# 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 ..................................... 14
  3.6. Threading Behavior ...................................................... 15
  3.7. Valid Usage ................................................................. 24
  3.8. VkResult Return Codes ................................................. 30
  3.9. Numeric Representation and Computation ...................... 34
  3.10. Fixed-Point Data Conversions ...................................... 35
  3.11. Common Object Types ................................................ 37
  3.12. API Name Aliases ........................................................ 52
4. Initialization ........................................................................... 53
  4.1. Command Function Pointers ......................................... 53
  4.2. Instances ........................................................................ 56
5. Devices and Queues ................................................................ 67
  5.1. Physical Devices ............................................................ 67
  5.2. Devices ......................................................................... 98
  5.3. Queues ......................................................................... 117
6. Command Buffers .................................................................. 126
  6.1. Command Buffer Lifecycle ............................................ 126
  6.2. Command Pools ............................................................. 128
  6.3. Command Buffer Allocation and Management ............... 136
  6.4. Command Buffer Recording ......................................... 141
  6.5. Command Buffer Submission ....................................... 148
  6.6. Queue Forward Progress ............................................... 165
  6.7. Secondary Command Buffer Execution ......................... 166
  6.8. Command Buffer Device Mask ...................................... 169
7. Synchronization and Cache Control ....................................... 172
  7.1. Execution and Memory Dependencies ............................. 172
  7.2. Implicit Synchronization Guarantees ............................... 194
  7.3. Fences ........................................................................ 196
  7.4. Semaphores ................................................................... 226
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.11. Pipeline Binding</td>
<td>476</td>
</tr>
<tr>
<td>10.12. Dynamic State</td>
<td>479</td>
</tr>
<tr>
<td>11. Memory Allocation</td>
<td>480</td>
</tr>
<tr>
<td>11.1. Host Memory</td>
<td>480</td>
</tr>
<tr>
<td>11.2. Device Memory</td>
<td>480</td>
</tr>
<tr>
<td>12. Resource Creation</td>
<td>526</td>
</tr>
<tr>
<td>12.1. Buffers</td>
<td>526</td>
</tr>
<tr>
<td>12.2. Buffer Views</td>
<td>533</td>
</tr>
<tr>
<td>12.3. Images</td>
<td>538</td>
</tr>
<tr>
<td