear weights.

The linear weights are derived from the fraction computed earlier:

\[ w_{hi} = (1 - \delta) \]
\[ w_{lo} = (\delta) \]

The values of multiple mipmap levels, together with their weights, are combined to produce a final filtered value.

The `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:

\[ \tau = (w_{hi})\tau[hi] + (w_{lo})\tau[lo] \]

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 Anisotropic Filtering**

Anisotropic filtering is enabled by the `anisotropyEnable` in the sampler. When enabled, the image filtering scheme accounts for a degree of anisotropy.

The particular scheme for anisotropic texture filtering is implementation-dependent.
Implementations should consider the `magFilter`, `minFilter` and `mipmapMode` of the sampler to control the specifics of the anisotropic filtering scheme used. In addition, implementations should consider `minLod` and `maxLod` of the sampler.

**Note**

For historical reasons, vendor implementations of anisotropic filtering interpret these sampler parameters in different ways, particularly in corner cases such as `magFilter`, `minFilter` of `NEAREST` or `maxAnisotropy` equal to 1.0. Applications should not expect consistent behavior in such cases, and should use anisotropic filtering only with parameters which are expected to give a quality improvement relative to `LINEAR` filtering.

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 $\tau_{2D\text{aniso}}$ is defined using the single isotropic $\tau_{2D}(u,v)$ at level d.

$$
\tau_{2D\text{aniso}} = \frac{1}{N} \sum_{i=1}^{N} \tau_{2D}\left(u\left(x - \frac{1}{2} + \frac{i}{N+1}, v\right), v\left(x - \frac{1}{2} + \frac{i}{N+1}, y\right)\right), \quad \text{when } \rho_x > \rho_y
$$

$$
\tau_{2D\text{aniso}} = \frac{1}{N} \sum_{i=1}^{N} \tau_{2D}\left(u\left(x, y - \frac{1}{2} + \frac{i}{N+1}, v\right), v\left(x, y - \frac{1}{2} + \frac{i}{N+1}, \right)\right), \quad \text{when } \rho_y \geq \rho_x
$$

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, LOD 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 (i,j,k,l,n) Transformation And Array Layer Selection: Performed by all OpImageSample, OpImageSparseSample, and OpImage*Gather instructions.

• Texel Gathering: Performed by OpImage*Gather instructions.

• Texel Filtering: Performed by all OpImageSample* and OpImageSparseSample* instructions.

• Sparse Residency: Performed by all OpImageSparse* instructions.

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, LOD Operation and Image Level(s) Selection. The return value is the vector (λ', d). 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, // device is the logical device that creates the query pool.
    const VkQueryPoolCreateInfo* pCreateInfo, // pCreateInfo is a pointer to a VkQueryPoolCreateInfo structure containing the number and type of queries to be managed by the pool.
    const VkAllocationCallbacks* pAllocator, // pAllocator controls host memory allocation as described in the Memory Allocation chapter.
    VkQueryPool* pQueryPool) // pQueryPool is a pointer to a VkQueryPool handle in which the resulting query pool object is returned.
```

If VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations is VK_TRUE, vkCreateQueryPool must not return VK_ERROR_OUT_OF_HOST_MEMORY.

Valid Usage

- VUID-vkCreateQueryPool-device-05068
  The number of query pools currently allocated from device plus 1 must be less than or equal to the total number of query pools requested via VkDeviceObjectReservationCreateInfo::queryPoolRequestCount specified when device was
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-null
pAllocator must be NULL

- 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 a VkStructureType value identifying 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

- **VUID-VkQueryPoolCreateInfo-queryType-05046**
  If queryType is VK_QUERY_TYPE_OCCLUSION then queryCount must be less than or equal to the maximum of all VkDeviceObjectReservationCreateInfo::maxOcclusionQueriesPerPool values specified when device was created

- **VUID-VkQueryPoolCreateInfo-queryType-05047**
  If queryType is VK_QUERY_TYPE_PIPELINE_STATISTICS then queryCount must be less than or equal to the maximum of all VkDeviceObjectReservationCreateInfo::maxPipelineStatisticsQueriesPerPool values specified when device was created

- **VUID-VkQueryPoolCreateInfo-queryType-05048**
  If queryType is VK_QUERY_TYPE_TIMESTAMP then queryCount must be less than or equal to the maximum of all VkDeviceObjectReservationCreateInfo::maxTimestampQueriesPerPool values specified when device was created

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

```c
// 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.

Query pools cannot be destroyed [SCID-4]. If VkPhysicalDeviceVulkanSC10Properties
deviceDestroyFreesMemory is VK_TRUE, the memory is returned to the system when the device is destroyed.

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`, 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`, or `vkCmdWriteTimestamp`. Both the availability status and numerical results can be retrieved by calling either `vkGetQueryPoolResults` or `vkCmdCopyQueryPoolResults`.

After query pool creation, each query is in an uninitialized state and 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:

```c
// Provided by VK_VERSION_1_0
void vkCmdResetQueryPool(
    VkCommandBuffer commandBuffer,
    VkQueryPool      queryPool,
```
uint32_t firstQuery,
uint32_t queryCount);

- **commandBuffer** is the command buffer into which this command will be recorded.
- **queryPool** is the handle of the query pool managing the queries being reset.
- **firstQuery** is the initial query index to reset.
- **queryCount** is the number of queries to reset.

When executed on a queue, this command sets the status of query indices \([\text{firstQuery}, \text{firstQuery} + \text{queryCount} - 1]\) 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-09436**
  \(\text{firstQuery must be less than the number of queries in queryPool}\)

- **VUID-vkCmdResetQueryPool-firstQuery-09437**
  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**
  \(\text{commandBuffer must be a valid VkCommandBuffer handle}\)

- **VUID-vkCmdResetQueryPool-queryPool-parameter**
  \(\text{queryPool must be a valid VkQueryPool handle}\)

- **VUID-vkCmdResetQueryPool-commandBuffer-recording**
  \(\text{commandBuffer must be in the recording state}\)

- **VUID-vkCmdResetQueryPool-commandBuffer-cmdpool**
  The **VkCommandPool** that **commandBuffer** was allocated from must support graphics, or compute operations
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.
The sum of `firstQuery` and `queryCount` must be less than or equal to the number of queries in `queryPool`.

- VUID-vkResetQueryPool-None-02665
  The `hostQueryReset` feature must be enabled

- 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_TYPEPIPELINE_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_TYPEPIPELINE_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, or compute 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</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td>State</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-vkCmdEndQuery-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-vkCmdEndQuery-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, or compute 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

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>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td>State</td>
</tr>
</tbody>
</table>

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,
    VkQueryPool queryPool,
    uint32_t firstQuery,
    uint32_t queryCount,
    size_t dataSize,
    void* pData,
    VkDeviceSize stride,
);```
device is the logical device that owns the query pool.
queryPool is the query pool managing the queries containing the desired results.
firstQuery is the initial query index.
queryCount is the number of queries to read.
dataSize is the size in bytes of the buffer pointed to by pData.
pData is a pointer to a user-allocated buffer where the results will be written.
stride is the stride in bytes between results for individual queries within pData.
flags is a bitmask of VkQueryResultFlagBits specifying how and when results are returned.

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. The size of the availability values is 64 bits if VK_QUERY_RESULT_64_BIT is set in flags. Otherwise, it is 32 bits.

Note
If VK_QUERY_RESULT_WITH_AVAILABILITY_BIT is set, the layout of data in the buffer is a (result, availability) pair for each query returned, and stride is the stride between each pair.

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, if returned, 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, 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 VK_QUERY_RESULT_WAIT_BIT is used in combination with VKQUERY_RESULT_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.

If VkPhysicalDeviceVulkanSC10Properties:deviceNoDynamicHostAllocations is VK_TRUE, vkGetQueryPoolResults must not return VK_ERROR_OUT_OF_HOST_MEMORY.

Valid Usage

- VUID-vkGetQueryPoolResults-firstQuery-09436
  firstQuery must be less than the number of queries in queryPool

- VUID-vkGetQueryPoolResults-firstQuery-09437
  The sum of firstQuery and queryCount must be less than or equal to the number of queries in queryPool

- VUID-vkGetQueryPoolResults-queryCount-09438
  If queryCount is greater than 1, stride must not be zero

- VUID-vkGetQueryPoolResults-queryType-09439
  If the queryType used to create queryPool was VK_QUERY_TYPE_TIMESTAMP, flags must not contain VK_QUERY_RESULT_PARTIAL_BIT

- VUID-vkGetQueryPoolResults-None-09401
  All queries used by the command must not be uninitialized

- VUID-vkGetQueryPoolResults-flags-02828
  If VK_QUERY_RESULT_64_BIT is not set in flags then pData and stride must be multiples of 4

- VUID-vkGetQueryPoolResults-flags-00815
  639
If `VK_QUERY_RESULT_64_BIT` is set in `flags` then `pData` and `stride` must be multiples of 8.

- **VUID-vkGetQueryPoolResults-stride-08993**
  If `VK_QUERY_RESULT_WITH_AVAILABILITY_BIT` is set, `stride` must be large enough to contain the unsigned integer representing availability in addition to the query result.

- **VUID-vkGetQueryPoolResults-dataSize-00817**
  `dataSize` must be large enough to contain the result of each query, as described [here](#).

## 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.

// 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:

// Provided by VK_VERSION_1_0
void vkCmdCopyQueryPoolResults(
    VkCommandBuffer commandBuffer,
    VkQueryPool queryPool,
    uint32_t firstQuery,
    uint32_t queryCount,
    VkBuffer dstBuffer,
    VkDeviceSize dstOffset,
    VkDeviceSize stride,
    VkQueryResultFlags flags);

- **commandBuffer** is the command buffer into which this command will be recorded.

- **queryPool** is the query pool managing the queries containing the desired results.

- **firstQuery** is the initial query index.

- **queryCount** is the number of queries. **firstQuery** and **queryCount** together define a range of queries.

- **dstBuffer** is a **VkBuffer** object that will receive the results of the copy command.

- **dstOffset** is an offset into **dstBuffer**.

- **stride** is the stride in bytes between results for individual queries within **dstBuffer**. The required size of the backing memory for **dstBuffer** is determined as described above for **vkGetQueryPoolResults**.

- **flags** is a bitmask of **VkQueryResultFlagBits** specifying how and when results are returned.

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`, `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-firstQuery-09436
  
  `firstQuery` must be less than the number of queries in `queryPool`

- VUID-vkCmdCopyQueryPoolResults-firstQuery-09437
  
  The sum of `firstQuery` and `queryCount` must be less than or equal to the number of queries in `queryPool`

- VUID-vkCmdCopyQueryPoolResults-queryCount-09438
  
  If `queryCount` is greater than 1, `stride` must not be zero

- VUID-vkCmdCopyQueryPoolResults-queryType-09439
  
  If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TIMESTAMP`, `flags` must not contain `VK_QUERY_RESULT_PARTIAL_BIT`
All queries used by the command **must** not be uninitialized when the command is executed.

**dstOffset** **must** be less than the size of **dstBuffer**

If **VK_QUERY_RESULT_64_BIT** is not set in **flags** then **dstOffset** and **stride** **must** be multiples of 4

If **VK_QUERY_RESULT_64_BIT** is set in **flags** then **dstOffset** and **stride** **must** be multiples of 8

**dstBuffer** **must** have enough storage, from **dstOffset**, to contain the result of each query, as described [here](#)

**dstBuffer** **must** have been created with **VK_BUFFER_USAGE_TRANSFER_DST_BIT** usage flag

If **dstBuffer** is non-sparse then it **must** be bound completely and contiguously to a single **VkDeviceMemory** object

All queries used by the command **must** not be active

All queries used by the command **must** have been made **available** by prior executed commands

---

**Valid Usage (Implicit)**

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

**queryPool** **must** be a valid **VkQueryPool** handle

**dstBuffer** **must** be a valid **VkBuffer** handle

**flags** **must** be a valid combination of **VkQueryResultFlagBits** values

**commandBuffer** **must** be in the **recording state**

The **VkCommandPool** that **commandBuffer** was allocated from **must** support graphics, or compute operations

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

### Host Synchronization

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

### Command Properties

<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.

Setting `VK_QUERY_CONTROL_PRECISE_BIT` does not guarantee that different implementations return the same number of samples in an occlusion query. Some implementations may kill fragments in the pre-rasterization shader stage, and
these killed fragments do not contribute to the final result of the query. It is possible that some implementations generate a zero result value for the query, while others generate a non-zero value.

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:

```
// 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,
    VK_QUERY_PIPELINE_STATISTIC_CLIPPING_PRIMITIVES_BIT = 0x00000040,
    VK_QUERY_PIPELINE_STATISTIC_FRAGMENT_SHADER_INVOCATIONS_BIT = 0x00000080,
    VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_CONTROL_SHADER_PATCHES_BIT = 0x00000100,
    VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_EVALUATION_SHADER_INVOCATIONS_BIT = 0x00000200,
    VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT = 0x00000400,
} VkQueryPipelineStatisticFlagBits;
```
• **VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_VERTICES_BIT** specifies that queries managed by the pool will count the number of vertices processed by the **input assembly** stage. Vertices corresponding to incomplete primitives **may** contribute to the count.

• **VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_PRIMITIVES_BIT** specifies that queries managed by the pool will count the number of primitives processed by the **input assembly** stage. If primitive restart is enabled, restarting the primitive topology has no effect on the count. Incomplete primitives **may** be counted.

• **VK_QUERY_PIPELINE_STATISTIC_VERTEX_SHADER_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of vertex shader invocations. This counter’s value is incremented each time a vertex shader is **invoked**.

• **VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of geometry shader invocations. This counter’s value is incremented each time a geometry shader is **invoked**. In the case of **instanced geometry shaders**, the geometry shader invocations count is incremented for each separate instanced invocation.

• **VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_PRIMITIVES_BIT** specifies that queries managed by the pool will count the number of primitives generated by geometry shader invocations. The counter’s value is incremented each time the geometry shader emits a primitive. Restarting primitive topology using the SPIR-V instructions `OpEndPrimitive` or `OpEndStreamPrimitive` has no effect on the geometry shader output primitives count.

• **VK_QUERY_PIPELINE_STATISTIC_CLIPPING_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of primitives processed by the **Primitive Clipping** stage of the pipeline. The counter’s value is incremented each time a primitive reaches the primitive clipping stage.

• **VK_QUERY_PIPELINE_STATISTIC_CLIPPING_PRIMITIVES_BIT** specifies that queries managed by the pool will count the number of primitives output by the **Primitive Clipping** stage of the pipeline. The actual number of primitives output by the primitive clipping stage for a particular input primitive is implementation-dependent but **must** satisfy the following conditions:
  ◦ If at least one vertex of the input primitive lies inside the clipping volume, the counter is incremented by one or more.
  ◦ Otherwise, the counter is incremented by zero or more.

• **VK_QUERY_PIPELINE_STATISTIC_FRAGMENT_SHADER_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of fragment shader invocations. The counter’s value is incremented each time the fragment shader is **invoked**.

• **VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_CONTROL_SHADER_PATCHES_BIT** specifies that queries managed by the pool will count the number of patches processed by the tessellation control shader. The counter’s value is incremented once for each patch for which a tessellation control shader is **invoked**.

• **VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_EVALUATION_SHADER_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of invocations of the tessellation evaluation
shader. The counter's value is incremented each time the tessellation evaluation shader is invoked.

- **VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of compute shader invocations. The counter's value is incremented every time the compute shader is invoked. Implementations may skip the execution of certain compute shader invocations or execute additional compute shader invocations for implementation-dependent reasons as long as the results of rendering otherwise remain unchanged.

These values are intended to measure relative statistics on one implementation. Various device architectures will count these values differently. Any or all counters may be affected by the issues described in Query Operation.

**Note**

For example, tile-based rendering devices may need to replay the scene multiple times, affecting some of the counts.

If a pipeline has `rasterizerDiscardEnable` enabled, implementations may discard primitives after the final 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_0
void vkCmdWriteTimestamp(
    VkCommandBuffer commandBuffer,
    VkPipelineStageFlagBits pipelineStage,
    VkQueryPool queryPool,
    uint32_t query);
```

- commandBuffer is the command buffer into which the command will be recorded.
- pipelineStage is 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.

**Note**

Implementations may write the timestamp at any stage that is logically later than stage.

Any timestamp write that happens-after another timestamp write in the same submission must not have a lower value unless its value overflows the maximum supported integer bit width of the query. If an overflow occurs, the timestamp value must wrap back to zero.

**Note**

Comparisons between timestamps should be done between timestamps where they are guaranteed to not decrease. For example, subtracting an older timestamp from a newer one to determine the execution time of a sequence of commands is only a reliable measurement if the two timestamp writes were performed in the same submission.

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-pipelineStage-06490
  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-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, or compute 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</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>
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 \( \texttt{vkCmdClearColorImage} \) or \( \texttt{vkCmdClearDepthStencilImage} \), 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);
```

- \texttt{commandBuffer} is the command buffer into which the command will be recorded.
- \texttt{image} is the image to be cleared.
- \texttt{imageLayout} specifies the current layout of the image subresource ranges to be cleared, and must be \texttt{VK_IMAGE_LAYOUT_GENERAL} or \texttt{VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL}.
- \texttt{pColor} is a pointer to a \texttt{VkClearColorValue} structure containing the values that the image subresource ranges will be cleared to (see Clear Values below).
- \texttt{rangeCount} is the number of image subresource range structures in \texttt{pRanges}.
- \texttt{pRanges} is a pointer to an array of \texttt{VkImageSubresourceRange} structures describing a range of mipmap levels, array layers, and aspects to be cleared, as described in Image Views.

Each specified range in \texttt{pRanges} is cleared to the value specified by \texttt{pColor}.

Valid Usage

- VUID-vkCmdClearColorImage-image-01993
  The format features of \texttt{image} must contain \texttt{VK_FORMAT_FEATURE_TRANSFER_DST_BIT}
- VUID-vkCmdClearColorImage-image-00002
  \texttt{image} must have been created with \texttt{VK_IMAGE_USAGE_TRANSFER_DST_BIT} usage flag
- VUID-vkCmdClearColorImage-image-01545
  \texttt{image} must not use any of the formats that require a sampler \( Y'C_bC_r \) conversion
- VUID-vkCmdClearColorImage-image-00003
  If \texttt{image} is non-sparse then it must be bound completely and contiguously to a single \texttt{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`.

- **VUID-vkCmdClearColorImage-imageLayout-01394**
  `imageLayout` must be `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`.

- **VUID-vkCmdClearColorImage-aspectMask-02498**
  The `VkImageSubresourceRange::aspectMask` members of the elements of the `pRanges` array must each only include `VK_IMAGE_ASPECT_COLOR_BIT`.

- **VUID-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.

- **VUID-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.

- **VUID-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.

- **VUID-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.

- **VUID-vkCmdClearColorImage-image-00007**
  `image` must not have a compressed or depth/stencil format.

- **VUID-vkCmdClearColorImage-pColor-04961**
  `pColor` must be a valid pointer to a `VkClearColorValue` union.

- **VUID-vkCmdClearColorImage-commandBuffer-01805**
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `image` must not be a protected image.

- **VUID-vkCmdClearColorImage-commandBuffer-01806**
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `must` not be an unprotected image.

---

**Valid Usage (Implicit)**

- **VUID-vkCmdClearColorImage-commandBuffer-parameter**
  `commandBuffer` must be a valid `VkCommandBuffer` handle.

- **VUID-vkCmdClearColorImage-image-parameter**
  `image` must be a valid `VkImage` handle.

- **VUID-vkCmdClearColorImage-imageLayout-parameter**
  `imageLayout` must be a valid `VkImageLayout` value.
- **VUID-vkCmdClearColorImage-pRanges-parameter**
  
  pRanges must be a valid pointer to an array of rangeCount valid VkImageSubresourceRange structures

- **VUID-vkCmdClearColorImage-commandBuffer-recording**
  
  commandBuffer must be in the recording state

- **VUID-vkCmdClearColorImage-commandBuffer-cmdpool**
  
  The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations

- **VUID-vkCmdClearColorImage-renderpass**
  
  This command must only be called outside of a render pass instance

- **VUID-vkCmdClearColorImage-rangeCount-arraylength**
  
  rangeCount must be greater than 0

- **VUID-vkCmdClearColorImage-commonparent**
  
  Both of commandBuffer, and image must have been created, allocated, or retrieved from the same VkDevice

---

**Host Synchronization**

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

---

**Command Properties**

<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, Compute</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To clear one or more subranges of a depth/stencil image, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdClearDepthStencilImage(
    VkCommandBuffer commandBuffer,
    VkImage image,
    VkImageLayout imageLayout,
    const VkClearDepthStencilValue* pDepthStencil,
    uint32_t rangeCount,
    const VkImageSubresourceRange* pRanges);
```

- commandBuffer is the command buffer into which the command will be recorded.
- `image` is the image to be cleared.
- `imageLayout` specifies the current layout of the image subresource ranges to be cleared, and must be `VK_IMAGE_LAYOUT_GENERAL` or `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`.
- `pDepthStencil` is a pointer to a `VkClearDepthStencilValue` structure 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 `VkImageStencilUsageCreateInfo::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.

image must have a depth/stencil format.

If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, image must not be a protected image.

If commandBuffer is a protected command buffer and protectedNoFault is not supported, image must not be an unprotected image.

Valid Usage (Implicit)

commandBuffer must be a valid VkCommandBuffer handle.

depthStencil must be a valid pointer to a valid VkClearDepthStencilValue structure.

pRanges must be a valid pointer to an array of rangeCount valid VkImageSubresourceRange structures.
commandBuffer must be in the recording state

The VkCommandPool that commandBuffer was allocated from must support graphics operations.

This command must only be called outside of a render pass instance.

rangeCount must be greater than 0

Both of commandBuffer, and image must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

Host access to commandBuffer must be externally synchronized.

Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized.

Command Properties

<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>

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.

If an attachment being cleared refers to an image view created with an `aspectMask` equal to one of `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT` or `VK_IMAGE_ASPECT_PLANE_2_BIT`, it is considered to be `VK_IMAGE_ASPECT_COLOR_BIT` for purposes of this command, and must be cleared with the `VK_IMAGE_ASPECT_COLOR_BIT` aspect as specified by image view creation.

---

**Valid Usage**

- VUID-vkCmdClearAttachments-aspectMask-07884
  If the `aspectMask` member of any element of `pAttachments` contains `VK_IMAGE_ASPECT_DEPTH_BIT`, the current subpass instance's depth-stencil attachment must be either `VK_ATTACHMENT_UNUSED` or the attachment format must contain a depth component.

- VUID-vkCmdClearAttachments-aspectMask-07885
  If the `aspectMask` member of any element of `pAttachments` contains `VK_IMAGE_ASPECT_STENCIL_BIT`, the current subpass instance's depth-stencil attachment must be either `VK_ATTACHMENT_UNUSED` or the attachment format must contain a stencil component.

- 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
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 VkClearColor structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkClearColor {
    VkRect2D     rect;
    uint32_t     baseArrayLayer;
    uint32_t     layerCount;
} VkClearColor;
```

- `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;
    VkClearValue     clearValue;
} VkClearAttachment;
```

- `aspectMask` is a mask selecting the color, depth and/or stencil aspects of the attachment to be
• 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 VK_IMAGE_ASPECT_DEPTH_BIT or VK_IMAGE_ASPECT_STENCIL_BIT

- **VUID-VkClearAttachment-aspectMask-00020**
  aspectMask must not include VK_IMAGE_ASPECT_METADATA_BIT

### Valid Usage (Implicit)

- **VUID-VkClearAttachment-aspectMask-parameter**
  aspectMask must be a valid combination of VkImageAspectFlagBits values

- **VUID-VkClearAttachment-aspectMask-required bitmask**
  aspectMask must not be 0

### 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 numeric formats with a numeric type that is floating-point. 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 has a numeric type that 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 has a numeric type
type that is unsigned integer (\texttt{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 \texttt{VkClearDepthStencilValue} structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkClearDepthStencilValue {
  float depth;
  uint32_t stencil;
} VkClearDepthStencilValue;
```

- \texttt{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.
- \texttt{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

- \texttt{VUID-VkClearDepthStencilValue-depth-00022}
  - \texttt{depth} must be between 0.0 and 1.0, inclusive

The \texttt{VkClearValue} union is defined as:

```c
// Provided by VK_VERSION_1_0
typedef union VkClearValue {
  VkClearColorValue color;
  VkClearDepthStencilValue depthStencil;
} VkClearValue;
```

- \texttt{color} specifies the color image clear values to use when clearing a color image or attachment.
- \texttt{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 \texttt{VkRenderPassBeginInfo} 
structure.
18.4. Filling Buffers

To clear buffer data, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdFillBuffer(
    VkCommandBuffer commandBuffer,
    VkBuffer dstBuffer,
    VkDeviceSize dstOffset,
    VkDeviceSize size,
    uint32_t data);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `dstBuffer` is the buffer to be filled.
- `dstOffset` is the byte offset into the buffer at which to start filling, and **must** be a multiple of 4.
- `size` is the number of bytes to fill, and **must** be either a multiple of 4, or `VK_WHOLE_SIZE` to fill the range from `offset` to the end of the buffer. If `VK_WHOLE_SIZE` is used and the remaining size of the buffer is not a multiple of 4, then the nearest smaller multiple is used.
- `data` is the 4-byte word written repeatedly to the buffer to fill `size` bytes of data. The data word is written to memory according to the host endianness.

`vkCmdFillBuffer` is treated as 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-apiVersion-07894
  `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics or compute operations
If \textit{dstBuffer} is non-sparse then it \textbf{must} be bound completely and contiguously to a single \textit{VkDeviceMemory} object.

If \textit{commandBuffer} is an unprotected command buffer and \textit{protectedNoFault} is not supported, \textit{dstBuffer} \textbf{must} not be a protected buffer.

If \textit{commandBuffer} is a protected command buffer and \textit{protectedNoFault} is not supported, \textit{dstBuffer} \textbf{must} not be an unprotected buffer.

### Valid Usage (Implicit)

- \textbf{VUID-vkCmdFillBuffer-commandBuffer-parameter} \\
  \textit{commandBuffer} \textbf{must} be a valid \textit{VkCommandBuffer} handle

- \textbf{VUID-vkCmdFillBuffer-dstBuffer-parameter} \\
  \textit{dstBuffer} \textbf{must} be a valid \textit{VkBuffer} handle

- \textbf{VUID-vkCmdFillBuffer-commandBuffer-recording} \\
  \textit{commandBuffer} \textbf{must} be in the \textit{recording} state

- \textbf{VUID-vkCmdFillBuffer-commandBuffer-cmdpool} \\
  The \textit{VkCommandPool} that \textit{commandBuffer} was allocated from \textbf{must} support transfer, graphics or compute operations

- \textbf{VUID-vkCmdFillBuffer-renderpass} \\
  This command \textbf{must} only be called outside of a render pass instance

- \textbf{VUID-vkCmdFillBuffer-commonparent} \\
  Both of \textit{commandBuffer}, and \textit{dstBuffer} \textbf{must} have been created, allocated, or retrieved from the same \textit{VkDevice}

### Host Synchronization

- Host access to \textit{commandBuffer} \textbf{must} be externally synchronized

- Host access to the \textit{VkCommandPool} that \textit{commandBuffer} was allocated from \textbf{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
dataSize must be less than or equal to 65536

- VUID-vkCmdUpdateBuffer-dataSize-00038
dataSize must be a multiple of 4

- VUID-vkCmdUpdateBuffer-commandBuffer-01813
If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, dstBuffer must not be a protected buffer

- VUID-vkCmdUpdateBuffer-commandBuffer-01814
If commandBuffer is a protected command buffer and protectedNoFault is not supported, dstBuffer must not be an unprotected buffer

Valid Usage (Implicit)

- VUID-vkCmdUpdateBuffer-commandBuffer-parameter
commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdUpdateBuffer-dstBuffer-parameter
dstBuffer must be a valid VkBuffer handle

- VUID-vkCmdUpdateBuffer-pData-parameter
pData must be a valid pointer to an array of dataSize bytes

- VUID-vkCmdUpdateBuffer-commandBuffer-recording
commandBuffer must be in the recording state

- VUID-vkCmdUpdateBuffer-commandBuffer-cmdpool
The VkCommandPool that commandBuffer was allocated from must support transfer, graphics, or compute operations

- VUID-vkCmdUpdateBuffer-renderpass
This command must only be called outside of a render pass instance

- VUID-vkCmdUpdateBuffer-dataSize-arraylength
dataSize must be greater than 0
Both of commandBuffer, and dstBuffer must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

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

Command Properties

<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>Transfer Graphics Compute</td>
<td>Action</td>
</tr>
</tbody>
</table>
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
19.2. Copying Data Between Images

To copy data between image objects, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdCopyImage(
    VkCommandBuffer commandBuffer,
    VkImage srcImage,
    VkImageLayout srcImageLayout,
    VkImage dstImage,
    VkImageLayout dstImageLayout,
    uint32_t regionCount,
    const VkImageCopy* pRegions);
```

- **commandBuffer** is the command buffer into which the command will be recorded.
- **srcImage** is the source image.
- **srcImageLayout** is the current layout of the source image subresource.
- **dstImage** is the destination image.
- **dstImageLayout** is the current layout of the destination image subresource.
- **regionCount** is the number of regions to copy.
- **pRegions** is a pointer to an array of `VkImageCopy` structures specifying the regions to copy.

Each 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. Other combinations of image types are disallowed.
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-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-vkCmdCopyImage-srcImageLayout-01917
  `srcImageLayout` must be `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL`, or `VK_IMAGE_LAYOUT_GENERAL`

- VUID-vkCmdCopyImage-dstImage-01996
  The format features of `dstImage` must contain `VK_FORMAT_FEATURE_TRANSFER_DST_BIT`

- VUID-vkCmdCopyImage-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-vkCmdCopyImage-dstImageLayout-01395
  `dstImageLayout` must be `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`, or `VK_IMAGE_LAYOUT_GENERAL`

- VUID-vkCmdCopyImage-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-vkCmdCopyImage-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-vkCmdCopyImage-srcImage-09247
  If the `VkFormat` of each of `srcImage` and `dstImage` is a compressed image format, the formats must have the same texel block extent

- VUID-vkCmdCopyImage-srcImage-00136
  The sample count of `srcImage` and `dstImage` must match

- VUID-vkCmdCopyImage-srcOffset-01783
The \texttt{srcOffset} and \texttt{extent} members of each element of \texttt{pRegions} \textbf{must} respect the image transfer granularity requirements of \texttt{commandBuffer}'s command pool's queue family, as described in \texttt{VkQueueFamilyProperties}.

- VUID-vkCmdCopyImage-dstOffset-01784
  The \texttt{dstOffset} and \texttt{extent} members of each element of \texttt{pRegions} \textbf{must} respect the image transfer granularity requirements of \texttt{commandBuffer}'s command pool's queue family, as described in \texttt{VkQueueFamilyProperties}.

- VUID-vkCmdCopyImage-srcImage-01551
  If neither \texttt{srcImage} nor \texttt{dstImage} has a \texttt{multi-planar image format} then for each element of \texttt{pRegions}, \texttt{srcSubresource.aspectMask} and \texttt{dstSubresource.aspectMask} \textbf{must} match.

- VUID-vkCmdCopyImage-srcImage-08713
  If \texttt{srcImage} has a \texttt{multi-planar image format}, then for each element of \texttt{pRegions}, \texttt{srcSubresource.aspectMask} \textbf{must} be a single valid \texttt{multi-planar aspect mask} bit.

- VUID-vkCmdCopyImage-dstImage-08714
  If \texttt{dstImage} has a \texttt{multi-planar image format}, then for each element of \texttt{pRegions}, \texttt{dstSubresource.aspectMask} \textbf{must} be a single valid \texttt{multi-planar aspect mask} bit.

- VUID-vkCmdCopyImage-srcImage-01556
  If \texttt{srcImage} has a \texttt{multi-planar image format} and the \texttt{dstImage} does not have a \texttt{multi-planar image format}, then for each element of \texttt{pRegions}, \texttt{dstSubresource.aspectMask} \textbf{must} be \texttt{VK_IMAGE_ASPECT_COLOR_BIT}.

- VUID-vkCmdCopyImage-dstImage-01557
  If \texttt{dstImage} has a \texttt{multi-planar image format} and the \texttt{srcImage} does not have a \texttt{multi-planar image format}, then for each element of \texttt{pRegions}, \texttt{srcSubresource.aspectMask} \textbf{must} be \texttt{VK_IMAGE_ASPECT_COLOR_BIT}.

- VUID-vkCmdCopyImage-srcImage-04443
  If \texttt{srcImage} is of type \texttt{VK_IMAGE_TYPE_3D}, then for each element of \texttt{pRegions}, \texttt{srcSubresource.baseArrayLayer} \textbf{must} be 0 and \texttt{srcSubresource.layerCount} \textbf{must} be 1.

- VUID-vkCmdCopyImage-dstImage-04444
  If \texttt{dstImage} is of type \texttt{VK_IMAGE_TYPE_3D}, then for each element of \texttt{pRegions}, \texttt{dstSubresource.baseArrayLayer} \textbf{must} be 0 and \texttt{dstSubresource.layerCount} \textbf{must} be 1.

- VUID-vkCmdCopyImage-aspectMask-00142
  For each element of \texttt{pRegions}, \texttt{srcSubresource.aspectMask} \textbf{must} specify aspects present in \texttt{srcImage}.

- VUID-vkCmdCopyImage-aspectMask-00143
  For each element of \texttt{pRegions}, \texttt{dstSubresource.aspectMask} \textbf{must} specify aspects present in \texttt{dstImage}.

- VUID-vkCmdCopyImage-srcOffset-00144
  For each element of \texttt{pRegions}, \texttt{srcOffset.x} and \((\text{extent.width} + \text{srcOffset.x})\) \textbf{must} both be greater than or equal to \texttt{0} and less than or equal to the width of the specified \texttt{srcSubresource} of \texttt{srcImage}.

- VUID-vkCmdCopyImage-srcOffset-00145
  For each element of \texttt{pRegions}, \texttt{srcOffset.y} and \((\text{extent.height} + \text{srcOffset.y})\) \textbf{must} both be greater than or equal to \texttt{0} and less than or equal to the height of the specified.
srcSubresource of srcImage

- VUID-vkCmdCopyImage-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-vkCmdCopyImage-srcOffset-00147
  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

- VUID-vkCmdCopyImage-srcImage-01785
  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

- VUID-vkCmdCopyImage-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-vkCmdCopyImage-srcImage-01787
  If srcImage is of type VK_IMAGE_TYPE_2D, then for each element of pRegions, dstOffset.z must be 0

- VUID-vkCmdCopyImage-dstImage-01788
  If dstImage is of type VK_IMAGE_TYPE_2D, then for each element of pRegions, dstOffset.z must be 0

- VUID-vkCmdCopyImage-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-vkCmdCopyImage-srcImage-08793
  If srcImage and dstImage have the same VkImageType, for each element of pRegions, the layerCount members of srcSubresource or dstSubresource must match

- VUID-vkCmdCopyImage-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-vkCmdCopyImage-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-vkCmdCopyImage-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-vkCmdCopyImage-dstOffset-00150
  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-vkCmdCopyImage-dstOffset-00151
  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.

If dstImage is of type VK_IMAGE_TYPE_3D, then 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.

For each element of pRegions, srcOffset.x must be a multiple of the texel block extent width of the VkFormat of srcImage.

For each element of pRegions, srcOffset.y must be a multiple of the texel block extent height of the VkFormat of srcImage.

For each element of pRegions, srcOffset.z must be a multiple of the texel block extent depth of the VkFormat of srcImage.

For each element of pRegions, dstOffset.x must be a multiple of the texel block extent width of the VkFormat of dstImage.

For each element of pRegions, dstOffset.y must be a multiple of the texel block extent height of the VkFormat of dstImage.

For each element of pRegions, dstOffset.z must be a multiple of the texel block extent depth of the VkFormat of dstImage.

For each element of pRegions, if the sum of srcOffset.x and extent.width does not equal the width of 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 srcOffset.y and extent.height does not equal the height of 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 srcOffset.z and extent.depth does not equal the depth of 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, if the sum of dstOffset.x and extent.width does not equal the width of the subresource specified by dstSubresource, extent.width must be a multiple of the texel block extent width of the VkFormat of dstImage.

For each element of pRegions, if the sum of dstOffset.y and extent.height does not equal
the height of 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 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-07966
  If srcImage is non-sparse then the image or the specified disjoint plane must be bound completely and contiguously to a single VkDeviceMemory object

• VUID-vkCmdCopyImage-srcSubresource-07967
  The srcSubresource.mipLevel member of each element of pRegions must be less than the mipLevels specified in VkImageCreateInfo when srcImage was created

• VUID-vkCmdCopyImage-srcSubresource-07968
  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-vkCmdCopyImage-dstImage-07966
  If dstImage is non-sparse then the image or the specified disjoint plane must be bound completely and contiguously to a single VkDeviceMemory object

• VUID-vkCmdCopyImage-dstSubresource-07967
  The dstSubresource.mipLevel member of each element of pRegions must be less than the mipLevels specified in VkImageCreateInfo when dstImage was created

• VUID-vkCmdCopyImage-dstSubresource-07968
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.

**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
  srcImageLayout must be a valid VkImageLayout value
- VUID-vkCmdCopyImage-dstImage-parameter
  dstImage must be a valid VkImage handle
- VUID-vkCmdCopyImage-dstImageLayout-parameter
  dstImageLayout must be a valid VkImageLayout value
- VUID-vkCmdCopyImage-pRegions-parameter
  pRegions must be a valid pointer to an array of regionCount valid VkImageCopy structures
- VUID-vkCmdCopyImage-commandBuffer-recording
  commandBuffer must be in the recording state
- VUID-vkCmdCopyImage-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support transfer, graphics, or compute operations
- VUID-vkCmdCopyImage-renderpass
  This command must only be called outside of a render pass instance
- VUID-vkCmdCopyImage-regionCount-arraylength
  regionCount must be greater than 0
- VUID-vkCmdCopyImage-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
The `VkImageCopy` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkImageCopy {
    VkImageSubresourceLayers srcSubresource;
    VkOffset3D srcOffset;
    VkImageSubresourceLayers dstSubresource;
    VkOffset3D dstOffset;
    VkExtent3D extent;
} VkImageCopy;
```

- `srcSubresource` and `dstSubresource` are `VkImageSubresourceLayers` structures specifying the image subresources of the images used for the source and destination image data, respectively.
- `srcOffset` and `dstOffset` select the initial `x`, `y`, and `z` offsets in texels of the sub-regions of the source and destination image data.
- `extent` is the size in texels of the image to copy in `width`, `height` and `depth`.

### 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-09243**
  `layerCount` must not be `VK_REMAINING_ARRAY_LAYERS`

- **VUID-VkImageSubresourceLayers-layerCount-01700**
  If `layerCount` is not `VK_REMAINING_ARRAY_LAYERS`, it 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`

### 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:

- \(x\) is in the range \([\text{imageOffset.x} / \text{blockWidth}, \lceil (\text{imageOffset.x} + \text{imageExtent.width}) / \text{blockWidth}\rceil)\),

- \(y\) is in the range \([\text{imageOffset.y} / \text{blockHeight}, \lceil (\text{imageOffset.y} + \text{imageExtent.height}) / \text{blockHeight}\rceil)\),

- \(z\) is in the range \([\text{imageOffset.z} / \text{blockDepth}, \lceil (\text{imageOffset.z} + \text{imageExtent.depth}) / \text{blockDepth}\rceil)\),

- \(\text{layer}\) is in the range \([\text{imageSubresource.baseArrayLayer}, \text{imageSubresource.baseArrayLayer} + \text{imageSubresource.layerCount})\),

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:

\[
\begin{align*}
[x \times \text{blockWidth}, \max \left( (x \times \text{blockWidth}) + \text{blockWidth}, \text{imageWidth} \right) ) \\
[y \times \text{blockHeight}, \max \left( (y \times \text{blockHeight}) + \text{blockHeight}, \text{imageHeight} \right) ) \\
[z \times \text{blockDepth}, \max \left( (z \times \text{blockDepth}) + \text{blockDepth}, \text{imageDepth} \right) )
\end{align*}
\]

where \(\text{imageWidth}\), \(\text{imageHeight}\), and \(\text{imageDepth}\) 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

\[
\text{rowExtent} = \max (\text{bufferRowLength}, \lceil \text{imageExtent.width} / \text{blockWidth}\rceil \times \text{blockSize})
\]

\[
\text{sliceExtent} = \max (\text{bufferImageHeight}, \text{imageExtent.height} \times \text{rowExtent})
\]
layerExtent = imageExtent.depth × 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>
<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 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-dstImage-07966
  If dstImage is non-sparse then the image or the specified disjoint plane must be bound completely and contiguously to a single VkDeviceMemory object

- VUID-vkCmdCopyBufferToImage-imageSubresource-07967
  The imageSubresource.mipLevel member of each element of pRegions must be less than the mipLevels specified in VkImageCreateInfo when dstImage was created

- VUID-vkCmdCopyBufferToImage-imageSubresource-07968
  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

- VUID-vkCmdCopyBufferToImage-imageSubresource-07970
  The image region specified by each element of pRegions must be contained within the specified imageSubresource of dstImage

- VUID-vkCmdCopyBufferToImage-imageSubresource-07971
  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

- VUID-vkCmdCopyBufferToImage-imageSubresource-07972
  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

- VUID-vkCmdCopyBufferToImage-dstImage-07973
  dstImage must have a sample count equal to VK_SAMPLE_COUNT_1_BIT

- 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-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-vkCmdCopyBufferToImage-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-vkCmdCopyBufferToImage-srcBuffer-00174**
  *srcBuffer* must have been created with `VK_BUFFER_USAGE_TRANSFER_SRC_BIT` usage flag

- **VUID-vkCmdCopyBufferToImage-dstImage-01997**
  The format features of *dstImage* must contain `VK_FORMAT_FEATURE_TRANSFER_DST_BIT`

- **VUID-vkCmdCopyBufferToImage-dstImage-00177**
  *dstImage* must have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag

- **VUID-vkCmdCopyBufferToImage-dstImageLayout-00180**
  *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-vkCmdCopyBufferToImage-dstImageLayout-01396**
  *dstImageLayout* must be `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`, or `VK_IMAGE_LAYOUT_GENERAL`

- **VUID-vkCmdCopyBufferToImage-pRegions-07931**
  For each element of *pRegions* whose *imageSubresource* contains a depth aspect, the data in *srcBuffer* must be in the range [0,1]

- **VUID-vkCmdCopyBufferToImage-dstImage-07979**
  If *dstImage* 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-vkCmdCopyBufferToImage-imageOffset-09104**
For each element of \texttt{pRegions}, \texttt{imageOffset.z} and \((\texttt{imageExtent.depth} + \texttt{imageOffset.z})\) must both be greater than or equal to 0 and less than or equal to the depth of the specified \texttt{imageSubresource} of \texttt{dstImage}.

- **VUID-vkCmdCopyBufferToImage-dstImage-07980**
  
  If \texttt{dstImage} is of type \texttt{VK_IMAGE_TYPE_1D} or \texttt{VK_IMAGE_TYPE_2D}, then for each element of \texttt{pRegions}, \texttt{imageOffset.z} must be 0 and \texttt{imageExtent.depth} must be 1.

- **VUID-vkCmdCopyBufferToImage-dstImage-07274**
  
  For each element of \texttt{pRegions}, \texttt{imageOffset.x} must be a multiple of the \texttt{texel block extent width} of the \texttt{VkFormat} of \texttt{dstImage}.

- **VUID-vkCmdCopyBufferToImage-dstImage-07275**
  
  For each element of \texttt{pRegions}, \texttt{imageOffset.y} must be a multiple of the \texttt{texel block extent height} of the \texttt{VkFormat} of \texttt{dstImage}.

- **VUID-vkCmdCopyBufferToImage-dstImage-07276**
  
  For each element of \texttt{pRegions}, \texttt{imageOffset.z} must be a multiple of the \texttt{texel block extent depth} of the \texttt{VkFormat} of \texttt{dstImage}.

- **VUID-vkCmdCopyBufferToImage-dstImage-07277**
  
  For each element of \texttt{pRegions}, if the sum of \texttt{imageOffset.x} and \texttt{extent.width} does not equal the width of the subresource specified by \texttt{srcSubresource}, \texttt{extent.width} must be a multiple of the \texttt{texel block extent width} of the \texttt{VkFormat} of \texttt{dstImage}.

- **VUID-vkCmdCopyBufferToImage-dstImage-07278**
  
  For each element of \texttt{pRegions}, if the sum of \texttt{imageOffset.y} and \texttt{extent.height} does not equal the height of the subresource specified by \texttt{srcSubresource}, \texttt{extent.height} must be a multiple of the \texttt{texel block extent height} of the \texttt{VkFormat} of \texttt{dstImage}.

- **VUID-vkCmdCopyBufferToImage-dstImage-07279**
  
  For each element of \texttt{pRegions}, if the sum of \texttt{imageOffset.z} and \texttt{extent.depth} does not equal the depth of the subresource specified by \texttt{srcSubresource}, \texttt{extent.depth} must be a multiple of the \texttt{texel block extent depth} of the \texttt{VkFormat} of \texttt{dstImage}.

- **VUID-vkCmdCopyBufferToImage-dstImage-00207**
  
  For each element of \texttt{pRegions}, \texttt{imageSubresource.aspectMask} must specify aspects present in \texttt{dstImage}.

- **VUID-vkCmdCopyBufferToImage-dstImage-07981**
  
  If \texttt{dstImage} has a \texttt{multi-planar image format}, then for each element of \texttt{pRegions}, \texttt{imageSubresource.aspectMask} must be a single valid \texttt{multi-planar aspect mask} bit.

- **VUID-vkCmdCopyBufferToImage-dstImage-07983**
  
  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-vkCmdCopyBufferToImage-bufferRowLength-09106**
  
  For each element of \texttt{pRegions}, \texttt{bufferRowLength} must be a multiple of the \texttt{texel block extent width} of the \texttt{VkFormat} of \texttt{dstImage}.

- **VUID-vkCmdCopyBufferToImage-bufferImageHeight-09107**
  
  For each element of \texttt{pRegions}, \texttt{bufferImageHeight} must be a multiple of the \texttt{texel block extent height} of the \texttt{VkFormat} of \texttt{dstImage}.
For each element of `pRegions`, `bufferRowLength` divided by the texel block extent width and then multiplied by the texel block size of `dstImage` must be less than or equal to 2^{31}-1.

If `dstImage` does not have either a depth/stencil format or a multi-planar format, then for each element of `pRegions`, `bufferOffset` must be a multiple of the texel block size.

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.

If `dstImage` has a depth/stencil format, the `bufferOffset` member of any element of `pRegions` must be a multiple of 4.

Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle.
- `srcBuffer` must be a valid `VkBuffer` handle.
- `dstImage` must be a valid `VkImage` handle.
- `dstImageLayout` must be a valid `VkImageLayout` value.
- `pRegions` must be a valid pointer to an array of `regionCount` valid `VkBufferImageCopy` structures.
- `commandBuffer` must be in the recording state.
- The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations.
- This command must only be called outside of a render pass instance.
- `regionCount` must be greater than 0.
- Each of `commandBuffer`, `dstImage`, and `srcBuffer` must have been created, allocated, or retrieved from the same `VkDevice`. 
Host Synchronization

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

Command Properties

<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>

To copy data from an image object to a buffer object, call:

```c
// Provided by VK_VERSION_1_0
def 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-srcImage-07966
  If `srcImage` is non-sparse then the image or the specified `disjoint` plane must be bound completely and contiguously to a single `VkDeviceMemory` object.

- VUID-vkCmdCopyImageToBuffer-imageSubresource-07967
  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-07968
  `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-imageSubresource-07970
  The image region specified by each element of `pRegions` must be contained within the specified `imageSubresource` of `srcImage`.

- VUID-vkCmdCopyImageToBuffer-imageSubresource-07971
  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-imageSubresource-07972
  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`.

- VUID-vkCmdCopyImageToBuffer-srcImage-07973
  `srcImage` must have a sample count equal to `VK_SAMPLE_COUNT_1_BIT`.

- 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`.
dstBuffer must be large enough to contain all buffer locations that are accessed according to Buffer and Image Addressing, for each element of pRegions.

The union of all source regions, and the union of all destination regions, specified by the elements of pRegions, must not overlap in memory.

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.

dstBuffer must have been created with VK_BUFFER_USAGE_TRANSFER_DST_BIT usage flag.

If dstBuffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object.

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.

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.

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.

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, 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 subresource specified by srcSubresource, extent.width must be a multiple...
of the texel block extent width of the VkFormat of srcImage

- VUID-vkCmdCopyImageToBuffer-srcImage-00208
  For each element of pRegions, if the sum of imageOffset.y and extent.height does not equal the height of the subresource specified by srcSubresource, extent.height must be a multiple of the texel block extent height of the VkFormat of srcImage

- VUID-vkCmdCopyImageToBuffer-srcImage-00209
  For each element of pRegions, if the sum of imageOffset.z and extent.depth does not equal the depth of the subresource specified by srcSubresource, extent.depth must be a multiple of the texel block extent depth of the VkFormat of srcImage

- VUID-vkCmdCopyImageToBuffer-imageSubresource-09105
  For each element of pRegions, imageSubresource.aspectMask must specify aspects present in srcImage

- VUID-vkCmdCopyImageToBuffer-srcImage-07981
  If srcImage has a multi-planar image format, then for each element of pRegions, imageSubresource.aspectMask must be a single valid multi-planar aspect mask bit

- VUID-vkCmdCopyImageToBuffer-srcImage-07983
  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-bufferRowLength-09106
  For each element of pRegions, bufferRowLength must be a multiple of the texel block extent width of the VkFormat of srcImage

- VUID-vkCmdCopyImageToBuffer-bufferImageHeight-09107
  For each element of pRegions, bufferImageHeight must be a multiple of the texel block extent height of the VkFormat of srcImage

- VUID-vkCmdCopyImageToBuffer-bufferRowLength-09108
  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-07975
  If srcImage does not have either a depth/stencil format or a multi-planar format, then for each element of pRegions, bufferOffset must be a multiple of the texel block size

- VUID-vkCmdCopyImageToBuffer-srcImage-07976
  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-07978
  If srcImage has a depth/stencil format, the bufferOffset member of any element of pRegions must be a multiple of 4

Valid Usage (Implicit)

- VUID-vkCmdCopyImageToBuffer-commandBuffer-parameter
commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdCopyImageToBuffer-srclang-parameter
  srcImage must be a valid VkImage handle

- VUID-vkCmdCopyImageToBuffer-srcImageLayout-parameter
  srcImageLayout must be a valid VkImageLayout value

- VUID-vkCmdCopyImageToBuffer-dstlang-parameter
  dstBuffer must be a valid VkBuffer handle

- VUID-vkCmdCopyImageToBuffer-param-api-change
  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

### 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>

For both vkCmdCopyBufferToImage and vkCmdCopyImageToBuffer, each element of pRegions is a structure defined as:
typedef struct VkBufferImageCopy {
    VkDeviceSize bufferOffset;
    uint32_t bufferRowLength;
    uint32_t bufferImageHeight;
    VkImageSubresourceLayers imageSubresource;
    VkOffset3D imageOffset;
    VkExtent3D imageExtent;
} VkBufferImageCopy;

- **bufferOffset** is the offset in bytes from the start of the buffer object where the image data is copied from or to.
- **bufferRowLength** and **bufferImageHeight** specify in texels a subregion of a larger two- or three-dimensional image in buffer memory, and control the addressing calculations. If either of these values is zero, that aspect of the buffer memory is considered to be tightly packed according to the **imageExtent**.
- **imageSubresource** is a **VkImageSubresourceLayers** used to specify the specific image subresources of the image used for the source or destination image data.
- **imageOffset** selects the initial x, y, z offsets in texels of the sub-region of the source or destination image data.
- **imageExtent** is the size in texels of the image to copy in width, height and depth.

**Valid Usage**

- VUID-VkBufferImageCopy-bufferRowLength-09101
  bufferRowLength must be 0, or greater than or equal to the width member of imageExtent
- VUID-VkBufferImageCopy-bufferImageHeight-09102
  bufferImageHeight must be 0, or greater than or equal to the height member of imageExtent
- VUID-VkBufferImageCopy-aspectMask-09103
  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
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,  // 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 VkImageBlit* pRegions,     // Provided by VK_VERSION_1_0
    VkFilter filter);                // 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 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:

  \[
  \begin{align*}
  u_{base} &= i + \frac{1}{2} \\
  v_{base} &= j + \frac{1}{2} \\
  w_{base} &= k + \frac{1}{2}
  \end{align*}
  \]

  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:

\[ \text{scale}_u = \frac{(x_{\text{src1}} - x_{\text{src0}})}{(x_{\text{dst1}} - x_{\text{dst0}})} \]

\[ \text{scale}_v = \frac{(y_{\text{src1}} - y_{\text{src0}})}{(y_{\text{dst1}} - y_{\text{dst0}})} \]

\[ \text{scale}_w = \frac{(z_{\text{src1}} - z_{\text{src0}})}{(z_{\text{dst1}} - z_{\text{dst0}})} \]

\[ u_{\text{scaled}} = u_{\text{offset}} \times \text{scale}_u \]

\[ v_{\text{scaled}} = v_{\text{offset}} \times \text{scale}_v \]

\[ w_{\text{scaled}} = w_{\text{offset}} \times \text{scale}_w \]

- Finally the source offset is added to the scaled coordinates, to determine the final unnormalized coordinates used to sample from \textit{srcImage}:

\[ u = u_{\text{scaled}} + x_{\text{src0}} \]

\[ v = v_{\text{scaled}} + y_{\text{src0}} \]

\[ w = w_{\text{scaled}} + z_{\text{src0}} \]

\[ q = \text{mipLevel} \]
These coordinates are used to sample from the source image, as described in the Image Operations chapter, with the filter mode equal to that of \( \text{filter} \), a mipmap mode of \text{VK_SAMPLER_MIPMAP_MODE_NEAREST} and an address mode of \text{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 \text{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 \text{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.
If \texttt{commandBuffer} is an unprotected command buffer and \texttt{protectedNoFault} is not supported, \texttt{dstImage} must not be a protected image.

If \texttt{commandBuffer} is a protected command buffer and \texttt{protectedNoFault} is not supported, \texttt{dstImage} must not be an unprotected image.

The source region specified by each element of \texttt{pRegions} must be a region that is contained within \texttt{srcImage}.

The destination region specified by each element of \texttt{pRegions} must be a region that is contained within \texttt{dstImage}.

The union of all destination regions, specified by the elements of \texttt{pRegions}, must not overlap in memory with any texel that may be sampled during the blit operation.

The format features of \texttt{srcImage} must contain \texttt{VK_FORMAT_FEATURE_BLIT_SRC_BIT}.

\texttt{srcImage} must not use a format that requires a sampler \(Y'CbCr\) conversion.

\texttt{srcImage} must have been created with \texttt{VK_IMAGE_USAGE_TRANSFER_SRC_BIT} usage flag.

If \texttt{srcImage} is non-sparse then it must be bound completely and contiguously to a single \texttt{VkDeviceMemory} object.

\texttt{srcImageLayout} must specify the layout of the image subresources of \texttt{srcImage} specified in \texttt{pRegions} at the time this command is executed on a \texttt{VkDevice}.

\texttt{srcImageLayout} must be \texttt{VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL} or \texttt{VK_IMAGE_LAYOUT_GENERAL}.

The format features of \texttt{dstImage} must contain \texttt{VK_FORMAT_FEATURE_BLIT_DST_BIT}.

\texttt{dstImage} must not use a format that requires a sampler \(Y'CbCr\) conversion.

\texttt{dstImage} must have been created with \texttt{VK_IMAGE_USAGE_TRANSFER_DST_BIT} usage flag.

If \texttt{dstImage} is non-sparse then it must be bound completely and contiguously to a single \texttt{VkDeviceMemory} object.

\texttt{dstImageLayout} must specify the layout of the image subresources of \texttt{dstImage} specified in \texttt{pRegions} at the time this command is executed on a \texttt{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-dstImage-00234
dstImage must have been created with a samples value of VK_SAMPLE_COUNT_1_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
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
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
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-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:
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-08800
  The layerCount members of srcSubresource or dstSubresource must match

### Valid Usage (Implicit)

- VUID-VkImageBlit-srcSubresource-parameter
  srcSubresource must be a valid VkImageSubresourceLayers structure
- VUID-VkImageBlit-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-01400**
  **srcImageLayout** must be **VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL** or **VK_IMAGE_LAYOUT_GENERAL**

- **VUID-vkCmdResolveImage-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-vkCmdResolveImage-dstImageLayout-01401**
  **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**
  **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**
  **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 **dstImage** is of type **VK_IMAGE_TYPE_3D**, then for each element of **pRegions**, **srcSubresource.layerCount** must be 1

- **VUID-vkCmdResolveImage-srcImage-04447**
  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-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-srcImage-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-dstImage-00275**
  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>

702
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-08803**
  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
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 a VkStructureType value identifying 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-06252
  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, or VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY, primitiveRestartEnable must be VK_FALSE
If `topology` is `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)

- `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO`.
- `pNext` must be `NULL`.
- `flags` must be `0`.
- `topology` must be a valid `VkPrimitiveTopology` value.

```c
// Provided by VK_VERSION_1_0
typedef VkPipelineInputAssemblyStateCreateFlags;
```

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

## 20.1. Primitive Topologies

*Primitive topology* determines how consecutive vertices are organized into primitives, and determines the type of primitive that is used at the beginning of the graphics pipeline. The effective topology for later stages of the pipeline is altered by tessellation or geometry shading (if either is in use) and depends on the execution modes of those shaders.

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,
    ... // other topologies
};
```
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>Vertex</td>
<td>A point in 3-dimensional space. Positions chosen within the diagrams are arbitrary and for illustration only.</td>
</tr>
<tr>
<td>5</td>
<td>Vertex Number</td>
</tr>
<tr>
<td>Provoking Vertex</td>
<td>Provoking vertex within the main primitive. The tail is angled towards the relevant primitive. Used in flat shading.</td>
</tr>
<tr>
<td>Primitive Edge</td>
<td>An edge connecting the points of a main primitive.</td>
</tr>
<tr>
<td>Adjacency Edge</td>
<td>Points connected by these lines do not contribute to a main primitive, and are only accessible in a geometry shader.</td>
</tr>
</tbody>
</table>
Winding Order

The relative order in which vertices are defined within a primitive, used in the facing determination. This ordering has no specific start or end point.

The diagrams are supported with mathematical definitions where the vertices (v) and primitives (p) are numbered starting from 0; v₀ is the first vertex in the provided data and p₀ is the first primitive in the set of primitives defined by the vertices and topology.

20.1.1. Point Lists

When the topology is VK_PRIMITIVE_TOPOLOGY_POINT_LIST, each consecutive vertex defines a single point primitive, according to the equation:

\[ p_i = \{v_i\} \]

As there is only one vertex, that vertex is the provoking vertex. The number of primitives generated is equal to vertexCount.

20.1.2. Line Lists

When the 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 \(\lceil\text{vertexCount}/2\rceil\).

The provoking vertex for \(p_i\) is \(v_{2i}\).
20.1.3. 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 \( v_i \).

20.1.4. 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 \( v_{3i} \).

20.1.5. 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 \( v_i \).
Note
The ordering of the vertices in each successive triangle is reversed, so that the winding order is consistent throughout the strip.

20.1.6. Triangle Fans

When the 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 \( v_{i+1} \).

20.1.7. Line Lists With Adjacency

When the primitive topology is `VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY`, each consecutive set of four vertices defines a single line primitive with adjacency, according to the equation:

\[ p_i = \{v_{4i}, v_{4i+1}, v_{4i+2}, v_{4i+3}\} \]

A line primitive is described by the second and third vertices of the total primitive, with the remaining two vertices only accessible in a geometry shader.

The number of primitives generated is equal to \( \lfloor \text{vertexCount}/4 \rfloor \).

The provoking vertex for \( p_i \) is \( v_{4i-1} \).
20.1.8. Line Strips With Adjacency

When the primitive topology is `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY`, one line primitive with adjacency is defined by each vertex and the following vertex, according to the equation:

\[ p_i = \{v_i, v_{i+1}, v_{i+2}, v_{i+3}\} \]

A line primitive is described by the second and third vertices of the total primitive, with the remaining two vertices only accessible in a geometry shader.

The number of primitives generated is equal to \(\text{max}(0, \text{vertexCount} - 3)\).

The provoking vertex for \(p_i\) is \(v_{i+1}\).

20.1.9. 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 \(v_{6i}\).
20.1.10. 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 \( \lceil \max(0, \text{vertexCount} - 4)/2 \rceil \).

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:

\[
p_i = \{v_{2i}, v_{2i+1}, v_{2i+2}, v_{2i+6}, v_{2i+4}, v_{2i+3}\} \text{ when } i=0
\]

\[
p_i = \{v_{2i}, v_{2i+3}, v_{2i+4}, v_{2i+6}, v_{2i+2}, v_{2i+3}\} \text{ when } i>0, i<n-1, \text{ and } i\%2=1
\]

\[
p_i = \{v_{2i}, v_{2i+2}, v_{2i+3}, v_{2i+6}, v_{2i+4}, v_{2i+3}\} \text{ when } i>0, i<n-1, \text{ and } i\%2=0
\]

\[
p_i = \{v_{2i}, v_{2i+3}, v_{2i+4}, v_{2i+5}, v_{2i+2}, v_{2i+3}\} \text{ when } i=n-1 \text{ and } i\%2=1
\]

\[
p_i = \{v_{2i}, v_{2i+3}, v_{2i+4}, v_{2i+5}, v_{2i+2}, v_{2i+3}\} \text{ when } i=n-1 \text{ and } i\%2=0
\]

A triangle primitive is described by the first, third, and fifth vertices of the total primitive in all cases, with the remaining three vertices only accessible in a geometry shader.

**Note**

The ordering of the vertices in each successive triangle is altered so that the winding order is consistent throughout the strip.

The provoking vertex for \( p_i \) is always \( v_{2i} \).
20.1.11. Patch Lists

When the 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:
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-08782
  `offset` must be less than the size of `buffer`

- VUID-vkCmdBindIndexBuffer-offset-08783
  The sum of `offset` and the base address of the range of `VkDeviceMemory` object that is backing `buffer`, must be a multiple of the size of the type indicated by `indexType`

- VUID-vkCmdBindIndexBuffer-buffer-08784
  `buffer` must have been created with the `VK_BUFFER_USAGE_INDEX_BUFFER_BIT` flag

- VUID-vkCmdBindIndexBuffer-buffer-08785
  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</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 `vkCmdBindIndexBuffer::indexType`, specifying the size of indices, are:

```cpp
// 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:

```cpp
// 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-aspectMask-06478**
  If a `VkImageView` is sampled with depth comparison, the image view must have been created with an `aspectMask` that contains `VK_IMAGE_ASPECT_DEPTH_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-None-07888**
  For each set `n` that is statically used by a bound shader, a descriptor set must have been bound to `n` at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set `n`, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility
For each push constant that is statically used by a bound shader, 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 a bound shader.

A valid pipeline must be bound to the pipeline bind point used by this command.

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 any stage of the VkPipeline object bound to the pipeline bind point used by this command accesses a uniform buffer, and the robustBufferAccess feature is not enabled, that stage 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 any stage of the VkPipeline object bound to the pipeline bind point used by this command accesses a storage buffer, and the robustBufferAccess feature is not enabled, that stage 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 bound shaders must not be a protected resource.

If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, any resource accessed by bound shaders must not be a protected resource.
If a bound shader accesses a VkSampler or VkImageView object that enables sampler Y’C_aC_b conversion, that object must only be used with OpImageSample* or OpImageSparseSample* instructions.

- VUID-vkCmdDraw-ConstOffset-06551
  If a bound shader accesses a VkSampler or VkImageView object that enables sampler Y’C_aC_b conversion, that object must not use the ConstOffset and Offset operands.

- VUID-vkCmdDraw-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-vkCmdDraw-format-07753
  If a VkImageView is accessed as a result of this command, then the numeric type of the image view’s format and the Sampled Type operand of the OpTypeImage must match.

- VUID-vkCmdDraw-OpImageWrite-08795
  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-vkCmdDraw-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-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 renderPass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS.

- VUID-vkCmdDraw-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-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 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-vkCmdDraw-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-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-09000**
  If a color attachment is written by any prior command in this subpass or by the load, store, or resolve operations for this subpass, it **must** not be accessed in any way other than as an attachment by this command

- **VUID-vkCmdDraw-None-09001**
  If a depth attachment is written by any prior command in this subpass or by the load, store, or resolve operations for this subpass, it **must** not be accessed in any way other than as an attachment by this command

- **VUID-vkCmdDraw-None-09002**
  If a stencil attachment is written by any prior command in this subpass or by the load, store, or resolve operations for this subpass, it **must** not be accessed in any way other than as an attachment by this command

- **VUID-vkCmdDraw-None-06539**
  If any previously recorded command in the current subpass accessed an image subresource used as an attachment in this subpass 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-None-07831**
  If the bound graphics pipeline state was created with the **VK_DYNAMIC_STATE_VIEWPORT** dynamic state enabled then **vkCmdSetViewport must** have been called in the current command buffer prior to this drawing command

- **VUID-vkCmdDraw-None-07832**
  If the bound graphics pipeline state was created with the **VK_DYNAMIC_STATE_SCISSOR** dynamic state enabled then **vkCmdSetScissor must** have been called in the current command buffer prior to this drawing command

- **VUID-vkCmdDraw-None-07833**
  If the bound graphics pipeline state was created with the **VK_DYNAMIC_STATE_LINE_WIDTH** dynamic state enabled then **vkCmdSetLineWidth must** have been called in the current command buffer prior to this drawing command

- **VUID-vkCmdDraw-None-07834**
  If the bound graphics pipeline state was created with the **VK_DYNAMIC_STATE_DEPTH_BIAS** dynamic state enabled then **vkCmdSetDepthBias 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_BLEND_CONSTANTS` dynamic state enabled then `vkCmdSetBlendConstants` 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_BOUNDS` dynamic state enabled, and if the current `depthBoundsTestEnable` state is `VK_TRUE`, then `vkCmdSetDepthBounds` 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_STENCIL_COMPARE_MASK` dynamic state enabled, and if the current `stencilTestEnable` state is `VK_TRUE`, then `vkCmdSetStencilCompareMask` 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_STENCIL_WRITE_MASK` dynamic state enabled, and if the current `stencilTestEnable` state is `VK_TRUE`, then `vkCmdSetStencilWriteMask` 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_STENCIL_REFERENCE` dynamic state enabled, and if the current `stencilTestEnable` state is `VK_TRUE`, then `vkCmdSetStencilReference` must have been called in the current command buffer prior to this drawing command.

If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index must be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`.

If blending is enabled for any attachment where either the source or destination blend factors for that attachment use the secondary color input, 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, 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, then `rasterizationSamples` for the currently bound graphics pipeline must be the same as the current subpass color and/or depth/stencil attachments.

If blending is enabled for any attachment where either the source or destination blend factors for that attachment use the secondary color input, the maximum value of `Location` for any output attachment statically used in the Fragment Execution Model executed by this command must be less than `maxFragmentDualSrcAttachments`.
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.

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.

**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.
To record an indexed draw, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdDrawIndexed(
    VkCommandBuffer commandBuffer,
    uint32_t indexCount,
    uint32_t instanceCount,
    uint32_t firstIndex,
    int32_t vertexOffset,
    uint32_t firstInstance);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `indexCount` is the number of vertices to draw.
- `instanceCount` is the number of instances to draw.
- `firstIndex` is the base index within the index buffer.
- `vertexOffset` is the value added to the vertex index before indexing into the vertex buffer.
- `firstInstance` is the instance ID of the first instance to draw.

When the command is executed, primitives are assembled using the current primitive topology and `indexCount` vertices whose indices are retrieved from the index buffer. The index buffer is treated as an array of tightly packed unsigned integers of size defined by the `vkCmdBindIndexBuffer ::indexType` parameter with which the buffer was bound.

The first vertex index is at an offset of `firstIndex × indexSize + offset` within the 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-aspectMask-06478
  If a VkImageView is sampled with depth comparison, the image view must have been created with an aspectMask that contains VK_IMAGE_ASPECT_DEPTH_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-None-07888
  If a VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER descriptor is accessed using atomic operations as a result of this command, then the storage texel buffer's format features must contain VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT

- VUID-vkCmdDrawIndexed-None-08600
  For each set \( n \) that is statically used by a bound shader, 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-None-08601
  For each push constant that is statically used by a bound shader, 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-08114
  Descriptors in each bound descriptor set, specified via vkCmdBindDescriptorSets, must be valid as described by descriptor validity if they are statically used by a bound shader

- VUID-vkCmdDrawIndexed-None-08606
  A valid pipeline must be bound to the pipeline bind point used by this command

- VUID-vkCmdDrawIndexed-None-08608
  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-08609
  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-08610**
  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-08611**
  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-uniformBuffers-06935**
  If any stage of the VkPipeline object bound to the pipeline bind point used by this command accesses a uniform buffer, and the robustBufferAccess feature is not enabled, that stage **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-storageBuffers-06936**
  If any stage of the VkPipeline object bound to the pipeline bind point used by this command accesses a storage buffer, and the robustBufferAccess feature is not enabled, that stage **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 bound shaders **must** not be a protected resource

- **VUID-vkCmdDrawIndexed-None-06550**
  If a bound shader accesses a VkSampler or VkImageView object that enables sampler Y’CpCqCr conversion, that object **must** only be used with OpImageSample* or OpImageSparseSample* instructions

- **VUID-vkCmdDrawIndexed-ConstOffset-06551**
  If a bound shader accesses a VkSampler or VkImageView object that enables sampler Y’CpCqCr 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 numeric type of the image view’s format and the Sampled Type operand of the OpTypeImage **must** match

- **VUID-vkCmdDrawIndexed-OpImageWrite-08795**
  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.

- **VUID-vkCmdDrawIndexed-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-vkCmdDrawIndexed-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-vkCmdDrawIndexed-None-07748**
  If any shader statically accesses an input attachment, a valid descriptor must be bound to the pipeline via a descriptor set.

- **VUID-vkCmdDrawIndexed-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-vkCmdDrawIndexed-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-vkCmdDrawIndexed-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-vkCmdDrawIndexed-None-09000**
  If a color attachment is written by any prior command in this subpass or by the load, store, or resolve operations for this subpass, it must not be accessed in any way other than as an attachment by this command.

- **VUID-vkCmdDrawIndexed-None-09001**
  If a depth attachment is written by any prior command in this subpass or by the load, store, or resolve operations for this subpass, it must not be accessed in any way other than as an attachment by this command.

- **VUID-vkCmdDrawIndexed-None-09002**
  If a stencil attachment is written by any prior command in this subpass or by the load, store, or resolve operations for this subpass, it must not be accessed in any way other than as an attachment by this command.

- **VUID-vkCmdDrawIndexed-None-06539**
  If any previously recorded command in the current subpass accessed an image...
subresource used as an attachment in this subpass in any way other than as an
attachment, this command must not write to that image subresource as an attachment

• VUID-vkCmdDrawIndexed-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-vkCmdDrawIndexed-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-vkCmdDrawIndexed-None-07831
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT
dynamic state enabled then vkCmdSetViewport must have been called in the current
command buffer prior to this drawing command

• VUID-vkCmdDrawIndexed-None-07832
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_SCISSOR
dynamic state enabled then vkCmdSetScissor must have been called in the current
command buffer prior to this drawing command

• VUID-vkCmdDrawIndexed-None-07833
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_LINE_WIDTH
dynamic state enabled then vkCmdSetLineWidth must have been called in the current
command buffer prior to this drawing command

• VUID-vkCmdDrawIndexed-None-07834
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_DEPTH_BIAS
dynamic state enabled then vkCmdSetDepthBias must have been called in the current
command buffer prior to this drawing command

• VUID-vkCmdDrawIndexed-None-07835
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_BLEND_CONSTANTS
dynamic state enabled then vkCmdSetBlendConstants must have been called in the current
command buffer prior to this drawing command

• VUID-vkCmdDrawIndexed-None-07836
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_DEPTH_BOUNDS
dynamic state enabled, and if the current depthBoundsTestEnable state is VK_TRUE, then
vkCmdSetDepthBounds must have been called in the current command buffer prior to this drawing command

• VUID-vkCmdDrawIndexed-None-07837
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_STENCILCOMPARE_MASK
dynamic state enabled, and if the current stencilTestEnable state is VK_TRUE, then
vkCmdSetStencilCompareMask must have been called in the current command buffer prior to this drawing command

• VUID-vkCmdDrawIndexed-None-07838
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_STENCIL_WRITE_MASK
dynamic state enabled, and if the current stencilTestEnable state is VK_TRUE, then
vkCmdSetStencilWriteMask 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_STENCIL_REFERENCE` dynamic state enabled, and if the current `stencilTestEnable` state is `VK_TRUE`, then `vkCmdSetStencilReference` must have been called in the current command buffer prior to this drawing command.

If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index must be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`.

If 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, then `rasterizationSamples` for the currently bound graphics pipeline must be the same as the current subpass color and/or depth/stencil attachments.

If blending is enabled for any attachment where either the source or destination blend factors for that attachment use the secondary color input, the maximum value of `Location` for any output attachment statically used in the `Fragment Execution Model` executed by this command must be less than `maxFragmentDualSrcAttachments`.

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.

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`.

• 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-None-07312**
  An index buffer must be bound.

- **VUID-vkCmdDrawIndexed-robustBufferAccess2-07825**
  If robustBufferAccess2 is not enabled, \((\text{indexSize} \times (\text{firstIndex} + \text{indexCount}) + \text{offset})\) must be less than or equal to the size of the bound index buffer, with indexSize being based on the type specified by indexType, where the index buffer, indexType, and offset are specified via `vkCmdBindIndexBuffer`.

- **VUID-vkCmdDrawIndexed-robustBufferAccess2-08798**
  If robustBufferAccess2 is not enabled, \((\text{indexSize} \times (\text{firstIndex} + \text{indexCount}) + \text{offset})\) must be less than or equal to the size of the bound index buffer, with indexSize being based on the type specified by indexType, where the index buffer, indexType, and offset are specified via `vkCmdBindIndexBuffer`.

### Valid Usage (Implicit)

- **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 `VkIndirectCommand` 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-aspectMask-06478**
  If a `VkImageView` is sampled with depth comparison, the image view must have been created with an `aspectMask` that contains `VK_IMAGE_ASPECT_DEPTH_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-None-07888**
  If a `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` descriptor is accessed using atomic operations as a result of this command, then the storage texel buffer's format features...
must contain VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT

- VUID-vkCmdDrawIndirect-None-08600
  For each set $n$ that is statically used by a bound shader, 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-08601
  For each push constant that is statically used by a bound shader, 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-vkCmdDrawIndirect-None-08114
  Descriptors in each bound descriptor set, specified via vkCmdBindDescriptorSets, must be valid as described by descriptor validity if they are statically used by a bound shader

- VUID-vkCmdDrawIndirect-None-08606
  A valid pipeline must be bound to the pipeline bind point used by this command

- VUID-vkCmdDrawIndirect-None-08608
  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-08609
  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-08610
  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-08611
  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-uniformBuffers-06935
  If any stage of the VkPipeline object bound to the pipeline bind point used by this command accesses a uniform buffer, and the robustBufferAccess feature is not enabled, that stage 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-storageBuffers-06936
  If any stage of the VkPipeline object bound to the pipeline bind point used by this command accesses a storage buffer, and the robustBufferAccess feature is not enabled,
that stage **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 **bound shaders must** not be a protected resource

- VUID-vkCmdDrawIndirect-ConstOffset-06550
  
  If a **bound shader** accesses a `VkSampler` or `VkImageView` object that enables sampler YC\(_{\alpha \beta}C_{\gamma}\) conversion, that object **must** only be used with `OpImageSample*` or `OpImageSparseSample*` instructions

- VUID-vkCmdDrawIndirect-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-vkCmdDrawIndirect-format-07753
  
  If a `VkImageView` is accessed as a result of this command, then the numeric type of the image view's `format` and the `Sampled Type` operand of the `OpTypeImage` **must** match

- VUID-vkCmdDrawIndirect-OpImageWrite-08795
  
  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-vkCmdDrawIndirect-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-vkCmdDrawIndirect-None-07288
  
  Any shader invocation executed by this command **must** terminate

- VUID-vkCmdDrawIndirect-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-vkCmdDrawIndirect-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-vkCmdDrawIndirect-None-07748
  
  If any shader statically accesses an input attachment, a valid descriptor **must** be bound to the pipeline via a descriptor set

- VUID-vkCmdDrawIndirect-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-vkCmdDrawIndirect-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-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-09000**
  If a color attachment is written by any prior command in this subpass or by the load, store, or resolve operations for this subpass, it **must** not be accessed in any way other than as an attachment by this command

- **VUID-vkCmdDrawIndirect-None-09001**
  If a depth attachment is written by any prior command in this subpass or by the load, store, or resolve operations for this subpass, it **must** not be accessed in any way other than as an attachment by this command

- **VUID-vkCmdDrawIndirect-None-09002**
  If a stencil attachment is written by any prior command in this subpass or by the load, store, or resolve operations for this subpass, it **must** not be accessed in any way other than as an attachment by this command

- **VUID-vkCmdDrawIndirect-None-06539**
  If any previously recorded command in the current subpass accessed an image subresource used as an attachment in this subpass 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-None-07831**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT` dynamic state enabled then `vkCmdSetViewport` **must** have been called in the current command buffer prior to this drawing command

- **VUID-vkCmdDrawIndirect-None-07832**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_SCISSOR` dynamic state enabled then `vkCmdSetScissor` **must** have been called in the current command buffer prior to this drawing command

- **VUID-vkCmdDrawIndirect-None-07833**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_LINE_WIDTH`
dynamic state enabled then \texttt{vkCmdSetLineWidth} must have been called in the current command buffer prior to this drawing command

- **VUID-vkCmdDrawIndirect-None-07834**
  If the bound graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_DEPTH_BIAS} dynamic state enabled then \texttt{vkCmdSetDepthBias} must have been called in the current command buffer prior to this drawing command

- **VUID-vkCmdDrawIndirect-None-07835**
  If the bound graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_BLEND_CONSTANTS} dynamic state enabled then \texttt{vkCmdSetBlendConstants} must have been called in the current command buffer prior to this drawing command

- **VUID-vkCmdDrawIndirect-None-07836**
  If the bound graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_DEPTH_BOUNDS} dynamic state enabled, and if the current \texttt{depthBoundsTestEnable} state is \texttt{VK_TRUE}, then \texttt{vkCmdSetDepthBounds} must have been called in the current command buffer prior to this drawing command

- **VUID-vkCmdDrawIndirect-None-07837**
  If the bound graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK} dynamic state enabled, and if the current \texttt{stencilTestEnable} state is \texttt{VK_TRUE}, then \texttt{vkCmdSetStencilCompareMask} must have been called in the current command buffer prior to this drawing command

- **VUID-vkCmdDrawIndirect-None-07838**
  If the bound graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_STENCIL_WRITE_MASK} dynamic state enabled, and if the current \texttt{stencilTestEnable} state is \texttt{VK_TRUE}, then \texttt{vkCmdSetStencilWriteMask} must have been called in the current command buffer prior to this drawing command

- **VUID-vkCmdDrawIndirect-None-07839**
  If the bound graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_STENCIL_REFERENCE} dynamic state enabled, and if the current \texttt{stencilTestEnable} state is \texttt{VK_TRUE}, then \texttt{vkCmdSetStencilReference} must have been called in the current command buffer prior to this drawing command

- **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 \texttt{VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex}

- **VUID-vkCmdDrawIndirect-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 \texttt{format features} do not contain \texttt{VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT}, then the \texttt{blendEnable} member of the corresponding element of the \texttt{pAttachments} member of \texttt{pColorBlendState} must be \texttt{VK_FALSE}

- **VUID-vkCmdDrawIndirect-multisampledRenderToSingleSampled-07284**
  If rasterization is not disabled in the bound graphics pipeline, then \texttt{rasterizationSamples} for the currently bound graphics pipeline must be the same as
the current subpass color and/or depth/stencil attachments

- VUID-vkCmdDrawIndirect-maxFragmentDualSrcAttachments-09239
  If blending is enabled for any attachment where either the source or destination blend factors for that attachment use the secondary color input, the maximum value of Location for any output attachment statically used in the Fragment Execution Model executed by this command must be less than maxFragmentDualSrcAttachments

- VUID-vkCmdDrawIndirect-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-vkCmdDrawIndirect-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-vkCmdDrawIndirect-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-vkCmdDrawIndirect-buffer-02708
  If buffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

- VUID-vkCmdDrawIndirect-buffer-02709
  buffer must have been created with the VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT bit set

- VUID-vkCmdDrawIndirect-offset-02710
  offset must be a multiple of 4

- VUID-vkCmdDrawIndirect-commandBuffer-02711
  commandBuffer must not be a protected command buffer

- VUID-vkCmdDrawIndirect-drawCount-02718
  If the multiDrawIndirect feature is not enabled, drawCount must be 0 or 1

- VUID-vkCmdDrawIndirect-drawCount-02719
  drawCount must be less than or equal to VkPhysicalDeviceLimits::maxDrawIndirectCount

- VUID-vkCmdDrawIndirect-drawCount-00476
  If drawCount is greater than 1, stride must be a multiple of 4 and must be greater than or equal to sizeof(VkDrawIndirectCommand)

- VUID-vkCmdDrawIndirect-drawCount-00487
  If drawCount is equal to 1, (offset + sizeof(VkDrawIndirectCommand)) must be less than or equal to the size of buffer

- VUID-vkCmdDrawIndirect-drawCount-00488
  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)

• VUID-vkCmdDrawIndirect-commandBuffer-parameter
  
  commandBuffer must be a valid VkCommandBuffer handle

• VUID-vkCmdDrawIndirect-buffer-parameter
  
  buffer must be a valid VkBuffer handle

• VUID-vkCmdDrawIndirect-commandBuffer-recording
  
  commandBuffer must be in the recording state

• VUID-vkCmdDrawIndirect-commandBuffer-cmdpool
  
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

• VUID-vkCmdDrawIndirect-renderpass
  
  This command must only be called inside of a render pass instance

• VUID-vkCmdDrawIndirect-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 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,  // commandBuffer is the command buffer into which the command is recorded.
    VkBuffer buffer,               // buffer is the buffer containing draw parameters.
    VkDeviceSize offset,           // offset is the byte offset into buffer where parameters begin.
    VkBuffer countBuffer,          // countBuffer is the buffer containing the draw count.
    VkDeviceSize countBufferOffset, // countBufferOffset is the byte offset into countBuffer where the draw count begins.
    uint32_t maxDrawCount,         // 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.
    uint32_t stride);              // 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-aspectMask-06478**
  If a `VkImageView` is sampled with depth comparison, the image view must have been created with an `aspectMask` that contains `VK_IMAGE_ASPECT_DEPTH_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-None-07888**
  If a `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` descriptor is accessed using atomic operations as a result of this command, then the storage texel buffer's format features must contain `VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT`.

- **VUID-vkCmdDrawIndirectCount-None-08600**
  For each set `n` that is statically used by a bound shader, 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-None-08601**
  For each push constant that is statically used by a bound shader, 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-08114**
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid as described by descriptor validity if they are statically used by a bound shader.

- **VUID-vkCmdDrawIndirectCount-None-08606**
  A valid pipeline must be bound to the pipeline bind point used by this command.

- **VUID-vkCmdDrawIndirectCount-None-08608**
  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-08609**
  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-08610
  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-vkCmdDrawIndirectCount-None-08611
  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-vkCmdDrawIndirectCount-uniformBuffers-06935
  If any stage of the VkPipeline object bound to the pipeline bind point used by this command accesses a uniform buffer, and the robustBufferAccess feature is not enabled, that stage 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-vkCmdDrawIndirectCount-storageBuffers-06936
  If any stage of the VkPipeline object bound to the pipeline bind point used by this command accesses a storage buffer, and the robustBufferAccess feature is not enabled, that stage 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-vkCmdDrawIndirectCount-commandBuffer-02707
  If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, any resource accessed by bound shaders must not be a protected resource

- VUID-vkCmdDrawIndirectCount-None-06550
  If a bound shader accesses a VkSampler or VkImageView object that enables sampler Y'CbCr conversion, that object must only be used with OpImageSample* or OpImageSparseSample* instructions

- VUID-vkCmdDrawIndirectCount-ConstOffset-06551
  If a bound shader accesses a VkSampler or VkImageView object that enables sampler Y'CbCr conversion, that object must not use the ConstOffset and Offset operands

- VUID-vkCmdDrawIndirectCount-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-vkCmdDrawIndirectCount-format-07753
  If a VkImageView is accessed as a result of this command, then the numeric type of the image view's format and the Sampled Type operand of the OpTypeImage must match

- VUID-vkCmdDrawIndirectCount-OpImageWrite-08795
  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-vkCmdDrawIndirectCount-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-vkCmdDrawIndirectCount-None-07288**
  Any shader invocation executed by this command must terminate.

- **VUID-vkCmdDrawIndirectCount-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-vkCmdDrawIndirectCount-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-vkCmdDrawIndirectCount-None-07748**
  If any shader statically accesses an input attachment, a valid descriptor must be bound to the pipeline via a descriptor set.

- **VUID-vkCmdDrawIndirectCount-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-vkCmdDrawIndirectCount-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-vkCmdDrawIndirectCount-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-vkCmdDrawIndirectCount-None-09000**
  If a color attachment is written by any prior command in this subpass or by the load, store, or resolve operations for this subpass, it must not be accessed in any way other than as an attachment by this command.

- **VUID-vkCmdDrawIndirectCount-None-09001**
  If a depth attachment is written by any prior command in this subpass or by the load, store, or resolve operations for this subpass, it must not be accessed in any way other than as an attachment by this command.

- **VUID-vkCmdDrawIndirectCount-None-09002**
  If a stencil attachment is written by any prior command in this subpass or by the load, store, or resolve operations for this subpass, it must not be accessed in any way other than as an attachment by this command.

- **VUID-vkCmdDrawIndirectCount-None-06539**
  If any previously recorded command in the current subpass accessed an image...
subresource used as an attachment in this subpass in any way other than as an attachment, this command must not write to that image subresource as an attachment

- VUID-vkCmdDrawIndirectCount-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-vkCmdDrawIndirectCount-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-vkCmdDrawIndirectCount-None-07831
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT dynamic state enabled then vkCmdSetViewport must have been called in the current command buffer prior to this drawing command

- VUID-vkCmdDrawIndirectCount-None-07832
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_SCISSOR dynamic state enabled then vkCmdSetScissor must have been called in the current command buffer prior to this drawing command

- VUID-vkCmdDrawIndirectCount-None-07833
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_LINE_WIDTH dynamic state enabled then vkCmdSetLineWidth must have been called in the current command buffer prior to this drawing command

- VUID-vkCmdDrawIndirectCount-None-07834
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_DEPTH_BIAS dynamic state enabled then vkCmdSetDepthBias must have been called in the current command buffer prior to this drawing command

- VUID-vkCmdDrawIndirectCount-None-07835
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_BLEND_CONSTANTS dynamic state enabled then vkCmdSetBlendConstants must have been called in the current command buffer prior to this drawing command

- VUID-vkCmdDrawIndirectCount-None-07836
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_DEPTH_BOUNDS dynamic state enabled, and if the current depthBoundsTestEnable state is VK_TRUE, then vkCmdSetDepthBounds must have been called in the current command buffer prior to this drawing command

- VUID-vkCmdDrawIndirectCount-None-07837
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK dynamic state enabled, and if the current stencilTestEnable state is VK_TRUE, then vkCmdSetStencilCompareMask must have been called in the current command buffer prior to this drawing command

- VUID-vkCmdDrawIndirectCount-None-07838
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_STENCIL_WRITE_MASK dynamic state enabled, and if the current stencilTestEnable state is VK_TRUE, then vkCmdSetStencilWriteMask 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_STENCIL_REFERENCE` dynamic state enabled, and if the current `stencilTestEnable` state is `VK_TRUE`, then `vkCmdSetStencilReference` must have been called in the current command buffer prior to this drawing command.

If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index must be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`.

If 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, then `rasterizationSamples` for the currently bound graphics pipeline must be the same as the current subpass color and/or depth/stencil attachments.

If blending is enabled for any attachment where either the source or destination blend factors for that attachment use the secondary color input, the maximum value of `Location` for any output attachment statically used in the Fragment Execution Model executed by this command must be less than `maxFragmentDualSrcAttachments`.

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 Vertext Input Description.

If `buffer` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.

`buffer` must have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set.

Offset must be a multiple of 4.
• VUID-vkCmdDrawIndirectCount-commandBuffer-02711
commandBuffer must not be a protected command buffer

• VUID-vkCmdDrawIndirectCount-countBuffer-02714
If countBuffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

• VUID-vkCmdDrawIndirectCount-countBuffer-02715
countBuffer must have been created with the VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT bit set

• VUID-vkCmdDrawIndirectCount-countBufferOffset-02716
countBufferOffset must be a multiple of 4

• VUID-vkCmdDrawIndirectCount-countBuffer-02717
The count stored in countBuffer must be less than or equal to VkPhysicalDeviceLimits::maxDrawIndirectCount

• VUID-vkCmdDrawIndirectCount-countBufferOffset-04129
(countBufferOffset + sizeof(uint32_t)) must be less than or equal to the size of countBuffer

• VUID-vkCmdDrawIndirectCount-None-04445
If drawIndirectCount is not enabled this function must not be used

• VUID-vkCmdDrawIndirectCount-stride-03110
stride must be a multiple of 4 and must be greater than or equal to sizeof(VkDrawIndirectCommand)

• VUID-vkCmdDrawIndirectCount-maxDrawCount-03111
If maxDrawCount is greater than or equal to 1, (stride × (maxDrawCount - 1) + offset + sizeof(VkDrawIndirectCommand)) must be less than or equal to the size of buffer

• VUID-vkCmdDrawIndirectCount-countBuffer-03121
If the count stored in countBuffer is equal to 1, (offset + sizeof(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, (stride × (drawCount - 1) + offset + sizeof(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
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-aspectMask-06478
If a VkImageView is sampled with depth comparison, the image view must have been created with an aspectMask that contains VK_IMAGE_ASPECT_DEPTH_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-None-07888
If a VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER descriptor is accessed using atomic operations as a result of this command, then the storage texel buffer’s format features must contain VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT

• VUID-vkCmdDrawIndexedIndirect-None-08600
For each set n that is statically used by a bound shader, 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-vkCmdDrawIndexedIndirect-None-08601
For each push constant that is statically used by a bound shader, 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-vkCmdDrawIndexedIndirect-None-08606
A valid pipeline must be bound to the pipeline bind point used by this command

• VUID-vkCmdDrawIndexedIndirect-None-08608
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-vkCmdDrawIndexedIndirect-None-08609

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-08610

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-08611

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-uniformBuffers-06935

If any stage of the VkPipeline object bound to the pipeline bind point used by this command accesses a uniform buffer, and the robustBufferAccess feature is not enabled, that stage 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-storageBuffers-06936

If any stage of the VkPipeline object bound to the pipeline bind point used by this command accesses a storage buffer, and the robustBufferAccess feature is not enabled, that stage 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 bound shaders must not be a protected resource.

• VUID-vkCmdDrawIndexedIndirect-None-06550

If a bound shader accesses a VkSampler or VkImageView object that enables sampler YCₐCᵣ conversion, that object must only be used with OpImageSample* or OpImageSparseSample* instructions.

• VUID-vkCmdDrawIndexedIndirect-ConstOffset-06551

If a bound shader accesses a VkSampler or VkImageView object that enables sampler YCₐCᵣ 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 numeric type of the image view's format and the Sampled Type operand of the OpTypeImage must match.

• VUID-vkCmdDrawIndexedIndirect-OpImageWrite-08795
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-09000
  If a color attachment is written by any prior command in this subpass or by the load, store, or resolve operations for this subpass, it **must** not be accessed in any way other than as an attachment by this command

- VUID-vkCmdDrawIndexedIndirect-None-09001
  If a depth attachment is written by any prior command in this subpass or by the load, store, or resolve operations for this subpass, it **must** not be accessed in any way other than as an attachment by this command

- VUID-vkCmdDrawIndexedIndirect-None-09002
  If a stencil attachment is written by any prior command in this subpass or by the load,
store, or resolve operations for this subpass, it **must** not be accessed in any way other than as an attachment by this command

- VUID-vkCmdDrawIndexedIndirect-None-06539
  If any previously recorded command in the current subpass accessed an image subresource used as an attachment in this subpass 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 must be VK_STENCIL_OP_KEEP**

- VUID-vkCmdDrawIndexedIndirect-None-07831
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT` dynamic state enabled then `vkCmdSetViewport` **must** have been called in the current command buffer prior to this drawing command

- VUID-vkCmdDrawIndexedIndirect-None-07832
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_SCISSOR` dynamic state enabled then `vkCmdSetScissor` **must** have been called in the current command buffer prior to this drawing command

- VUID-vkCmdDrawIndexedIndirect-None-07833
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_LINE_WIDTH` dynamic state enabled then `vkCmdSetLineWidth` **must** have been called in the current command buffer prior to this drawing command

- VUID-vkCmdDrawIndexedIndirect-None-07834
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_DEPTH_BIAS` dynamic state enabled then `vkCmdSetDepthBias` **must** have been called in the current command buffer prior to this drawing command

- VUID-vkCmdDrawIndexedIndirect-None-07835
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_BLEND_CONSTANTS` dynamic state enabled then `vkCmdSetBlendConstants` **must** have been called in the current command buffer prior to this drawing command

- VUID-vkCmdDrawIndexedIndirect-None-07836
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_DEPTH_BOUNDS` dynamic state enabled, and if the current `depthBoundsTestEnable` state is `VK_TRUE`, then `vkCmdSetDepthBounds` **must** have been called in the current command buffer prior to this drawing command

- VUID-vkCmdDrawIndexedIndirect-None-07837
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK` dynamic state enabled, and if the current `stencilTestEnable` state is `VK_TRUE`, then `vkCmdSetStencilCompareMask` **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_STENCIL_WRITE_MASK dynamic state enabled, and if the current
stencilTestEnable state is VK_TRUE, then vkCmdSetStencilWriteMask 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_STENCIL_REFERENCE dynamic state enabled, and if the current
stencilTestEnable state is VK_TRUE, then vkCmdSetStencilReference must have been
called in the current command buffer prior to this drawing command.

If the draw is recorded in a render pass instance with multiview enabled, the maximum
instance index must be less than or equal to VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex.

If 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, then rasterizationSamples for the currently bound graphics pipeline must be the same as the current subpass color and/or depth/stencil attachments.

If blending is enabled for any attachment where either the source or destination blend
factors for that attachment use the secondary color input, the maximum value of Location
for any output attachment statically used in the Fragment Execution Model executed by this
command must be less than maxFragmentDualSrcAttachments.

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 buffer is non-sparse then it must be bound completely and contiguously to a single
VkDeviceMemory object

- VUID-vkCmdDrawIndexedIndirect-buffer-02709
  buffer must have been created with the VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT bit set

- VUID-vkCmdDrawIndexedIndirect-offset-02710
  offset must be a multiple of 4

- VUID-vkCmdDrawIndexedIndirect-commandBuffer-02711
  commandBuffer must not be a protected command buffer

- VUID-vkCmdDrawIndexedIndirect-drawCount-02718
  If the multiDrawIndirect feature is not enabled, drawCount must be 0 or 1

- VUID-vkCmdDrawIndexedIndirect-drawCount-02719
  drawCount must be less than or equal to VkPhysicalDeviceLimits::maxDrawIndirectCount

- VUID-vkCmdDrawIndexedIndirect-None-07312
  An index buffer must be bound

- VUID-vkCmdDrawIndexedIndirect-robustBufferAccess2-07825
  If robustBufferAccess2 is not enabled, \((\text{indexSize} \times (\text{firstIndex} + \text{indexCount}) + \text{offset})\) must be less than or equal to the size of the bound index buffer, with indexSize being based on the type specified by indexType, where the index buffer, indexType, and offset are specified via vkCmdBindIndexBuffer

- VUID-vkCmdDrawIndexedIndirect-drawCount-00528
  If drawCount is greater than 1, stride must be a multiple of 4 and must be greater than or equal to sizeof(VkDrawIndexedIndirectCommand)

- VUID-vkCmdDrawIndexedIndirect-drawCount-00539
  If drawCount is equal to 1, \((\text{offset} + \text{sizeof(VkDrawIndexedIndirectCommand)})\) must be less than or equal to the size of buffer

- VUID-vkCmdDrawIndexedIndirect-drawCount-00540
  If drawCount is greater than 1, \((\text{stride} \times (\text{drawCount} - 1) + \text{offset} + \text{sizeof(VkDrawIndexedIndirectCommand)})\) 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-robustBufferAccess2-08798
If robustBufferAccess2 is not enabled, \((\text{indexSize} \times (\text{firstIndex} + \text{indexCount}) + \text{offset})\) must be less than or equal to the size of the bound index buffer, with indexSize being based on the type specified by indexType, where the index buffer, indexType, and offset are specified via vkCmdBindIndexBuffer

• 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-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,
  VkBuffer buffer,
  VkDeviceSize offset,
  VkBuffer countBuffer,
  VkDeviceSize countBufferOffset,
  uint32_t maxDrawCount,
  uint32_t stride);
```

• commandBuffer is the command buffer into which the command is recorded.

• buffer is the buffer containing draw parameters.

• offset is the byte offset into buffer where parameters begin.

• countBuffer is the buffer containing the draw count.

• countBufferOffset is the byte offset into countBuffer where the draw count begins.

• maxDrawCount specifies the maximum number of draws that will be executed. The actual number of executed draw calls is the minimum of the count specified in countBuffer and maxDrawCount.

• stride is the byte stride between successive sets of draw parameters.

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
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`.

If a `VkImageView` is sampled with depth comparison, the image view **must** have been created with an `aspectMask` that contains `VK_IMAGE_ASPECT_DEPTH_BIT`.

If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`.

If a `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` descriptor is accessed using atomic operations as a result of this command, then the storage texel buffer's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT`.

For each set `n` that is statically used by a bound shader, a descriptor set **must** have been bound to `n` at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set `n`, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in **Pipeline Layout Compatibility**.

For each push constant that is statically used by a bound shader, 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 a bound shader.

A valid pipeline **must** be bound to the pipeline bind point used by this command.

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

- VUID-vkCmdDrawIndexedIndirectCount-None-08611
  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-vkCmdDrawIndexedIndirectCount-uniformBuffers-06935
  If any stage of the VkPipeline object bound to the pipeline bind point used by this command accesses a uniform buffer, and the robustBufferAccess feature is not enabled, that stage 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-vkCmdDrawIndexedIndirectCount-storageBuffers-06936
  If any stage of the VkPipeline object bound to the pipeline bind point used by this command accesses a storage buffer, and the robustBufferAccess feature is not enabled, that stage 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-vkCmdDrawIndexedIndirectCount-commandBuffer-02707
  If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, any resource accessed by bound shaders must not be a protected resource

- VUID-vkCmdDrawIndexedIndirectCount-None-06550
  If a bound shader accesses a VkSampler or VkImageView object that enables sampler Y’C_aC_R conversion, that object must only be used with OpImageSample* or OpImageSparseSample* instructions

- VUID-vkCmdDrawIndexedIndirectCount-ConstOffset-06551
  If a bound shader accesses a VkSampler or VkImageView object that enables sampler Y’C_aC_R conversion, that object must not use the ConstOffset and Offset operands

- VUID-vkCmdDrawIndexedIndirectCount-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-vkCmdDrawIndexedIndirectCount-format-07753
  If a VkImageView is accessed as a result of this command, then the numeric type of the image view’s format and the Sampled Type operand of the OpTypeImage must match

- VUID-vkCmdDrawIndexedIndirectCount-OpImageWrite-08795
  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-vkCmdDrawIndexedIndirectCount-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-vkCmdDrawIndexedIndirectCount-None-07288
Any shader invocation executed by this command **must** terminate

- **VUID-vkCmdDrawIndexedIndirectCount-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-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-09000**
  If a color attachment is written by any prior command in this subpass or by the load, store, or resolve operations for this subpass, it **must** not be accessed in any way other than as an attachment by this command

- **VUID-vkCmdDrawIndexedIndirectCount-None-09001**
  If a depth attachment is written by any prior command in this subpass or by the load, store, or resolve operations for this subpass, it **must** not be accessed in any way other than as an attachment by this command

- **VUID-vkCmdDrawIndexedIndirectCount-None-09002**
  If a stencil attachment is written by any prior command in this subpass or by the load, store, or resolve operations for this subpass, it **must** not be accessed in any way other than as an attachment by this command

- **VUID-vkCmdDrawIndexedIndirectCount-None-06539**
  If any previously recorded command in the current subpass accessed an image subresource used as an attachment in this subpass 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
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-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-None-07831**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT` dynamic state enabled then `vkCmdSetViewport` **must** have been called in the current command buffer prior to this drawing command.

- **VUID-vkCmdDrawIndexedIndirectCount-None-07832**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_SCISSOR` dynamic state enabled then `vkCmdSetScissor` **must** have been called in the current command buffer prior to this drawing command.

- **VUID-vkCmdDrawIndexedIndirectCount-None-07833**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_LINE_WIDTH` dynamic state enabled then `vkCmdSetLineWidth` **must** have been called in the current command buffer prior to this drawing command.

- **VUID-vkCmdDrawIndexedIndirectCount-None-07834**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_DEPTH_BIAS` dynamic state enabled then `vkCmdSetDepthBias` **must** have been called in the current command buffer prior to this drawing command.

- **VUID-vkCmdDrawIndexedIndirectCount-None-07835**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_BLEND_CONSTANTS` dynamic state enabled then `vkCmdSetBlendConstants` **must** have been called in the current command buffer prior to this drawing command.

- **VUID-vkCmdDrawIndexedIndirectCount-None-07836**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_BLEND_CONSTANTS` dynamic state enabled then `vkCmdSetBlendConstants` **must** have been called in the current command buffer prior to this drawing command.

- **VUID-vkCmdDrawIndexedIndirectCount-None-07837**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK` dynamic state enabled, and if the current `depthBoundsTestEnable` state is `VK_TRUE`, then `vkCmdSetDepthBounds` **must** have been called in the current command buffer prior to this drawing command.

- **VUID-vkCmdDrawIndexedIndirectCount-None-07838**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK` dynamic state enabled, and if the current `stencilTestEnable` state is `VK_TRUE`, then `vkCmdSetStencilCompareMask` **must** have been called in the current command buffer prior to this drawing command.

- **VUID-vkCmdDrawIndexedIndirectCount-None-07839**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_STENCIL_WRITE_MASK` dynamic state enabled, and if the current `stencilTestEnable` state is `VK_TRUE`, then `vkCmdSetStencilWriteMask` **must** have been called in the current command buffer prior to this drawing command.

- **VUID-vkCmdDrawIndexedIndirectCount-None-07840**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_STENCIL_REFERENCE` dynamic state enabled, and if the current `stencilTestEnable` state is `VK_TRUE`, then `vkCmdSetStencilReference` **must** have been called in the current command buffer prior to this drawing command.
in the current command buffer prior to this drawing command

- 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-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-multisampledRenderToSingleSampled-07284
  If rasterization is not disabled in the bound graphics pipeline, then `rasterizationSamples` for the currently bound graphics pipeline must be the same as the current subpass color and/or depth/stencil attachments

- VUID-vkCmdDrawIndexedIndirectCount-maxFragmentDualSrcAttachments-09239
  If blending is enabled for any attachment where either the source or destination blend factors for that attachment use the secondary color input, the maximum value of `Location` for any output attachment statically used in the Fragment Execution Model executed by this command must be less than `maxFragmentDualSrcAttachments`

- 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-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-robustBufferAccess2-07825**
  If `robustBufferAccess2` is not enabled, `(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-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</td>
<td>Inside</td>
<td>Graphics</td>
<td>Action</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></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**

```spirv
...  
%1 = OpExtInstImport "GLSL.std.450"  
...
OpName %9 "variableName"
OpName %15 "variableNameArray"
OpDecorate %18 BuiltIn VertexIndex
```
21.1.1. Attribute Location and Component Assignment

The `Location` decoration specifies which vertex input attribute is used to read and interpret the data that a variable will consume.

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 `Input attribute components accessed by 16-bit and 32-bit input variables`. Any 16-bit or 32-bit scalar or vector input will consume a single `Location`. For 16-bit and 32-bit data types, missing components are filled in with default values as described below.

If an implementation supports `storageInputOutput16`, vertex shader input variables can have a width of 16 bits.

*Table 21. Input attribute components accessed by 16-bit and 32-bit input variables*

<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>
<tr>
<td>two-component vector</td>
<td>1</td>
<td>(o, y, z, o)</td>
</tr>
<tr>
<td>two-component vector</td>
<td>2</td>
<td>(o, o, 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 \( \text{VkVertexInputAttributeDescription}::\text{location} \). Such matrices are treated as an array of column vectors with values taken from the input attributes identified in Input attributes accessed by 32-bit input matrix variables. The \( \text{VkVertexInputAttributeDescription}::\text{format} \) must be specified with a \( \text{VkFormat} \) 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
The `VkVertexInputAttributeDescription::location` location slots and component words 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.

<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), (x, y, -, -)</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>(o, o, 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>R64G64B64A64</td>
<td>i, i+1</td>
<td>scalar</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, o, o), (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), (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. Two variables are allowed to share the same Location slot only if their Component words do not overlap. If multiple variables share the same Location slot, they must all have the same SPIR-V floating-point component type or all have the same width scalar 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 a VkStructureType value identifying 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
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:

```c
// 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 binding.

### 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
  - The **format features** of **format** must contain `VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT`

### 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,
```

---

766
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>

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 VkPipelineVertexInputStateCreateInfo::pVertexBindingDescriptions with VkVertexInputAttributeDescription::binding equal to attribDesc.binding.

- Let vertexIndex be the index of the vertex within the draw (a value between firstVertex and firstVertex+vertexCount for vkCmdDraw, or a value taken from the index buffer plus vertexOffset for vkCmdDrawIndexed), and let instanceIndex be the instance number of the draw (a value
between \texttt{firstInstance} and \texttt{firstInstance+instanceCount}).

- Let \texttt{offset} be an array of offsets into the currently bound vertex buffers specified during \texttt{vkCmdBindVertexBuffers} with \texttt{pOffsets}.

```python
bufferBindingAddress = buffer[binding].baseAddress + offset[binding];

if (bindingDesc.inputRate == VK_VERTEX_INPUT_RATE_VERTEX)
    effectiveVertexOffset = vertexIndex * bindingDesc.stride;
else
    effectiveVertexOffset = instanceIndex * bindingDesc.stride;

attribAddress = bufferBindingAddress + effectiveVertexOffset + attribDesc.offset;
```

### 21.3.1. Vertex Input Extraction

For each attribute, raw data is extracted starting at \texttt{attribAddress} and is converted from the \texttt{VkVertexInputAttributeDescription}’s \texttt{format} to either floating-point, unsigned integer, or signed integer based on the numeric type of \texttt{format}. The numeric type of \texttt{format} must match the numeric 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 \texttt{format} is a 64-bit data type. If \texttt{format} is a packed format, \texttt{attribAddress} must be a multiple of the size in bytes of the whole attribute data type as described in Packed Formats. Otherwise, \texttt{attribAddress} must be a multiple of the size in bytes of the component type indicated by \texttt{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 \texttt{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 using one of the **Triangles**, **Quads**, or **IsoLines** execution modes. This instruction may be specified in either the tessellation evaluation or tessellation control shader. Other tessellation-related execution modes can also be specified in either the tessellation control or tessellation evaluation shaders.

Any tessellation-related modes specified in both the tessellation control and tessellation evaluation shaders 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 \texttt{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 \texttt{domainOrigin} member of \texttt{VkPipelineTessellationDomainOriginStateCreateInfo}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{domain_parameterization.png}
\caption{Domain parameterization for tessellation primitive modes (upper-left origin)}
\end{figure}

\textit{Figure 11. Domain parameterization for tessellation primitive modes (upper-left origin)}
In the domain parameterization diagrams, the coordinates illustrate the value of \textit{TessCoord} at the corners of the domain. The labels on the edges indicate the inner (IL0 and IL1) and outer (OL0 through OL3) tessellation level values used to control the number of subdivisions along each edge of the domain.

For triangles, the vertex's position is a barycentric coordinate \((u,v,w)\), where \(u + v + w = 1.0\), and indicates the relative influence of the three vertices of the triangle on the position of the vertex. For quads and isolines, the position is a \((u,v)\) coordinate indicating the relative horizontal and vertical position of the vertex relative to the subdivided rectangle. The subdivision process is explained in more detail in subsequent sections.

### 22.2. Tessellator Patch Discard

A patch is discarded by the tessellator if any relevant outer tessellation level is less than or equal to zero.

Patches will also be discarded if any relevant outer tessellation level corresponds to a floating-point
NaN (not a number) in implementations supporting NaN.

No new primitives are generated and the tessellation evaluation shader is not executed for patches that are discarded. For Quads, all four outer levels are relevant. For Triangles and IsoLines, only the first three or two outer levels, respectively, are relevant. Negative inner levels will not cause a patch to be discarded; they will be clamped as described below.

22.3. Tessellator Spacing

Each of the tessellation levels is used to determine the number and spacing of segments used to subdivide a corresponding edge. The method used to derive the number and spacing of segments is specified by an OpExecutionMode in the tessellation control or tessellation evaluation shader using one of the identifiers SpacingEqual, SpacingFractionalEven, or SpacingFractionalOdd.

If SpacingEqual is used, the floating-point tessellation level is first clamped to \([1, \text{maxLevel}]\), where maxLevel is the implementation-dependent maximum tessellation level (VkPhysicalDeviceLimits::maxTessellationGenerationLevel). The result is rounded up to the nearest integer \(n\), and the corresponding edge is divided into \(n\) segments of equal length in \((u,v)\) space.

If SpacingFractionalEven is used, the tessellation level is first clamped to \([2, \text{maxLevel}]\) and then rounded up to the nearest even integer \(n\). If SpacingFractionalOdd is used, the tessellation level is clamped to \([1, \text{maxLevel} - 1]\) and then rounded up to the nearest odd integer \(n\). If \(n\) is one, the edge will not be subdivided. Otherwise, the corresponding edge will be divided into \(n - 2\) segments of equal length, and two additional segments of equal length that are typically shorter than the other segments. The length of the two additional segments relative to the others will decrease monotonically with \(n - f\), where \(f\) is the clamped floating-point tessellation level. When \(n - f\) is zero, the additional segments will have equal length to the other segments. As \(n - f\) approaches 2.0, the relative length of the additional segments approaches zero. The two additional segments must be placed symmetrically on opposite sides of the subdivided edge. The relative location of these two segments is implementation-dependent, but must be identical for any pair of subdivided edges with identical values of \(f\).

When tessellating triangles or quads using point mode with fractional odd spacing, the tessellator may produce interior vertices that are positioned on the edge of the patch if an inner tessellation level is less than or equal to one. Such vertices are considered distinct from vertices produced by subdividing the outer edge of the patch, even if there are pairs of vertices with identical coordinates.

22.4. Tessellation Primitive Ordering

Few guarantees are provided for the relative ordering of primitives produced by tessellation, as they pertain to primitive order.

- The output primitives generated from each input primitive are passed to subsequent pipeline stages in an implementation-dependent order.
- All output primitives generated from a given input primitive are passed to subsequent pipeline stages before any output primitives generated from subsequent input primitives.
22.5. Tessellator Vertex Winding Order

When the tessellator produces triangles (in the Triangles or Quads modes), the orientation of all triangles is specified with an OpExecutionMode of VertexOrderCw or VertexOrderCcw in the tessellation control or tessellation evaluation shaders. If the order is VertexOrderCw, the vertices of all generated triangles will have clockwise ordering in (u,v) or (u,v,w) space. If the order is VertexOrderCcw, the vertices will have counter-clockwise ordering in that space.

If the tessellation domain has an upper-left origin, the vertices of a triangle have counter-clockwise ordering if

\[ a = u_0 v_1 - u_1 v_0 + u_1 v_2 - u_2 v_1 + u_2 v_0 - u_0 v_2 \]

is negative, and clockwise ordering if \( a \) is positive. \( u_i \) and \( v_i \) are the \( u \) and \( v \) coordinates in normalized parameter space of the \( i \)th vertex of the triangle. If the tessellation domain has a lower-left origin, the vertices of a triangle have counter-clockwise ordering if \( a \) is positive, and clockwise ordering if \( a \) is negative.

\[ \text{Note} \]

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.

![Inner Triangle Tessellation](image)

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 \( 1 + \epsilon \) and will result in a two- or three-segment subdivision depending on the tessellation spacing. When used with fractional odd spacing, the three-segment subdivision may produce inner vertices positioned on the edge of the 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.

![Inner Quad Tessellation Diagram]

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;
```
uint32_t patchControlPoints;
} VkPipelineTessellationStateCreateInfo;

- `sType` is a `VkStructureType` value identifying 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

// 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:

// Provided by VK_VERSION_1_1
typedef struct VkPipelineTessellationDomainOriginStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkTessellationDomainOrigin domainOrigin;
} VkPipelineTessellationDomainOriginStateCreateInfo;

- `sType` is a `VkStructureType` value identifying 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:

```c
// 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, OutputLineStrip, or OutputTriangleStrip 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 \texttt{VkPhysicalDeviceLimits::maxCullDistances} client-defined half-spaces (if no client-defined cull half-spaces are enabled, culling against the cull volume is skipped).

A shader \textbf{must} write a single cull distance for each enabled cull half-space to elements of the \texttt{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 \texttt{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 \textbf{must} write a single clip distance for each enabled clip half-space to elements of the \texttt{ClipDistance} array. Clip half-space \(i\) is then given by the set of points satisfying the inequality

\[
\mathbf{c}_i(\mathbf{P}) \geq 0
\]

where \(\mathbf{c}_i(\mathbf{P})\) is the clip distance \(i\) at point \(\mathbf{P}\). For point primitives, \(\mathbf{c}_i(\mathbf{P})\) is simply the clip distance for the vertex in question. For line and triangle primitives, per-vertex clip distances are interpolated using a weighted mean, with weights derived according to the algorithms described in sections \textit{Basic Line Segment Rasterization} and \textit{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 \texttt{ClipDistance} and \texttt{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 \texttt{depthClampEnable} enable of the \texttt{VkPipelineRasterizationStateCreateInfo} structure. Depth clipping is disabled when \texttt{depthClampEnable} is \texttt{VK_TRUE}.

When depth clipping is disabled, the plane equation

\[
z_m \leq z \leq w_c
\]

(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 \texttt{VkPhysicalDevicePointClippingProperties::pointClippingBehavior}, specifying clipping behavior of a point primitive whose vertex lies outside the clip volume, are:

// 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 \texttt{NoPerspective}, the value of \(t\) used to obtain the output value associated with \(\mathbf{P}\) 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 \texttt{Basic Line Segment Rasterization} and \texttt{Basic Polygon Rasterization}), and no interpolation is performed. The output value \(c\) is taken from either \(\mathbf{c}_1\) or \(\mathbf{c}_2\), since flat shading has already occurred and the two values are identical.

### 24.4. Coordinate Transformations

\textit{Clip coordinates} for a vertex result from shader execution, which yields a vertex coordinate \texttt{Position}.

Perspective division on clip coordinates yields \textit{normalized device coordinates}, followed by a \texttt{viewport} transformation (see \texttt{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

\[
\begin{align*}
    x_f &= (p_x / 2) x_d + o_x \\
    y_f &= (p_y / 2) y_d + o_y \\
    z_f &= p_z \times z_d + o_z
\end{align*}
\]

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;
```

- **sType** is a $\text{VkStructureType}$ value identifying this structure.
- **pNext** is $\text{NULL}$ or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **viewportCount** is the number of viewports used by the pipeline.
- **pViewports** is a pointer to an array of $\text{VkViewport}$ structures, defining the viewport transforms. If the viewport state is dynamic, this member is ignored.
- **scissorCount** is the number of $\text{scissors}$ and must match the number of viewports.
- **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, \textit{viewportCount} must not be greater than 1

- VUID-VkPipelineViewportStateCreateInfo-scissorCount-01217
  If the multiViewport feature is not enabled, \textit{scissorCount} must not be greater than 1

- VUID-VkPipelineViewportStateCreateInfo-viewportCount-01218
  \textit{viewportCount} must be less than or equal to \textit{VkPhysicalDeviceLimits::maxViewports}

- VUID-VkPipelineViewportStateCreateInfo-scissorCount-01219
  \textit{scissorCount} must be less than or equal to \textit{VkPhysicalDeviceLimits::maxViewports}

- VUID-VkPipelineViewportStateCreateInfo-x-02821
  The \textit{x} and \textit{y} members of \textit{offset} member of any element of \textit{pScissors} must be greater than or equal to 0

- VUID-VkPipelineViewportStateCreateInfo-offset-02822
  Evaluation of \((\text{offset.x} + \text{extent.width})\) must not cause a signed integer addition overflow for any element of \textit{pScissors}

- VUID-VkPipelineViewportStateCreateInfo-offset-02823
  Evaluation of \((\text{offset.y} + \text{extent.height})\) must not cause a signed integer addition overflow for any element of \textit{pScissors}

- VUID-VkPipelineViewportStateCreateInfo-scissorCount-04134
  If \textit{scissorCount} and \textit{viewportCount} are both not dynamic, then \textit{scissorCount} and \textit{viewportCount} must be identical

- VUID-VkPipelineViewportStateCreateInfo-viewportCount-04135
  \textit{viewportCount} must be greater than 0

- VUID-VkPipelineViewportStateCreateInfo-scissorCount-04136
  \textit{scissorCount} must be greater than 0

Valid Usage (Implicit)

- VUID-VkPipelineViewportStateCreateInfo-sType-sType
  \textit{sType} must be \textit{VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO}

- VUID-VkPipelineViewportStateCreateInfo-pNext-pNext
  \textit{pNext} must be NULL

- VUID-VkPipelineViewportStateCreateInfo-flags-zerobitmask
  \textit{flags} must be 0

// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineViewportStateCreateFlags;

\textbf{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,  // Provided by VK_VERSION_1_0
    uint32_t firstViewport,         // Provided by VK_VERSION_1_0
    uint32_t viewportCount,         // Provided by VK_VERSION_1_0
    const VkViewport* pViewports);  // Provided by VK_VERSION_1_0
```

- `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
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;
};
```


```c
float height;
float minDepth;
float maxDepth;
} VkViewport;
```

- \( x \) and \( y \) are the viewport’s upper left corner \((x,y)\).
- \( \text{width} \) and \( \text{height} \) are the viewport’s width and height, respectively.
- \( \text{minDepth} \) and \( \text{maxDepth} \) are the depth range for the viewport.

**Note**
Despite their names, \( \text{minDepth} \) can be less than, equal to, or greater than \( \text{maxDepth} \).

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 \( k \in \{0, 1, \ldots, 2^m-1\} \), 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

\[
\begin{align*}
o_x &= x + \text{width} / 2 \\
o_y &= y + \text{height} / 2 \\
o_z &= \text{minDepth} \\
p_x &= \text{width} \\
p_y &= \text{height} \\
p_z &= \text{maxDepth} - \text{minDepth}
\end{align*}
\]

The application can specify a negative term for \( \text{height} \), which has the effect of negating the \( y \) coordinate in clip space before performing the transform. When using a negative \( \text{height} \), 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 \( \text{height} \) allows the application to avoid having to negate the \( y \) component of the \text{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-01234**
  - `minDepth` **must** be between **0.0** and **1.0**, inclusive

- **VUID-VkViewport-maxDepth-01235**
  - `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;
    const void* pNext;
    VkPipelineRasterizationStateCreateFlags flags;
    VkBool32 depthClampEnable;
    VkBool32 rasterizerDiscardEnable;
    VkPolygonMode polygonMode;
    VkCullModeFlags cullMode;
    VkFrontFace frontFace;
    float lineWidth;
} VkPipelineRasterizationStateCreateInfo;
```

- `sType` is a `VkStructureType` value identifying this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
• **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 frustum as described in Primitive Clipping.

• **rasterizerDiscardEnable** controls whether primitives are discarded immediately before the rasterization stage.

• **polygonMode** is the triangle rendering mode. See VkPolygonMode.

• **cullMode** is the triangle facing direction used for primitive culling. See VkCullModeFlagBits.

• **frontFace** is 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-01507
  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 a VkStructureType value identifying 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 × w + b. pSampleMask has a length equal to ⌈rasterizationSamples / 32⌉ 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 \( \left\lceil \frac{\text{rasterizationSamples}}{32} \right\rceil \) `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.

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 24. Standard sample locations

<table>
<thead>
<tr>
<th>Sample count</th>
<th>Sample Locations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VK_SAMPLE_COUNT_1_BIT</strong></td>
<td>(0.5, 0.5)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>VK_SAMPLE_COUNT_2_BIT</strong></td>
<td>(0.75, 0.75) (0.25, 0.25)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>VK_SAMPLE_COUNT_4_BIT</strong></td>
<td>(0.375, 0.125) (0.875, 0.375) (0.125, 0.625) (0.625, 0.875)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>VK_SAMPLE_COUNT_8_BIT</strong></td>
<td>(0.5625, 0.3125) (0.4375, 0.6875) (0.8125, 0.5625) (0.3125, 0.1875) (0.1875, 0.8125) (0.0625, 0.4375) (0.6875, 0.9375) (0.9375, 0.0625)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>VK_SAMPLE_COUNT_16_BIT</strong></td>
<td>(0.5625, 0.5625) (0.4375, 0.3125) (0.3125, 0.625) (0.75, 0.4375) (0.1875, 0.375) (0.625, 0.8125) (0.8125, 0.6875) (0.6875, 0.1875) (0.375, 0.875) (0.5, 0.0625) (0.25, 0.125) (0.125, 0.75) (0.0, 0.5) (0.9375, 0.25) (0.875, 0.9375) (0.0625, 0.0)</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 \( \max(\lceil \frac{\text{VkPipelineMultisampleStateCreateInfo::minSampleShading}}{\text{VkPipelineMultisampleStateCreateInfo::rasterizationSamples}}, 1) \) times per fragment. If \text{VkPipelineMultisampleStateCreateInfo::sampleShadingEnable} is set to \text{VK_TRUE}, sample shading is enabled.

If a fragment shader entry point statically uses an input variable decorated with a \text{BuiltIn} of \text{SampleId} or \text{SamplePosition}, sample shading is enabled and a value of 1.0 is used instead of \text{minSampleShading}. If a fragment shader entry point statically uses an input variable decorated with \text{Sample}, sample shading may be enabled and a value of 1.0 will be used instead of \text{minSampleShading} if it is.

\[\text{Note}\]

If a shader decorates an input variable with \text{Sample} and that value meaningfully impacts the output of a shader, sample shading will be enabled to ensure that the input is in fact interpolated per-sample. This is inherent to the specification and not spelled out here - if an application simply declares such a variable it is implementation-defined whether sample shading is enabled or not. It is possible to see the effects of this by using atomics in the shader or using a pipeline statistics query to query the number of fragment invocations, even if the shader itself does not use any per-sample variables.

If there are fewer fragment 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, and each invocation that is not a helper invocation covers at least one sample.

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 \text{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 \text{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 \text{pointSizeRange} and \text{pointSizeGranularity} members of \text{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 `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 \(t\):

\[
\begin{align*}
    s &= \frac{1}{2} + \frac{(x_p - x_f)}{\text{size}} \\
    t &= \frac{1}{2} + \frac{(y_p - y_f)}{\text{size}}
\end{align*}
\]

where \(\text{size}\) is the point’s size; \((x_p,y_p)\) is the location at which the point sprite coordinates are evaluated - this may be the framebuffer coordinates of the fragment center, or the location of a sample; and \((x_o,y_o)\) is the exact, unrounded framebuffer coordinate of the vertex for the point.

### 25.6. Line Segments

To dynamically set the line width, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetLineWidth(
    VkCommandBuffer commandBuffer, 
    float lineWidth);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `lineWidth` is the width of rasterized line segments.

This command sets the line width for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_LINE_WIDTH` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineRasterizationStateCreateInfo::lineWidth` value used to create the currently active pipeline.

**Valid Usage**

- VUID-vkCmdSetLineWidth-lineWidth-00788
  If the wideLines feature is not enabled, `lineWidth` must be 1.0
Valid Usage (Implicit)

- VUID-vkCmdSetLineWidth-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdSetLineWidth-commandBuffer-recording
  `commandBuffer` must be in the recording state
- VUID-vkCmdSetLineWidth-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>

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 \( \text{strictLines} \) is \( \text{VK_TRUE} \), 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

\[
 f = \frac{(1 - t) f_a / w_a + t f_b / w_b}{(1 - t) / w_a + t / w_b}
\]

where \( f_a \) and \( f_b \) are the data associated with the starting and ending endpoints of the segment, respectively; \( w_a \) and \( w_b \) are the clip \( w \) coordinates of the starting and ending endpoints of the segment, respectively.

**Linear interpolation** for a line segment directly interpolates two values, and an interpolated value \( f \) can be determined by

\[
 f = (1 - t) f_a + t f_b
\]

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 \( \text{strictLines} \) limit in \( \text{VkPhysicalDeviceLimits} \) as \( \text{VK_TRUE} \).

When \( \text{strictLines} \) is \( \text{VK_FALSE} \), 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 \( \text{lineWidth} \) 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:

\[
R_f = \{(x, y) | |x - x_f| + |y - y_f| \leq \frac{1}{2}\}
\]

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 $(x_a, y_a) - (\epsilon, \epsilon^2)$ and $p'_b$ given by $(x_b, y_b) - (\epsilon, \epsilon^2)$. Rasterizing the line segment starting at $p_a$ and ending at $p_b$ produces those fragments $f$ for which the segment starting at $p'_a$ and ending on $p'_b$ intersects $R_f$, except if $p'_b$ is contained in $R_f$. $\epsilon$ 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 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 \( \text{lineWidthRange} \) (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).

\[\text{Note}\]

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).

### 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 back-facing or front-facing. This determination is made based on the sign of the (clipped or unclipped) polygon’s area computed in framebuffer coordinates. One way to compute this area is:
\[ a = -\frac{1}{2} \sum_{i=0}^{n-1} x_i^f y_i^{f+1} - x_i^{f+1} y_i^f \]

where \( x_i^f \) and \( y_i^f \) 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} \texttt{::frontFace} property of the currently active pipeline. Possible values are:

```cpp
// Provided by VK_VERSION_1_0
typedef enum VkFrontFace {
    VK_FRONT_FACE_COUNTER_CLOCKWISE = 0,
    VK_FRONT_FACE_CLOCKWISE = 1,
} VkFrontFace;
```

- \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.

Once the orientation of triangles is determined, they are culled according to the \texttt{VkPipelineRasterizationStateCreateInfo} \texttt{::cullMode} property of the currently active pipeline. Possible values are:

```cpp
// 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;
```

- \texttt{VK_CULL_MODE_NONE} specifies that no triangles are discarded
- \texttt{VK_CULL_MODE_FRONT_BIT} specifies that front-facing triangles are discarded
- \texttt{VK_CULL_MODE_BACK_BIT} specifies that back-facing triangles are discarded
- \texttt{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.

```cpp
// Provided by VK_VERSION_1_0
typedef VkFlags VkCullModeFlags;
```

\texttt{VkCullModeFlags} is a bitmask type for setting a mask of zero or more \texttt{VkCullModeFlagBits}. 
The rule for determining which fragments are produced by polygon rasterization is called point sampling. The two-dimensional projection obtained by taking the x and y framebuffer coordinates of the polygon’s vertices is formed. Fragments are produced for any fragment area groups of pixels for which any sample points lie inside of this polygon. Coverage bits that correspond to sample points that satisfy the point sampling criteria are 1, other coverage bits are 0. Special treatment is given to a sample whose sample location lies on a polygon edge. In such a case, if two polygons lie on either side of a common edge (with identical endpoints) on which a sample point lies, then exactly one of the polygons must result in a covered sample for that fragment during rasterization. As for the data associated with each fragment produced by rasterizing a polygon, we begin by specifying how these values are produced for fragments in a triangle.

**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:

\[
a = \frac{A(p_b p_c)}{A(p_a p_b p_c)}, \quad b = \frac{A(p_a p_c)}{A(p_a p_b p_c)}, \quad c = \frac{A(p_a p_b)}{A(p_a p_b p_c)},
\]

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.

**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

\[
f = \frac{af_a / w_a + bf_b / w_b + cf_c / w_c}{a / w_a + b / w_b + c / w_c}
\]

where \( w_a, w_b, \) and \( w_c \) are the clip w coordinates of \( p_a, p_b, \) and \( p_c \) respectively. \( a, b, \) and \( c \) are the barycentric coordinates of the location at which the data are produced.

**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 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 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 \( f \) 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:

```c
// 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.

The point size of the final rasterization of polygons when polygon mode is **VK_POLYGON_MODE_POINT** is implementation-dependent, and the point size **may** either be `PointSize` or 1.0.

### 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 \texttt{VkPipelineRasterizationStateCreateInfo::depthBiasEnable} value used to create the currently active pipeline. If the depth bias enable is \texttt{VK_FALSE}, no bias is applied and the fragment’s depth values are unchanged.

Depth Bias Computation

The depth bias depends on three parameters:

- \texttt{depthBiasSlopeFactor} scales the maximum depth slope \(m\) of the polygon
- \texttt{depthBiasConstantFactor} scales the parameter \(r\) 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 \texttt{depthBiasClamp}

\texttt{depthBiasSlopeFactor}, \texttt{depthBiasConstantFactor}, and \texttt{depthBiasClamp} can each be positive, negative, or zero. These parameters are set as described for \texttt{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

\[
m = \max\left(\left|\frac{\partial z_f}{\partial x_f}\right|, \left|\frac{\partial z_f}{\partial y_f}\right|\right).
\]

\(r\) is the minimum resolvable difference 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

\[
r = 2 \times 2^n
\]

where \(n\) is the number of bits used for the depth aspect.

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
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:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetDepthBias(
    VkCommandBuffer commandBuffer,
    float depthBiasConstantFactor,
    float depthBiasClamp,
    float depthBiasSlopeFactor);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `depthBiasConstantFactor` is a scalar factor controlling the constant depth value added to each fragment.
- `depthBiasClamp` is the maximum (or minimum) depth bias of a fragment.
- `depthBiasSlopeFactor` is a scalar factor applied to a fragment's slope in depth bias calculations.

This command sets the depth bias parameters for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_DEPTH_BIAS` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the corresponding `VkPipelineRasterizationStateCreateInfo::depthBiasConstantFactor`, `depthBiasClamp`, and `depthBiasSlopeFactor` values used to create the currently active pipeline.

**Valid Usage**

- VUID-vkCmdSetDepthBias-depthBiasClamp-00790
  If the `depthBiasClamp` feature is not enabled, `depthBiasClamp` must be 0.0
Valid Usage (Implicit)

- VUID-vkCmdSetDepthBias-commandBuffer-parameter
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetDepthBias-commandBuffer-recording
  
  `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 Secondary</td>
<td>Both</td>
<td>Graphics</td>
<td>State</td>
</tr>
</tbody>
</table>
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

• 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, and sample mask test may 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 DepthLess 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 DepthGreater 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 
\[
\text{if } x_f \in [\text{VkRect2D}::\text{offset}.x, \text{VkRect2D}::\text{offset}.x + \text{VkRect2D}::\text{extent}.x),
\]
and
\[
\text{if } y_f \in [\text{VkRect2D}::\text{offset}.y, \text{VkRect2D}::\text{offset}.y + \text{VkRect2D}::\text{extent}.y).
\]

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,
    uint32_t firstScissor,
    uint32_t scissorCount,
    const VkRect2D* pScissors);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `firstScissor` is the index of the first scissor whose state is updated by the command.
- `scissorCount` is the number of scissors whose rectangles are updated by the command.
- `pScissors` is a pointer to an array of VkRect2D structures defining scissor rectangles.

The scissor rectangles taken from element \(i\) of `pScissors` replace the current state for the scissor index \(firstScissor + i\), for \(i\) in \([0, \text{scissorCount})\).

This command sets the scissor rectangles for subsequent drawing commands when the graphics pipeline is created with VK_DYNAMIC_STATE_SCISSOR set in VkPipelineDynamicStateCreateInfo.
: DynamicStates. 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 synchronized
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 $z_f$.

### 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 `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 may be
different at different framebuffer coordinates.

Note
Using different algorithms at different framebuffer coordinates may help to avoid
artifacts caused by regular coverage sample locations.

Finally, if 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.

26.5. Depth and Stencil Operations

Pipeline state controlling the depth bounds tests, stencil test, and depth test is specified through the
members of the VkPipelineDepthStencilStateCreateInfo structure.

The VkPipelineDepthStencilStateCreateInfo structure is defined as:

```c
// 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;
```

- **sType** is a VkStructureType value identifying this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **depthTestEnable** controls whether depth testing is enabled.
- **depthWriteEnable** controls whether depth writes are enabled when depthTestEnable is VK_TRUE.
  Depth writes are always disabled when depthTestEnable is VK_FALSE.
- **depthCompareOp** is a VkCompareOp value specifying the comparison operator to use in the Depth
  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 \( z_a \) in the depth/stencil attachment at each sample’s framebuffer coordinates \((x_f, y_f)\) 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 VkPipelineDepthStencilStateCreateInfo structure during pipeline creation, or dynamically by vkCmdSetDepthBounds.

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 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-00600
  `minDepthBounds` must be between 0.0 and 1.0, inclusive

- VUID-vkCmdSetDepthBounds-maxDepthBounds-00601
  `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 `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 `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 `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:

$$s_a = (s_a \& \neg s_w) | (s_g \& s_w)$$

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_STENCILCOMPAREMASK 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,
} 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,  // command buffer into which the command will be recorded.
    VkStencilFaceFlags faceMask,    // bitmask of VkStencilFaceFlagBits specifying the set of stencil state for which to update the write mask, as described above for vkCmdSetStencilCompareMask.
    uint32_t writeMask);             // new value to use as the stencil write mask.
```

- **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
• VUID-vkCmdSetStencilReference-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>

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 $z_f$. 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, $z_t$ is clamped to $[z_{\min}, z_{\max}]$, where $z_{\min} = \min(n,f)$, $z_{\max} = \max(n,f)$, and $n$ and $f$ are the minDepth and maxDepth depth range values of the viewport used by this fragment, respectively.

Following depth clamping:

- If $z_t$ is not in the range $[z_{\min}, z_{\max}]$, then $z_t$ is undefined following this step.

26.8.2. Depth Comparison

If the depth test is not enabled, as specified by VkPipelineDepthStencilStateCreateInfo::depthTestEnable, then this step is skipped.

The comparison operation performed is determined by the VkCompareOp value set by
VkPipelineDepthStencilStateCreateInfo::depthCompareOp during pipeline creation. \( z_r \) 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 VkPipelineDepthStencilStateCreateInfo::depthWriteEnable, and the comparison evaluated to true, the depth attachment value \( z_a \) is set to the sample's depth value \( z_f \). If there is no depth attachment, no value is written.

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 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 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 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:

```c
// 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 a VkStructureType value identifying 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-07353**
  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

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

VkPipelineColorBlendStateCreateFlags is a bitmask type for setting a mask, but is currently reserved
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

- **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,
};
```
The semantics of the enum values are described in the table below:

<table>
<thead>
<tr>
<th>VkBlendFactor</th>
<th>RGB Blend Factors ((S_{r0}, S_{g0}, S_{b0})) or ((D_{r0}, D_{g0}, D_{b0}))</th>
<th>Alpha Blend Factor ((S_{a0}) or (D_{a0}))</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>(S_{a0})</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>(S_{a0})</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 = \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.
Rd,Gd,Bd and Ad 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.

Rc,Gc,Bc and Ac 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 Rc, Gc, Bc, and Ac 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.

Color outputs can be 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 value of the second input is undefined.

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:

```c
// 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 26. Basic Blend Operations**

<table>
<thead>
<tr>
<th>VkBlendOp</th>
<th>RGB Components</th>
<th>Alpha Component</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VK_BLEND_OP_ADD</strong></td>
<td>$R = R_{s0} \times S_r + R_d \times D_r$</td>
<td>$A = A_{s0} \times S_a + A_d \times D_a$</td>
</tr>
<tr>
<td></td>
<td>$G = G_{s0} \times S_g + G_d \times D_g$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$B = B_{s0} \times S_b + B_d \times D_b$</td>
<td></td>
</tr>
<tr>
<td><strong>VK_BLEND_OP_SUBTRACT</strong></td>
<td>$R = R_{s0} \times S_r - R_d \times D_r$</td>
<td>$A = A_{s0} \times S_a - A_d \times D_a$</td>
</tr>
<tr>
<td></td>
<td>$G = G_{s0} \times S_g - G_d \times D_g$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$B = B_{s0} \times S_b - B_d \times D_b$</td>
<td></td>
</tr>
<tr>
<td><strong>VK_BLEND_OP_REVERSE_SUBTRACT</strong></td>
<td>$R = R_d \times D_r - R_{s0} \times S_r$</td>
<td>$A = A_d \times D_a - A_{s0} \times S_a$</td>
</tr>
<tr>
<td></td>
<td>$G = G_d \times D_g - G_{s0} \times S_g$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$B = B_d \times D_b - B_{s0} \times S_b$</td>
<td></td>
</tr>
<tr>
<td><strong>VK_BLEND_OP_MIN</strong></td>
<td>$R = \min(R_{s0}, R_d)$</td>
<td>$A = \min(A_{s0}, A_d)$</td>
</tr>
<tr>
<td></td>
<td>$G = \min(G_{s0}, G_d)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$B = \min(B_{s0}, B_d)$</td>
<td></td>
</tr>
<tr>
<td><strong>VK_BLEND_OP_MAX</strong></td>
<td>$R = \max(R_{s0}, R_d)$</td>
<td>$A = \max(A_{s0}, A_d)$</td>
</tr>
<tr>
<td></td>
<td>$G = \max(G_{s0}, G_d)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$B = \max(B_{s0}, B_d)$</td>
<td></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.
- $R_d, G_d, B_d$ and $A_d$ represent the R, G, B, and A components of the destination color. That is, the color currently in the corresponding color attachment for this fragment/sample.
- $S_r, S_g, S_b$ and $S_a$ represent the source blend factor R, G, B, and A components, respectively.
- $D_r, D_g, D_b$ and $D_a$ represent the destination blend factor R, G, B, and A components, respectively.

The blending operation produces a new set of values R, G, B and A, which are written to the framebuffer attachment. If blending is not enabled for this attachment, then R, G, B and A are assigned $R_{s0}, G_{s0}, B_{s0}$ and $A_{s0}$, respectively.

If the color attachment is fixed-point, the components of the source and destination values and blend factors are each clamped to [0,1] or [-1,1] respectively for an unsigned normalized or signed normalized color attachment prior to evaluating the blend operations. If the color attachment is floating-point, no clamping occurs.

If the numeric format of a framebuffer attachment uses sRGB encoding, the R, G, and B destination color values (after conversion from fixed-point to floating-point) are considered to be encoded for the sRGB color space and hence are linearized prior to their use in blending. Each R, G, and B component is converted from nonlinear to linear as described in the “sRGB EOTF” section of the Khronos Data Format Specification. If the format is not sRGB, no linearization is performed.

If the numeric format of a framebuffer attachment uses sRGB encoding, then the final R, G and B values are converted into the nonlinear sRGB representation before being written to the framebuffer attachment as described in the “sRGB EOTF” section of the Khronos Data Format Specification.
If the numeric format of a framebuffer color attachment is not sRGB encoded then the resulting 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 \texttt{VK\_ATTACHMENT\_UNUSED}, no writes are performed through that attachment. Writes are not performed to framebuffer color attachments greater than or equal to the \texttt{VkSubpassDescription::colorAttachmentCount} or \texttt{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 \texttt{logicOpEnable} and \texttt{logicOp} members of \texttt{VkPipelineColorBlendStateCreateInfo}. If \texttt{logicOpEnable} is \texttt{VK\_TRUE}, then a logical operation selected by \texttt{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 \texttt{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 \lor 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 \lor d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_NAND</td>
<td>( \neg (s \lor 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:

```c
// 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
definition 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 \( \text{groupCountX} \times \text{groupCountY} \times \text{groupCountZ} \) 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-aspectMask-06478**
  If a `VkImageView` is sampled with depth comparison, the image view must have been created with an `aspectMask` that contains `VK_IMAGE_ASPECT_DEPTH_BIT`

- **VUID-vkCmdDispatch-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`
If a `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` descriptor is accessed using atomic operations as a result of this command, then the storage texel buffer's format features must contain `VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT`.

For each set \( n \) that is statically used by a bound shader, a descriptor set must have been bound to \( n \) at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set \( n \), with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility.

For each push constant that is statically used by a bound shader, 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 a bound shader.

A valid pipeline must be bound to the pipeline bind point used by this command.

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 any stage of the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, and the `robustBufferAccess` feature is not enabled, that stage 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 any stage of the VkPipeline object bound to the pipeline bind point used by this command accesses a storage buffer, and the robustBufferAccess feature is not enabled, that stage 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 bound shaders must not be a protected resource.

If a bound shader accesses a VkSampler or VkImageView object that enables sampler YC_bconversion, that object must only be used with OpImageSample* or OpImageSparseSample* instructions.

If a bound shader accesses a VkSampler or VkImageView object that enables sampler YC_bconversion, 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 numeric type of the image view's format and the Sampled Type operand of the OpTypeImage must match.

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.
• VUID-vkCmdDispatch-groupCountY-00387
groupCountY must be less than or equal to VkPhysicalDeviceLimits

• VUID-vkCmdDispatch-groupCountZ-00388
groupCountZ must be less than or equal to VkPhysicalDeviceLimits

Valid Usage (Implicit)

• VUID-vkCmdDispatch-commandBuffer-parameter
commandBuffer must be a valid VkCommandBuffer handle

• VUID-vkCmdDispatch-commandBuffer-recording
commandBuffer must be in the recording state

• VUID-vkCmdDispatch-commandBuffer-cmdpool
The VkCommandPool that commandBuffer was allocated from must support compute operations

• VUID-vkCmdDispatch-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>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);
```
- `commandBuffer` is the command buffer into which the command will be recorded.
- `buffer` is the buffer containing dispatch parameters.
- `offset` is the byte offset into `buffer` where parameters begin.

`vkCmdDispatchIndirect` behaves similarly to `vkCmdDispatch` except that the parameters are read by the device from a buffer during execution. The parameters of the dispatch are encoded in a `VkDispatchIndirectCommand` structure taken from `buffer` starting at `offset`.

### Valid Usage

- **VUID-vkCmdDispatchIndirect-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-vkCmdDispatchIndirect-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-vkCmdDispatchIndirect-aspectMask-06478**
  If a `VkImageView` is sampled with depth comparison, the image view must have been created with an `aspectMask` that contains `VK_IMAGE_ASPECT_DEPTH_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-None-07888**
  If a `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` descriptor is accessed using atomic operations as a result of this command, the storage texel buffer's format features must contain `VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT`.

- **VUID-vkCmdDispatchIndirect-None-08600**
  For each set `n` that is statically used by a bound shader, 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-None-08601**
  For each push constant that is statically used by a bound shader, 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-08114**
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid as described by descriptor validity if they are statically used by a bound shader.
A valid pipeline **must** be bound to the pipeline bind point used by this command

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 any stage of the * VkPipeline* object bound to the pipeline bind point used by this command accesses a uniform buffer, and the *robustBufferAccess* feature is not enabled, that stage **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 any stage of the * VkPipeline* object bound to the pipeline bind point used by this command accesses a storage buffer, and the *robustBufferAccess* feature is not enabled, that stage **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 bound shaders **must** not be a protected resource

If a bound shader accesses a * VkSampler* or * VkImageView* object that enables sampler Y’CₐC_b conversion, that object **must** only be used with *OpImageSample* or *OpImageSparseSample* instructions

If a bound shader accesses a * VkSampler* or * VkImageView* object that enables sampler Y’CₐC_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 Dim operand of the OpTypeImage as described in Instruction/Sampler/Image View Validation

- VUID-vkCmdDispatchIndirect-format-07753
  If a VkImageView is accessed as a result of this command, then the numeric type of the image view’s format and the Sampled Type operand of the OpTypeImage must match

- VUID-vkCmdDispatchIndirect-OpImageWrite-08795
  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-vkCmdDispatchIndirect-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-vkCmdDispatchIndirect-None-07288
  Any shader invocation executed by this command must terminate

- VUID-vkCmdDispatchIndirect-buffer-02708
  If buffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

- VUID-vkCmdDispatchIndirect-buffer-02709
  buffer must have been created with the VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT bit set

- VUID-vkCmdDispatchIndirect-offset-02710
  offset must be a multiple of 4

- VUID-vkCmdDispatchIndirect-commandBuffer-02711
  commandBuffer must not be a protected command buffer

- VUID-vkCmdDispatchIndirect-offset-00407
  The sum of offset and the size of VkDispatchIndirectCommand must be less than or equal to the size of buffer

Valid Usage (Implicit)

- VUID-vkCmdDispatchIndirect-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdDispatchIndirect-buffer-parameter
  buffer must be a valid VkBuffer handle

- VUID-vkCmdDispatchIndirect-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdDispatchIndirect-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support compute operations

- VUID-vkCmdDispatchIndirect-renderpass
  This command must only be called outside of a render pass instance
Both of buffer, and commandBuffer must have been created, allocated, or retrieved from the same VkDevice

**Host Synchronization**

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

**Command Properties**

<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]
- VUID-VkDispatchIndirectCommand-z-00419  
  z must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2]
To record a dispatch using non-zero base values for the components of WorkgroupId, call:

```cpp
// 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 `groupCountX × groupCountY × groupCountZ` local workgroups is assembled, with WorkgroupId values ranging from `[baseGroup*, baseGroup* + groupCount*)` in each component. `vkCmdDispatch` is equivalent to `vkCmdDispatchBase(0, 0, 0, groupCountX, groupCountY, groupCountZ)`.

### Valid Usage

- **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](https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/vkGetSamplerProperties.html#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](https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/vkGetSamplerProperties.html#VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT).

- **VUID-vkCmdDispatchBase-aspectMask-06478**
  If a `VkImageView` is sampled with depth comparison, the image view must have been created with an aspectMask that contains [VK_IMAGE_ASPECT_DEPTH_BIT](https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/vkGetImageSubresourceLayout.html#VK_IMAGE_ASPECT_DEPTH_BIT).

- **VUID-vkCmdDispatchBase-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](https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/vkGetImageSubresourceLayout.html#VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT).
• VUID-vkCmdDispatchBase-None-07888
If a `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` descriptor is accessed using atomic operations as a result of this command, then the storage texel buffer’s format features must contain `VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT`.

• VUID-vkCmdDispatchBase-None-08600
For each set `n` that is statically used by a bound shader, 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-vkCmdDispatchBase-None-08601
For each push constant that is statically used by a bound shader, 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-vkCmdDispatchBase-None-08114
Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid as described by descriptor validity if they are statically used by a bound shader.

• VUID-vkCmdDispatchBase-None-08606
A valid pipeline must be bound to the pipeline bind point used by this command.

• VUID-vkCmdDispatchBase-None-08608
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-08609
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-08610
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-08611
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-uniformBuffers-06935
If any stage of the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, and the `robustBufferAccess` feature is not enabled, that stage 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 any stage of the VkPipeline object bound to the pipeline bind point used by this command accesses a storage buffer, and the robustBufferAccess feature is not enabled, that stage 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 bound shaders must not be a protected resource.

If a bound shader accesses a VkSampler or VkImageView object that enables sampler YC_aC_b conversion, that object must only be used with OpImageSample* or OpImageSparseSample* instructions.

If a bound shader accesses a VkSampler or VkImageView object that enables sampler YC_aC_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 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 numeric type of the image view’s format and the Sampled Type operand of the OpTypeImage must match.

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.

baseGroupX must be less than VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0]
- **VUID-vkCmdDispatchBase-baseGroupX-00422**
  \(\text{baseGroupY must be less than } \text{VkPhysicalDeviceLimits::maxComputeWorkGroupCount}[1]\)

- **VUID-vkCmdDispatchBase-baseGroupZ-00423**
  \(\text{baseGroupZ must be less than } \text{VkPhysicalDeviceLimits::maxComputeWorkGroupCount}[2]\)

- **VUID-vkCmdDispatchBase-groupCountX-00424**
  \(\text{groupCountX must be less than or equal to } \text{VkPhysicalDeviceLimits::maxComputeWorkGroupCount}[0] \text{ minus baseGroupX}\)

- **VUID-vkCmdDispatchBase-groupCountY-00425**
  \(\text{groupCountY must be less than or equal to } \text{VkPhysicalDeviceLimits::maxComputeWorkGroupCount}[1] \text{ minus baseGroupY}\)

- **VUID-vkCmdDispatchBase-groupCountZ-00426**
  \(\text{groupCountZ must be less than or equal to } \text{VkPhysicalDeviceLimits::maxComputeWorkGroupCount}[2] \text{ minus baseGroupZ}\)

- **VUID-vkCmdDispatchBase-baseGroupX-00427**
  If any of \(\text{baseGroupX, baseGroupY, or baseGroupZ}\) are not zero, then the bound compute pipeline \(\text{must}\) have been created with the \textit{VK_PIPELINE_CREATE_DISPATCH_BASE} flag

---

**Valid Usage (Implicit)**

- **VUID-vkCmdDispatchBase-commandBuffer-parameter**
  \(\text{commandBuffer must be a valid } \text{VkCommandBuffer handle}\)

- **VUID-vkCmdDispatchBase-commandBuffer-recording**
  \(\text{commandBuffer must be in the recording state}\)

- **VUID-vkCmdDispatchBase-commandBuffer-cmdpool**
  The \text{VkCommandPool} that \text{commandBuffer} was allocated from \(\text{must}\) support compute operations

- **VUID-vkCmdDispatchBase-renderpass**
  This command \(\text{must}\) only be called outside of a render pass instance

---

**Host Synchronization**

- Host access to \text{commandBuffer} \(\text{must}\) be externally synchronized

- Host access to the \text{VkCommandPool} that \text{commandBuffer} was allocated from \(\text{must}\) 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>
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.

Sparse resources are not supported in Vulkan SC, due to complexity and the necessity of being able to update page table mappings at runtime [SCID-8]. However, the sparse resource features, properties, resource creation flags, and definitions have been retained for completeness and compatibility.

All sparse resource physical device features must not be advertised as supported, and the related physical device sparse properties and physical device limits must be reported accordingly.

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 \( \text{VkPhysicalDeviceSparseProperties}::\text{residencyNonResidentStrict} \). 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 \( \text{VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT} \) and \( \text{VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT} \) bits.
- Sparse residency support is advertised on a finer grain via the following features:
  - The \( \text{sparseResidencyBuffer} \) feature provides support for creating \( \text{VkBuffer} \) objects with the \( \text{VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT} \).
  - The \( \text{sparseResidencyImage2D} \) feature provides support for creating 2D single-sampled \( \text{VkImage} \) objects with \( \text{VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT} \).
  - The \( \text{sparseResidencyImage3D} \) feature provides support for creating 3D \( \text{VkImage} \) objects with \( \text{VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT} \).
  - The \( \text{sparseResidency2Samples} \) feature provides support for creating 2D \( \text{VkImage} \) objects with 2 samples and \( \text{VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT} \).
  - The \( \text{sparseResidency4Samples} \) feature provides support for creating 2D \( \text{VkImage} \) objects with 4 samples and \( \text{VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT} \).
  - The \( \text{sparseResidency8Samples} \) feature provides support for creating 2D \( \text{VkImage} \) objects with 8 samples and \( \text{VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT} \).
  - The \( \text{sparseResidency16Samples} \) feature provides support for creating 2D \( \text{VkImage} \) objects with 16 samples and \( \text{VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT} \).

Implementations supporting \( \text{sparseResidencyImage2D} \) are only \textbf{required} to support sparse 2D, single-sampled images. Support for sparse 3D and MSAA images is \textbf{optional} and \textbf{can} be enabled via \( \text{sparseResidencyImage3D} \), \( \text{sparseResidency2Samples} \), \( \text{sparseResidency4Samples} \), \( \text{sparseResidency8Samples} \), and \( \text{sparseResidency16Samples} \).

- A sparse image created using \( \text{VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT} \) supports all non-compressed color formats with power-of-two element size that non-sparse usage supports. Additional formats \textbf{may} also be supported and \textbf{can} be queried via \( \text{vkGetPhysicalDeviceSparseImageFormatProperties} \). \( \text{VK_IMAGE_TILING_LINEAR} \) tiling is not supported.

- The \( \text{sparseResidencyAliased} \) feature provides the following capability that \textbf{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.
29.2. Sparse Resource API

The APIs related to sparse resources are grouped into the following categories:

- Physical Device Features
- Physical Device Sparse Properties

29.2.1. 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.

29.2.2. 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:

```
// Provided by VK_VERSION_1_0
```
typedef struct VkPhysicalDeviceSparseProperties {
    VkBool32 residencyStandard2DBlockShape;
    VkBool32 residencyStandard2DMultisampleBlockShape;
    VkBool32 residencyStandard3DBlockShape;
    VkBool32 residencyAlignedMipSize;
    VkBool32 residencyNonResidentStrict;
} VkPhysicalDeviceSparseProperties;

• residencyStandard2DBlockShape must be VK_FALSE in Vulkan SC [SCID-8].
• residencyStandard2DMultisampleBlockShape must be VK_FALSE in Vulkan SC [SCID-8].
• residencyStandard3DBlockShape must be VK_FALSE in Vulkan SC [SCID-8].
• residencyAlignedMipSize must be VK_FALSE in Vulkan SC [SCID-8].
• residencyNonResidentStrict must be VK_FALSE in Vulkan SC [SCID-8].
Chapter 30. 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.

30.1. 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 dispatch from a `VkDevice` object or a child object of a `VkDevice`, or take any of them as a parameter, are considered device-level functionality. Types defined by a device extension are also considered device-level functionality.

Commands that dispatch from `VkPhysicalDevice`, or accept a `VkPhysicalDevice` object as a parameter, are considered either instance-level or device-level functionality depending if the functionality is specified by an instance extension or device extension respectively.

Additionally, commands that enumerate physical device properties 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.

30.2. 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.

30.2.1. 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 1 for the Vulkan SC API. The Base Vulkan API is variant 0.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 Base Vulkan Specification.</td>
</tr>
</tbody>
</table>

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) >> 29U)
```

`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) >> 22U) & 0x7FU)

**VK_API_VERSION_MINOR** extracts the API minor version number from a packed version number:

```c
// Provided by VK_VERSION_1_0
#define VK_API_VERSION_MINOR(version) (((uint32_t)(version) >> 12U) & 0x3FFU)
```

**VK_API_VERSION_PATCH** extracts the API patch version number from a packed version number:

```c
// Provided by VK_VERSION_1_0
#define VK_API_VERSION_PATCH(version) ((uint32_t)(version) & 0xFFFU)
```

**VK_MAKE_API_VERSION** constructs an API version number.

```c
// Provided by VK_VERSION_1_0
#define VK_MAKE_API_VERSION(variant, major, minor, patch) \
    (((uint32_t)(variant)) << 29U) | (((uint32_t)(major)) << 22U) | \
    (((uint32_t)(minor)) << 12U) | ((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_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
```
30.2.2. 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.

30.3. 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:
typedef struct VkLayerProperties {
    char    layerName[VK_MAX_EXTENSION_NAME_SIZE];
    uint32_t specVersion;
    uint32_t implementationVersion;
    char    description[VK_MAX_DESCRIPTION_SIZE];
} VkLayerProperties;

• **layerName** is an array of **VK_MAX_EXTENSION_NAME_SIZE** char containing a null-terminated UTF-8 string which is the name of the layer. Use this name in the **ppEnabledLayerNames** array passed in the **VkInstanceCreateInfo** structure to enable this layer for an instance.

• **specVersion** is the Vulkan version the layer was written to, encoded as described in **Version Numbers**.

• **implementationVersion** is the version of this layer. It is an integer, increasing with backward compatible changes.

• **description** is an array of **VK_MAX_DESCRIPTION_SIZE** char containing a null-terminated UTF-8 string which provides additional details that can be used by the application to identify the layer.

**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.

```
#define VK_MAX_EXTENSION_NAME_SIZE        256U
```

**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.

```
#define VK_MAX_DESCRIPTION_SIZE           256U
```

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.
30.3.1. 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.

To enumerate device layers, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkEnumerateDeviceLayerProperties(
    VkPhysicalDevice physicalDevice,
    uint32_t* pPropertyCount,
    VkLayerProperties* pProperties);
```

- `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.

Physical device layers are not supported. `pPropertyCount` is set to 0 and `VK_SUCCESS` is returned.

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`

The `ppEnabledLayerNames` and `enabledLayerCount` members of `VkDeviceCreateInfo` are deprecated
and their values **must** be ignored by implementations.

The sequence of layers active for a device will be exactly the sequence of layers enabled when the parent instance was created.

### 30.4. 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_sc_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.

#### 30.4.1. Instance Extensions

Instance extensions add new **instance-level functionality** to the API, outside of the core specification.

To query the available instance extensions, call:
VkResult vkEnumerateInstanceExtensionProperties(
    const char* pLayerName,
    uint32_t* pPropertyCount,
    VkExtensionProperties* pProperties);

• **pLayerName** is either **NULL** or a pointer to a null-terminated UTF-8 string naming the layer to retrieve extensions from.

• **pPropertyCount** is a pointer to an integer related to the number of extension properties available or queried, as described below.

• **pProperties** is either **NULL** or a pointer to an array of **VkExtensionProperties** structures.

When **pLayerName** parameter is **NULL**, only extensions provided by the Vulkan implementation or by implicitly enabled layers are returned. When **pLayerName** is the name of a layer, the instance extensions provided by that layer are returned.

If **pProperties** is **NULL**, then the number of extensions properties available is returned in **pPropertyCount**. Otherwise, **pPropertyCount** must point to a variable set by the user to the number of elements in the **pProperties** array, and on return the variable is overwritten with the number of structures actually written to **pProperties**. If **pPropertyCount** is less than the number of extension properties available, at most **pPropertyCount** structures will be written, 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
### 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

### 30.4.2. 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,  // physicalDevice is the physical device that will be queried.
    const char* pLayerName,          // pLayerName is either NULL or a pointer to a null-terminated UTF-8 string naming the layer to retrieve extensions from.
    uint32_t* pPropertyCount,        // 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.
    VkExtensionProperties* pProperties) // 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.

---

**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.
If \( \text{VkPhysicalDeviceVulkanSC10Properties}::\text{deviceNoDynamicHostAllocations} \) is \( \text{VK_TRUE} \), \( \text{vkEnumerateDeviceExtensionProperties} \) must not return \( \text{VK_ERROR_OUT_OF_HOST_MEMORY} \).

**Valid Usage (Implicit)**

- VUID-vkEnumerateDeviceExtensionProperties-physicalDevice-parameter
  - \( \text{physicalDevice} \) must be a valid \( \text{VkPhysicalDevice} \) handle
- VUID-vkEnumerateDeviceExtensionProperties-pLayerName-parameter
  - If \( \text{pLayerName} \) is not \( \text{NULL} \), \( \text{pLayerName} \) must be a null-terminated UTF-8 string
- VUID-vkEnumerateDeviceExtensionProperties-pPropertyCount-parameter
  - \( \text{pPropertyCount} \) must be a valid pointer to a \( \text{uint32_t} \) value
- VUID-vkEnumerateDeviceExtensionProperties-pProperties-parameter
  - If the value referenced by \( \text{pPropertyCount} \) is not \( 0 \), and \( \text{pProperties} \) is not \( \text{NULL} \), \( \text{pProperties} \) must be a valid pointer to an array of \( \text{pPropertyCount} \) \( \text{VkExtensionProperties} \) structures

**Return Codes**

**Success**

- \( \text{VK_SUCCESS} \)
- \( \text{VK_INCOMPLETE} \)

**Failure**

- \( \text{VK_ERROR_OUT_OF_HOST_MEMORY} \)
- \( \text{VK_ERROR_OUT_OF_DEVICE_MEMORY} \)
- \( \text{VK_ERROR_LAYER_NOT_PRESENT} \)

The \( \text{VkExtensionProperties} \) 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;
```

- \( \text{extensionName} \) is an array of \( \text{VK_MAX_EXTENSION_NAME_SIZE} \) \( \text{char} \) containing a null-terminated UTF-8 string which is the name of the extension.
- \( \text{specVersion} \) is the version of this extension. It is an integer, incremented with backward compatible changes.

**Accessing Device-Level Functionality From a \text{VkPhysicalDevice}**

Some device extensions also add support for physical-device-level functionality. Physical-device-level functionality can be used, if the required extension is supported as advertised by
vkEnumerateDeviceExtensionProperties for a given VkPhysicalDevice.

**Accessing Device-Level Functionality From a VkDevice**

For commands that are dispatched from a VkDevice, or from a child object of a VkDevice, device extensions must be enabled in vkCreateDevice.

### 30.5. 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. As discussed in more detail for Promotion below, when an extension is promoted it does not mean that a mechanical substitution of an extension API by the corresponding promoted API will work in exactly the same fashion; be supported at runtime; or even exist.

### 30.6. 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.
30.6.1. 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.

### 30.6.2. 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 parameter 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 28. 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 applications.</td>
</tr>
<tr>
<td>D3D support</td>
<td>d3demulation</td>
<td>Extension is intended to support D3D emulation layers, and applications ported from D3D, by adding functionality specific to 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 applications 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 applications 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 31. 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 29. 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 a `VkStructureType` value identifying 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;
};
```
VkBool32 samplerAnisotropy;
VkBool32 textureCompressionETC2;
VkBool32 textureCompressionASTC_LDR;
VkBool32 textureCompressionBC;
VkBool32 occlusionQueryPrecise;
VkBool32 pipelineStatisticsQuery;
VkBool32 vertexPipelineStoresAndAtomics;
VkBool32 fragmentStoresAndAtomics;
VkBool32 shaderTessellationAndGeometryPointSize;
VkBool32 shaderImageGatherExtended;
VkBool32 shaderStorageImageExtendedFormats;
VkBool32 shaderStorageImageMultisample;
VkBool32 shaderStorageImageReadWithoutFormat;
VkBool32 shaderStorageImageWriteWithoutFormat;
VkBool32 shaderUniformBufferArrayDynamicIndexing;
VkBool32 shaderSampledImageArrayDynamicIndexing;
VkBool32 shaderStorageBufferArrayDynamicIndexing;
VkBool32 shaderStorageImageArrayDynamicIndexing;
VkBool32 shaderClipDistance;
VkBool32 shaderCullDistance;
VkBool32 shaderFloat64;
VkBool32 shaderInt64;
VkBool32 shaderInt16;
VkBool32 shaderResourceResidency;
VkBool32 shaderResourceMinLod;
VkBool32 sparseBinding;
VkBool32 sparseResidencyBuffer;
VkBool32 sparseResidencyImage2D;
VkBool32 sparseResidencyImage3D;
VkBool32 sparseResidency2Samples;
VkBool32 sparseResidency4Samples;
VkBool32 sparseResidency8Samples;
VkBool32 sparseResidency16Samples;
VkBool32 sparseResidencyAliased;
VkBool32 variableMultisampleRate;
VkBool32 inheritedQueries;
} VkPhysicalDeviceFeatures;

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:

- Values from anywhere within the memory range(s) bound to the buffer, converted according to the format of the attribute.
- Zero values, format converted according to the format of the attribute.
- Zero values, or (0,0,0,x) vectors, as described above.

  If robustBufferAccess is not enabled, applications must not perform out of bounds accesses.

- fullDrawIndexUint32 specifies the full 32-bit range of indices is supported for indexed draw calls when using a VkIndexType of VK_INDEX_TYPE_UINT32. maxDrawIndexedIndexValue is the maximum index value that may be used (aside from the primitive restart index, which is always 2^{32}-1 when the VkIndexType is VK_INDEX_TYPE_UINT32). If this feature is supported, maxDrawIndexedIndexValue must be 2^{32}-1; otherwise it must be no smaller than 2^{24}-1. See maxDrawIndexedIndexValue.

- imageCubeArray specifies whether image views with a VkImageViewType of VK_IMAGE_VIEW_TYPE_CUBE_ARRAY can be created, and that the corresponding SampledCubeArray and ImageCubeArray SPIR-V capabilities can be used in shader code.

- independentBlend specifies whether the VkPipelineColorBlendAttachmentState settings are controlled independently per-attachment. If this feature is not enabled, the VkPipelineColorBlendAttachmentState settings for all color attachments must be identical. Otherwise, a different VkPipelineColorBlendAttachmentState can be provided for each bound color attachment.

- geometryShader specifies whether geometry shaders are supported. If this feature is not enabled, the VK_SHADER_STAGE_GEOMETRY_BIT and VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT enum values must not be used. This also specifies whether shader modules can declare the Geometry capability.

- tessellationShader specifies whether tessellation control and evaluation shaders are supported. If this feature is not enabled, the VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT, VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT, VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT, VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT, and VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO enum values must not be used. This also specifies whether shader modules can declare the Tessellation capability.

- sampleRateShading specifies whether Sample Shading and multisample interpolation are supported. If this feature is not enabled, the sampleShadingEnable member of the VkPipelineMultisampleStateCreateInfo structure must be set to VK_FALSE and the minSampleShading member is ignored. This also specifies whether shader modules can declare the SampleRateShading capability.

- dualSrcBlend specifies whether blend operations which take two sources are supported. If this feature is not enabled, the VK_BLEND_FACTOR_SRC1_COLOR, VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR, VK_BLEND_FACTOR_SRC1_ALPHA, and VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA enum values must not be used as source or destination blending factors. See Dual-Source Blending.

- logicOp specifies whether logic operations are supported. If this feature is not enabled, the logicOpEnable member of the VkPipelineColorBlendStateCreateInfo structure must be set to VK_FALSE, and the logicOp member is ignored.

- multiDrawIndirect specifies whether multiple draw indirect is supported. If this feature is not
enabled, the drawCount parameter to the vkCmdDrawIndirect and vkCmdDrawIndexedIndirect commands must be 0 or 1. The maxDrawIndirectCount member of the VkPhysicalDeviceLimits structure must also be 1 if this feature is not supported. See maxDrawIndirectCount.

- drawIndirectFirstInstance specifies whether indirect 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.

- lineWidth 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_10x5_UNORM_BLOCK
VK_FORMAT_ASTC_10x5_SRGB_BLOCK
VK_FORMAT_ASTC_10x6_UNORM_BLOCK
VK_FORMAT_ASTC_10x6_SRGB_BLOCK
VK_FORMAT_ASTC_10x8_UNORM_BLOCK
VK_FORMAT_ASTC_10x8_SRGB_BLOCK
VK_FORMAT_ASTC_10x10_UNORM_BLOCK
VK_FORMAT_ASTC_10x10_SRGB_BLOCK
VK_FORMAT_ASTC_12x10_UNORM_BLOCK
VK_FORMAT_ASTC_12x10_SRGB_BLOCK
VK_FORMAT_ASTC_12x12_UNORM_BLOCK
VK_FORMAT_ASTC_12x12_SRGB_BLOCK

To query for additional properties, or if the feature is not enabled, `vkGetPhysicalDeviceFormatProperties` and `vkGetPhysicalDeviceImageFormatProperties` can be used to check for supported properties of individual formats as normal.

- `textureCompressionBC` specifies whether all of the BC compressed texture formats are supported. If this feature is enabled, then the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `VK_FORMAT_FEATURE_BLIT_SRC_BIT` and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` features must be supported in `optimalTilingFeatures` for the following formats:

  - VK_FORMAT_BC1_RGB_UNORM_BLOCK
  - VK_FORMAT_BC1_RGB_SRGB_BLOCK
  - VK_FORMAT_BC1_RGBA_UNORM_BLOCK
  - VK_FORMAT_BC1_RGBA_SRGB_BLOCK
  - VK_FORMAT_BC2_UNORM_BLOCK
  - VK_FORMAT_BC2_SRGB_BLOCK
  - VK_FORMAT_BC3_UNORM_BLOCK
  - VK_FORMAT_BC3_SRGB_BLOCK
  - VK_FORMAT_BC4_UNORM_BLOCK
  - VK_FORMAT_BC4_SNORM_BLOCK
  - VK_FORMAT_BC5_UNORM_BLOCK
  - VK_FORMAT_BC5_SNORM_BLOCK
VK_FORMAT_BC6H_UFLOAT_BLOCK
VK_FORMAT_BC6H_SFLOAT_BLOCK
VK_FORMAT_BC7_UNORM_BLOCK
VK_FORMAT_BC7_SRGB_BLOCK

To query for additional properties, or if the feature is not enabled, `vkGetPhysicalDeviceFormatProperties` and `vkGetPhysicalDeviceImageFormatProperties` can be used to check for supported properties of individual formats as normal.

- **occlusionQueryPrecise** specifies whether occlusion queries returning actual sample counts are supported. Occlusion queries are created in a `VkQueryPool` by specifying the `queryType` of `VK_QUERY_TYPE_OCCLUSION` in the `VkQueryPoolCreateInfo` structure which is passed to `vkCreateQueryPool`. If this feature is enabled, queries of this type can enable `VK_QUERY_CONTROL_PRECISE_BIT` in the `flags` parameter to `vkCmdBeginQuery`. If this feature is not supported, the implementation supports only boolean occlusion queries. When any samples are passed, boolean queries will return a non-zero result value, otherwise a result value of zero is returned. When this feature is enabled and `VK_QUERY_CONTROL_PRECISE_BIT` is set, occlusion queries will report the actual number of samples passed.

- **pipelineStatisticsQuery** specifies whether the pipeline statistics queries are supported. If this feature is not enabled, queries of type `VK_QUERY_TYPE_PIPELINE_STATISTICS` cannot be created, and none of the `VkQueryPipelineStatisticFlagBits` bits can be set in the `pipelineStatistics` member of the `VkQueryPoolCreateInfo` structure.

- **vertexPipelineStoresAndAtomics** specifies whether storage buffers and images support stores and atomic operations in the vertex, tessellation, and geometry shader stages. If this feature is not enabled, all storage image, storage texel 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_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 and storage texel buffers require a format qualifier to be specified when reading.

• **shaderStorageImageWriteWithoutFormat** specifies whether storage images and storage texel buffers require a format qualifier to be specified when writing.

• **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. This **must** be VK_FALSE in Vulkan SC [SCID-8].

- **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. This **must** be VK_FALSE in Vulkan SC [SCID-8].

- **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. This **must** be VK_FALSE in Vulkan SC [SCID-8].

- **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. This **must** be VK_FALSE in Vulkan SC [SCID-8].

- **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. This **must** be VK_FALSE in Vulkan SC [SCID-8].

- **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. This must be VK_FALSE in Vulkan SC [SCID-8].

- 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. This must be VK_FALSE in Vulkan SC [SCID-8].

- 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. This must be VK_FALSE in Vulkan SC [SCID-8].

- 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. This must be VK_FALSE in Vulkan SC [SCID-8].

- 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. This must be VK_FALSE in Vulkan SC [SCID-8].

- 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
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 a `VkStructureType` value identifying 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'CbCr conversion. If samplerYcbcrConversion is VK_FALSE, sampler Y'CbCr conversion is not supported, and samplers using sampler Y'CbCr conversion must not be used.

• 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 a `VkStructureType` value identifying 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.

• **separateDepth_STENCILLayouts** 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 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 `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 a `VkStructureType` value identifying 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
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 a `VkStructureType` value identifying 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 a `VkStructureType` value identifying 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 a `VkStructureType` value identifying 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 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 a `VkStructureType` value identifying 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:

```
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceSamplerYcbcrConversionFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 samplerYcbcrConversion;
} VkPhysicalDeviceSamplerYcbcrConversionFeatures;
```

This structure describes the following feature:

- **sType** is a `VkStructureType` value identifying this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **samplerYcbcrConversion** specifies whether the implementation supports sampler Y’CbCr conversion. If `samplerYcbcrConversion` is `VK_FALSE`, sampler Y’CbCr conversion is not supported, and samplers using sampler Y’CbCr conversion must not be used.

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 a `VkStructureType` value identifying 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 a `VkStructureType` value identifying 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 shaderStorageBufferArrayNonUniformIndexing;
  VkBool32 shaderStorageImageArrayNonUniformIndexing;
  VkBool32 shaderInputAttachmentArrayNonUniformIndexing;
  VkBool32 shaderUniformTexelBufferArrayNonUniformIndexing;
  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 a **VkStructureType** value identifying 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**
• **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 buffer.
buffers can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of \texttt{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 \texttt{UniformTexelBufferArrayNonUniformIndexing} capability.

- \texttt{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 \texttt{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 \texttt{StorageTexelBufferArrayNonUniformIndexing} capability.

- \texttt{descriptorBindingUniformBufferUpdateAfterBind} indicates whether the implementation supports updating uniform buffer descriptors after a set is bound. If this feature is not enabled, \texttt{VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT} must not be used with \texttt{VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER}.

- \texttt{descriptorBindingSampledImageUpdateAfterBind} indicates whether the implementation supports updating sampled image descriptors after a set is bound. If this feature is not enabled, \texttt{VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT} must not be used with \texttt{VK_DESCRIPTOR_TYPE_SAMPLER}, \texttt{VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER}, or \texttt{VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE}.

- \texttt{descriptorBindingStorageImageUpdateAfterBind} indicates whether the implementation supports updating storage image descriptors after a set is bound. If this feature is not enabled, \texttt{VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT} must not be used with \texttt{VK_DESCRIPTOR_TYPE_STORAGE_IMAGE}.

- \texttt{descriptorBindingStorageBufferUpdateAfterBind} indicates whether the implementation supports updating storage buffer descriptors after a set is bound. If this feature is not enabled, \texttt{VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT} must not be used with \texttt{VK_DESCRIPTOR_TYPE_STORAGE_BUFFER}.

- \texttt{descriptorBindingUniformTexelBufferUpdateAfterBind} indicates whether the implementation supports updating uniform texel buffer descriptors after a set is bound. If this feature is not enabled, \texttt{VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT} must not be used with \texttt{VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER}.

- \texttt{descriptorBindingStorageTexelBufferUpdateAfterBind} indicates whether the implementation supports updating storage texel buffer descriptors after a set is bound. If this feature is not enabled, \texttt{VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT} must not be used with \texttt{VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER}.

- \texttt{descriptorBindingUpdateUnusedWhilePending} indicates whether the implementation supports updating descriptors while the set is in use. If this feature is not enabled, \texttt{VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT} must not be used.

- \texttt{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, \texttt{VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT} must not be used.

- \texttt{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 a VkStructureType value identifying this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- vulkanMemoryModel 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 VkPhysicalDeviceScalarBlockLayoutFeatures structure is defined as:

```c
// Provided by VK_VERSION_1_2
typdef struct VkPhysicalDeviceScalarBlockLayoutFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 scalarBlockLayout;
} VkPhysicalDeviceScalarBlockLayoutFeatures;
```

This structure describes the following feature:

- **sType** is a VkStructureType value identifying 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
typdef struct VkPhysicalDeviceUniformBufferStandardLayoutFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 uniformBufferStandardLayout;
} VkPhysicalDeviceUniformBufferStandardLayoutFeatures;
```

This structure describes the following feature:

- **sType** is a VkStructureType value identifying 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 a `VkStructureType` value identifying 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_ADDRESSFEATURES`

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 a `VkStructureType` value identifying 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;
    // ...
} VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures;
```
This structure describes the following feature:

- **sType** is a *VkStructureType* value identifying 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 a *VkStructureType* value identifying 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 a VkStructureType value identifying 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 a VkStructureType value identifying 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` nullDescriptor support requires the `VK_EXT_robustness2` extension.

The `VkPhysicalDeviceVulkanSC10Features` structure is defined as:

```c
// Provided by VKSC_VERSION_1_0
typedef struct VkPhysicalDeviceVulkanSC10Features {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderAtomicInstructions;
} VkPhysicalDeviceVulkanSC10Features;
```

This structure describes the following features:

• `sType` is a `VkStructureType` value identifying this structure.

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

• `shaderAtomicInstructions` indicates whether this implementation supports shaders which use the SPIR-V `OpAtomic*` instructions.

If the `VkPhysicalDeviceVulkanSC10Features` 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. `VkPhysicalDeviceVulkanSC10Features` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

Valid Usage (Implicit)

• `VUID-VkPhysicalDeviceVulkanSC10Features-sType-sType`
31.1. Feature Requirements

All Vulkan graphics implementations **must** support the following features:

- **robustBufferAccess**
- **multiview**, if Vulkan 1.1 is supported. Vulkan SC 1.0 does not require **multiview** to be supported [SCID-8].
- **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**
- **subgroupBroadcastDynamicId**, if Vulkan 1.2 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_depth_stencil_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. Vulkan SC 1.0 does not require **timelineSemaphore** to be supported [SCID-8].
• shaderSubgroupExtendedTypes, if Vulkan 1.2 or the VK_KHR_shader_subgroup_extended_types extension is supported.

• vulkanMemoryModel, if Vulkan SC 1.0 [SCID-5] or if the VK_KHR_vulkan_memory_model extension is supported.

• shaderInt64, if the shaderSharedInt64Atomics or shaderBufferInt64Atomics features are supported.

• storageBuffer16BitAccess, if uniformAndStorageBuffer16BitAccess is enabled.

All other features defined in the Specification are optional.
Chapter 32. 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 maxVertexInputAttributes;
    uint32_t maxVertexInputBindings;
    uint32_t maxVertexInputAttributeOffset;
    uint32_t maxVertexInputAttributes;
    uint32_t maxVertexInputBindings;
    uint32_t maxVertexInputAttributeOffset;
} VkPhysicalDeviceLimits;
```
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, `(offset + size)` must be less than or equal to this limit.

- **maxMemoryAllocationCount** is the maximum number of device memory allocations, as created by `vkAllocateMemory`, which can simultaneously exist.

- **maxSamplerAllocationCount** is the maximum number of sampler objects, as created by `vkCreateSampler`, which can simultaneously exist on a device.

- **bufferImageGranularity** is the granularity, in bytes, at which buffer or linear image resources, and optimal image resources can be bound to adjacent offsets in the same `VkDeviceMemory` object without aliasing. See Buffer-Image Granularity for more details.

- **sparseAddressSpaceSize** is the total amount of address space available, in bytes, for sparse memory resources. This is an upper bound on the sum of the 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_UNIFORM_BUFFER_DYNAMIC`, `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`, or `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` 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.

• `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 execution mode in shader modules or by the object decorated by the WorkgroupSize decoration, must be less than or equal to this limit.

• `maxComputeWorkGroupSize[3]` is the maximum size of a local compute workgroup, per dimension. These three values represent the maximum local workgroup size in the X, Y, and Z dimensions, respectively. The x, y, and z sizes, as specified by the LocalSize execution mode or by the object decorated by the WorkgroupSize decoration in shader modules, must be less than or equal to the corresponding limit.

• `subPixelPrecisionBits` is the number of bits of subpixel precision in framebuffer coordinates $x_f$ and $y_f$. See Rasterization.

• `subTexelPrecisionBits` is the number of bits of precision in the division along an axis of an image used for minification and magnification filters. $2^{\text{subTexelPrecisionBits}}$ is the actual number of divisions along each axis of the image represented. Sub-texel values calculated during image sampling will snap to these locations when generating the filtered results.

• `mipmapPrecisionBits` is the number of bits of division that the LOD calculation for mipmap fetching get snapped to when determining the contribution from each mip level to the mip filtered results. $2^{\text{mipmapPrecisionBits}}$ is the actual number of divisions.
- **maxDrawIndexedIndexValue** is the maximum index value that can be used for indexed draw calls when using 32-bit indices. This excludes the primitive restart index value of 0xFFFFFFFF. See **fullDrawIndexUint32**.

- **maxDrawIndirectCount** is the maximum draw count that is supported for indirect 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.

  ```
  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. **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 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 distances that can be used in a single shader stage. The sum of the sizes of any pair of arrays declared with the `ClipDistance` and `CullDistance` built-in decoration used by a single shader stage in a shader module must be less than or equal to this limit.

• `discreteQueuePriorities` is the number of discrete priorities that can be assigned to a queue based on the value of each member of `VkDeviceQueueCreateInfo::pQueuePriorities`. This must be at least 2, and levels must be spread evenly over the range, with at least one level at 1.0, and another at 0.0. See Queue Priority.

• `pointSizeRange[2]` is the range `[minimum, maximum]` of supported sizes for points. Values written to variables decorated with the `PointSize` built-in decoration are clamped to this range.
• `lineWidthRange[2]` is the range `[minimum, maximum]` of supported widths for lines. Values specified by the `lineWidth` member of the `VkPipelineRasterizationStateCreateInfo` or the `lineWidth` parameter to `vkCmdSetLineWidth` are clamped to this range.

• `PointSizeGranularity` is the granularity of supported point sizes. Not all point sizes in the range defined by `PointSizeRange` are supported. This limit specifies the granularity (or increment) between successive supported point sizes.

• `LineWidthGranularity` is the granularity of supported line widths. Not all line widths in the range defined by `LineWidthRange` are supported. This limit specifies the granularity (or increment) between successive supported line widths.

• `strictLines` specifies whether lines are rasterized according to the preferred method of rasterization. If set to `VK_FALSE`, lines may be rasterized under a relaxed set of rules. If set to `VK_TRUE`, lines are rasterized as per the strict definition. See Basic Line Segment Rasterization.

• `standardSampleLocations` specifies whether rasterization uses the standard sample locations as documented in Multisampling. If set to `VK_TRUE`, the implementation uses the documented sample locations. If set to `VK_FALSE`, the implementation may use different sample locations.

• `optimalBufferCopyOffsetAlignment` is the optimal buffer offset alignment in bytes for `vkCmdCopyBufferToImage` and `vkCmdCopyImageToBuffer`. The per texel alignment requirements are enforced, but applications should use the optimal alignment for optimal performance and power use. The value must be a power of two.

• `optimalBufferCopyRowPitchAlignment` is the optimal buffer row pitch alignment in bytes for `vkCmdCopyBufferToImage` and `vkCmdCopyImageToBuffer`. Row pitch is the number of bytes between texels with the same X coordinate in adjacent rows (Y coordinates differ by one). The per texel alignment requirements are enforced, but applications should use the optimal alignment for optimal performance and power use. 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,
};
```
• **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 a *VkStructureType* value identifying 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 instance.

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 shaderRoundingModeRTZFloat16;
    VkBool32 shaderRoundingModeRTZFloat32;
    VkBool32 shaderRoundingModeRTZFloat64;
} VkPhysicalDeviceFloatControlsProperties;
```

- `sType` is a `VkStructureType` value identifying 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 \( \pm \infty \) can be preserved in 16-bit floating-point computations. It also indicates whether the `SignedZeroInfNanPreserve` execution mode can be used for 16-bit floating-point types.
- `shaderSignedZeroInfNanPreserveFloat32` is a boolean value indicating whether sign of a zero, Nans and \( \pm \infty \) can be preserved in 32-bit floating-point computations. It also indicates whether the `SignedZeroInfNanPreserve` execution mode can be used for 32-bit floating-point types.
- `shaderSignedZeroInfNanPreserveFloat64` is a boolean value indicating whether sign of a zero, Nans and \( \pm \infty \) can be preserved in 64-bit floating-point computations. It also indicates whether the `SignedZeroInfNanPreserve` execution mode can be used for 64-bit floating-point types.
- `shaderDenormPreserveFloat16` is a boolean value indicating whether denormals can be preserved in 16-bit floating-point computations. It also indicates whether the `DenormPreserve` execution mode can be used for 16-bit floating-point types.
- `shaderDenormPreserveFloat32` is a boolean value indicating whether denormals can be preserved in 32-bit floating-point computations. It also indicates whether the `DenormPreserve` execution mode can be used for 32-bit floating-point types.
mode can be used for 32-bit floating-point types.

- \texttt{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.

- \texttt{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.

- \texttt{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.

- \texttt{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.

- \texttt{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.

- \texttt{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.

- \texttt{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.

- \texttt{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.

- \texttt{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.

- \texttt{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 \texttt{VkPhysicalDeviceFloatControlsProperties} structure is included in the \texttt{pNext} chain of the \texttt{VkPhysicalDeviceProperties2} structure passed to \texttt{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 a **VkStructureType** value identifying 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:

```
// 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 a `VkStructureType` value identifying 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:
- **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.
VkPhysicalDeviceSamplerFilterMinmaxProperties;

- `sType` is a `VkStructureType` value identifying 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.
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 a `VkStructureType` value identifying 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.

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 a `VkStructureType` value identifying 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 `VkPhysicalDeviceDescriptorIndexingProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceDescriptorIndexingProperties {
    VkStructureType sType;
    void* pNext;
    uint32_t maxUpdateAfterBindDescriptorsInAllPools;
    VkBool32 shaderUniformBufferArrayNonUniformIndexingNative;
    VkBool32 shaderSampledImageArrayNonUniformIndexingNative;
    VkBool32 shaderStorageBufferArrayNonUniformIndexingNative;
    VkBool32 shaderStorageImageArrayNonUniformIndexingNative;
    VkBool32 shaderInputAttachmentArrayNonUniformIndexingNative;
    VkBool32 robustBufferAccessUpdateAfterBind;
    VkBool32 quadDivergentImplicitLod;
    uint32_t maxPerStageDescriptorUpdateAfterBindSamplers;
    uint32_t maxPerStageDescriptorUpdateAfterBindUniformBuffers;
    uint32_t maxPerStageDescriptorUpdateAfterBindUniformBuffersDynamic;
    uint32_t maxPerStageDescriptorUpdateAfterBindStorageBuffers;
    uint32_t maxPerStageDescriptorUpdateAfterBindStorageBuffersDynamic;
    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;
} VkPhysicalDeviceDescriptorIndexingProperties;
```
• **sType** is a `VkStructureType` value identifying 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 LOD 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. 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. 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 VkPhysicalDeviceDepthStencilResolveProperties structure is defined as:

```
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceDepthStencilResolveProperties {
    VkStructureType sType;
    void* pNext;
    VkResolveModeFlags supportedDepthResolveModes;
    VkResolveModeFlags supportedStencilResolveModes;
    VkBool32 independentResolveNone;
    VkBool32 independentResolve;
} VkPhysicalDeviceDepthStencilResolveProperties;
```

• sType is a VkStructureType value identifying 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. A value of VK_RESOLVE_MODE_NONE indicates that depth resolve operations are disallowed [SCID-8]. If any bits are set then 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. A value of VK_RESOLVE_MODE_NONE indicates that stencil resolve operations are disallowed [SCID-8]. If any bits are set then 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 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 a VkStructureType value identifying 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

32.1. 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 30. Required Limit Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Limit</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32_t</td>
<td>maxImageDimension1D</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><code>uint32_t</code></td>
<td>maxImageDimension2D</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxImageDimension3D</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxImageDimensionCube</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxImageArrayLayers</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxTexelBufferElements</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxUniformBufferRange</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxStorageBufferRange</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxPushConstantsSize</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxMemoryAllocationCount</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxSamplerAllocationCount</td>
<td>-</td>
</tr>
<tr>
<td><code>VkDeviceSize</code></td>
<td>bufferImageGranularity</td>
<td>-</td>
</tr>
<tr>
<td><code>VkDeviceSize</code></td>
<td>sparseAddressSpaceSize</td>
<td><code>sparseBinding</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxBoundDescriptorSets</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxPerStageDescriptorSamplers</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxPerStageDescriptorUniformBuffers</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxPerStageDescriptorStorageBuffers</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxPerStageDescriptorSampledImages</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxPerStageDescriptorStorageImages</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxPerStageDescriptorInputAttachments</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxPerStageResources</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxDescriptorSetSamplers</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxDescriptorSetUniformBuffers</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxDescriptorSetUniformBuffersDynamic</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxDescriptorSetStorageBuffers</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxDescriptorSetStorageBuffersDynamic</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxDescriptorSetSampledImages</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxDescriptorSetStorageImages</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxDescriptorSetInputAttachments</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxVertexInputAttributes</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxVertexInputBindings</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxVertexInputAttributeOffset</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxVertexInputBindingStride</td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxVertexOutputComponents</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>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>maxTessellationTotalOutputComponents</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>3 × 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>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>Type</td>
<td>Limit</td>
<td>Feature</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------------------------------------------------------------------</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>geometryShader, shaderOutputLayer</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>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>
</tbody>
</table>

940
<table>
<thead>
<tr>
<th>Type</th>
<th>Limit</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxCullDistances</code></td>
<td><code>shaderCullDistance</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxCombinedClipAndCullDistances</code></td>
<td><code>shaderCullDistance</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>discreteQueuePriorities</code></td>
<td>-</td>
</tr>
<tr>
<td><code>2 × float</code></td>
<td><code>pointSizeRange</code></td>
<td><code>largePoints</code></td>
</tr>
<tr>
<td><code>2 × float</code></td>
<td><code>lineWidthRange</code></td>
<td><code>wideLines</code></td>
</tr>
<tr>
<td><code>float</code></td>
<td><code>pointSizeGranularity</code></td>
<td><code>largePoints</code></td>
</tr>
<tr>
<td><code>float</code></td>
<td><code>lineWidthGranularity</code></td>
<td><code>wideLines</code></td>
</tr>
<tr>
<td>VkBool32</td>
<td><code>strictLines</code></td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td><code>standardSampleLocations</code></td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td><code>optimalBufferCopyOffsetAlignment</code></td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td><code>optimalBufferCopyRowPitchAlignment</code></td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td><code>nonCoherentAtomSize</code></td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td><code>filterMinmaxSingleComponentFormats</code></td>
<td><code>samplerFilterMinmax</code></td>
</tr>
<tr>
<td>VkBool32</td>
<td><code>filterMinmaxImageComponentMapping</code></td>
<td><code>samplerFilterMinmax</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxUpdateAfterBindDescriptorsInAllPools</code></td>
<td><code>descriptorIndexing</code></td>
</tr>
<tr>
<td>VkBool32</td>
<td><code>shaderUniformBufferArrayNonUniformIndexingNative</code></td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td><code>shaderSampledImageArrayNonUniformIndexingNative</code></td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td><code>shaderStorageBufferArrayNonUniformIndexingNative</code></td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td><code>shaderStorageImageArrayNonUniformIndexingNative</code></td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td><code>shaderInputAttachmentArrayNonUniformIndexingNative</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageDescriptorUpdateAfterBindSamplers</code></td>
<td><code>descriptorIndexing</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageDescriptorUpdateAfterBindUniformBuffers</code></td>
<td><code>descriptorIndexing</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageDescriptorUpdateAfterBindStorageBuffers</code></td>
<td><code>descriptorIndexing</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageDescriptorUpdateAfterBindSampledImages</code></td>
<td><code>descriptorIndexing</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageDescriptorUpdateAfterBindStorageImages</code></td>
<td><code>descriptorIndexing</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageDescriptorUpdateAfterBindInputAttachments</code></td>
<td><code>descriptorIndexing</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageUpdateAfterBindResources</code></td>
<td><code>descriptorIndexing</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxDescriptorSetUpdateAfterBindSamplers</code></td>
<td><code>descriptorIndexing</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxDescriptorSetUpdateAfterBindUniformBuffers</code></td>
<td><code>descriptorIndexing</code></td>
</tr>
<tr>
<td>Type</td>
<td>Limit</td>
<td>Feature</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxDescriptorUpdateAfterBindUniformBuffersDynamic</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxDescriptorUpdateAfterBindStorageBuffers</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxDescriptorUpdateAfterBindStorageBuffersDynamic</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxDescriptorUpdateAfterBindSampledImages</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxDescriptorUpdateAfterBindStorageImages</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxDescriptorUpdateAfterBindInputAttachments</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint64_t</code></td>
<td><code>maxTimelineSemaphoreValueDifference</code></td>
<td>timelineSemaphore</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td><code>deviceNoDynamicHostAllocations</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td><code>deviceDestroyFreesMemory</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td><code>commandPoolMultipleCommandBuffersRecording</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td><code>commandPoolResetCommandBuffer</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td><code>commandBufferSimultaneousUse</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td><code>secondaryCommandBufferNullOrImagelessFramebuffer</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td><code>recycleDescriptorSetMemory</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td><code>recyclePipelineMemory</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxRenderPassSubpasses</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxRenderPassDependencies</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxSubpassInputAttachments</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxSubpassPreserveAttachments</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxFramebufferAttachments</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxDescriptorSetLayoutBindings</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxQueryFaultCount</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxCallbackFaultCount</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxCommandPoolCommandBuffers</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkDeviceSize</code></td>
<td><code>maxCommandBufferSize</code></td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 31. 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><code>maxImageDimension1D</code></td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td><code>maxImageDimension2D</code></td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td><code>maxImageDimension3D</code></td>
<td>-</td>
<td>256</td>
<td>min</td>
</tr>
<tr>
<td><code>maxImageDimensionCube</code></td>
<td>-</td>
<td>4096</td>
<td>min</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>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>maxBoundDescriptorSets</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>(96^8)</td>
<td>min, (n \times ) PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetUniformBuffers</td>
<td>-</td>
<td>(72^8)</td>
<td>min, (n \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>min, (n \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>min, (n \times ) PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetStorageImages</td>
<td>-</td>
<td>(24^8)</td>
<td>min, (n \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>
<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>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)³</td>
<td>min</td>
</tr>
<tr>
<td>viewportBoundsRange</td>
<td>-</td>
<td>(-8192,8191)⁴</td>
<td>(max,min)</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>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>-0.5⁵</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>4⁵</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>1</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 Type¹</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-------------------</td>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>sampledImageIntegerSampleCounts</td>
<td>-</td>
<td>VK_SAMPLE_COUNT_1_BIT</td>
<td>min</td>
</tr>
<tr>
<td>sampledImageDepthSampleCounts</td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>min</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td>sampledImageStencilSampleCounts</td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>min</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td>storageImageSampleCounts</td>
<td>VK_SAMPLE_COUNT_1_BIT</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>min</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td>maxSampleMaskWords</td>
<td>-</td>
<td>1</td>
<td>min</td>
</tr>
<tr>
<td>timestampComputeAndGraphics</td>
<td>-</td>
<td>-</td>
<td>implementatio n-dependent</td>
</tr>
<tr>
<td>timestampPeriod</td>
<td>-</td>
<td>-</td>
<td>duration</td>
</tr>
<tr>
<td>maxClipDistances</td>
<td>0</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td>maxCullDistances</td>
<td>0</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td>maxCombinedClipAndCullDistances</td>
<td>0</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td>discreteQueuePriorities</td>
<td>-</td>
<td>2</td>
<td>min</td>
</tr>
<tr>
<td>pointSizeRange</td>
<td>(1.0,1.0)</td>
<td>(1.0,64.0 - ULP)⁶</td>
<td>(max,min)</td>
</tr>
<tr>
<td>lineWidthRange</td>
<td>(1.0,1.0)</td>
<td>(1.0,8.0 - ULP)⁷</td>
<td>(max,min)</td>
</tr>
<tr>
<td>pointSizeGranularity</td>
<td>0.0</td>
<td>1.0⁶</td>
<td>max, fixed point increment</td>
</tr>
<tr>
<td>lineWidthGranularity</td>
<td>0.0</td>
<td>1.0⁷</td>
<td>max, fixed point increment</td>
</tr>
<tr>
<td>strictLines</td>
<td>-</td>
<td>-</td>
<td>implementatio n-dependent</td>
</tr>
<tr>
<td>standardSampleLocations</td>
<td>-</td>
<td>-</td>
<td>implementatio n-dependent</td>
</tr>
<tr>
<td>optimalBufferCopyOffsetAlignment</td>
<td>-</td>
<td>-</td>
<td>recommendation</td>
</tr>
<tr>
<td>optimalBufferCopyRowPitchAlignment</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</td>
<td>max</td>
</tr>
<tr>
<td>maxMultiviewViewCount</td>
<td>-</td>
<td>6</td>
<td>min</td>
</tr>
<tr>
<td>maxMultiviewInstanceIndex</td>
<td>-</td>
<td>(2^{27}-1)</td>
<td>min</td>
</tr>
<tr>
<td>filterMinmaxSingleComponentFormats</td>
<td>-</td>
<td>-</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>filterMinmaxImageComponentMapping</td>
<td>-</td>
<td>-</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>maxPerSetDescriptors</td>
<td>-</td>
<td>1024</td>
<td>min</td>
</tr>
<tr>
<td>maxMemoryAllocationSize</td>
<td>-</td>
<td>(2^{30})</td>
<td>min</td>
</tr>
<tr>
<td>maxUpdateAfterBindDescriptorsInAllPools</td>
<td>0</td>
<td>500000</td>
<td>min</td>
</tr>
<tr>
<td>shaderUniformBufferArrayNonUniformIndexingNative</td>
<td>-</td>
<td>false</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>shaderSampledImageArrayNonUniformIndexingNative</td>
<td>-</td>
<td>false</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>shaderStorageBufferArrayNonUniformIndexingNative</td>
<td>-</td>
<td>false</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>shaderStorageImageArrayNonUniformIndexingNative</td>
<td>-</td>
<td>false</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>shaderInputAttachmentArrayNonUniformIndexingNative</td>
<td>-</td>
<td>false</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindSamplers</td>
<td>(0^9)</td>
<td>500000 (9)</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindUniformBuffers</td>
<td>(0^9)</td>
<td>12 (9)</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindStorageBuffers</td>
<td>(0^9)</td>
<td>500000 (9)</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindSampledImages</td>
<td>(0^9)</td>
<td>500000 (9)</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindStorageImages</td>
<td>(0^9)</td>
<td>500000 (9)</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindInputAttachments</td>
<td>(0^9)</td>
<td>4 (9)</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageUpdateAfterBindResources</td>
<td>(0^9)</td>
<td>500000 (9)</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindSamplers</td>
<td>(0^9)</td>
<td>500000 (9)</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindUniformBuffers</td>
<td>(0^9)</td>
<td>72 (8) (9)</td>
<td>min, (n \times) PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindUniformBuffersDynamic</td>
<td>(0^9)</td>
<td>8 (9)</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindStorageBuffers</td>
<td>(0^9)</td>
<td>500000 (9)</td>
<td>min</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>maxDescriptorSetUpdateAfterBindStorageBuffersDynamic</td>
<td>0⁹</td>
<td>4⁹</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindSampledImages</td>
<td>0⁹</td>
<td>500000⁹</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindStorageImages</td>
<td>0⁹</td>
<td>500000⁹</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindInputAttachments</td>
<td>0⁹</td>
<td>4⁹</td>
<td>min</td>
</tr>
<tr>
<td>maxTimelineSemaphoreValueDifference</td>
<td></td>
<td>2³¹-1</td>
<td>min</td>
</tr>
<tr>
<td>deviceNoDynamicHostAllocations</td>
<td>-</td>
<td>-</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>deviceDestroyFreesMemory</td>
<td>-</td>
<td>-</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>commandPoolMultipleCommandBuffersRecording</td>
<td>-</td>
<td>-</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>commandPoolResetCommandBuffer</td>
<td>-</td>
<td>-</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>commandBufferSimultaneousUse</td>
<td>-</td>
<td>-</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>secondaryCommandBufferNullOrImagelessFramebuffer</td>
<td>-</td>
<td>-</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>recycleDescriptorSetMemory</td>
<td>-</td>
<td>-</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>recyclePipelineMemory</td>
<td>-</td>
<td>-</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>maxRenderPassSubpasses</td>
<td>-</td>
<td>1</td>
<td>min</td>
</tr>
<tr>
<td>maxRenderPassDependencies</td>
<td>-</td>
<td>18</td>
<td>min</td>
</tr>
<tr>
<td>maxSubpassInputAttachments</td>
<td>-</td>
<td>0</td>
<td>min</td>
</tr>
<tr>
<td>maxSubpassPreserveAttachments</td>
<td>-</td>
<td>0</td>
<td>min</td>
</tr>
<tr>
<td>maxFramebufferAttachments</td>
<td>-</td>
<td>9¹¹</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetLayoutBindings</td>
<td>-</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxQueryFaultCount</td>
<td>-</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxCallbackFaultCount</td>
<td>-</td>
<td>1</td>
<td>min</td>
</tr>
<tr>
<td>maxCommandPoolCommandBuffers</td>
<td>-</td>
<td>2⁵⁶</td>
<td>min</td>
</tr>
<tr>
<td>maxCommandBufferSize</td>
<td>-</td>
<td>2²⁰</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: \([\text{minInterpolationOffset}, \text{maxInterpolationOffset}]\). The ULP is determined by **subPixelInterpolationOffsetBits**. If **subPixelInterpolationOffsetBits** is 4, this provides increments of \((1/2^4) = 0.0625\), and thus the range of supported interpolation offsets would be \([-0.5, 0.4375]\).

6

The point size ULP is determined by **PointSizeGranularity**. If the **PointSizeGranularity** is 0.125, the range of supported point sizes **must** be at least \([1.0, 63.875]\).

7

The line width ULP is determined by **LineWidthGranularity**. If the **LineWidthGranularity** is 0.0625, the range of supported line widths **must** be at least \([1.0, 7.9375]\).

8

The minimum **maxDescriptorSet** limit is \(n\) times the corresponding **specification** minimum **maxPerStageDescriptor** limit, where \(n\) is the number of shader stages supported by the **VkPhysicalDevice**. If all shader stages are supported, \(n = 6\) (vertex, tessellation control, tessellation evaluation, geometry, fragment, compute).

9

The **UpdateAfterBind** descriptor limits **must** each be greater than or equal to the corresponding **non**-**UpdateAfterBind** limit.

11

**maxFramebufferAttachments** **must** be greater than or equal to two times **maxColorAttachments** (for color and resolve attachments) plus one (for the depth/stencil attachment), or else **must** be equal to \(2^{32} - 1\).
Chapter 33. 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.

33.1. 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.

// 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_A2R10G10B10_UNORM_PACK32 = 53,
VK_FORMAT_A2R10G10B10_SNORM_PACK32 = 54,
VK_FORMAT_A2R10G10B10_USCALED_PACK32 = 55,
VK_FORMAT_A2R10G10B10_SSCALED_PACK32 = 56,
VK_FORMAT_A2B10G10R10_UNORM_PACK32 = 57,
VK_FORMAT_A2B10G10R10_SNORM_PACK32 = 58,
VK_FORMAT_A2B10G10R10_USCALED_PACK32 = 59,
VK_FORMAT_A2B10G10R10_SSCALED_PACK32 = 60,
VK_FORMAT_A2B10G10R10_UINT_PACK32 = 61,
VK_FORMAT_A2B10G10R10_SINT PACK32 = 62,
VK_FORMAT_A2B10G10R10_SRGB PACK32 = 63,
VK_FORMAT_R16_UNORM = 64,
VK_FORMAT_R16_SNORM = 65,
VK_FORMAT_R16_USCALED = 66,
VK_FORMAT_R16_SSCALED = 67,
VK_FORMAT_R16_UINT = 68,
VK_FORMAT_R16_SINT = 69,
VK_FORMAT_R16_SFLOAT = 70,
VK_FORMAT_R16G16_UNORM = 71,
VK_FORMAT_R16G16_SNORM = 72,
VK_FORMAT_R16G16_USCALED = 73,
VK_FORMAT_R16G16_SSCALED = 74,
VK_FORMAT_R16G16_UINT = 75,
VK_FORMAT_R16G16_SINT = 76,
VK_FORMAT_R16G16_SFLOAT = 83,
VK_FORMAT_R16G16B16_UNORM = 84,
VK_FORMAT_R16G16B16_SNORM = 85,
VK_FORMAT_R16G16B16_USCALED = 86,
VK_FORMAT_R16G16B16_SSCALED = 87,
VK_FORMAT_R16G16B16_UINT = 88,
VK_FORMAT_R16G16B16_SINT = 89,
VK_FORMAT_R16G16B16_SFLOAT = 90,
VK_FORMAT_R16G16B16A16_UNORM = 91,
VK_FORMAT_R16G16B16A16_SNORM = 92,
VK_FORMAT_R16G16B16A16_USCALED = 93,
VK_FORMAT_R16G16B16A16_SSCALED = 94,
VK_FORMAT_R16G16B16A16_UINT = 95,
VK_FORMAT_R16G16B16A16_SINT = 96,
VK_FORMAT_R16G16B16A16_SFLOAT = 97,
VK_FORMAT_R32_UINT = 98,
VK_FORMAT_R32_SINT = 99,
VK_FORMAT_R32_SFLOAT = 100,
VK_FORMAT_R32G32_UINT = 101,
VK_FORMAT_R32G32_SINT = 102,
VK_FORMAT_R32G32_SFLOAT = 103,
VK_FORMAT_R32G32B32_UINT = 104,
VK_FORMAT_R32G32B32_SINT = 105,
VK_FORMAT_R32G32B32_SFLOAT = 106,
VK_FORMAT_R32G32B32A32_UINT = 107,
VK_FORMAT_R32G32B32A32_SINT = 108,
VK_FORMAT_R32G32B32A32_SFLOAT = 109,
VK_FORMAT_R64_UINT = 110,
VK_FORMAT_R64_SINT = 111,
VK_FORMAT_R64_SFLOAT = 112,
VK_FORMAT_R64G64_UINT = 113,
VK_FORMAT_R64G64_SINT = 114,
VK_FORMAT_R64G64_SFLOAT = 115,
VK_FORMAT_R64G64B64_UINT = 116,
VK_FORMAT_R64G64B64_SINT = 117,
VK_FORMAT_R64G64B64_SFLOAT = 118,
VK_FORMAT_R64G64B64A64_UINT = 119,
VK_FORMAT_R64G64B64A64_SINT = 120,
VK_FORMAT_R64G64B64A64_SFLOAT = 121,
VK_FORMAT_B10G11R11_UFLOAT_PACK32 = 122,
VK_FORMAT_E5B9G9R9_UFLOAT_PACK32 = 123,
VK_FORMAT_D16_UNORM = 124,
VK_FORMAT_X8_D24_UNORM_PACK32 = 125,
VK_FORMAT_D32_SFLOAT = 126,
VK_FORMAT_S8_UINT = 127,
VK_FORMAT_D16_UNORM_S8_UINT = 128,
VK_FORMAT_D24_UNORM_S8_UINT = 129,
VK_FORMAT_D32_SFLOAT_S8_UINT = 130,
VK_FORMAT_BC1_RGB_UNORM_BLOCK = 131,
VK_FORMAT_BC1_RGB_SRGB_BLOCK = 132,
VK_FORMAT_BC1_RGBA_UNORM_BLOCK = 133,
<table>
<thead>
<tr>
<th>Format Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_BC1_RGBA_SRGB_BLOCK</td>
<td>134</td>
</tr>
<tr>
<td>VK_FORMAT_BC2_UNORM_BLOCK</td>
<td>135</td>
</tr>
<tr>
<td>VK_FORMAT_BC2_SRGB_BLOCK</td>
<td>136</td>
</tr>
<tr>
<td>VK_FORMAT_BC3_UNORM_BLOCK</td>
<td>137</td>
</tr>
<tr>
<td>VK_FORMAT_BC3_SRGB_BLOCK</td>
<td>138</td>
</tr>
<tr>
<td>VK_FORMAT_BC4_UNORM_BLOCK</td>
<td>139</td>
</tr>
<tr>
<td>VK_FORMAT_BC4_SNORM_BLOCK</td>
<td>140</td>
</tr>
<tr>
<td>VK_FORMAT_BC5_UNORM_BLOCK</td>
<td>141</td>
</tr>
<tr>
<td>VK_FORMAT_BC5_SNORM_BLOCK</td>
<td>142</td>
</tr>
<tr>
<td>VK_FORMAT_BCGH_UFLOAT_BLOCK</td>
<td>143</td>
</tr>
<tr>
<td>VK_FORMAT_BCGH_SFLOAT_BLOCK</td>
<td>144</td>
</tr>
<tr>
<td>VK_FORMAT_BC7_UNORM_BLOCK</td>
<td>145</td>
</tr>
<tr>
<td>VK_FORMAT_BC7_SRGB_BLOCK</td>
<td>146</td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK</td>
<td>147</td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK</td>
<td>148</td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK</td>
<td>149</td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK</td>
<td>150</td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK</td>
<td>151</td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK</td>
<td>152</td>
</tr>
<tr>
<td>VK_FORMAT_EAC_R11_UNORM_BLOCK</td>
<td>153</td>
</tr>
<tr>
<td>VK_FORMAT_EAC_R11_SNORM_BLOCK</td>
<td>154</td>
</tr>
<tr>
<td>VK_FORMAT_EAC_R11G11_UNORM_BLOCK</td>
<td>155</td>
</tr>
<tr>
<td>VK_FORMAT_EAC_R11G11_SNORM_BLOCK</td>
<td>156</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_4x4_UNORM_BLOCK</td>
<td>157</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_4x4_SRGB_BLOCK</td>
<td>158</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x4_UNORM_BLOCK</td>
<td>159</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x4_SRGB_BLOCK</td>
<td>160</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x5_UNORM_BLOCK</td>
<td>161</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x5_SRGB_BLOCK</td>
<td>162</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x5_UNORM_BLOCK</td>
<td>163</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x5_SRGB_BLOCK</td>
<td>164</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x6_UNORM_BLOCK</td>
<td>165</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x6_SRGB_BLOCK</td>
<td>166</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x5_UNORM_BLOCK</td>
<td>167</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x5_SRGB_BLOCK</td>
<td>168</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x6_UNORM_BLOCK</td>
<td>169</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x6_SRGB_BLOCK</td>
<td>170</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x8_UNORM_BLOCK</td>
<td>171</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x8_SRGB_BLOCK</td>
<td>172</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x5_UNORM_BLOCK</td>
<td>173</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x5_SRGB_BLOCK</td>
<td>174</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x6_UNORM_BLOCK</td>
<td>175</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x6_SRGB_BLOCK</td>
<td>176</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x8_UNORM_BLOCK</td>
<td>177</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x8_SRGB_BLOCK</td>
<td>178</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x10_UNORM_BLOCK</td>
<td>179</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x10_SRGB_BLOCK</td>
<td>180</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x10_UNORM_BLOCK</td>
<td>181</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x10_SRGB_BLOCK</td>
<td>182</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x12_UNORM_BLOCK</td>
<td>183</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x12_SRGB_BLOCK</td>
<td>184</td>
</tr>
</tbody>
</table>
• `VK_FORMAT_UNDEFINED` specifies that the format is not specified.

• `VK_FORMAT_R4G4_UNORM_PACK8` specifies a two-component, 8-bit packed unsigned normalized format that has a 4-bit R component in bits 4..7, and a 4-bit G component in bits 0..3.

• `VK_FORMAT_R4G4B4A4_UNORM_PACK16` specifies a four-component, 16-bit packed unsigned normalized format that has a 4-bit R component in bits 12..15, a 4-bit G component in bits 8..11, a 4-bit B component in bits 4..7, and a 4-bit A component in bits 0..3.

• `VK_FORMAT_B4G4R4A4_UNORM_PACK16` specifies a four-component, 16-bit packed unsigned normalized format that has a 4-bit B component in bits 12..15, a 4-bit G component in bits 8..11, a 4-bit R component in bits 4..7, and a 4-bit A component in bits 0..3.

• `VK_FORMAT_R5G6B5_UNORM_PACK16` specifies a three-component, 16-bit packed unsigned normalized format that has a 5-bit R component in bits 11..15, a 6-bit G component in bits 5..10, and a 5-bit B component in bits 0..4.

• `VK_FORMAT_B5G5R5A1_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 format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.

• **VK_FORMAT_R8G8B8_UINT** specifies a three-component, 24-bit unsigned integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.

• **VK_FORMAT_R8G8B8_SINT** specifies a three-component, 24-bit signed integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
• **VK_FORMAT_R8G8B8_SRGB** specifies a three-component, 24-bit unsigned normalized format that has an 8-bit R component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, and an 8-bit B component stored with sRGB nonlinear encoding in byte 2.

• **VK_FORMAT_B8G8R8_UNORM** specifies a three-component, 24-bit unsigned normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

• **VK_FORMAT_B8G8R8_SNORM** specifies a three-component, 24-bit signed normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

• **VK_FORMAT_B8G8R8_USCALED** specifies a three-component, 24-bit unsigned scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

• **VK_FORMAT_B8G8R8_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 stored with sRGB nonlinear encoding in byte 3.
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 RGBA texel data, and provides 1 bit of alpha.

• **VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK** specifies a four-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB
texel data with sRGB nonlinear encoding, and provides 1 bit of alpha.

- **VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK** specifies a four-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values.

- **VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK** specifies a four-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values with sRGB nonlinear encoding applied.

- **VK_FORMAT_EAC_R11_UNORM_BLOCK** specifies a one-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 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_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_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_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.
data with sRGB nonlinear encoding applied to the RGB components.

- **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_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_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_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 with sRGB nonlinear encoding applied to the RGB components.

- **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_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_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.
• **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_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.

• **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_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_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_G8_B8_R8_3PLANE_420_UNORM** specifies an unsigned normalized multi-planar format that has an 8-bit G component in plane 0, an 8-bit B component in plane 1, and an 8-bit R component in plane 2. The horizontal and vertical dimensions of the R and B planes are halved relative to the image dimensions, and each R and B component is shared with the G components for which \( |i \odot 0.5| = i_R \) and \( |j \odot 0.5| = j_B = j_R \). 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_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_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. This format only supports images with a width that is a multiple of two.

- **VK_FORMAT_G8_B8R8_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_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 $2 \times 1$ rectangle of unsigned normalized RGB texel data. One G value is present at each $i$ coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 10-bit G component for the even $i$ coordinate in the top 10 bits of the word in bytes $0..1$, a 10-bit B component in the top 10 bits of the word in bytes $2..3$, a 10-bit G component for the odd $i$ coordinate in the top 10 bits of the word in bytes $4..5$, and a 10-bit R component in the top 10 bits of the word in bytes $6..7$, with the bottom 6 bits of each word 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 $2 \times 1$ compressed texel block.

• **VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16** specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a $2 \times 1$ rectangle of unsigned normalized RGB texel data. One G value is present at each $i$ coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 10-bit B component in the top 10 bits of the word in bytes $0..1$, a 10-bit G component for the even $i$ coordinate in the top 10 bits of the word in bytes $2..3$, a 10-bit R component in the top 10 bits of the word in bytes $4..5$, and a 10-bit G component for the odd $i$ coordinate in the top 10 bits of the word in bytes $6..7$, with the bottom 6 bits of each word 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 $2 \times 1$ compressed texel block.

• **VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16** specifies an unsigned normalized multi-planar format that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, a 10-bit B component in the top 10 bits of each 16-bit word of plane 1, and a 10-bit R component in the top 10 bits of each 16-bit word of plane 2, with the bottom 6 bits of each word 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 `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_B10X6R10X6_2PLANE_420_UNORM_3PACK16** specifies an unsigned normalized multi-planar format that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 10-bit B component in the top 10 bits of the word in bytes $0..1$, and a 10-bit R component in the top 10 bits of the word in bytes $2..3$, 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 $|i_G \times 0.5| = i_B = i_R$ 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_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 \(i_G \times 0.5 = i_B = i_R\). 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_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 \(i_G \times 0.5 = i_B = i_R\). 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. 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 `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane.

- **`VK_FORMAT_R12X4_UNORM_PACK16`** specifies a one-component, 16-bit unsigned normalized format that has a single 12-bit R component in the top 12 bits of a 16-bit word, with the bottom 4 bits 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 RGBA 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 \(i_G \times 0.5 = i_R\) 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_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 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 \(j_G \times 0.5 = j_B = j_R\). 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_3PLANE_422_UNORM_3PACK16** specifies an unsigned normalized multi-planar format that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, a 12-bit B component in the top 12 bits of each 16-bit word of plane 1, and a 12-bit R component in the top 12 bits of each 16-bit word of plane 2, with the bottom 4 bits of each word 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 \(i_G \times 0.5 = i_R\). 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, 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_B16_R16_3PLANE_420_UNORM** specifies an unsigned normalized multi-planar format that has a 16-bit G component in each 16-bit word of plane 0, a 16-bit B component in each 16-bit word of plane 1, and a 16-bit R component in each 16-bit word of plane 2. The horizontal and vertical dimensions of the R and B planes are halved relative to the image dimensions, and each R and B component is shared with the G components for which \( \lfloor i_G \times 0.5 \rfloor = i_B = i_R \) and \( \lceil i_G \times 0.5 \rceil = j_B = j_R \). 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 \( |i_G \times 0.5| = i_B = i_R \) 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_G16_B16_R16_3PLANE_422_UNORM** specifies an unsigned normalized multi-planar format that has a 16-bit G component in each 16-bit word of plane 0, a 16-bit B component in each 16-bit word of plane 1, and a 16-bit R component in each 16-bit word of plane 2. The horizontal dimension of the R and B plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which \( |i_G \times 0.5| = i_B = i_R \). The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane. 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 \( |i_G \times 0.5| = i_B = i_R \). 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. 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.

### 33.1.1. 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.

_Please note: the table is not shown in this text._
<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>VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM</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_R8_UNORM</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R8_UNORM</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td>VK_FORMAT_G8_B8R8_2PLANE_420_UNORM</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>$h/2$</td>
</tr>
<tr>
<td>VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM</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_R8_UNORM</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R8_UNORM</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td>VK_FORMAT_G8_B8R8_2PLANE_422_UNORM</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>$h/2$</td>
</tr>
<tr>
<td>VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM</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>2</td>
<td>VK_FORMAT_R8_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16</td>
<td></td>
<td>$w$</td>
<td>$h$</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>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16</td>
<td></td>
<td>$w$</td>
<td>$h$</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>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16</td>
<td></td>
<td>$w$</td>
<td>$h$</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>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16</td>
<td></td>
<td>$w$</td>
<td>$h$</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>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>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16</td>
<td>w</td>
<td>h</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</td>
<td>h</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16</td>
<td>w</td>
<td>h</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>w</td>
<td>h</td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>w/2</td>
<td>h/2</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>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16</td>
<td>VK_FORMAT_R12X4G12X4_UNORM_2PACK16</td>
<td>w</td>
<td>h</td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R12X4G12X4_UNORM_2PACK16</td>
<td>w/2</td>
<td>h/2</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R12X4G12X4_UNORM_2PACK16</td>
<td>w/2</td>
<td>h/2</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16</td>
<td>VK_FORMAT_R12X4G12X4_UNORM_2PACK16</td>
<td>w</td>
<td>h</td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R12X4G12X4_UNORM_2PACK16</td>
<td>w/2</td>
<td>h/2</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R12X4G12X4_UNORM_2PACK16</td>
<td>w/2</td>
<td>h/2</td>
</tr>
<tr>
<td>VR_FORMAT_G16_B16_R16_3PLANE_420_UNORM</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>w</td>
<td>h</td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>w/2</td>
<td>h/2</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>VK_FORMAT_G16_B16R16_2PLANE_420_UNORM</td>
<td>VK_FORMAT_R16G1616_UNORM</td>
<td>w</td>
<td>h</td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R16G1616_UNORM</td>
<td>w/2</td>
<td>h/2</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R16G1616_UNORM</td>
<td>w/2</td>
<td>h/2</td>
</tr>
<tr>
<td>VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>w</td>
<td>h</td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>w/2</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>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>2</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_G16_B16R16_2PLANE_422_UNORM</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_R16_3PLANE_444_UNORM</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
</tbody>
</table>

### 33.1.2. Multi-planar Format Image Aspect

When using `VkImageAspectFlagBits` to select a plane of a multi-planar format, the following are the valid options:

- Two planes
  - `VK_IMAGE_ASPECT_PLANE_0_BIT`
  - `VK_IMAGE_ASPECT_PLANE_1_BIT`

- Three planes
  - `VK_IMAGE_ASPECT_PLANE_0_BIT`
  - `VK_IMAGE_ASPECT_PLANE_1_BIT`
  - `VK_IMAGE_ASPECT_PLANE_2_BIT`

### 33.1.3. 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_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_R10X6_UNORM_PACK16
• VK_FORMAT_R10X6G10X6_UNORM_2PACK16
• VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16
• VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16
• VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16
• VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16
• VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16
• VK_FORMAT_G10X6_B10X6R10X6_3PLANE_422_UNORM_3PACK16
• VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16
• VK_FORMAT_G10X6_B10X6R10X6_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_B12X4R12X4_3PLANE_444_UNORM_3PACK16

• 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_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_SRGB_PACK32
33.1.4. Identification of Formats

A “format” is represented by a single enum value. The name of a format is usually built up by using the following pattern:

```
VK_FORMAT_{component-format|compression-scheme}_{numeric-format}
```

The component-format indicates either the size of the R, G, B, and A components (if they are present) in the case of a color format, or the size of the depth (D) and stencil (S) components (if they are present) in the case of a depth/stencil format (see below). An X indicates a component that is unused, but may be present for padding.
Table 33. Interpretation of Numeric Format

<table>
<thead>
<tr>
<th>Numeric format</th>
<th>Type-Declaration instructions</th>
<th>Numeric type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNORM</td>
<td>OpTypeFloat</td>
<td>floating-point</td>
<td>The components are unsigned normalized values in the range [0,1]</td>
</tr>
<tr>
<td>SNORM</td>
<td>OpTypeFloat</td>
<td>floating-point</td>
<td>The components are signed normalized values in the range [-1,1]</td>
</tr>
<tr>
<td>USCALED</td>
<td>OpTypeFloat</td>
<td>floating-point</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>floating-point</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>unsigned integer</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>signed integer</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>floating-point</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>floating-point</td>
<td>The components are signed floating-point numbers</td>
</tr>
<tr>
<td>SRGB</td>
<td>OpTypeFloat</td>
<td>floating-point</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 34. 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>Compression scheme</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</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 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.

33.1.5. 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 35. Byte mappings for non-packed/compressed color formats

<table>
<thead>
<tr>
<th>R</th>
<th>G</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>G</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>G</td>
<td>R</td>
</tr>
<tr>
<td>R</td>
<td>G</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>G</td>
<td>R</td>
</tr>
<tr>
<td>G0</td>
<td>B</td>
<td>G1</td>
</tr>
<tr>
<td>B</td>
<td>G0</td>
<td>R</td>
</tr>
</tbody>
</table>

Note
No common format has a single plane containing both R and B components but does not store these components at reduced horizontal resolution.

980
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 36. 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>6</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

```
VK_FORMAT_R4G4_UNORM_PACK8

3  2  1  0
```

Table 37. 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>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
VK_FORMAT_R4G4B4A4_UNORM_PACK16

3  2  1  0
```

```
VK_FORMAT_B4G4R4A4_UNORM_PACK16
```

```
B  G  R  A
```

981
Table 38. Bit mappings for packed 32-bit formats

<table>
<thead>
<tr>
<th>Bit</th>
<th>VK_FORMAT_R5G6B5_UNORM_PACK16</th>
<th>VK_FORMAT_B5G6R5_UNORM_PACK16</th>
<th>VK_FORMAT_R5G5B5A1_UNORM_PACK16</th>
<th>VK_FORMAT_B5G5R5A1_UNORM_PACK16</th>
<th>VK_FORMAT_A8B8G8R8_*_PACK32</th>
<th>VK_FORMAT_A2R10G10B10_*_PACK32</th>
<th>VK_FORMAT_A2B10G10R10_*_PACK32</th>
<th>VK_FORMAT_B10G11R11_UFLOAT_PACK32</th>
<th>VK_FORMAT_E5B9G9R9_UFLOAT_PACK32</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0 1 0 3 2 1 0 3 2 1 0</td>
<td>0 1 0 3 2 1 0 3 2 1 0</td>
<td>0 1 0 3 2 1 0 3 2 1 0</td>
<td>0 1 0 3 2 1 0 3 2 1 0</td>
<td>0 1 0 3 2 1 0 3 2 1 0</td>
<td>0 1 0 3 2 1 0 3 2 1 0</td>
<td>0 1 0 3 2 1 0 3 2 1 0</td>
<td>0 1 0 3 2 1 0 3 2 1 0</td>
<td>0 1 0 3 2 1 0 3 2 1 0</td>
</tr>
<tr>
<td>R</td>
<td>4 3 2 1 0 5 4 3 2 1 0</td>
<td>4 3 2 1 0 5 4 3 2 1 0</td>
<td>4 3 2 1 0 5 4 3 2 1 0</td>
<td>4 3 2 1 0 5 4 3 2 1 0</td>
<td>4 3 2 1 0 5 4 3 2 1 0</td>
<td>4 3 2 1 0 5 4 3 2 1 0</td>
<td>4 3 2 1 0 5 4 3 2 1 0</td>
<td>4 3 2 1 0 5 4 3 2 1 0</td>
<td>4 3 2 1 0 5 4 3 2 1 0</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td>4 3 2 1 0 5 4 3 2 1 0</td>
<td>4 3 2 1 0 5 4 3 2 1 0</td>
<td>4 3 2 1 0 5 4 3 2 1 0</td>
<td>4 3 2 1 0 5 4 3 2 1 0</td>
<td>4 3 2 1 0 5 4 3 2 1 0</td>
<td>4 3 2 1 0 5 4 3 2 1 0</td>
<td>4 3 2 1 0 5 4 3 2 1 0</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>A</td>
<td>0 4 3 2 1 0 4 3 2 1 0</td>
<td>0 4 3 2 1 0 4 3 2 1 0</td>
<td>0 4 3 2 1 0 4 3 2 1 0</td>
<td>0 4 3 2 1 0 4 3 2 1 0</td>
<td>0 4 3 2 1 0 4 3 2 1 0</td>
<td>0 4 3 2 1 0 4 3 2 1 0</td>
<td>0 4 3 2 1 0 4 3 2 1 0</td>
<td>0 4 3 2 1 0 4 3 2 1 0</td>
<td>0 4 3 2 1 0 4 3 2 1 0</td>
</tr>
<tr>
<td>R</td>
<td>9 8 7 6 5 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

982
33.1.6. 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).

33.1.7. 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 numeric format of the uncompressed pixels. 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 39. 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_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 Block size 4 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_R10X6G10X6_UNORM_2PACK16, VK_FORMAT_R12X4G12X4_UNORM_2PACK16, VK_FORMAT_R8G8B8B8_UNORM, VK_FORMAT_R8G8B8B8_SNORM, VK_FORMAT_R8G8B8B8_USCALED, VK_FORMAT_R8G8B8B8_SSCALED, VK_FORMAT_R8G8B8B8_UINT, VK_FORMAT_R8G8B8B8_SINT, VK_FORMAT_R8G8B8B8_SRGB, VK_FORMAT_B8G8R8B8_UNORM, VK_FORMAT_B8G8R8B8_SNORM, VK_FORMAT_B8G8R8B8_USCALED, VK_FORMAT_B8G8R8B8_SSCALED, VK_FORMAT_B8G8R8B8_UINT, VK_FORMAT_B8G8R8B8_SINT, VK_FORMAT_B8G8R8B8_SRGB, VK_FORMAT_A8B8G8R8_UNORM_PACK32, VK_FORMAT_A8B8G8R8_SNORM_PACK32, VK_FORMAT_A8B8G8R8_USCALED_PACK32, VK_FORMAT_A8B8G8R8_SSCALED_PACK32, VK_FORMAT_A8B8G8R8_UINT_PACK32, VK_FORMAT_A8B8G8R8_SINT PACK32, VK_FORMAT_A8B8G8R8_SRGB_PACK32, VK_FORMAT_A2R10G10B10_UNORM_PACK32, VK_FORMAT_A2R10G10B10_SNORM PACK32, VK_FORMAT_A2R10G10B10_USCALED_PACK32, VK_FORMAT_A2R10G10B10_SSCALED PACK32, VK_FORMAT_A2R10G10B10_UINT PACK32, VK_FORMAT_A2R10G10B10_SINT PACK32, VK_FORMAT_A2R10G10B10_SRGB PACK32, VK_FORMAT_R16G16_UNORM, VK_FORMAT_R16G16_SNORM, VK_FORMAT_R16G16_USCALED, VK_FORMAT_R16G16_SSCALED, VK_FORMAT_R16G16_UINT, VK_FORMAT_R16G16_SINT, VK_FORMAT_R16G16_SFLOAT, VK_FORMAT_R32_UINT, VK_FORMAT_R32_SINT, VK_FORMAT_R32_SFLOAT, VK_FORMAT_B10G11R11_UFLOAT_PACK32, VK_FORMAT_E5B9G9R9_UFLOAT_PACK32</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>D16 Block size 2 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_D16_UNORM</td>
</tr>
<tr>
<td>D24 Block size 4 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_X8_D24_UNORM_PACK32</td>
</tr>
<tr>
<td>D32 Block size 4 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_D32_SFLOAT</td>
</tr>
<tr>
<td>S8 Block size 1 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_S8_UINT</td>
</tr>
<tr>
<td>D16S8 Block size 3 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_D16_UNORM_S8_UINT</td>
</tr>
<tr>
<td>D24S8 Block size 4 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_D24_UNORM_S8_UINT</td>
</tr>
<tr>
<td>D32S8 Block size 5 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_D32_SFLOAT_S8_UINT</td>
</tr>
<tr>
<td>BC1_RGB 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>BC1_RGBA 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>BC2 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>BC3 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>BC4 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>BC5 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>BC6H 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>BC7 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>ETC2_RGB 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>ETC2_RGBA 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>ETC2_EAC_RGBA 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>EAC_R 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>EAC_RG 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>ASTC_4x4 Block size 16 byte 4x4x1 block extent 16 texel/block</td>
<td>VK_FORMAT_ASTC_4x4_UNORM_BLOCK, VK_FORMAT_ASTC_4x4_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_5x4 Block size 16 byte 5x4x1 block extent 20 texel/block</td>
<td>VK_FORMAT_ASTC_5x4_UNORM_BLOCK, VK_FORMAT_ASTC_5x4_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_5x5 Block size 16 byte 5x5x1 block extent 25 texel/block</td>
<td>VK_FORMAT_ASTC_5x5_UNORM_BLOCK, VK_FORMAT_ASTC_5x5_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_6x5 Block size 16 byte 6x5x1 block extent 30 texel/block</td>
<td>VK_FORMAT_ASTC_6x5_UNORM_BLOCK, VK_FORMAT_ASTC_6x5_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_6x6 Block size 16 byte 6x6x1 block extent 36 texel/block</td>
<td>VK_FORMAT_ASTC_6x6_UNORM_BLOCK, VK_FORMAT_ASTC_6x6_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_8x5 Block size 16 byte 8x5x1 block extent 40 texel/block</td>
<td>VK_FORMAT_ASTC_8x5_UNORM_BLOCK, VK_FORMAT_ASTC_8x5_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_8x6 Block size 16 byte 8x6x1 block extent 48 texel/block</td>
<td>VK_FORMAT_ASTC_8x6_UNORM_BLOCK, VK_FORMAT_ASTC_8x6_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_8x8 Block size 16 byte 8x8x1 block extent 64 texel/block</td>
<td>VK_FORMAT_ASTC_8x8_UNORM_BLOCK, VK_FORMAT_ASTC_8x8_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_10x5 Block size 16 byte 10x5x1 block extent 50 texel/block</td>
<td>VK_FORMAT_ASTC_10x5_UNORM_BLOCK, VK_FORMAT_ASTC_10x5_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_10x6 Block size 16 byte 10x6x1 block extent 60 texel/block</td>
<td>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_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_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_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_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_B10X6G10X6R10X6_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</td>
<td>VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16</td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>64-bit B12G12R12G12 Block size 8 byte 2x1x1 block extent</td>
<td>VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16</td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>12-bit 3-plane 420 Block size 6 byte 1x1x1 block extent</td>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16</td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>12-bit 2-plane 420 Block size 6 byte 1x1x1 block extent</td>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16</td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>12-bit 3-plane 422 Block size 6 byte 1x1x1 block extent</td>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16</td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>12-bit 2-plane 422 Block size 6 byte 1x1x1 block extent</td>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16</td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>12-bit 3-plane 444 Block size 6 byte 1x1x1 block extent</td>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16</td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>64-bit G16B16G16R16 Block size 8 byte 2x1x1 block extent</td>
<td>VK_FORMAT_G16B16G16R16_422_UNORM</td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>64-bit B16G16R16G16 Block size 8 byte 2x1x1 block extent</td>
<td>VK_FORMAT_B16G16R16G16_422_UNORM</td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>16-bit 3-plane 420 Block size 6 byte 1x1x1 block extent</td>
<td>VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM</td>
</tr>
<tr>
<td>1 texel/block</td>
<td></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>
</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

\[
\lceil A/a \rceil = \lceil X/x \rceil \\
\lceil B/b \rceil = \lceil Y/y \rceil \\
\lceil C/c \rceil = \lceil Z/z \rceil
\]

**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...
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.

### 33.2. 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; // A bitmask of VkFormatFeatureFlagBits specifying features supported by images created with a tiling parameter of VK_IMAGE_TILING_LINEAR.
    VkFormatFeatureFlags optimalTilingFeatures; // A bitmask of VkFormatFeatureFlagBits specifying features supported by images created with a tiling parameter of VK_IMAGE_TILING_OPTIMAL.
    VkFormatFeatureFlags bufferFeatures; // A bitmask of VkFormatFeatureFlagBits specifying features supported by buffers.
} VkFormatProperties;
```
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
    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,
};
```
These values 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 `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.
- **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 the `srcImage` for the `vkCmdBlitImage` command.
- **VK_FORMAT_FEATURE_BLIT_DST_BIT** specifies that an image can be used as the `dstImage` for the `vkCmdBlitImage` command.
- **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT** specifies that if `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT` is also set, an image view can be used with a sampler that has either of `magFilter` or `minFilter` set to `VK_FILTER_LINEAR`, or `mipmapMode` set to `VK_SAMPLER_MIPMAP_MODE_LINEAR`. If `VK_FORMAT_FEATURE_BLIT_SRC_BIT` is also set, an image can be used as the `srcImage` to `vkCmdBlitImage` with a filter of `VK_FILTER_LINEAR`. This bit must only be exposed for formats that also support the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT` or `VK_FORMAT_FEATURE_BLIT_SRC_BIT`.

If the format being queried is a depth/stencil format, this bit only specifies that the depth aspect (not the stencil aspect) of an image of this format supports linear filtering, and that linear filtering of the depth aspect is supported whether depth compare is enabled in the sampler or not. 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 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_SAMPLED_IMAGE_BIT.

- **VK_FORMAT_FEATURE_MIDPOINT_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_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_bC_r 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_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_COSITED_CHROMA_SAMPLES_BIT nor VK_FORMAT_FEATURE_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_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_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**).

**Note**

**VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT** and **VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT** are only intended to be advertised for single-component formats, since SPIR-V atomic operations require a scalar type.

```c
// 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:

```c
// 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 a VkStructureType value identifying 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

### 33.2.1. Potential Format Features

Some valid usage conditions depend on the format features supported by a VkImage whose VkImageTiling is unknown. In such cases the exact VkFormatFeatureFlagBits supported by the VkImage cannot be determined, so the valid usage conditions are expressed in terms of the potential format features of the VkImage format.

The potential format features of a VkFormat are defined as follows:

- The union of VkFormatFeatureFlagBits supported when the VkImageTiling is VK_IMAGE_TILING_OPTIMAL or VK_IMAGE_TILING_LINEAR

### 33.3. Required Format Support

Implementations must support at least the following set of features on the listed formats. For images, these features must be supported for every VkImageType (including arrayed and cube variants) unless otherwise noted. These features are supported on existing formats without needing to advertise an extension or needing to explicitly enable them. Support for additional functionality beyond the requirements listed here is queried using the vkGetPhysicalDeviceFormatProperties command.
Unless otherwise excluded below, the required formats are supported for all VkImageCreateFlags values as long as those flag values are otherwise allowed.

The following tables show which feature bits must be supported for each format. Formats that are required to support VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT must also support VK_FORMAT_FEATURE_TRANSFER_SRC_BIT and VK_FORMAT_FEATURE_TRANSFER_DST_BIT.

**Table 40. Key for format feature tables**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>This feature must be supported on the named format</td>
</tr>
<tr>
<td>†</td>
<td>This feature must be supported on at least some of the named formats, with more information in the table where the symbol appears</td>
</tr>
<tr>
<td>‡</td>
<td>This feature must be supported with some caveats or preconditions, with more information in the table where the symbol appears</td>
</tr>
</tbody>
</table>

**Table 41. Feature bits in optimalTilingFeatures**

<table>
<thead>
<tr>
<th>Feature Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_FEATURE_TRANSFER_SRC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_TRANSFER_DST_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_SRC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_DST_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT</td>
</tr>
</tbody>
</table>

**Table 42. Feature bits in bufferFeatures**

<table>
<thead>
<tr>
<th>Feature Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
</tr>
</tbody>
</table>
Table 43. Mandatory format support: sub-byte 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_UNDEFINED</td>
<td>☑</td>
<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_R4G4_UNORM_PACK8</td>
<td>☑</td>
<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_R4G4B4A4_UNORM_PACK16</td>
<td>☑</td>
<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_B4G4R4A4_UNORM_PACK16</td>
<td>☑</td>
<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_R5G6B5_UNORM_PACK16</td>
<td>☑</td>
<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_R5G5B5A1_UNORM_PACK16</td>
<td>☑</td>
<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_B5G6R5A1_UNORM_PACK16</td>
<td>☑</td>
<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_R5G5B5A1_UNORM_PACK16</td>
<td>☑</td>
<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_B5G5R5A1_UNORM_PACK16</td>
<td>☑</td>
<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_A1R5G5B5_UNORM_PACK16</td>
<td>☑</td>
<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>
Table 44. Mandatory format support: 1-3 byte-sized components

| Format                        | VK_FORMAT_R8_UNORM | VK_FORMAT_R8_SNORM | VK_FORMAT_R8_USCALED | VK_FORMAT_R8_SSCALED | VK_FORMAT_R8_UINT | VK_FORMAT_R8_SINT | VK_FORMAT_R8_SRGBA | VK_FORMAT_R8_G8B8_UNORM | VK_FORMAT_R8_G8B8_SNORM | VK_FORMAT_R8_G8B8_USCALED | VK_FORMAT_R8_G8B8_SSCALED | VK_FORMAT_R8_G8B8_UINT | VK_FORMAT_R8_G8B8_SINT | VK_FORMAT_R8_G8B8_SRGB | VK_FORMAT_R8888_UNORM | VK_FORMAT_R8888_SNORM | VK_FORMAT_R8888_USCALED | VK_FORMAT_R8888_SSCALED | VK_FORMAT_R8888_UINT | VK_FORMAT_R8888_SINT | VK_FORMAT_R8888_SRGB | VK_FORMAT_B8G8R8_UNORM | VK_FORMAT_B8G8R8_SNORM | VK_FORMAT_B8G8R8_SINT | VK_FORMAT_B8G8R8_SRGB |
|-------------------------------|---------------------|---------------------|-----------------------|-----------------------|-------------------|-------------------|----------------------|-----------------------------|------------------------------|--------------------------|-----------------------------|--------------------------|--------------------------|--------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|
| VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT | ✓                   | ✓                   | ✓                    | ✓                    | ✓                 | ✓                 | ✓                    | ✓                            | ✓                            | ✓                          | ✓                            | ✓                          | ✓                          | ✓                          | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        |
| VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT | ✓                   | ✓                   | ✓                    | ✓                    | ✓                 | ✓                 | ✓                    | ✓                            | ✓                            | ✓                          | ✓                            | ✓                          | ✓                          | ✓                          | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        |
| VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT | ✓                   | ✓                   | ✓                    | ✓                    | ✓                 | ✓                 | ✓                    | ✓                            | ✓                            | ✓                          | ✓                            | ✓                          | ✓                          | ✓                          | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        |
| VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT | ✓                   | ✓                   | ✓                    | ✓                    | ✓                 | ✓                 | ✓                    | ✓                            | ✓                            | ✓                          | ✓                            | ✓                          | ✓                          | ✓                          | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        |
| VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT | ✓                   | ✓                   | ✓                    | ✓                    | ✓                 | ✓                 | ✓                    | ✓                            | ✓                            | ✓                          | ✓                            | ✓                          | ✓                          | ✓                          | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        |
| VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT | ✓                   | ✓                   | ✓                    | ✓                    | ✓                 | ✓                 | ✓                    | ✓                            | ✓                            | ✓                          | ✓                            | ✓                          | ✓                          | ✓                          | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        |
| VK_FORMAT_FEATURE_BLIT_DST_BIT | ✓                   | ✓                   | ✓                    | ✓                    | ✓                 | ✓                 | ✓                    | ✓                            | ✓                            | ✓                          | ✓                            | ✓                          | ✓                          | ✓                          | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        |
| VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT | ✓                   | ✓                   | ✓                    | ✓                    | ✓                 | ✓                 | ✓                    | ✓                            | ✓                            | ✓                          | ✓                            | ✓                          | ✓                          | ✓                          | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        |
| VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT | ✓                   | ✓                   | ✓                    | ✓                    | ✓                 | ✓                 | ✓                    | ✓                            | ✓                            | ✓                          | ✓                            | ✓                          | ✓                          | ✓                          | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        |
| VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT | ✓                   | ✓                   | ✓                    | ✓                    | ✓                 | ✓                 | ✓                    | ✓                            | ✓                            | ✓                          | ✓                            | ✓                          | ✓                          | ✓                          | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        |
| VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT | ✓                   | ✓                   | ✓                    | ✓                    | ✓                 | ✓                 | ✓                    | ✓                            | ✓                            | ✓                          | ✓                            | ✓                          | ✓                          | ✓                          | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        |
| VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT | ✓                   | ✓                   | ✓                    | ✓                    | ✓                 | ✓                 | ✓                    | ✓                            | ✓                            | ✓                          | ✓                            | ✓                          | ✓                          | ✓                          | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        | ✓                        |

1002
<table>
<thead>
<tr>
<th>Format Features</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_B8G8R8_USCALED</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8_SSCALED</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8_UINT</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8_SINT</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8_SRGB</td>
<td></td>
</tr>
</tbody>
</table>

Format features marked with ‡ must be supported for optimalTilingFeatures if the VkPhysicalDevice supports the shaderStorageImageExtendedFormats feature.
<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>
<th>VK_FORMAT_FEATURE_SAMPLED_IMAGE_ATOMIC_BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_R8G8B8A8_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>
<td>✗</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8A8_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>
<td>✗</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8A8_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>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8A8_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>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8A8_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>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8A8_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>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8A8_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>
<td>✗</td>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8A8_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>
<td>✗</td>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8A8_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>
<td>✗</td>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8A8_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>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8A8_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>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8A8_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>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8A8_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>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8A8_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>
<td>✗</td>
</tr>
<tr>
<td>VK_FORMAT_A8B8G8R8_UNORM_PACK32</td>
<td>✗</td>
<td>✗</td>
<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_A8B8G8R8_SNORM_PACK32</td>
<td>✗</td>
<td>✗</td>
<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_A8B8G8R8_USCALED_PACK32</td>
<td>✗</td>
<td>✗</td>
<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_A8B8G8R8_SSCALED_PACK32</td>
<td>✗</td>
<td>✗</td>
<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_A8B8G8R8_UINT_PACK32</td>
<td>✗</td>
<td>✗</td>
<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_A8B8G8R8_SINT_PACK32</td>
<td>✗</td>
<td>✗</td>
<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_A8B8G8R8_SRGB_PACK32</td>
<td>✗</td>
<td>✗</td>
<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>
Table 46. Mandatory format support: 10- and 12-bit components

<table>
<thead>
<tr>
<th>Format Features</th>
<th>hyperlink VK_FORMAT_A2R10G10B10_UNORM_PACK32</th>
<th>hyperlink VK_FORMAT_A2R10G10B10_SNORM_PACK32</th>
<th>hyperlink VK_FORMAT_A2R10G10B10_USCALED_PACK32</th>
<th>hyperlink VK_FORMAT_A2R10G10B10_SSCALED_PACK32</th>
<th>hyperlink VK_FORMAT_A2R10G10B10_UINT_PACK32</th>
<th>hyperlink VK_FORMAT_A2R10G10B10_SINT_PACK32</th>
<th>hyperlink VK_FORMAT_A2B10G10R10_UNORM_PACK32</th>
<th>hyperlink VK_FORMAT_A2B10G10R10_SNORM_PACK32</th>
<th>hyperlink VK_FORMAT_A2B10G10R10_USCALED_PACK32</th>
<th>hyperlink VK_FORMAT_A2B10G10R10_SSCALED_PACK32</th>
<th>hyperlink VK_FORMAT_A2B10G10R10_UINT_PACK32</th>
<th>hyperlink VK_FORMAT_A2B10G10R10_SINT_PACK32</th>
<th>hyperlink VK_FORMAT_R10X6_UNORM_PACK16</th>
<th>hyperlink VK_FORMAT_R10X6G10X6_UNORM_2PACK16</th>
<th>hyperlink VK_FORMAT_R12X4_UNORM_PACK16</th>
<th>hyperlink VK_FORMAT_R12X4G12X4_UNORM_2PACK16</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>
<td>✗</td>
<td>✗</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>
<td>✗</td>
<td>✗</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>
<td>✗</td>
<td>✗</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>
<td>✗</td>
<td>✗</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>
<td>✗</td>
<td>✗</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>
<td>✗</td>
<td>✗</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>
<td>✗</td>
<td>✗</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>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<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_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<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_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<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_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<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 optimal Tiling Features if the VkPhysicalDevice supports the shaderStorageImageExtendedFormats feature.
Table 47. 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>
</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>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<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_FEATURE_STORAGE_TEXEL_BUFFER_BIT</td>
<td>✓</td>
<td>✧</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<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_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</td>
<td>✓</td>
<td>✧</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<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_FEATURE_VERTEX_BUFFER_BIT</td>
<td>✓</td>
<td>✧</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<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_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
<td>✓</td>
<td>✧</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<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_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</td>
<td>✓</td>
<td>✧</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<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_FEATURE_BLIT_DST_BIT</td>
<td>✓</td>
<td>✧</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<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_FEATURE_COLOR_ATTACHMENT_BIT</td>
<td>✓</td>
<td>✧</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<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_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</td>
<td>✓</td>
<td>✧</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<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_FEATURE_STORAGE_IMAGE_BIT</td>
<td>✓</td>
<td>✧</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<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_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</td>
<td>✧</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<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_FEATURE_BLIT_SRC_BIT</td>
<td>✓</td>
<td>✧</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<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_FEATURE_SAMPLED_IMAGE_BIT</td>
<td>✓</td>
<td>✧</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<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</td>
<td>Supported Features</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>--------</td>
<td>--------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>
</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>
</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>
</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>
</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>
</tr>
</tbody>
</table>

Format features marked with ‡ must be supported for optimal Tiling Features if the VkPhysicalDevice supports the shaderStorageImageExtendedFormats feature.
Table 48. Mandatory format support: 32-bit 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_R32_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_R32_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_R32_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>
</tr>
<tr>
<td>VK_FORMAT_R32G32_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_R32G32_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_R32G32_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>
</tr>
<tr>
<td>VK_FORMAT_R32G32B32_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_R32G32B32_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_R32G32B32_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>
</tr>
<tr>
<td>VK_FORMAT_R32G32B32A32_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_R32G32B32A32_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_R32G32B32A32_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>
</tr>
</tbody>
</table>
### Table 49. Mandatory format support: 64-bit/uneven 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_R64_UINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</td>
<td>VK_FORMAT_R64_SINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</td>
<td>VK_FORMAT_R64_SFLOAT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT</td>
<td>VK_FORMAT_R64G64_UINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
<td>VK_FORMAT_R64G64_SINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</td>
<td>VK_FORMAT_R64G64SINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_DST_BIT</td>
<td>VK_FORMAT_R64G64B64_UINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</td>
<td>VK_FORMAT_R64G64B64_SINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</td>
<td>VK_FORMAT_R64G64B64A64_UINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT</td>
<td>VK_FORMAT_R64G64B64A64_SINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</td>
<td>VK_FORMAT_B10G11R11_UFLOAT_PACK32</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_SRC_BIT</td>
<td>VK_FORMAT_E5B9G9R9_UFLOAT_PACK32</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: depth/stencil with 

<table>
<thead>
<tr>
<th>Format</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_D16_UNORM</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_X8_D24_UNORM_PACK32</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_D32_SFLOAT</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_S8_UINT</td>
<td>➥</td>
</tr>
<tr>
<td>VK_FORMAT_D16_UNORM_S8_UINT</td>
<td>➥</td>
</tr>
<tr>
<td>VK_FORMAT_D24_UNORM_S8_UINT</td>
<td>➥</td>
</tr>
<tr>
<td>VK_FORMAT_D32_SFLOAT_S8_UINT</td>
<td>➥</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
<td>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.</td>
</tr>
<tr>
<td>bufferFeatures</td>
<td>must not support any features for these formats</td>
</tr>
</tbody>
</table>
Table 51. Mandatory format support: BC compressed formats with \texttt{VkImageType VK\_IMAGE\_TYPE\_2D} and \texttt{VK\_IMAGE\_TYPE\_3D}

<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_BC1_RGB_UNORM_BLOCK †††</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_BC1_RGB_SRGB_BLOCK †††</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_BC1_RGBA_UNORM_BLOCK †††</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_BC1_RGBA_SRGB_BLOCK †††</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_BC2_UNORM_BLOCK †††</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_BC2_SRGB_BLOCK †††</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_BC3_UNORM_BLOCK †††</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_BC3_SRGB_BLOCK †††</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_BC4_UNORM_BLOCK †††</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_BC4_SNORM_BLOCK †††</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_BC5_UNORM_BLOCK †††</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_BC5_SNORM_BLOCK †††</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_BC6H_UFLOAT_BLOCK †††</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_BC6H_SFLOAT_BLOCK †††</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_BC7_UNORM_BLOCK †††</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_BC7_SRGB_BLOCK †††</td>
</tr>
</tbody>
</table>

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 must be supported in \texttt{optimalTilingFeatures} for all the formats in at least one of: this table, Mandatory format support: ETC2 and EAC compressed formats with \texttt{VkImageType VK\_IMAGE\_TYPE\_2D}, or Mandatory format support: ASTC LDR compressed formats with \texttt{VkImageType VK\_IMAGE\_TYPE\_2D}.
<table>
<thead>
<tr>
<th>Format</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK</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>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK</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>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK</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>
</tr>
<tr>
<td>VK_FORMAT_EAC_R11_SNORM_BLOCK</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>
</tr>
<tr>
<td>VK_FORMAT_EAC_R11G11_SNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></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: ASTC LDR compressed formats with VkImageType VK_IMAGE_TYPE_2D**.
Table 53. Mandatory format support: ASTC LDR compressed formats with `VkImageType VK_IMAGE_TYPE_2D`

<table>
<thead>
<tr>
<th>Format</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VK_FORMAT_ASTC_4x4_UNORM_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_4x4_SRGB_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_5x4_UNORM_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_5x4_SRGB_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_5x5_UNORM_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_5x5_SRGB_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_6x5_UNORM_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_6x5_SRGB_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_6x6_UNORM_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_6x6_SRGB_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_8x5_UNORM_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_8x5_SRGB_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_8x6_UNORM_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_8x6_SRGB_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_8x8_UNORM_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_8x8_SRGB_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_10x5_UNORM_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_10x5_SRGB_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_10x6_UNORM_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_10x6_SRGB_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>VK_FORMAT_ASTC_10x8_UNORM_BLOCK</code></td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></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 54. 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_B8G8_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_2PLANE_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 YCbCr conversion. To determine whether the
implementation supports sparse image creation flags with these formats use

33.3.1. Formats Without Shader Storage Format

The device-level features for using a storage image or a storage texel buffer 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
Note
This list of formats is the union of required storage formats from Required Format Support section and formats listed in shaderStorageImageExtendedFormats.

33.3.2. Format Feature Dependent Usage Flags

Certain resource usage flags depend on support for the corresponding format feature flag for the format in question. The following tables list the VkBufferUsageFlagBits and VkImageUsageFlagBits that have such dependencies, and the format feature flags they depend on. Additional restrictions, including, but not limited to, further required format feature flags specific to the particular use of the resource may apply, as described in the respective sections of this specification.

Table 55. Format feature dependent buffer usage flags

<table>
<thead>
<tr>
<th>Buffer usage flag</th>
<th>Required format feature flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT</td>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</td>
</tr>
<tr>
<td>VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT</td>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</td>
</tr>
<tr>
<td>VK_BUFFER_USAGE_VERTEX_BUFFER_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT</td>
</tr>
</tbody>
</table>

Table 56. Format feature dependent image usage flags

<table>
<thead>
<tr>
<th>Image usage flag</th>
<th>Required format feature flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_IMAGE_USAGE_SAMPLED_BIT</td>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT</td>
</tr>
<tr>
<td>VK_IMAGE_USAGE_STORAGE_BIT</td>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT</td>
</tr>
<tr>
<td>VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</td>
</tr>
<tr>
<td>VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
</tr>
<tr>
<td>VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT or VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
</tr>
</tbody>
</table>
Chapter 34. 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

34.1. 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 `format` is not a supported image format, or if the combination of `format`, `type`, `tiling`, `usage`, and `flags` is not supported for images, then `vkGetPhysicalDeviceImageFormatProperties` returns `VK_ERROR_FORMAT_NOT_SUPPORTED`.

The limitations on an image format that are reported by `vkGetPhysicalDeviceImageFormatProperties` have the following property: if `usage1` and `usage2` of type `VkImageUsageFlags` are such that the bits set in `usage1` are a subset of the bits set in `usage2`, and `flags1` and `flags2` of type `VkImageCreateFlags` are such that the bits set in `flags1` are a subset of the bits set in `flags2`, then the limitations for `usage1` and `flags1` must be no more strict than the limitations for `usage2` and `flags2`, for all values of `format`, `type`, and `tiling`.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkGetPhysicalDeviceImageFormatProperties` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

**Valid Usage (Implicit)**

- VUID-vkGetPhysicalDeviceImageFormatProperties-physicalDevice-parameter `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceImageFormatProperties-format-parameter `format` must be a valid `VkFormat` value
- VUID-vkGetPhysicalDeviceImageFormatProperties-type-parameter `type` must be a valid `VkImageType` value
- VUID-vkGetPhysicalDeviceImageFormatProperties-tiling-parameter `tiling` must be a valid `VkImageTiling` value
- VUID-vkGetPhysicalDeviceImageFormatProperties-usage-parameter `usage` must be a valid combination of `VkImageUsageFlagBits` values
- VUID-vkGetPhysicalDeviceImageFormatProperties-usage-requiredbitsetmask `usage` must not be 0
- VUID-vkGetPhysicalDeviceImageFormatProperties-flags-parameter `flags` must be a valid combination of `VkImageCreateFlagBits` values
- VUID-vkGetPhysicalDeviceImageFormatProperties-pImageFormatProperties-parameter `pImageFormatProperties` must be a valid pointer to a `VkImageFormatProperties` structure

**Return Codes**

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OFDEVICE_MEMORY`
- `VK_ERROR_FORMAT_NOT_SUPPORTED`
The `VkImageFormatProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_0
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:

```c
// 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.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkGetPhysicalDeviceImageFormatProperties2` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

**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`
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 a `VkStructureType` value identifying 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_PHYSICAL_DEVICE_IMAGE_FORMAT_INFO_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:

```c
// Provided by VK_VERSION_1_1
typedef struct VkImageFormatProperties2 {
    VkStructureType sType;
    void* pNext;
    VkImageFormatProperties imageFormatProperties;
} VkImageFormatProperties2;
```

- sType is a VkStructureType value identifying 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 a VkStructureType value identifying 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
Possible values of `VkPhysicalDeviceExternalImageFormatInfo::handleType`, specifying an external memory handle type, are:

```c
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>

// 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:

// Provided by VK_VERSION_1_1
typedef struct VkExternalImageFormatProperties {
    VkStructureType sType;
    void* pNext;
    VkExternalMemoryProperties externalMemoryProperties;
} VkExternalImageFormatProperties;

- *sType* is a *VkStructureType* value identifying 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
sType must be VK_STRUCTURE_TYPE_EXTERNAL_IMAGE_FORMAT_PROPERTIES

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`

```c
// 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 `VkImageFormatProperties2` structure in a call to `vkGetPhysicalDeviceImageFormatProperties2`.

The `VkSamplerYcbcrConversionImageFormatProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkSamplerYcbcrConversionImageFormatProperties {
    VkStructureType sType;
    void* pNext;
    uint32_t combinedImageSamplerDescriptorCount;
} VkSamplerYcbcrConversionImageFormatProperties;
```

- `sType` is a `VkStructureType` value identifying 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.

### 34.1.1. 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 component, a superset of `VkPhysicalDeviceLimits::framebufferDepthSampleCounts`
- If `usage` includes `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, and `format` includes a stencil component, a superset of `VkPhysicalDeviceLimits::framebufferStencilSampleCounts`
- If `usage` includes `VK_IMAGE_USAGE_SAMPLED_BIT`, and `format` includes a color component, a superset of `VkPhysicalDeviceLimits::sampledImageColorSampleCounts`
• If usage includes `VK_IMAGE_USAGE_SAMPLED_BIT`, and format includes a depth component, 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`.

### 34.1.2. 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 ≥ VkPhysicalDeviceLimits::maxImageDimension1D`
- `maxExtent.height = 1`
- `maxExtent.depth = 1`

For `VK_IMAGE_TYPE_2D` when flags does not contain `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`:

- `maxExtent.width ≥ VkPhysicalDeviceLimits::maxImageDimension2D`
- `maxExtent.height ≥ VkPhysicalDeviceLimits::maxImageDimension2D`
- `maxExtent.depth = 1`

For `VK_IMAGE_TYPE_2D` when flags contains `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`:

- `maxExtent.width ≥ VkPhysicalDeviceLimits::maxImageDimensionCube`
- `maxExtent.height ≥ VkPhysicalDeviceLimits::maxImageDimensionCube`
- `maxExtent.depth = 1`

For `VK_IMAGE_TYPE_3D`:

- `maxExtent.width ≥ VkPhysicalDeviceLimits::maxImageDimension3D`
• \( \text{maxExtent.height} \geq \text{VkPhysicalDeviceLimits::maxImageDimension3D} \)
• \( \text{maxExtent.depth} \geq \text{VkPhysicalDeviceLimits::maxImageDimension3D} \)

## 34.2. 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 a `VkStructureType` value identifying 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.

Only usage flags representable in **VkBufferUsageFlagBits** are returned in this structure’s **usage**.

---

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceExternalBufferInfo-sType-sType
  - **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_BUFFER_INFO**

- VUID-VkPhysicalDeviceExternalBufferInfo-pNext-pNext
  - **pNext** must be NULL

- VUID-VkPhysicalDeviceExternalBufferInfo-flags-parameter
  - **flags** must be a valid combination of **VkBufferCreateFlagBits** values

- VUID-VkPhysicalDeviceExternalBufferInfo-usage-parameter
  - **usage** must be a valid combination of **VkBufferUsageFlagBits** values

- VUID-VkPhysicalDeviceExternalBufferInfo-usage-requiredbitmap
  - **usage** must not be 0

- VUID-VkPhysicalDeviceExternalBufferInfo-handleType-parameter
  - **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 a **VkStructureType** value identifying 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)

- VUID-VkExternalBufferProperties-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_EXTERNAL_BUFFER_PROPERTIES`.

- VUID-VkExternalBufferProperties-pNext-pNext
  
  `pNext` must be `NULL`.

34.3. 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 a **VkStructureType** value identifying 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**
  
  The **sType** value of each struct in the **pNext** chain must be unique.

- **VUID-VkPhysicalDeviceExternalSemaphoreInfo-pNext-pNext**
  
  **pNext** must be **NULL** or a pointer to a valid instance of **VkSemaphoreTypeCreateInfo**.

- **VUID-VkPhysicalDeviceExternalSemaphoreInfo-sType-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 =
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_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.

1034
- **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 58. External semaphore 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_SEMAPHOREHANDLE_TYPE_OPAQUE_FD_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHOREHANDLE_TYPE_OPAQUE_WIN32_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHOREHANDLE_TYPE_OPAQUE_WIN32_KMT_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHOREHANDLE_TYPE_D3D12_FENCE_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHOREHANDLE_TYPE_SYNC_FD_BIT</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHOREHANDLE_TYPE_ZIRCON_EVENT_BIT_FUCHSIA</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
</tbody>
</table>

```c
// 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:

```c
// Provided by VK_VERSION_1_1
typedef struct VkExternalSemaphoreProperties {
  VkStructureType sType;
  void*pNext;
  VkExternalSemaphoreHandleTypeFlags exportFromImportedHandleTypes;
  VkExternalSemaphoreHandleTypeFlags compatibleHandleTypes;
  VkExternalSemaphoreFeatureFlags externalSemaphoreFeatures;
} VkExternalSemaphoreProperties;
```

- `sType` is a `VkStructureType` value identifying 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`. 

1036
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`.

### 34.4. 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 a `VkStructureType` value identifying 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
// Provided by VK_VERSION_1_1
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_BIT</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:

```cpp
// 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.

```cpp
// Provided by VK_VERSION_1_1
typedef VkFlags VkExternalFenceFeatureFlags;
```

`VkExternalFenceFeatureFlags` is a bitmask type for setting a mask of zero or more `VkExternalFenceFeatureFlagBits`. 
Chapter 35. 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,
} 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>
<tr>
<td>VK_OBJECT_TYPE_INSTANCE</td>
<td>VkInstance</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_PHYSICAL_DEVICE</td>
<td>VkPhysicalDevice</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_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>VkCommandBuffer</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_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>
</tbody>
</table>

If this Specification was generated with any such extensions included, they will be described in the remainder of this chapter.

### 35.1. Fault Handling

The fault handling mechanism provides a method for the implementation to pass fault information to the application. A fault indicates that an issue has occurred with the host or device that could impact the implementation's ability to function correctly. It consists of a `VkFaultData` structure that is used to communicate information about the fault between the implementation and the application, with two methods to obtain the data. The application **can** obtain the fault data from the implementation using `vkGetFaultData`. Alternatively, the implementation **can** directly call a pre-registered fault handler function (`PFN_vkFaultCallbackFunction`) in the application when a fault occurs.
The `VkFaultData` structure provides categories the implementation must set to provide basic information on a fault. These allow the implementation to provide a coarse classification of a fault to the application. As the potential faults that could occur will vary between different platforms, it is expected that an implementation would also provide additional implementation-specific data on the fault, enabling the application to take appropriate action.

The implementation must also define whether a particular fault results in the fault callback function being called, is communicated via `vkGetFaultData`, or both. This will be decided by several factors including:

- the severity of the fault,
- the application’s ability to handle the fault, and
- how the application should handle the fault.

The implementation must document the implementation-specific fault data, how the faults are communicated, and expected responses from the application for each of the faults that it can report.

### 35.1.1. Fault Data

The information on a single fault is returned using the `VkFaultData` structure. The `VkFaultData` structure is defined as:

```c
// Provided by VKSC_VERSION_1_0
typedef struct VkFaultData {
    VkStructureType sType;
    void* pNext;
    VkFaultLevel faultLevel;
    VkFaultType faultType;
} VkFaultData;
```

- `sType` is a `VkStructureType` value identifying this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure that provides implementation-specific data on the fault.
- `faultLevel` is a `VkFaultLevel` that provides the severity of the fault.
- `faultType` is a `VkFaultType` that provides the type of the fault.

To retrieve implementation-specific fault data, `pNext` can point to one or more implementation-defined fault structures or `NULL` to not retrieve implementation-specific data.
Valid Usage (Implicit)

- **VUID-VkFaultData-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_FAULT_DATA`

Possible values of `VkFaultData::faultLevel`, specifying the fault severity, are:

```c
// Provided by VKSC_VERSION_1_0
typedef enum VkFaultLevel {
    VK_FAULT_LEVEL_UNASSIGNED = 0,
    VK_FAULT_LEVEL_CRITICAL = 1,
    VK_FAULT_LEVEL_RECOVERABLE = 2,
    VK_FAULT_LEVEL_WARNING = 3,
} VkFaultLevel;
```

- `VK_FAULT_LEVEL_UNASSIGNED` A fault level has not been assigned.
- `VK_FAULT_LEVEL_CRITICAL` A fault that **cannot** be recovered by the application.
- `VK_FAULT_LEVEL_RECOVERABLE` A fault that **can** be recovered by the application.
- `VK_FAULT_LEVEL_WARNING` A fault that indicates a non-optimal condition has occurred, but no recovery is necessary at this point.

Possible values of `VkFaultData::faultType`, specifying the fault type, are:

```c
// Provided by VKSC_VERSION_1_0
typedef enum VkFaultType {
    VK_FAULT_TYPE_INVALID = 0,
    VK_FAULT_TYPE_UNASSIGNED = 1,
    VK_FAULT_TYPE_IMPLEMENTATION = 2,
    VK_FAULT_TYPE_SYSTEM = 3,
    VK_FAULT_TYPE_PHYSICAL_DEVICE = 4,
    VK_FAULT_TYPE_COMMAND_BUFFER_FULL = 5,
    VK_FAULT_TYPE_INVALID_API_USAGE = 6,
} VkFaultType;
```

- `VK_FAULT_TYPE_INVALID` The fault data does not contain a valid fault.
- `VK_FAULT_TYPE_UNASSIGNED` A fault type has not been assigned.
- `VK_FAULT_TYPE_IMPLEMENTATION` Implementation-defined fault.
- `VK_FAULT_TYPE_SYSTEM` A fault occurred in the system components.
- `VK_FAULT_TYPE_PHYSICAL_DEVICE` A fault occurred with the physical device.
- `VK_FAULT_TYPE_COMMAND_BUFFER_FULL` Command buffer memory was exhausted before `vkEndCommandBuffer` was called.
- `VK_FAULT_TYPE_INVALID_API_USAGE` Invalid usage of the API was detected by the implementation.
35.1.2. Querying Fault Status

To query the number of current faults and obtain the fault data, call `vkGetFaultData`.

```c
// Provided by VKSC_VERSION_1_0
VkResult vkGetFaultData(
    VkDevice device,
    VkFaultQueryBehavior faultQueryBehavior,
    VkBool32* pUnrecordedFaults,
    uint32_t* pFaultCount,
    VkFaultData* pFaults);
```

- `device` is the logical device to obtain faults from.
- `faultQueryBehavior` is a `VkFaultQueryBehavior` that specifies the types of faults to obtain from the implementation, and how those faults should be handled.
- `pUnrecordedFaults` is a return boolean that specifies if the logged fault information is incomplete and does not contain entries for all faults that have been detected by the implementation and may be reported via `vkGetFaultData`.
- `pFaultCount` is a pointer to an integer that specifies the number of fault entries.
- `pFaults` is either `NULL` or a pointer to an array of `pFaultCount` `VkFaultData` structures to be updated with the recorded fault data.

Access to fault data is internally synchronized, meaning `vkGetFaultData` can be called from multiple threads simultaneously.

The implementation must not record more than `maxQueryFaultCount` faults to be reported by `vkGetFaultData`.

`pUnrecordedFaults` is set to `VK_TRUE` if the implementation has detected one or more faults since the last successful retrieval of fault data using this command, but was unable to record fault information for all faults. Otherwise, `pUnrecordedFaults` is set to `VK_FALSE`.

If `pFaults` is `NULL`, then the number of faults with the specified `faultQueryBehavior` characteristics associated with `device` is returned in `pFaultCount`, and `pUnrecordedFaults` is set as indicated above. Otherwise, `pFaultCount` must point to a variable set by the user to the number of elements in the `pFaults` array, and on return the variable is overwritten with the number of faults actually written to `pFaults`. If `pFaultCount` is less than the number of recorded `device` faults with the specified `faultQueryBehavior` characteristics, at most `pFaultCount` faults will be written, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available faults were returned.

On success, the fault information stored by the implementation for the faults that were returned will be handled as specified by `faultQueryBehavior`.

For each filled `pFaults` entry, if `pNext` is not `NULL`, the implementation will fill in any implementation-specific structures applicable to that fault that are included in the `pNext` chain.

**Note**
In order to simplify the application logic, an application could have a static allocation sized to \texttt{maxQueryFaultCount} which it passes in to each call of \texttt{vkGetFaultData}. This allows an application to obtain all the faults available at this time in a single call to \texttt{vkGetFaultData}. Furthermore, under this usage pattern, the command will never return \texttt{VK_INCOMPLETE}.

If \texttt{VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations} is \texttt{VK_TRUE}, \texttt{vkGetFaultData} must not return \texttt{VK_ERROR_OUT_OF_HOST_MEMORY}.

### Valid Usage

- \texttt{VUID-vkGetFaultData-pFaultCount-05020} \texttt{pFaultCount} must be less than or equal to \texttt{maxQueryFaultCount}

### Valid Usage (Implicit)

- \texttt{VUID-vkGetFaultData-device-parameter} \texttt{device} must be a valid \texttt{VkDevice} handle
- \texttt{VUID-vkGetFaultData-faultQueryBehavior-parameter} \texttt{faultQueryBehavior} must be a valid \texttt{VkFaultQueryBehavior} value
- \texttt{VUID-vkGetFaultData-pUnrecordedFaults-parameter} \texttt{pUnrecordedFaults} must be a valid pointer to a \texttt{VkBool32} value
- \texttt{VUID-vkGetFaultData-pFaultCount-parameter} \texttt{pFaultCount} must be a valid pointer to a \texttt{uint32_t} value
- \texttt{VUID-vkGetFaultData-pFaults-parameter} If the value referenced by \texttt{pFaultCount} is not \texttt{0}, and \texttt{pFaults} is not \texttt{NULL}, \texttt{pFaults} must be a valid pointer to an array of \texttt{VkFaultData} structures

### Return Codes

#### Success
- \texttt{VK_SUCCESS}
- \texttt{VK_INCOMPLETE}

#### Failure
- \texttt{VK_ERROR_OUT_OF_HOST_MEMORY}
- \texttt{VK_ERROR_OUT_OF_DEVICE_MEMORY}

Possible values that \texttt{can} be set in \texttt{VkFaultQueryBehavior}, specifying which faults to return, are:

```c
// Provided by VKSC_VERSION_1_0
typedef enum VkFaultQueryBehavior {
```
• **VK_FAULT_QUERY_BEHAVIOR_GET_AND_CLEAR_ALL_FAULTS** All fault types and severities are reported and are cleared from the internal fault storage after retrieval.

### 35.1.3. Fault Callback

The *VkFaultCallbackInfo* structure allows an application to register a function at device creation that the implementation can call to report faults when they occur. A callback function is registered by attaching a valid *VkFaultCallbackInfo* structure to the *pNext* chain of the *VkDeviceCreateInfo* structure. The callback function is only called by the implementation during a call to the API, using the same thread that is making the API call. The *VkFaultCallbackInfo* structure provides the function pointer to be called by the implementation, and optionally, application memory to store fault data.

The *VkFaultCallbackInfo* structure is defined as:

```c
// Provided by VK_SC_VERSION_1_0
typedef struct VkFaultCallbackInfo {
  VkStructureType sType;
  const void* pNext;
  uint32_t faultCount;
  VkFaultData* pFaults;
  PFN_vkFaultCallbackFunction pfnFaultCallback;
} VkFaultCallbackInfo;
```

- **sType** is a *VkStructureType* value identifying this structure.
- **pNext** is NULL or pointer to a structure extending this structure.
- **faultCount** is the number of reported faults in the array pointed to by **pFaults**.
- **pFaults** is either NULL or a pointer to an array of **faultCount** *VkFaultData* structures.
- **pfnFaultCallback** is a function pointer to the fault handler function that will be called by the implementation when a fault occurs.

If provided, the implementation may make use of the **pFaults** array to return fault data to the application when using the fault callback.

**Note**

Prior to Vulkan SC 1.0.11, the application was required to provide the **pFaults** array for fault callback data. This proved to be unwieldy for both applications and implementations and it was made optional as of version 1.0.11. It is expected that most implementations will ignore this and use stack or other preallocated memory for fault callback parameters.

If provided, the application memory referenced by **pFaults** must remain accessible throughout the lifetime of the logical device that was created with this structure.
The memory pointed to by \texttt{pFaults} will be updated by the implementation and should not be used or accessed by the application outside of the fault handling function pointed to by \texttt{pfnFaultCallback}. This restriction also applies to any implementation-specific structure chained to an element of \texttt{pFaults} by \texttt{pNext}.

It is expected that implementations will maintain separate storage for fault information and populate the array pointed to by \texttt{pFaults} ahead of calling the fault callback function.

---

### Valid Usage

- **VUID-VkFaultCallbackInfo-faultCount-05138**
  
  \texttt{faultCount} must either be 0, or equal to \texttt{VkPhysicalDeviceVulkanSC10Properties::maxCallbackFaultCount}.

---

### Valid Usage (Implicit)

- **VUID-VkFaultCallbackInfo-sType-sType**
  
  \texttt{sType} must be \texttt{VK_STRUCTURE_TYPE_FAULT_CALLBACK_INFO}.

- **VUID-VkFaultCallbackInfo-pFaults-parameter**
  
  If \texttt{faultCount} is not 0, and \texttt{pFaults} is not NULL, \texttt{pFaults} must be a valid pointer to an array of \texttt{VkFaultData} structures.

- **VUID-VkFaultCallbackInfo-pfnFaultCallback-parameter**
  
  \texttt{pfnFaultCallback} must be a valid \texttt{PFN_vkFaultCallbackFunction} value.

---

The function pointer \texttt{PFN_vkFaultCallbackFunction} is defined as:

```c
// Provided by VKSC_VERSION_1_0
typedef void (VKAPI_PTR *PFN_vkFaultCallbackFunction)(
    VkBool32 unrecordedFaults,
    uint32_t faultCount,
    const VkFaultData* pFaults);
```

- \texttt{unrecordedFaults} is a boolean that specifies if the supplied fault information is incomplete and does not contain entries for all faults that have been detected by the implementation and may be reported via \texttt{PFN_vkFaultCallbackFunction} since the last call to this callback.

- \texttt{faultCount} will contain the number of reported faults in the array pointed to by \texttt{pFaults}.

- \texttt{pFaults} will point to an array of \texttt{faultCount} \texttt{VkFaultData} structures containing the fault information.

An implementation must only make calls to \texttt{pfnFaultCallback} during the execution of an API command. An implementation must only make calls into the application-provided fault callback function.
from the same thread that called the API command. The implementation should not synchronize calls to the callback. If synchronization is needed, the callback must provide it.

The fault callback must not call any Vulkan commands.

It is implementation-dependent whether faults reported by this callback are also reported via vkGetFaultData, but each unique fault will be reported by at most one callback.
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 1.2 implementation **must** support the 1.0, 1.1, 1.2, 1.3, 1.4, and 1.5 versions of SPIR-V and the 1.0 version of the SPIR-V Extended Instructions for GLSL.

A SPIR-V module 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 select a pipeline cache entry, which was created by passing a SPIR-V module using any of these capabilities to the offline pipeline cache compiler, in a `vkCreate*Pipelines` command unless one of the following conditions is met for the `VkDevice` specified in the `device` parameter of the `vkCreate*Pipelines` command:

- 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>
</tbody>
</table>

Table 61. List of SPIR-V Capabilities and corresponding Vulkan features, extensions, or core version
### SPIR-V OpCapability

#### Vulkan feature, extension, or core version

<table>
<thead>
<tr>
<th>Feature</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image1D</td>
<td>VK_VERSION_1_0</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></td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceFeatures::geometryShader</td>
</tr>
<tr>
<td>Tessellation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceFeatures::tessellationShader</td>
</tr>
<tr>
<td>Float64</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceFeatures::shaderFloat64</td>
</tr>
<tr>
<td>Int64</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceFeatures::shaderInt64</td>
</tr>
<tr>
<td>Int64Atomics</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceVulkan12Features::shaderBufferInt64Atomics</td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceVulkan12Features::shaderSharedInt64Atomics</td>
</tr>
<tr>
<td>Int16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceFeatures::shaderInt16</td>
</tr>
<tr>
<td>TessellationPointSize</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceFeatures::shaderTessellationAndGeometryPointSize</td>
</tr>
<tr>
<td>GeometryPointSize</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceFeatures::shaderTessellationAndGeometryPointSize</td>
</tr>
<tr>
<td>ImageGatherExtended</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceFeatures::shaderImageGatherExtended</td>
</tr>
<tr>
<td>StorageImageMultisample</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceFeatures::shaderStorageImageMultisample</td>
</tr>
<tr>
<td>UniformBufferArrayDynamicIndexing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceFeatures::shaderUniformBufferArrayDynamicIndexing</td>
</tr>
<tr>
<td>SampledImageArrayDynamicIndexing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceFeatures::shaderSampledImageArrayDynamicIndexing</td>
</tr>
<tr>
<td>StorageBufferArrayDynamicIndexing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceFeatures::shaderStorageBufferArrayDynamicIndexing</td>
</tr>
</tbody>
</table>
### SPIR-V

#### OpCapability

**Vulkan feature, extension, or core version**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StorageImageArrayDynamicIndexing</td>
<td>VkPhysicalDeviceFeatures::shaderStorageImageArrayDynamicIndexing</td>
</tr>
<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></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>
</tbody>
</table>
SPIR-V OpCapability

Vulkan feature, extension, or core version

VariablePointers

VkPhysicalDeviceVulkan11Features::variablePointers

ShaderViewportIndex

VkPhysicalDeviceVulkan12Features::shaderOutputViewportIndex

ShaderLayer

VkPhysicalDeviceVulkan12Features::shaderOutputLayer

StorageBuffer16BitAccess

VkPhysicalDeviceVulkan11Features::storageBuffer16BitAccess

UniformAndStorageBuffer16BitAccess

VkPhysicalDeviceVulkan11Features::uniformAndStorageBuffer16BitAccess

StoragePushConstant16

VkPhysicalDeviceVulkan11Features::storagePushConstant16

StorageInputOutput16

VkPhysicalDeviceVulkan11Features::storageInputOutput16

GroupNonUniform

VK_SUBGROUP_FEATURE_BASIC_BIT

GroupNonUniformVote

VK_SUBGROUP_FEATURE_VOTE_BIT

GroupNonUniformArithmetic

VK_SUBGROUP_FEATURE_ARITHMETIC_BIT

GroupNonUniformBallot

VK_SUBGROUP_FEATURE_BALLOT_BIT

GroupNonUniformShuffle

VK_SUBGROUP_FEATURE_SHUFFLE_BIT

GroupNonUniformShuffleRelative

VK_SUBGROUP_FEATURE_SHUFFLE_RELATIVE_BIT

GroupNonUniformClustered

VK_SUBGROUP_FEATURE_CLUSTERED_BIT

GroupNonUniformQuad

VK_SUBGROUP_FEATURE_QUAD_BIT

ShaderNonUniform

VK_VERSION_1_2

RuntimeDescriptorArray

VkPhysicalDeviceVulkan12Features::runtimeDescriptorArray

InputAttachmentArrayDynamicIndexing

VkPhysicalDeviceVulkan12Features::shaderInputAttachmentArrayDynamicIndexing

UniformTexelBufferArrayDynamicIndexing

VkPhysicalDeviceVulkan12Features::shaderUniformTexelBufferArrayDynamicIndexing
<table>
<thead>
<tr>
<th>SPIR-V OpCapability</th>
<th>Vulkan feature, extension, or core version</th>
</tr>
</thead>
<tbody>
<tr>
<td>StorageTexelBufferArrayDynamicIndexing</td>
<td>VkPhysicalDeviceVulkan12Features::shaderStorageTexelBufferArrayDynamicIndexing</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>
The application must not select a pipeline cache entry, which was created by passing a SPIR-V module containing any of the following to the offline pipeline cache compiler, containing any of the following in a `vkCreate*Pipelines` command:

- 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 select a pipeline cache entry, which was created by passing a SPIR-V module using any of the following SPIR-V extensions to the offline pipeline cache compiler, in a `vkCreate*Pipelines` command unless one of the following conditions is met for the `VkDevice` specified in the `device` parameter of the `vkCreate*Pipelines` command:

- Any corresponding Vulkan extension is enabled.
- The corresponding core version is supported (as returned by `VkPhysicalDeviceProperties::apiVersion`).

<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>
</tbody>
</table>
Validation Rules Within a Module

Pipeline cache entries must have been compiled with the offline pipeline cache compiler using SPIR-V modules that 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

- VUID-StandaloneSpirv-None-04634
  The static function-call graph for an entry point must not contain cycles; that is, static recursion is not allowed

- VUID-StandaloneSpirv-None-04635
The Logical or PhysicalStorageBuffer64 addressing model must be selected

- VUID-StandaloneSpirv-None-04636
  Scope for execution must be limited to Workgroup or Subgroup

- VUID-StandaloneSpirv-None-04637
  If the Scope for execution is Workgroup, then it must only be used in the task, mesh, tessellation control, or compute Execution Model

- VUID-StandaloneSpirv-None-04638
  Scope for memory must be limited to Device, QueueFamily, Workgroup, ShaderCallKHR, Subgroup, or Invocation

- VUID-StandaloneSpirv-ExecutionModel-07320
  If the Execution Model is TessellationControl, and the MemoryModel is GLSL450, the Scope for memory must not be Workgroup

- VUID-StandaloneSpirv-None-07321
  If the Scope for memory is Workgroup, then it must only be used in the task, mesh, tessellation control, or compute Execution Model

- VUID-StandaloneSpirv-None-04640
  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 Model

- VUID-StandaloneSpirv-None-04641
  If the Scope for memory is Invocation, then memory semantics must be None

- VUID-StandaloneSpirv-None-04642
  Scope for group operations must be limited to Subgroup

- VUID-StandaloneSpirv-SubgroupVoteKHR-07951
  If none of the SubgroupVoteKHR, GroupNonUniform, or SubgroupBallotKHR capabilities are declared, Scope for memory must not be Subgroup

- VUID-StandaloneSpirv-None-04643
  Storage Class must be limited to UniformConstant, Input, Uniform, Output, Workgroup, Private, Function, PushConstant, Image, StorageBuffer, RayPayloadKHR, IncomingRayPayloadKHR, HitAttributeKHR, CallableDataKHR, IncomingCallableDataKHR, ShaderRecordBufferKHR, PhysicalStorageBuffer, or TileImageEXT

- VUID-StandaloneSpirv-None-04644
  If the Storage Class is Output, then it must not be used in the GlCompute, RayGenerationKHR, IntersectionKHR, AnyHitKHR, ClosestHitKHR, MissKHR, or CallableKHR Execution Model

- VUID-StandaloneSpirv-None-04645
  If the Storage Class is Workgroup, then it must only be used in the task, mesh, or compute Execution Model

- VUID-StandaloneSpirv-None-08720
  If the Storage Class is TileImageEXT, then it must only be used in the fragment execution model

- VUID-StandaloneSpirv-OpAtomicStore-04730
  OpAtomicStore must not use Acquire, AcquireRelease, or SequentiallyConsistent memory semantics
• VUID-StandaloneSpirv-OpAtomicLoad-04731
  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-OpControl Barrier-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-04734
  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 SPIR-V 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.

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.

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
  If a user-defined variable is not a pointer to a Block decorated OpTypeStruct, then the OpVariable must have a Location decoration

- VUID-StandaloneSpirv-Location-04918
  If a user-defined variable has a Location decoration, and the variable is a pointer to a OpTypeStruct, then the members of that structure must not have Location decorations

- VUID-StandaloneSpirv-Location-04919
  If a user-defined variable does not have a Location decoration, and the variable is a pointer to a Block decorated OpTypeStruct, then each member of the struct must have a Location decoration

- 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 be used with any type that is not a scalar or vector, or an array of such a type

- 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 GLSLPacked 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 not be used on variables
with the \textit{Output} storage class in a fragment shader

- \textbf{VUID-StandaloneSpirv-Flat-06202}
  The \textit{Flat}, \textit{NoPerspective}, \textit{Sample}, and \textit{Centroid} decorations \textbf{must} not be used on variables with the \textit{Input} storage class in a vertex shader

- \textbf{VUID-StandaloneSpirv-PerVertexKHR-06777}
  The \textit{PerVertexKHR} decoration \textbf{must} only be used on variables with the \textit{Input Storage Class} in a fragment shader

- \textbf{VUID-StandaloneSpirv-Flat-04744}
  Any variable with integer or double-precision floating-point type and with \textit{Input Storage Class} in a fragment shader, \textbf{must} be decorated \textit{Flat}

- \textbf{VUID-StandaloneSpirv-ViewportRelativeNV-04672}
  The \textit{ViewportRelativeNV} decoration \textbf{must} only be used on a variable decorated with \textit{Layer} in the vertex, tessellation evaluation, or geometry shader stages

- \textbf{VUID-StandaloneSpirv-ViewportRelativeNV-04673}
  The \textit{ViewportRelativeNV} decoration \textbf{must} not be used unless a variable decorated with one of \textit{ViewportIndex} or \textit{ViewportMaskNV} is also statically used by the same \textit{OpEntryPoint}

- \textbf{VUID-StandaloneSpirv-ViewportMaskNV-04674}
  The \textit{ViewportMaskNV} and \textit{ViewportIndex} decorations \textbf{must} not both be statically used by one or more \textit{OpEntryPoint}’s that form the \textit{pre-rasterization shader stages} of a graphics pipeline

- \textbf{VUID-StandaloneSpirv-FPRoundingMode-04675}
  Rounding modes other than round-to-nearest-even and round-towards-zero \textbf{must} not be used for the \textit{FPRoundingMode} decoration

- \textbf{VUID-StandaloneSpirv-Invariant-04677}
  Variables decorated with \textit{Invariant} and variables with structure types that have any members decorated with \textit{Invariant} \textbf{must} be in the \textit{Output} or \textit{Input Storage Class}. \textit{Invariant} used on an \textit{Input Storage Class} variable or structure member has no effect

- \textbf{VUID-StandaloneSpirv-VulkanMemoryModel-04678}
  If the \textit{VulkanMemoryModel} capability is not declared, the \textit{Volatile} decoration \textbf{must} be used on any variable declaration that includes one of the \textit{SMIDNV}, \textit{WarpIDNV}, \textit{SubgroupSize}, \textit{SubgroupLocalInvocationId}, \textit{SubgroupEqMask}, \textit{SubgroupGeMask}, \textit{SubgroupGtMask}, \textit{SubgroupLeMask}, or \textit{SubgroupLtMask BuiltIn} decorations when used in the ray generation, closest hit, miss, intersection, or callable shaders, or with the \textit{RayTmaxKHR BuiltIn} decoration when used in an intersection shader

- \textbf{VUID-StandaloneSpirv-VulkanMemoryModel-04679}
  If the \textit{VulkanMemoryModel} capability is declared, the \textit{OpLoad} instruction \textbf{must} use the \textit{Volatile} memory semantics when it accesses into any variable that includes one of the \textit{SMIDNV}, \textit{WarpIDNV}, \textit{SubgroupSize}, \textit{SubgroupLocalInvocationId}, \textit{SubgroupEqMask}, \textit{SubgroupGeMask}, \textit{SubgroupGtMask}, \textit{SubgroupLeMask}, or \textit{SubgroupLtMask BuiltIn} decorations when used in the ray generation, closest hit, miss, intersection, or callable shaders, or with the \textit{RayTmaxKHR BuiltIn} decoration when used in an intersection shader

- \textbf{VUID-StandaloneSpirv-OpTypeRuntimeArray-04680}
  \textit{OpTypeRuntimeArray} \textbf{must} only be used for:
  - the last member of a \textit{Block}-decorated \textit{OpTypeStruct} in \textit{StorageBuffer} or
PhysicalStorageBuffer storage Storage Class

- BufferBlock-decorated OpTypeStruct in the Uniform storage Storage Class
- the outermost dimension of an arrayed variable in the StorageBuffer, Uniform, or UniformConstant storage Storage Class
- variables in the NodePayloadAMDX storage Storage Class when the CoalescingAMDX Execution Mode is specified

- 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 Class, 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
Output variables or block members decorated with `Offset` that have a 32-bit type, or a composite type contains a 32-bit type, **must** specify an `Offset` value aligned to a 4 byte boundary.

Output variables, blocks or block members decorated with `Offset` **must** only contain base types that have components that are either 32-bit or 64-bit in size.

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.

All variables or block members in the output interface of the entry point being compiled decorated with a specific `XfbBuffer` value **must** all be decorated with identical `XfbStride` values.

If any variables or block members in the output interface of the entry point being compiled are decorated with `Stream`, then all variables belonging to the same `XfbBuffer` **must** specify the same `Stream` value.

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.

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.

`RayPayloadKHR Storage Class` **must** only be used in ray generation, closest hit or miss shaders.

`IncomingRayPayloadKHR Storage Class` **must** only be used in closest hit, any-hit, or miss shaders.

There **must** be at most one variable with the `IncomingRayPayloadKHR Storage Class` in the input interface of an entry point.

`HitAttributeKHR Storage Class` **must** only be used in intersection, any-hit, or closest hit shaders.

There **must** be at most one variable with the `HitAttributeKHR Storage Class` in the input interface of an entry point.
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.

ShaderRecordBufferKHR Storage Class must only be used in ray generation, intersection, any-hit, closest hit, callable, or miss shaders.

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.

Each `OpEntryPoint` must not have more than one `Input` variable assigned the same `Component` word inside a `Location` slot, either explicitly or implicitly.

Each `OpEntryPoint` must not have more than one `Output` variable assigned the same `Component` word inside a `Location` slot, either explicitly or implicitly.

The `Result Type` operand of any `OpImageRead` or `OpImageSparseRead` instruction must be a vector of four components.

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.

Any variable in the `PushConstant` or `StorageBuffer` storage class must be decorated as `Block`.

Any variable in the `UniformStorage Class` must be decorated as `Block` or `BufferBlock`.

Any variable in the `UniformConstant`, `StorageBuffer`, or `Uniform Storage Class` must be decorated with `DescriptorSet` and `Binding`.

Variables decorated with `InputAttachmentIndex` must be in the `UniformConstant Storage Class`.

If a variable is decorated by `DescriptorSet` or `Binding`, the `Storage Class` must correspond to an entry in `Shader Resource and Storage Class Correspondence`.

Variables with a `Storage Class` of `Input` in a fragment shader stage that are decorated with `PerVertexKHR` must be declared as arrays.

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`.

In mesh shaders using the `MeshEXT Execution Model OpSetMeshOutputsEXT` must be called before any outputs are written.

In mesh shaders using the `MeshEXT Execution Model` all variables declared as output must not be read from.

Any variable in the `PushConstant` or `StorageBuffer` storage class must be decorated as `Block`.

Any variable in the `UniformStorage Class` must be decorated as `Block` or `BufferBlock`.

Any variable in the `UniformConstant`, `StorageBuffer`, or `Uniform Storage Class` must be decorated with `DescriptorSet` and `Binding`.

Variables decorated with `InputAttachmentIndex` must be in the `UniformConstant Storage Class`.

If a variable is decorated by `DescriptorSet` or `Binding`, the `Storage Class` must correspond to an entry in `Shader Resource and Storage Class Correspondence`.

Variables with a `Storage Class` of `Input` in a fragment shader stage that are decorated with `PerVertexKHR` must be declared as arrays.

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`.

In mesh shaders using the `MeshEXT Execution Model OpSetMeshOutputsEXT` must be called before any outputs are written.

In mesh shaders using the `MeshEXT Execution Model` all variables declared as output must not be read from.
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-StandaloneSpirv-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-StandaloneSpirv-MeshEXT-07330
  In mesh shaders using the MeshEXT Execution Model the OutputVertices Execution Mode must be greater than 0

- VUID-StandaloneSpirv-MeshEXT-07331
  In mesh shaders using the MeshEXT Execution Model the OutputPrimitivesEXT Execution Mode must be greater than 0

- VUID-StandaloneSpirv-Input-07290
  Variables with a Storage Class of Input or Output and a type of OpTypeBool must be decorated with the BuiltIn decoration

- VUID-StandaloneSpirv-TileImageEXT-08723
  The tile image variable declarations must obey the constraints on the TileImageEXT Storage Class and the Location decoration described in Fragment Tile Image Interface

- VUID-StandaloneSpirv-None-08724
  The TileImageEXT Storage Class must only be used for declaring tile image variables.

- VUID-StandaloneSpirv-Pointer-08973
  The Storage Class of the Pointer operand to OpCooperativeMatrixLoadKHR or OpCooperativeMatrixStoreKHR must be limited to Workgroup, StorageBuffer, or PhysicalStorageBuffer.

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-OpTypeImage-06269**
  If `shaderStorageImageWriteWithoutFormat` is not enabled, any variable created with a “Type” of `OpTypeImage` that has a “Sampled” operand of 2 and an “Image Format” operand of `Unknown` **must** be decorated with `NonWritable`

- **VUID-RuntimeSpirv-OpTypeImage-06270**
  If `shaderStorageImageReadWithoutFormat` is not enabled, any variable created with a “Type” of `OpTypeImage` that has a “Sampled” operand of 2 and an “Image Format” operand of `Unknown` **must** be decorated with `NonReadable`

- **VUID-RuntimeSpirv-OpImageWrite-07112**
  `OpImageWrite` to any `Image` whose `Image Format` is `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-Location-06428**
  The maximum number of storage buffers, storage images, and output `Location` decorated color attachments written to in the `Fragment Execution Model` **must** be less than or equal to `maxFragmentCombinedOutputResources`

- **VUID-RuntimeSpirv-NonUniform-06274**
  If an instruction loads from or stores to a resource (including atomics and image instructions) and the resource descriptor being accessed is not dynamically uniform, then the operand corresponding to that resource (e.g. the pointer or sampled image operand) **must** be decorated with `NonUniform`

- **VUID-RuntimeSpirv-NonUniform-06275**
  `shaderSubgroupExtendedTypes` **must** be enabled for `group operations` to use 8-bit integer, 16-bit integer, 64-bit integer, 16-bit floating-point, and vectors of these types

- **VUID-RuntimeSpirv-subgroupBroadcastDynamicId-06276**
  If `subgroupBroadcastDynamicId` is `VK_TRUE`, and the shader module version is 1.5 or higher, the “Index” for `OpGroupNonUniformQuadBroadcast` **must** be dynamically uniform within the derivative group. Otherwise, “Index” **must** be a constant

- **VUID-RuntimeSpirv-subgroupBroadcastDynamicId-06277**
  If `subgroupBroadcastDynamicId` is `VK_TRUE`, and the shader module version is 1.5 or higher, the “Id” for `OpGroupNonUniformBroadcast` **must** be dynamically uniform within the subgroup. Otherwise, “Id” **must** be a constant

- **VUID-RuntimeSpirv-denormBehaviorIndependence-06289**
  If `denormBehaviorIndependence` is `VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_32_BIT_ONLY`, then the entry point **must** use the same denormals `Execution Mode` for both 16-bit and 64-bit floating-point types
VUID-RuntimeSpirv-denormBehaviorIndependence-06290
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.

VUID-RuntimeSpirv-roundingModeIndependence-06291
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.

VUID-RuntimeSpirv-roundingModeIndependence-06292
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.

VUID-RuntimeSpirv-shaderSignedZeroInfNanPreserveFloat16-06293
If \texttt{shaderSignedZeroInfNanPreserveFloat16} is \texttt{VK_FALSE}, then \texttt{SignedZeroInfNanPreserve} for 16-bit floating-point type \textbf{must} not be used.

VUID-RuntimeSpirv-shaderSignedZeroInfNanPreserveFloat32-06294
If \texttt{shaderSignedZeroInfNanPreserveFloat32} is \texttt{VK_FALSE}, then \texttt{SignedZeroInfNanPreserve} for 32-bit floating-point type \textbf{must} not be used.

VUID-RuntimeSpirv-shaderSignedZeroInfNanPreserveFloat64-06295
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 \texttt{shaderDenormPreserveFloat16} is \texttt{VK_FALSE}, then \texttt{DenormPreserve} for 16-bit floating-point type \textbf{must} not be used.

VUID-RuntimeSpirv-shaderDenormPreserveFloat32-06297
If \texttt{shaderDenormPreserveFloat32} is \texttt{VK_FALSE}, then \texttt{DenormPreserve} for 32-bit floating-point type \textbf{must} not be used.

VUID-RuntimeSpirv-shaderDenormPreserveFloat64-06298
If \texttt{shaderDenormPreserveFloat64} is \texttt{VK_FALSE}, then \texttt{DenormPreserve} for 64-bit floating-point type \textbf{must} not be used.

VUID-RuntimeSpirv-shaderDenormFlushToZeroFloat16-06299
If \texttt{shaderDenormFlushToZeroFloat16} is \texttt{VK_FALSE}, then \texttt{DenormFlushToZero} for 16-bit floating-point type \textbf{must} not be used.

VUID-RuntimeSpirv-shaderDenormFlushToZeroFloat32-06300
If \texttt{shaderDenormFlushToZeroFloat32} is \texttt{VK_FALSE}, then \texttt{DenormFlushToZero} for 32-bit floating-point type \textbf{must} not be used.

VUID-RuntimeSpirv-shaderDenormFlushToZeroFloat64-06301
If \texttt{shaderDenormFlushToZeroFloat64} is \texttt{VK_FALSE}, then \texttt{DenormFlushToZero} for 64-bit floating-point type \textbf{must} not be used.

VUID-RuntimeSpirv-shaderRoundingModeRTEFloat16-06302
If \texttt{shaderRoundingModeRTEFloat16} is \texttt{VK_FALSE}, then \texttt{RoundingModeRTE} for 16-bit floating-point type \textbf{must} not be used.

VUID-RuntimeSpirv-shaderRoundingModeRTEFloat32-06303
If \texttt{shaderRoundingModeRTEFloat32} is \texttt{VK_FALSE}, then \texttt{RoundingModeRTE} for 32-bit floating-point type \textbf{must} not be used.
If `shaderRoundingModeRTEFloat64` is `VK_FALSE`, then `RoundingModeRTE` for 64-bit floating-point type **must** not be used.

If `shaderRoundingModeRTZFloat16` is `VK_FALSE`, then `RoundingModeRTZ` for 16-bit floating-point type **must** not be used.

If `shaderRoundingModeRTZFloat32` is `VK_FALSE`, then `RoundingModeRTZ` for 32-bit floating-point type **must** not be used.

If `shaderRoundingModeRTZFloat64` is `VK_FALSE`, then `RoundingModeRTZ` for 64-bit floating-point type **must** not be used.

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.

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 operand.

**DescriptorSet** and **Binding** decorations **must** obey the constraints on **Storage Class**, type, and descriptor type described in [DescriptorSet and Binding Assignment](#).

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.

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.

If `shaderAtomicInstructions` is not enabled, the SPIR-V Atomic Instructions listed in 3.37.18 (`OpAtomic*`) **must** not be used.

If `subgroupQuadOperationsInAllStages` is `VK_FALSE`, then **quad subgroup operations** **must** not be used except for in fragment and compute stages.

Group operations with subgroup scope **must** not be used if the shader stage is not in subgroupSupportedStages.

The first element of the `Offset` operand of `InterpolateAtOffset` **must** be greater than or equal to:
fragwidth \times \text{minInterpolationOffset}
where \(\text{fragwidth}\) 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
\[\text{fragwidth} \times (\text{maxInterpolationOffset} + \text{ULP}) - \text{ULP}\]
where \(\text{fragwidth}\) is the width of the current fragment in pixels and \(\text{ULP} = 1 / 2^{\text{subPixelInterpolationOffsetBits}}\).

- **VUID-RuntimeSpirv-Offset-06346**
The second element of the Offset operand of InterpolateAtOffset must be greater than or equal to
\[\text{fragheight} \times \text{minInterpolationOffset}\]
where \(\text{fragheight}\) 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
\[\text{fragheight} \times (\text{maxInterpolationOffset} + \text{ULP}) - \text{ULP}\]
where \(\text{fragheight}\) is the height of the current fragment in pixels and \(\text{ULP} = 1 / 2^{\text{subPixelInterpolationOffsetBits}}\).

- **VUID-RuntimeSpirv-x-06429**
In compute shaders using the GLCompute Execution Model the x size in LocalSize or LocalSizeId must be less than or equal to \(\text{VkPhysicalDeviceLimits}::\text{maxComputeWorkGroupSize}[0]\)

- **VUID-RuntimeSpirv-y-06430**
In compute shaders using the GLCompute Execution Model the y size in LocalSize or LocalSizeId must be less than or equal to \(\text{VkPhysicalDeviceLimits}::\text{maxComputeWorkGroupSize}[1]\)

- **VUID-RuntimeSpirv-z-06431**
In compute shaders using the GLCompute Execution Model the z size in LocalSize or LocalSizeId must be less than or equal to \(\text{VkPhysicalDeviceLimits}::\text{maxComputeWorkGroupSize}[2]\)

- **VUID-RuntimeSpirv-x-06432**
In compute shaders using the GLCompute Execution Model the product of x size, y size, and z size in LocalSize or LocalSizeId must be less than or equal to \(\text{VkPhysicalDeviceLimits}::\text{maxComputeWorkGroupInvocations}\)

- **VUID-RuntimeSpirv-LocalSizeId-06433**
The Execution Mode LocalSizeId must not be used

- **VUID-RuntimeSpirv-OpTypeVector-06816**
Any OpTypeVector output interface variables must not have a higher Component Count than a matching OpTypeVector input interface variable

- **VUID-RuntimeSpirv-OpEntryPoint-08743**
Any user-defined variables shared between the OpEntryPoint of two shader stages, and declared with Input as its Storage Class for the subsequent shader stage, must have all Location slots and Component words declared in the preceding shader stage’s OpEntryPoint
with Output as the Storage Class

- VUID-RuntimeSpirv-OpEntryPoint-07754
  Any user-defined variables between the OpEntryPoint of two shader stages must have the same type and width for each Component

- VUID-RuntimeSpirv-OpVariable-08746
  Any OpVariable, Block-decorated OpTypeStruct, or Block-decorated OpTypeStruct members shared between the OpEntryPoint of two shader stages must have matching decorations as defined in interface matching

- VUID-RuntimeSpirv-Workgroup-06530
  The sum of size in bytes for variables and padding in the Workgroup Storage Class in the GLCompute Execution Model must be less than or equal to maxComputeSharedMemorySize

- VUID-RuntimeSpirv-OpVariable-06373
  Any OpVariable with Workgroup as its Storage Class must not have an Initializer operand

- VUID-RuntimeSpirv-OpImage-06376
  If an OpImage*Gather operation has an image operand of Offset, ConstOffset, or ConstOffsets the offset value must be greater than or equal to minTexelGatherOffset

- VUID-RuntimeSpirv-OpImage-06377
  If an OpImage*Gather operation has an image operand of Offset, ConstOffset, or ConstOffsets the offset value must be less than or equal to maxTexelGatherOffset

- VUID-RuntimeSpirv-OpImageSample-06435
  If an OpImageSample* or OpImageFetch* operation has an image operand of ConstOffset then the offset value must be greater than or equal to minTexelOffset

- VUID-RuntimeSpirv-OpImageSample-06436
  If an OpImageSample* or OpImageFetch* operation has an image operand of ConstOffset then the offset value must be less than or equal to maxTexelOffset

- VUID-RuntimeSpirv-samples-08725
  If an OpTypeImage has an MS operand 0, its bound image must have been created with VkImageCreateInfo::samples as VK_SAMPLE_COUNT_1_BIT

- VUID-RuntimeSpirv-samples-08726
  If an OpTypeImage has an MS operand 1, its bound image must not have been created with VkImageCreateInfo::samples as VK_SAMPLE_COUNT_1_BIT

- VUID-RuntimeSpirv-OpEntryPoint-08727
  Each OpEntryPoint must not have more than one variable decorated with InputAttachmentIndex per image aspect of the attachment image bound to it, either explicitly or implicitly as described by input attachment interface

- VUID-RuntimeSpirv-MeshEXT-09218
  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
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.

The following core SPIR-V instructions must respect the DenormPreserve Execution Mode:

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.

\textbf{Note}
For example, in the absence of the NoContraction decoration implementations are allowed to implement \(a + b - a\) and \(\frac{ab}{a}\) 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:

\textbf{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, finite 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, \( \text{ulp}(x) = \min_{a \neq b} |b - a| \). If such numbers do not exist then \( \text{ulp}(x) \) is defined to be the difference between the two non-equal, 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 DenormFlushToZero execution mode, then any intermediate denormal value(s) while evaluating the formula may be flushed to zero. Denormal final results must be flushed to zero. If the entry point is declared with the DenormPreserve Execution Mode, then denormals must be preserved throughout the formula.

For half- (16 bit) and single- (32 bit) precision instructions, precisions are required to be at least as follows:

**Table 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 \(\text{trunc}()\) and \(\text{floor}()\). 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>log(), log2()</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>pow(x, y)</td>
<td>Inherited from (\exp(2y \times \log2(x))).</td>
<td></td>
</tr>
<tr>
<td>sqrt()</td>
<td>Inherited from (1.0 / \text{inversesqrt}()).</td>
<td></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><code>inversesqrt()</code></td>
<td>2 ULP.</td>
<td></td>
</tr>
<tr>
<td><code>radians(x)</code></td>
<td>Inherited from $\frac{x \times C_{n,180}}{180}$, where $C_{n,180}$ is a correctly rounded approximation to $\frac{n}{180}$.</td>
<td></td>
</tr>
<tr>
<td><code>degrees(x)</code></td>
<td>Inherited from $\frac{x \times C_{180,n}}{180}$, where $C_{180,n}$ is a correctly rounded approximation to $\frac{n}{180}$.</td>
<td></td>
</tr>
<tr>
<td><code>sin()</code></td>
<td>Absolute error $\leq 2^{-11}$ inside the range $[-n, n]$.</td>
<td>Absolute error $\leq 2^{-7}$ inside the range $[-n, n]$.</td>
</tr>
<tr>
<td><code>cos()</code></td>
<td>Absolute error $\leq 2^{-11}$ inside the range $[-n, n]$.</td>
<td>Absolute error $\leq 2^{-7}$ inside the range $[-n, n]$.</td>
</tr>
<tr>
<td><code>tan()</code></td>
<td>Inherited from $\frac{\sin(x)}{\cos(x)}$.</td>
<td></td>
</tr>
<tr>
<td><code>asin(x)</code></td>
<td>Inherited from $\text{atan2}(x, \sqrt{1.0 - x \times x})$.</td>
<td></td>
</tr>
<tr>
<td><code>acos(x)</code></td>
<td>Inherited from $\text{atan2}(\sqrt{1.0 - x \times x}, x)$.</td>
<td></td>
</tr>
<tr>
<td><code>atan()</code>, <code>atan2()</code></td>
<td>4096 ULP</td>
<td>5 ULP.</td>
</tr>
<tr>
<td><code>sinh(x)</code></td>
<td>Inherited from $(\exp(x) - \exp(-x)) \times 0.5$.</td>
<td></td>
</tr>
<tr>
<td><code>cosh(x)</code></td>
<td>Inherited from $(\exp(x) + \exp(-x)) \times 0.5$.</td>
<td></td>
</tr>
<tr>
<td><code>tanh()</code></td>
<td>Inherited from $\frac{\sinh(x)}{\cosh(x)}$.</td>
<td></td>
</tr>
<tr>
<td><code>asinh(x)</code></td>
<td>Inherited from $\log(x + \sqrt{x \times x + 1.0})$.</td>
<td></td>
</tr>
<tr>
<td><code>acosh(x)</code></td>
<td>Inherited from $\log(x + \sqrt{x \times x - 1.0})$.</td>
<td></td>
</tr>
<tr>
<td><code>atanh(x)</code></td>
<td>Inherited from $\log\left(\frac{1.0 + x}{1.0 - x}\right) \times 0.5$.</td>
<td></td>
</tr>
<tr>
<td><code>frexp()</code></td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td><code>ldexp()</code></td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td><code>length(x)</code></td>
<td>Inherited from $\sqrt{\text{dot}(x, x)}$.</td>
<td></td>
</tr>
<tr>
<td><code>distance(x, y)</code></td>
<td>Inherited from $\sqrt{\text{dot}(x - y)}$.</td>
<td></td>
</tr>
<tr>
<td><code>normalize(x)</code></td>
<td>Inherited from $x \times \text{inversesqrt}(\text{dot}(x, x))$.</td>
<td></td>
</tr>
<tr>
<td><code>faceforward(N, I, NRef)</code></td>
<td>Inherited from $\text{dot}(NRef, I) &lt; 0.0 \ ? \ N : -N$.</td>
<td></td>
</tr>
<tr>
<td><code>reflect(x, y)</code></td>
<td>Inherited from $x - 2.0 \times \text{dot}(y, x) \times y$.</td>
<td></td>
</tr>
<tr>
<td><code>refract(I, N, eta)</code></td>
<td>Inherited from $k &lt; 0.0 \ ? \ 0.0 : \text{eta} \times I - (\text{eta} \times \text{dot}(N, I) + \sqrt{\text{dot}(k)}) \times N$, 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><code>round</code></td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<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>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>fsign</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>floor</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>ceil</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>fract</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>modf</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>fmin</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>fmax</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>fclamp</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>fmix(x, y, a)</td>
<td>Inherited from $x \times (1.0 - a) + y \times a.$</td>
<td></td>
</tr>
<tr>
<td>step</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>smoothStep(edge0, edge1, x)</td>
<td>Inherited from $t \times t \times (3.0 - 2.0 \times t)$, where $t = \text{clamp}\left(\frac{1 - edge0}{edge1 - edge0}, 0.0, 1.0\right)$.</td>
<td></td>
</tr>
<tr>
<td>nmin</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>nmax</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>nclamp</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-Declaration instructions</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>
### Type-Declaration instructions

<table>
<thead>
<tr>
<th>Image Format</th>
<th>Type-Declaration instructions</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></td>
<td></td>
<td></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>

The **SPIR-V Type** is defined by an instruction in SPIR-V, declared with the Type-Declaration Instruction, Bit Width, and Signedness from above.

### Compatibility Between SPIR-V Image Formats and Vulkan Formats

SPIR-V **Image Format** values are compatible with **VkFormat** values as defined below:

**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>SPIR-V Image Format</td>
<td>Compatible Vulkan Format</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------</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>Rgba8</td>
<td>VK_FORMAT_R8G8B8B8A8_UNORM</td>
</tr>
<tr>
<td>Rgba8Snorm</td>
<td>VK_FORMAT_R8G8B8B8A8_SNORM</td>
</tr>
<tr>
<td>Rgba8ui</td>
<td>VK_FORMAT_R8G8B8B8A8_UINT</td>
</tr>
<tr>
<td>Rgba8i</td>
<td>VK_FORMAT_R8G8B8B8A8_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

Note
This memory model describes synchronizations provided by all implementations; however, some of the synchronizations defined require extra features to be supported by the implementation. See `VkPhysicalDeviceVulkanMemoryModelFeatures`.

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 \texttt{OpAtomic}. An
atomic operation on the host is any operation performed with an \texttt{std::atomic} typed object.

Each atomic operation has a memory \textit{scope} and a \textit{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 \texttt{CrossDevice} scope, and uses a memory semantics equivalent to
a C++ \texttt{std::memory\_order} value of relaxed, acquire, release, acq\_rel, or seq\_cst.

Two atomic operations A and B are \textit{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 \textit{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.

\begin{note}

\end{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.
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.

Informally, these constraints limit how memory operations can be reordered, and these limits apply not only to the order of accesses as performed in the agent that executes the instruction, but also to the order the effects of writes become visible to all other agents within the same instance of the instruction’s memory scope.

Release and acquire operations in different threads can act as synchronization operations, to guarantee that writes that happened before the release are visible after the acquire. (This is not a formal definition, just an Informative forward reference.)

The 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. 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 W₂ 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 \(\neq\) Y. Let \((A_X, R_X)\) be the agent and reference used for X, and \((A_Y, R_Y)\) be the agent and reference used for Y. For now, let “ \(\rightarrow\) ” denote happens-before and “ \(\rightarrow_{rcpo}\) ” denote the reflexive closure of program-ordered before.

If \(D_1\) and \(D_2\) are different memory domains, then let \(\text{DOM}(D_1, D_2)\) be a memory domain operation from \(D_1\) to \(D_2\). Otherwise, let \(\text{DOM}(D, D)\) be a placeholder such that \(X \rightarrow\text{DOM}(D, D) \rightarrow Y\) if and only if \(X \rightarrow Y\).

X is **location-ordered** before Y for a location L in M if and only if any of the following is true:

- \(A_X = A_Y\) and \(R_X = R_Y\) and \(X \rightarrow Y\)
  - NOTE: this case means no availability/visibility ops 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, \(\text{AVC}(A_X, R_X, D, L)\) is an availability chain making \((X, L)\) available to domain D, and \(X \rightarrow_{rcpo} \text{AVC}(A_X, R_X, D, L) \rightarrow Y\)
  - X is a read, Y is a read, \(\text{AVC}(A_X, R_X, D, L)\) is an availability chain making \((X, L)\) available to domain D, \(\text{VISC}(A_Y, R_Y, D, L)\) is a visibility chain making writes to L available in domain D visible to Y, and \(X \rightarrow_{rcpo} \text{AVC}(A_X, R_X, D, L) \rightarrow \text{VISC}(A_Y, R_Y, D, L) \rightarrow_{rcpo} Y\)
  - If \(\text{VkPhysicalDeviceVulkanMemoryModelFeatures::vulkanMemoryModelAvailabilityVisibilityChains}\) 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 \(X \rightarrow \text{AV}(A_X, R_X, D_X, L) \rightarrow \text{DOM}(D_X, D_Y) \rightarrow Y\)
- X is a write and Y is a read, and \( X \rightarrow AV(A_X,R_X,D_X,L) \rightarrow DOM(D_X,D_Y) \rightarrow VIS(A_Y,R_Y,D_Y,L) \rightarrow Y \)

**Note**

The final bullet (synchronization through device/host domain) requires API-level synchronization operations, since the device/host domains are not accessible via shader instructions. And “device domain” is not to be confused with “device scope”, which synchronizes through the “shader domain”.

### Data Race

Let X and Y be operations that access overlapping sets of memory locations M, where X != 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.

**Note**

Data races can only occur due to instructions that are actually executed. For example, an instruction skipped due to control flow must not contribute to a data race.

### Visible-To

Let X be a write and Y be a read whose sets of memory locations overlap, and let M be the set of memory locations that overlap. Let \( M_2 \) be a non-empty subset of M. Then X is **visible-to** Y for memory locations \( M_2 \) if and only if all of the following are true:

- X is location-ordered before Y for each location L in \( M_2 \).
- There does not exist another write Z to any location L in \( M_2 \) such that X is location-ordered before Z for location L and Z is location-ordered before Y for location L.

If X is visible-to Y, then Y reads the value written by X for locations \( M_2 \).

**Note**

It is possible for there to be a write between X and Y that overwrites a subset of the memory locations, but the remaining memory locations (\( M_2 \)) will still be visible-to Y.

### Acyclicity

**Reads-from** is a relation between operations, where the first operation is a write, the second operation is a read, and the second operation reads the value written by the first operation. **From-reads** is a relation between operations, where the first operation is a read, the second operation is a write, and the first operation reads a value written earlier than the second operation in the second operation’s scoped modification order (or the first operation reads from the initial value, and the
second operation is any write to the same locations).

Then the implementation must guarantee that no cycles exist in the union of the following relations:

- location-ordered
- scoped modification order (over all atomic writes)
- reads-from
- from-reads

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, scoped 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

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>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>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>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/doc/ETC2-Compression/).

<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 × 4</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_4x4_SRGB_BLOCK</td>
<td>4 × 4</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x4_UNORM_BLOCK</td>
<td>5 × 4</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x4_SRGB_BLOCK</td>
<td>5 × 4</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x5_UNORM_BLOCK</td>
<td>5 × 5</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x5_SRGB_BLOCK</td>
<td>5 × 5</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x5_UNORM_BLOCK</td>
<td>6 × 5</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x5_SRGB_BLOCK</td>
<td>6 × 5</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x6_UNORM_BLOCK</td>
<td>6 × 6</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x6_SRGB_BLOCK</td>
<td>6 × 6</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x5_UNORM_BLOCK</td>
<td>8 × 5</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x5_SRGB_BLOCK</td>
<td>8 × 5</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x6_UNORM_BLOCK</td>
<td>8 × 6</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x6_SRGB_BLOCK</td>
<td>8 × 6</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x8_UNORM_BLOCK</td>
<td>8 × 8</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x8_SRGB_BLOCK</td>
<td>8 × 8</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x5_UNORM_BLOCK</td>
<td>10 × 5</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x5_SRGB_BLOCK</td>
<td>10 × 5</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x6_UNORM_BLOCK</td>
<td>10 × 6</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x6_SRGB_BLOCK</td>
<td>10 × 6</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x8_UNORM_BLOCK</td>
<td>10 × 8</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x8_SRGB_BLOCK</td>
<td>10 × 8</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x10_UNORM_BLOCK</td>
<td>10 × 10</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x10_SRGB_BLOCK</td>
<td>10 × 10</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x10_UNORM_BLOCK</td>
<td>12 × 10</td>
<td>Linear LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x10_SRGB_BLOCK</td>
<td>12 × 10</td>
<td>sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x12_UNORM_BLOCK</td>
<td>12 × 12</td>
<td>Linear LDR</td>
</tr>
</tbody>
</table>
## VkFormat

<table>
<thead>
<tr>
<th>VkFormat</th>
<th>Compressed texel block dimensions</th>
<th>Requested mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_ASTC_12x12_SRGB_BLOCK</td>
<td>12 × 12</td>
<td>sRGB</td>
</tr>
</tbody>
</table>

ASTC textures containing any HDR blocks **should** not be passed into the API using an sRGB or UNORM 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 1.2

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
- VK_KHR_uniform_buffer_standard_layout
- VK_KHR_vulkan_memory_model
- VK_EXT_descriptor_indexing
- VK_EXT_host_query_reset
- VK_EXT_sampler_filter_minmax
- VK_EXT_scalar_block_layout
- VK_EXT_separateStencilUsage
- VK_EXT_shader_viewport_index_layer

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 `OpGroupNonUniformBroadcast` to be dynamically uniform within a subgroup, and the “Index” operand of `OpGroupNonUniformQuadBroadcast` 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_COUNTALLOCATE_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_STORAGEFEATURES
- 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_EXTENDEDTYPES_FEATURES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_FEATURES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_PROPERTIES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_UNIFORM_BUFFER_STANDARD_LAYOUT_FEATURES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_1_FEATURES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_1_PROPERTIES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_2_FEATURES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_2_PROPERTIES
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_MEMORY_MODEL_FEATURES
- VK_STRUCTURE_TYPE_RENDER_PASS_ATTACHMENT_BEGIN_INFO
- VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO
- VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO_2
VK_STRUCTURE_TYPE_SAMPLER_REDUCTION_MODE_CREATE_INFO
VK_STRUCTURE_TYPE_SEMAPHORE_SIGNAL_INFO
VK_STRUCTURE_TYPE_SEMAPHORE_TYPE_CREATE_INFO
VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO
VK_STRUCTURE_TYPE_SUBPASS_BEGIN_INFO
VK_STRUCTURE_TYPE_SUBPASS_DEPENDENCY_2
VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_2
VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_DEPTH_STENCIL_RESOLVE
VK_STRUCTURE_TYPE_SUBPASS_END_INFO
VK_STRUCTURE_TYPE_TIMELINE_SEMAPHORE_SUBMIT_INFO

Version 1.1

Vulkan Version 1.1 promoted a number of key extensions into the core API:

- VK_KHR_16bit_storage
- VK_KHR_bind_memory2
- VK_KHR_dedicated_allocation
- VK_KHR_descriptor_update_template
- VK_KHR_device_group
- VK_KHR_device_group_creation
- VK_KHR_external_fence
- VK_KHR_external_fence_capabilities
- VK_KHR_external_memory
- VK_KHR_external_memory_capabilities
- VK_KHR_external_semaphore
- VK_KHR_external_semaphore_capabilities
- VK_KHR_get_memory_requirements2
- VK_KHR_get_physical_device_properties2
- VK_KHR_maintenance1
- VK_KHR_maintenance2
- VK_KHR_maintenance3
- VK_KHR_multiview
- VK_KHR_relaxed_block_layout
- VK_KHR_sampler_ycbcr_conversion
- VK_KHR_shader_draw_parameters
• VK_KHR_storage_buffer_storage_class
• VK_KHR_variable_pointers

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 `VkPhysicalDeviceVariablePointersFeatures::variablePointersStorageBuffer` or `VkPhysicalDeviceVulkan11Features::variablePointersStorageBuffer` when queried via `vkGetPhysicalDeviceFeatures2`.

**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
• vkGetDescriptorSetLayoutSupport
• vkGetDeviceGroupPeerMemoryFeatures
• vkGetDeviceQueue2
• vkGetImageMemoryRequirements2
• vkGetImageSparseMemoryRequirements2
• vkGetPhysicalDeviceExternalBufferProperties
• vkGetPhysicalDeviceExternalFenceProperties
• vkGetPhysicalDeviceExternalSemaphoreProperties
• vkGetPhysicalDeviceFeatures2
• vkGetPhysicalDeviceFormatProperties2
• vkGetPhysicalDeviceImageFormatProperties2
• vkGetPhysicalDeviceMemoryProperties2
• vkGetPhysicalDeviceProperties2
• vkGetPhysicalDeviceQueueFamilyProperties2
• vkGetPhysicalDeviceSparseImageFormatProperties2
• vkTrimCommandPool
New Structures

- VkBindBufferMemoryInfo
- VkBindImageMemoryInfo
- VkBufferMemoryRequirementsInfo2
- VkDescriptorSetLayoutSupport
- VkDescriptorUpdateTemplateCreateInfo
- VkDescriptorUpdateTemplateEntry
- VkDeviceQueueInfo2
- VkExternalBufferProperties
- VkExternalFenceProperties
- VkExternalMemoryProperties
- VkExternalSemaphoreProperties
- VkFormatProperties2
- VkImageFormatProperties2
- VkImageMemoryRequirementsInfo2
- VkImageSparseMemoryRequirementsInfo2
- 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`
- `VkProtectedSubmitInfo`

**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
• VkSubgroupFeatureFlags

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_422_UNORM_4PACK16
• 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_R10X6G10X6_2PACK16
- `VK_FORMAT_R10X6_UNORM_PACK16`
- `VK_FORMAT_R12X4G12X4B12X4A12X4_UNORM_4PACK16`
- `VK_FORMAT_R12X4G12X4_UNORM_2PACK16`
- `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`:
• **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_MEMORY_DEVICE_GROUP_INFO
  - VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO
  - VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_DEVICE_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_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO
  - VK_STRUCTURE_TYPE_DEVICE_GROUP_BIND_SPARSE_INFO
  - VK_STRUCTURE_TYPE_DEVICE_GROUP_COMMAND_BUFFER_BEGIN_INFO
  - VK_STRUCTURE_TYPE_DEVICE_GROUP_DEVICE_CREATE_INFO
  - VK_STRUCTURE_TYPE_DEVICE_GROUP_RENDER_PASS_BEGIN_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_PHYSICAL_DEVICE_EXTERNAL_BUFFER_INFO
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_FENCE_INFO
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_IMAGE_FORMAT_INFO
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SEMAPHORE_INFO
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GROUP_PROPERTIES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_PROPERTIES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_FORMAT_INFO_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_3_PROPERTIES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PROPERTIES_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_FEATURES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PROPERTIES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_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_SPARSE_IMAGE_FORMAT_INFO_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_PROPERTIES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTERS_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_SAMPLER_YCBCR_CONVERSION_INFO
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
• vkCmdBindVertexBuffer
• vkCmdBindRenderPass
• vkCmdBindPipeline
• 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
- 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

1133
• 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
Appendix E: Layers & Extensions (Informative)

Extensions to the Vulkan API can be defined by authors, groups of authors, and the Khronos Vulkan Safety Critical 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/vulkansc/

and allows generating versions of the Specification incorporating different extensions.

Authors creating extensions and layers must follow the mandatory procedures described in the Vulkan Documentation and Extensions document when creating extensions and layers.

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 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.

Extension Dependencies

Extensions which have dependencies on specific core versions or on other extensions will list such dependencies.

For core versions, the specified version must be supported at runtime. All extensions implicitly require support for Vulkan 1.0.

For a device extension, use of any device-level functionality defined by that extension requires that any extensions that extension depends on be enabled.

For any extension, use of any instance-level functionality defined by that extension requires only that any extensions that extension depends on be supported at runtime.

Extension Interactions

Some extensions define APIs which are only supported when other extensions or core versions are supported at runtime. Such interactions are noted as “API Interactions”.

List of Extensions
Appendix F: API Boilerplate

This appendix defines Vulkan API features that are infrastructure required for a complete functional description of Vulkan, but do not logically belong elsewhere in the Specification.

Vulkan Header Files

Vulkan is defined as an API in the C99 language. Khronos provides a corresponding set of header files for applications using the API, which may be used in either C or C++ code. The interface descriptions in the specification are the same as the interfaces defined in these header files, and both are derived from the *vk.xml* XML API Registry, which is the canonical machine-readable description of the Vulkan API. The Registry, scripts used for processing it into various forms, and documentation of the registry schema are available as described at https://registry.khronos.org/vulkansc/#apiregistry.

Language bindings for other languages can be defined using the information in the Specification and the Registry. Khronos does not provide any such bindings, but third-party developers have created some additional bindings.

Vulkan Combined API Header *vulkan_sc.h* (Informative)

Applications normally will include the header *vulkan_sc.h*. In turn, *vulkan_sc.h* always includes the following headers:

- *vk_platform.h*, defining platform-specific macros and headers.
- *vulkan_sc_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 *vulkan_sc.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 *vk_platform.h* (Informative)

Platform-specific macros and interfaces are defined in *vk_platform.h*. These macros are used to control platform-dependent behavior, and their exact definitions are under the control of specific platforms and Vulkan implementations.

Platform-Specific Calling Conventions

On many platforms the following macros are empty strings, causing platform- and compiler-specific default calling conventions to be used.

- **VKAPI_ATTR** is a macro placed before the return type in Vulkan API function declarations. This macro controls calling conventions for C++11 and GCC/Clang-style compilers.
- **VKAPI_CALL** is a macro placed after the return type in Vulkan API function declarations. This macro controls calling conventions for MSVC-style compilers.
VKAPI_PTR is a macro placed between the '(' and '*' in Vulkan API function pointer declarations. This macro also controls calling conventions, and typically has the same definition as VKAPI_ATTR or VKAPI_CALL, depending on the compiler.

With these macros, a Vulkan function declaration takes the form of:

```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_sc_core.h`

Applications that do not make use of window system-specific extensions may simply include `vulkan_sc_core.h` instead of `vulkan_sc.h`, although there is usually no reason to do so. In addition to the Vulkan API, `vulkan_sc_core.h` also defines a small number of C preprocessor macros that are described below.

`vulkan_sc_core.hpp` provides the same functionality as `vulkan_sc_core.h`, but does so in a manner that is aligned for compliance with MISRA C++. In contrast, `vulkan_sc_core.h` is aligned for compliance with MISRA C:2012.

Vulkan Header File Version Number

VK_HEADER_VERSION is the version number of the `vulkan_sc_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 14
```

VK_HEADER_VERSION_COMPLETE is the complete version number of the `vulkan_sc_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(VKSC_API_VARIANT, 1, 0,
VK_HEADER_VERSION)
```

**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
#elif (VK_USE_64_BIT_PTR_DEFINES==1)
#define VK_DEFINE_NON_DISPATCHABLE_HANDLE(object) typedef struct object##_T* (object);
#else
#define VK_DEFINE_NON_DISPATCHABLE_HANDLE(object) typedef uint64_t (object);
#endif
#endif
```

- `object` is the name of the resulting C type.

Most Vulkan handle types, such as `VkBuffer`, are non-dispatchable.

**Note**

The `vulkan_sc_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_sc_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
#endif
#endif
```

VK_DEFINE_NON_DISPATCHABLE_HANDLE 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__) || (defined(__riscv) && __riscv_xlen == 64)
        #define VK_USE_64_BIT_PTR_DEFINES 1
    #else
        #define VK_USE_64_BIT_PTR_DEFINES 0
    #endif
#endif
```

**Note**
The `vulkan_sc_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.

**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_sc_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_sc.h` includes `vulkan_sc_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_sc.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_sc.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_sc.h</code> Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_KHR_android_surface</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>VK_KHR_wayland_surface</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 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 Xr andr.h&gt;</td>
<td>VK Use PLATFORM_XLIB_XRANRD_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>
<tr>
<td>VK_NV external sci_sync, VK_NV external sci_sync2, VK_NV external memory sci_buf</td>
<td>NVIDIA Sci</td>
<td>vulkan sci h</td>
<td>&lt;nvscisync h&gt;, &lt;nvscibuf h&gt;</td>
<td>VK_USE_PLATFORM_SCI</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 provisionally header, `vulkan_beta.h`, allowing applications to decide whether or not to include them. The mechanism is similar to window systemspecific headers: before including `vulkan_beta.h`, applications **must** include `vulkan_sc_core.h`.

**Note**

Sometimes a provisional extension will include a subset of its interfaces in `vulkan_sc_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_sc.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_sc.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.
Appendix G: Invariance

The Vulkan specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different Vulkan implementations. However, the specification does specify exact matches, in some cases, for images produced by the same implementation. The purpose of this appendix is to identify and provide justification for those cases that require exact matches.

Repeatability

The obvious and most fundamental case is repeated issuance of a series of Vulkan commands. For any given Vulkan and framebuffer state vector, and for any Vulkan command, the resulting Vulkan and framebuffer state must be identical whenever the command is executed on that initial Vulkan and framebuffer state. This repeatability requirement does not apply when using shaders containing side effects (image and buffer variable stores and atomic operations), because these memory operations are not guaranteed to be processed in a defined order.

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 \(\text{TessCoord}\) will be in the range \([0, 1]\). Additionally, for any defined component \(x\) of \(\text{TessCoord}\), 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 H: Vulkan SC Deviations From Base Vulkan

Additions

The following extensions have been added to Vulkan SC:

<table>
<thead>
<tr>
<th>Extension</th>
<th>Level</th>
</tr>
</thead>
</table>

The following items have been added to Vulkan SC:

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Additions</th>
</tr>
</thead>
</table>
| **Fundamentals** | • extending VkResult  
  ◦ VK_ERROR_VALIDATION_FAILED [SCID-1]  
  ◦ VK_ERROR_INVALID_PIPELINE_CACHE_DATA [SCID-1]  
  ◦ VK_ERROR_NO_PIPELINE_MATCH [SCID-1] |
| **Devices and Queues** | • VkPhysicalDeviceVulkanSC10Properties [SCID-1]  
  • VkDeviceObjectReservationCreateInfo [SCID-4]  
  • VkPipelinePoolSize [SCID-4] |
| **Command Buffers** | • VkCommandPoolMemoryReservationCreateInfo [SCID-4]  
  • vkGetCommandPoolMemoryConsumption [SCID-1]  
  • VkCommandPoolMemoryConsumption [SCID-1] |
| **Pipelines** | • extending VkPipelineCacheCreateFlagBits  
  ◦ VK_PIPELINE_CACHE_CREATE_READ_ONLY_BIT [SCID-1], [SCID-8]  
  ◦ VK_PIPELINE_CACHE_CREATE_USE_APPLICATION_STORAGE_BIT [SCID-2]  
  • extending VkPipelineCacheHeaderVersion  
  ◦ VK_PIPELINE_CACHE_HEADER_VERSION_SAFETY_CRITICAL_ONE [SCID-1], [SCID-8]  
  ◦ VkPipelineCacheHeaderVersionSafetyCriticalOne [SCID-1], [SCID-8]  
  • VkPipelineCacheValidationVersion [SCID-1], [SCID-8]  
  • VkPipelineCacheSafetyCriticalIndexEntry [SCID-1], [SCID-8]  
  • VkPipelineCacheStageValidationIndexEntry[SCID-1], [SCID-8]  
  • VkPipelineOfflineCreateInfo [SCID-1], [SCID-8]  
  • VkPipelineMatchControl [SCID-1] |
| **Memory Allocation** | • extending VkMemoryHeapFlagBits  
  ◦ VK_MEMORY_HEAP_SEU_SAFE_BIT [SCID-1] |
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Additions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Features</strong></td>
<td>• VkPhysicalDeviceVulkanSC10Features [SCID-1]</td>
</tr>
<tr>
<td><strong>Debugging</strong></td>
<td>• VkFaultData [SCID-6]</td>
</tr>
<tr>
<td></td>
<td>• VkFaultCallbackInfo [SCID-6]</td>
</tr>
<tr>
<td></td>
<td>• VkFaultLevel [SCID-6]</td>
</tr>
<tr>
<td></td>
<td>• VkFaultType [SCID-6]</td>
</tr>
<tr>
<td></td>
<td>• VkFaultQueryBehavior [SCID-6]</td>
</tr>
<tr>
<td></td>
<td>• PFN_vkFaultCallbackFunction [SCID-6]</td>
</tr>
<tr>
<td></td>
<td>• vkGetFaultData [SCID-6]</td>
</tr>
</tbody>
</table>

**Modifications**

The following aspects of Base Vulkan have been modified for Vulkan SC:

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Modifications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fundamentals</strong></td>
<td>If VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations is VK_TRUE, VK_ERROR_OUT_OF_HOST_MEMORY must not be returned by physical or logical device commands which explicitly disallow it [SCID-4].</td>
</tr>
<tr>
<td><strong>Devices and Queues</strong></td>
<td>The VkDeviceCreateInfo::pNext chain must include a VkDeviceObjectReservationCreateInfo structure [SCID-4].</td>
</tr>
<tr>
<td></td>
<td>The VkDeviceCreateInfo::pNext chain must include a VkPhysicalDeviceVulkanSC10Features structure [SCID-1].</td>
</tr>
<tr>
<td></td>
<td>vkCreateDevice returns VK_ERROR_INVALID_PIPELINE_CACHE_DATA if the pInitialData member of any element of VkDeviceObjectReservationCreateInfo::pPipelineCacheCreateInfos is a pointer to incompatible pipeline cache data [SCID-1].</td>
</tr>
<tr>
<td>Chapter</td>
<td>Modifications</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Command Buffers</strong></td>
<td>• The \texttt{VkCommandPoolCreateInfo::pNext} chain must include a valid \texttt{VkCommandPoolMemoryReservationCreateInfo} structure \texttt{[SCID-4]}.</td>
</tr>
<tr>
<td></td>
<td>• If \texttt{commandPoolResetCommandBuffer} is not supported \texttt{[SCID-8]}, \texttt{vkResetCommandBuffer} must not be called.</td>
</tr>
<tr>
<td></td>
<td>• \texttt{vkFreeCommandBuffers} does not return the memory used by command recording back to its parent command pool \texttt{[SCID-4]}. This memory is reclaimed when \texttt{vkResetCommandPool} is next called.</td>
</tr>
<tr>
<td></td>
<td>• If \texttt{VkPhysicalDeviceVulkanSC10Properties::commandPoolMultipleCommandBuffersRecording} is \texttt{VK_FALSE}, then only one command buffer from a command pool can be in the recording state at a time \texttt{[SCID-8]}.</td>
</tr>
<tr>
<td></td>
<td>• If \texttt{VkPhysicalDeviceVulkanSC10Properties::commandBufferSimultaneousUse} is \texttt{VK_FALSE}, then \texttt{VkCommandBufferBeginInfo::flags} must not include \texttt{VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT} \texttt{[SCID-8]}.</td>
</tr>
<tr>
<td></td>
<td>• If \texttt{commandPoolResetCommandBuffer} is not supported, \texttt{commandBuffer} must be in the initial state when \texttt{vkBeginCommandBuffer} is called \texttt{[SCID-8]}.</td>
</tr>
<tr>
<td></td>
<td>• If \texttt{VkPhysicalDeviceVulkanSC10Properties::secondaryCommandBufferNullOrImagelessFramebuffer} is \texttt{VK_FALSE}, then \texttt{VkCommandBufferInheritanceInfo::framebuffer} must not be \texttt{VK_NULL_HANDLE} and \texttt{flags} must not have been created with a \texttt{VkFramebufferCreateInfo::flags} value that includes \texttt{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT} if the command buffer will be executed within a render pass instance \texttt{[SCID-8]}.</td>
</tr>
<tr>
<td>Chapter</td>
<td>Modifications</td>
</tr>
<tr>
<td>---------</td>
<td>---------------</td>
</tr>
</tbody>
</table>
| Pipelines | • **vkCreateComputePipelines** returns `VK_ERROR_NO_PIPELINE_MATCH` if the `VkComputePipelineCreateInfo::pNext` chain does not include a valid `VkPipelineOfflineCreateInfo` structure [SCID-1].
| | • `vkCreateComputePipelines::pipelineCache` must not be `VK_NULL_HANDLE` [SCID-1], [SCID-8].
| | • `VkComputePipelineCreateInfo::basePipelineHandle` must be `VK_NULL_HANDLE` [SCID-8].
| | • `VkComputePipelineCreateInfo::basePipelineIndex` must be zero [SCID-8].
| | • **vkCreateGraphicsPipelines** returns `VK_ERROR_NO_PIPELINE_MATCH` if the `VkGraphicsPipelineCreateInfo::pNext` chain does not include a valid `VkPipelineOfflineCreateInfo` structure [SCID-1].
| | • `vkCreateGraphicsPipelines::pipelineCache` must not be `VK_NULL_HANDLE` [SCID-1], [SCID-8].
| | • `VkGraphicsPipelineCreateInfo::basePipelineHandle` must be `VK_NULL_HANDLE` [SCID-8].
| | • `VkGraphicsPipelineCreateInfo::basePipelineIndex` must be zero [SCID-8].
| | • `VkPipelineCacheCreateInfo::pInitialData` must point to a valid pipeline cache that has been generated offline [SCID-1], [SCID-8].
| | • `VkPipelineCacheCreateInfo::initialDataSize` must not be 0 [SCID-1], [SCID-8].
| | • `VkPipelineCacheCreateInfo::pInitialData` must not be `NULL` [SCID-1], [SCID-8].
| | • `VkPipelineCacheCreateInfo::flags` must include `VK_PIPELINE_CACHE_CREATE_READ_ONLY_BIT` [SCID-1], [SCID-8].
| | • `VkPipelineCacheCreateInfo::flags` must include `VK_PIPELINE_CACHE_CREATE_USE_APPLICATION_STORAGE_BIT` [SCID-2].
| | • The contents of `VkPipelineCacheCreateInfo`, including the data pointed to by `VkPipelineCacheCreateInfo::pInitialData`, passed to `vkCreatePipelineCache` must be the same as specified in one of the `VkDeviceObjectReservationCreateInfo::pPipelineCacheCreateInfos` structures when the device was created [SCID-1].
| | • `VkPipelineCacheHeaderVersionOne::headerSize` must be 56 [SCID-1].
| | • `VkPipelineCacheHeaderVersionOne::headerVersion` must be `VK_PIPELINE_CACHE_HEADER_VERSION_SAFETY_CRITICAL_ONE` [SCID-1].
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Modifications</th>
</tr>
</thead>
</table>
| **Memory Allocation** | • `vkCreate*::pAllocator` must be `NULL` [SCID-2], [SCID-8].  
• `vkDestroy*::pAllocator` must be `NULL` [SCID-2], [SCID-8].  
• `vk*Memory::pAllocator` must be `NULL` [SCID-2], [SCID-8].  
• `vkRegisterDeviceEventEXT::pAllocator` must be `NULL` [SCID-8]. |
| **Resource Creation** | • `VkBufferCreateInfo::flags` must not contain any of the `VK_BUFFER_CREATE_SPARSE_BINDING_BIT`, `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT`, or `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT` flags [SCID-8].  
• `VkImageCreateInfo::flags` must not contain any of the `VK_IMAGE_CREATE_SPARSE_BINDING_BIT`, `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`, `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT`, or `VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT` flags [SCID-8].  
• `VkBindImageMemoryDeviceGroupInfo::splitInstanceBindRegionCount` must be zero [SCID-8]. |
| **Resource Descriptors** | • If `recycleDescriptorSetMemory` is `VK_FALSE`, then freeing a descriptor set does not make the pool memory it used available to be reallocated until the descriptor pool is reset [SCID-4]. |
| **Sparse Resources** | • `VkPhysicalDeviceSparseProperties::residencyStandard2DBlockShape` must be reported as `VK_FALSE` [SCID-8].  
• `VkPhysicalDeviceSparseProperties::residencyStandard2DMultisampleBlockShape` must be reported as `VK_FALSE` [SCID-8].  
• `VkPhysicalDeviceSparseProperties::residencyStandard3DBlockShape` must be reported as `VK_FALSE` [SCID-8].  
• `VkPhysicalDeviceSparseProperties::residencyAlignedMipSize` must be reported as `VK_FALSE` [SCID-8].  
• `VkPhysicalDeviceSparseProperties::residencyNonResidentStrict` must be reported as `VK_FALSE` [SCID-8]. |
| **WSI Swapchain** | • `VkSwapchainCreateInfoKHR::flags` must not contain `VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR` [SCID-8].  
• `VkSwapchainCreateInfoKHR::oldSwapchain` must be `VK_NULL_HANDLE` [SCID-4]. |
### Chapter

<table>
<thead>
<tr>
<th>Features</th>
<th>Modifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>• VkPhysicalDeviceFeatures::shaderResourceResidency must be reported as VK_FALSE [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• VkPhysicalDeviceFeatures::sparseBinding must be reported as VK_FALSE [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• VkPhysicalDeviceFeatures::sparseResidencyBuffer must be reported as VK_FALSE [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• VkPhysicalDeviceFeatures::sparseResidencyImage2D must be reported as VK_FALSE [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• VkPhysicalDeviceFeatures::sparseResidencyImage3D must be reported as VK_FALSE [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• VkPhysicalDeviceFeatures::sparseResidency2Samples must be reported as VK_FALSE [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• VkPhysicalDeviceFeatures::sparseResidency4Samples must be reported as VK_FALSE [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• VkPhysicalDeviceFeatures::sparseResidency8Samples must be reported as VK_FALSE [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• VkPhysicalDeviceFeatures::sparseResidency16Samples must be reported as VK_FALSE [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• VkPhysicalDeviceFeatures::sparseResidencyAliased must be reported as VK_FALSE [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• VkPhysicalDeviceVulkanSC10Features::shaderAtomicInstructions are made optional [SCID-1].</td>
<td></td>
</tr>
<tr>
<td>• VkPhysicalDeviceVulkan11Features::multiview is made optional [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• VkPhysicalDeviceVulkan12Features::timelineSemaphore is made optional [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• VkPhysicalDeviceVulkan12Features::vulkanMemoryModel must be reported as VK_TRUE [SCID-1].</td>
<td></td>
</tr>
<tr>
<td>Limits</td>
<td>Modifications</td>
</tr>
<tr>
<td>----------</td>
<td>---------------</td>
</tr>
<tr>
<td>• VkPhysicalDeviceLimits::maxFramebufferLayers may be 1 if neither geometryShader or shaderOutputLayer are supported [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• VkPhysicalDeviceVulkan12Properties::supportedDepthResolveModes may be only VK_RESOLVE_MODE_NONE [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• VkPhysicalDeviceVulkan12Properties::supportedStencilResolveModes may be only VK_RESOLVE_MODE_NONE [SCID-8].</td>
<td></td>
</tr>
</tbody>
</table>

### Removals

The following functionality has been removed from Base Vulkan in Vulkan SC:
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Removals</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fundamentals</strong></td>
<td>• <strong>VkStructureType</strong> (deprecated aliases)</td>
</tr>
<tr>
<td></td>
<td>◦ <strong>VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTER_FEATURES</strong> [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>◦ <strong>VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DRAW_PARAMETER_FEATURES</strong> [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>◦ <strong>VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES2_EXT</strong> [SCID-8]</td>
</tr>
<tr>
<td><strong>Devices and Queues</strong></td>
<td>• <strong>VkQueueFlagBits</strong></td>
</tr>
<tr>
<td></td>
<td>◦ <strong>VK_QUEUE_SPARSE_BINDING_BIT</strong> [SCID-8]</td>
</tr>
<tr>
<td><strong>Command Buffers</strong></td>
<td>• <strong>vkTrimCommandPool, vkTrimCommandPoolKHR</strong> [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• <strong>VkCommandPoolTrimFlags, VkCommandPoolTrimFlagsKHR</strong> [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• <strong>VkCommandPoolResetFlagBits</strong></td>
</tr>
<tr>
<td></td>
<td>◦ <strong>VK_COMMAND_POOL_RESET_RELEASE_RESOURCES_BIT</strong> [SCID-4]</td>
</tr>
<tr>
<td></td>
<td>• <strong>vkDestroyCommandPool</strong> [SCID-4]</td>
</tr>
<tr>
<td><strong>Synchronization and Cache Control</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Shaders</strong></td>
<td>• <strong>VkStructureType</strong></td>
</tr>
<tr>
<td></td>
<td>◦ <strong>VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO</strong> [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• <strong>VkObjectType</strong></td>
</tr>
<tr>
<td></td>
<td>◦ <strong>VK_OBJECT_TYPE_SHADER_MODULE</strong> [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• <strong>vkCreateShaderModule, vkDestroyShaderModule</strong> [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• <strong>VkShaderModule, VkShaderModuleCreateInfo</strong> [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• <strong>VkShaderModuleCreateFlags</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>VkShaderModuleCreateFlagBits</strong> [SCID-8]</td>
</tr>
<tr>
<td><strong>Pipelines</strong></td>
<td>• <strong>VkPipelineCreateFlagBits</strong></td>
</tr>
<tr>
<td></td>
<td>◦ <strong>VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT</strong> [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>◦ <strong>VK_PIPELINE_CREATE_DERIVATIVE_BIT</strong> [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• <strong>vkMergePipelineCaches, vkGetPipelineCacheData</strong> [SCID-1], [SCID-8]</td>
</tr>
<tr>
<td><strong>Memory Allocation</strong></td>
<td>• <strong>VkSystemAllocationScope, VkInternalAllocationType</strong> [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• <strong>vkFreeMemory</strong> [SCID-4]</td>
</tr>
<tr>
<td>Chapter</td>
<td>Removals</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Queries</td>
<td>• <code>vkDestroyQueryPool</code> [SCID-4]</td>
</tr>
<tr>
<td>Fragment Operations</td>
<td>• <code>VkStencilFaceFlagBits</code> (deprecated alias)&lt;br&gt;◦ <code>VK_STENCIL_FRONT_AND_BACK</code> [SCID-8]</td>
</tr>
</tbody>
</table>

1156
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Removals</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sparse Resources</strong></td>
<td>• VkStructureType</td>
</tr>
<tr>
<td></td>
<td>◦ VK_STRUCTURE_TYPE_BIND_SPARSE_INFO [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>◦ VK_STRUCTURE_TYPE_DEVICE_GROUP_BIND_SPARSE_INFO [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>◦ VK_STRUCTURE_TYPE_IMAGE_SPARSE_MEMORY_REQUIREMENTS_INFO_2 [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SPARSE_IMAGE_FORMAT_INFO_2 [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>◦ VK_STRUCTURE_TYPE_SPARSE_IMAGE_FORMAT_PROPERTIES_2 [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>◦ VK_STRUCTURE_TYPE_SPARSE_IMAGE_MEMORY_REQUIREMENTS_2 [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkSparseImageFormatProperties [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkSparseImageFormatFlagBits [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkSparseImageFormatFlags [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• vkGetPhysicalDeviceSparseImageFormatProperties [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• vkGetPhysicalDeviceSparseImageFormatProperties2 [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkPhysicalDeviceSparseImageFormatInfo2 [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkSparseImageFormatProperties2 [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkSparseImageMemoryRequirements [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• vkGetImageSparseMemoryRequirements [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• vkGetImageSparseMemoryRequirements2 [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkImageSparseMemoryRequirementsInfo2 [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkSparseImageMemoryRequirements2 [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkSparseMemoryBind [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkSparseMemoryBindFlagBits [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkSparseMemoryBindFlags [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkSparseBufferMemoryBindInfo [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkSparseImageOpaqueMemoryBindInfo [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkSparseImageMemoryBindInfo [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkSparseImageMemoryBind [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• vkQueueBindSparse [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkBindSparseInfo [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkDeviceGroupBindSparseInfo [SCID-8]</td>
</tr>
<tr>
<td><strong>Window System</strong></td>
<td>• VkColorSpaceKHR (deprecated aliases)</td>
</tr>
<tr>
<td></td>
<td>◦ VK_COLORSPACE_SRGB_NONLINEAR_KHR [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>◦ VK_COLOR_SPACE_DCI_P3_LINEAR_EXT [SCID-8]</td>
</tr>
</tbody>
</table>
Extension Support

Vulkan SC supports a subset of the extensions supported in Base Vulkan. This subset was decided by:

- Excluding any extensions that would pose significant difficulty to certify their implementations.
- Excluding any extension that would not be used in deployed devices. This was primarily extensions focused on application development and debug.
- Excluding any extensions that are specific to an Operating System or Windowing system that is highly unlikely to be used in the Safety Critical space.
- Non-KHR or EXT extension are supported on request.

Note
During development it is likely that application developers will need additional functionality in a Vulkan SC implementation beyond what is provided by the supported extensions. This can be achieved by implementing a development focused version of the implementation that exposes additional Vulkan extensions and tools support but is non-conformant to the Vulkan SC specification.

A Vulkan SC conformant implementation with this additional functionality removed will be used on the end device.

Fault and Error Handling

Vulkan SC maintains the use of VkResult Return Codes on a small number of commands. These allow the command to confirm it completed successfully or return an error code for situations where a failure could be detected at runtime during the execution of the command.

In addition to VkResult Return Codes Vulkan SC adds Fault Handling support. This provides the implementation the ability to communicate information on errors or faults to the application that have been detected but are not covered by VkResult Return Codes in the Vulkan SC API. These could be runtime failures of the system or application faults that are detected asynchronously to the Vulkan API commands.

Undefined Behavior in the API

If an application uses the API incorrectly the behavior of the API is undefined. The Vulkan SC runtime will perform minimal error and state checking and it is assumed that applications are using the API correctly, see Valid Usage.

With incorrect input to the API, the implementation could continue to function correctly, generate unexpected output, become unstable, or be terminated. The exact behavior will vary and be
dependent on the specifics of the invalid usage and the implementation.

It is primarily the application's responsibility to ensure it always uses the API correctly. Potential methods to detect incorrect API usage include performing manual code inspection, use of validation layers during development, use of validation layers at runtime, or adding runtime checking to the application. Outside of this, Vulkan SC implementations can add implementation-specific targeted checks to detect invalid API usage that could significantly impact the correct operation of the application or implementation. The Fault Handling extension allows implementations to communicate information on such occurrences.

**MISRA C:2012 Deviations**

*vulkan_sc_core.h* is intended to be compatible with safety coding standards like MISRA C:2012.

The following provides information on items a MISRA C code analysis tool may report for a project using Vulkan SC.


**Directives**

<table>
<thead>
<tr>
<th>Directive</th>
<th>4.6: &quot;typedefs that indicate size and signedness should be used in place of the basic numerical types&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>Advisory</td>
</tr>
<tr>
<td>Note</td>
<td>This is reported for every <code>char</code> and <code>float</code> variable used in the API.</td>
</tr>
<tr>
<td>Rationale</td>
<td>Vulkan SC maintains the Base Vulkan type conventions for compatibility between APIs.</td>
</tr>
</tbody>
</table>

**Rules**

<table>
<thead>
<tr>
<th>Rule</th>
<th>2.3: &quot;A project should not contain unused type declarations&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>Advisory</td>
</tr>
<tr>
<td>Note</td>
<td>This is reported for any unused type definitions.</td>
</tr>
<tr>
<td>Rationale</td>
<td>The <em>vulkan_sc_core.h</em> provides a complete API definition and it is expected that an application may not use all the provided type declarations.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>2.4: &quot;A project should not contain unused tag declarations&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>Advisory</td>
</tr>
<tr>
<td>Note</td>
<td>This is reported for each instance of <code>typedef struct VkStruct { … } VkStruct;</code> and <code>typedef enum VkEnum { … } VkEnum;</code> where the tag declaration is unused.</td>
</tr>
<tr>
<td>Rule</td>
<td>Note</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td><strong>2.4: &quot;A project should not contain unused tag declarations&quot;</strong></td>
<td>The <code>vulkan_sc_core.h</code> provides a complete API definition and it is expected that an application <strong>may</strong> not use all the provided tag declarations. Vulkan SC maintains the Base Vulkan type conventions for compatibility between APIs. Tag declarations are required in case an application wishes to make forward declarations to API-defined types.</td>
</tr>
</tbody>
</table>
| **2.5: "A project should not contain unused macro declarations"** | **Category**: Advisory  
**Note**: This is reported for every unused macro defined in the header.  
**Rationale**: The `vulkan_sc_core.h` provides a complete API definition and it is expected that an application **may** not use all the provided macro declarations. |
| **5.1: "External identifiers shall be distinct"** | **Category**: Required  
**Note**: This is reported for identifiers with names that do not differ in the first 31 characters, such as `vkGetPhysicalDeviceFormatProperties` and `vkGetPhysicalDeviceFormatProperties2`.  
**Rationale**: Vulkan SC maintains the Base Vulkan naming conventions for compatibility between APIs. Vulkan SC applications **must** be built using a compiler that treats enough characters as significant. |
| **5.2: "Identifiers declared in the same scope and name space shall be distinct"** | **Category**: Required  
**Note**: This is reported for many `typedef` statements with long identifiers.  
**Rationale**: Vulkan SC maintains the Base Vulkan type and naming conventions for compatibility between APIs. Vulkan SC applications **must** be built using a compiler that treats enough characters as significant. |
| **5.4: "Macro identifiers shall be distinct"** | **Category**: Required  
**Note**: This is reported for macros with names that do not differ in the first 31 characters, such as `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT` and `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`.  
**Rationale**: Vulkan SC maintains the Base Vulkan naming conventions for compatibility between APIs. Vulkan SC applications **must** be built using a compiler that treats enough characters as significant. |
<p>| <strong>8.6: &quot;An identifier with external linkage shall have exactly one external definition&quot;</strong> | <strong>Category</strong>: Required |</p>
<table>
<thead>
<tr>
<th>Rule</th>
<th>8.6: &quot;An identifier with external linkage shall have exactly one external definition&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note</td>
<td>This is reported for every API entry point declaration, and the external definitions are provided by the implementation.</td>
</tr>
<tr>
<td>Rationale</td>
<td>It is expected that a Vulkan SC application will link against an implementation that provides these definitions.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>19.2: &quot;The union keyword should not be used&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>Advisory</td>
</tr>
<tr>
<td>Note</td>
<td>This is reported on the VkClearColorValue, VkClearValue, and VkPerformanceCounterResultKHR unions.</td>
</tr>
<tr>
<td>Rationale</td>
<td>These are required to remain compatible with the Base Vulkan API.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>20.1: &quot;#include directives should only be preceded by preprocessor directives or comments&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>Advisory</td>
</tr>
<tr>
<td>Note</td>
<td>This is reported because the entire Vulkan SC API definition is wrapped in an extern &quot;C&quot; block.</td>
</tr>
<tr>
<td>Rationale</td>
<td>This is expected because the Vulkan SC API is a C ABI and the header may be included from C++ code.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>20.10: &quot;The # and ## preprocessor operators should not be used&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>Advisory</td>
</tr>
<tr>
<td>Note</td>
<td>This is reported for the two lines:</td>
</tr>
<tr>
<td></td>
<td>#define VK_DEFINE_HANDLE(object) typedef struct object##_T* (object);</td>
</tr>
<tr>
<td></td>
<td>#define VK_DEFINE_NON_DISPATCHABLE_HANDLE(object) typedef struct object##_T *(object);</td>
</tr>
<tr>
<td>Rationale</td>
<td>This is expected usage of the macro expansion operation and there are not multiple operators used in the statement.</td>
</tr>
</tbody>
</table>
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 or unsigned. 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, and using the `vkBindImageMemory` command for non-sparse image objects.

**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 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 (Position decoration) are written by pre-rasterization shader stages.

Clip Distance
A built-in output from 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 VkFormat where VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT is set in one of the following, depending on the image’s tiling:

- VkFormatProperties::linearTilingFeatures
- 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 VkCommandBuffer object.

Command Pool
An object that command buffer memory is allocated from, and that owns that memory. Command pools aid multithreaded performance by enabling different threads to use different allocators, without internal synchronization on each use. Represented by a VkCommandPool object.
**Compatible Allocator**

When allocators are compatible, allocations from each allocator *can* be freed by the other allocator.

**Compatible Image Formats**

When formats are compatible, images created with one of the formats *can* have image views created from it using any of the compatible formats. Also see *Size-Compatible Image Formats*.

**Compatible Queues**

Queues within a queue family. Compatible queues have identical properties.

**Complete Mipmap Chain**

The entire set of mip levels that can be provided for an image, from the largest application specified mip level size down to the *minimum mip level size*. See *Image Mip Level 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. Additionally, physical-device-level functionality defined by a device extension is also considered device-level functionality.

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
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 $Y' C_a C_b C_r$ 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
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’C₆C₈ 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.

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 Mip Level Size
The smallest size that is permitted for a mip level. For conventional images this is 1x1x1. See Image Mip Level 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’C₆C₈ 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 JSON Schema**

A JSON-based representation for encapsulating all pipeline state which is necessary for the offline pipeline cache compiler. This includes the SPIR-V shader module, pipeline layout, render pass information and pipeline state creation information.

**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.

**Pipeline Identifier**

An identifier that can be used to identify a specific pipeline independently from the pipeline description.

**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 Event Upset**

A change of physical device state, such as a register or memory bitflip, e.g. caused by ionizing radiation.

**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 SC 1.0 is the result of contributions from many people and companies participating in the Khronos Vulkan SC Working Group, building upon the Base Vulkan specification produced by the Khronos Vulkan Working Group, as well as input from the Vulkan Advisory Panel.

Members of the Working Group, including the company that they represented at the time of their most recent contribution, are listed in the following sections. Some specific contributions made by individuals are listed together with their name.

**Working Group Contributors to Vulkan SC 1.0**

- Aarusha Thakral, CoreAVI
- Aidan Fabius, CoreAVI
- Alastair Donaldson, Google
- Alastair Murray, Codeplay Software Ltd.
- Alex Crabb, Khronos
- Alexander Galazin, Arm
- Alis Ors, NXP Semiconductors
- Alon Or-bach, Samsung Electronics
- Andrew Garrard, Imagination Technologies
- Anna Buczkowska, Mobica
- Balajee Gurumoorah, Huawei Technologies Co., Ltd.
- Bogdan Naodovic, NVIDIA
- Boris Zanin, Mobica
- Brad Cain, NVIDIA
- Cary Ashby, Collins Aerospace
- Chris Forbes, Google
- Craig Davies, Huawei Technologies Co., Ltd.
- Daniel Bernal, Arm
- Daniel Koch, NVIDIA
- Dave Higham, Imagination Technologies
- Dave McCloskey, Juice Labs
- David Hayward, Imagination Technologies
- Donald Scorgie, Imagination Technologies
- Doug Singkofer, Collins Aerospace
- Emily Stearns, Khronos
• Erik Tomusk, Codeplay Software Ltd.
• Ewa Galamon, Mobica
• Greg Szober, CoreAVI
• Illya Rudkin, Codeplay Software Ltd.
• Jacek Wisniewski, Mobica
• James Helferty, NVIDIA
• Jan Hemes, Continental Corporation
• Jan-Harald Fredriksen, Arm
• Janos Lakatos, Imagination Technologies
• Jeff Bolz, NVIDIA
• Jim Carroll, Mobica
• John Zulauf, LunarG
• Jon Leech, Independent (XML toolchain, normative language, release wrangler)
• Jun Wang, Huawei Technologies Co., Ltd.
• Karen Ghavam, LunarG
• Karolina Palka, Mobica
• Ken Wenger, CoreAVI
• Lenny Komow, LunarG
• Lilja Tamminen, Basemark Oy
• Luca Di Mauro, Arm
• Lukasz Janyst, Daedalean
• Mark Bellamy, Arm
• Matthew Netsch, Qualcomm Technologies, Inc.
• Michael Wong, Codeplay Software Ltd.
• Mukund Keshava, NVIDIA
• Neil Stroud, CoreAVI
• Neil Trevett, NVIDIA
• Nick Blurton-Jones, CoreAVI
• Pawel Ksiezopolski, Mobica
• Piotr Byszewski, Mobica
• Rob Simpson, Qualcomm Technologies, Inc.
• Stephne Strahn, Kalray
• Steve Viggers, CoreAVI (working group chair)
• Tim Lewis, Khronos
• Todd Brown, Collins Aerospace
• Tom Malnar, CoreAVI
• Tom Olson, Arm
• Tony Zlatinski, NVIDIA
• Vladyslav Zakkarchenko, Huawei Technologies Co., Ltd.

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)
• Lina 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, LunarG (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)
• Faith 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
- 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 Przybylski, 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 (Sheng-Wen) 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)
• Sujeewan 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)
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.

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.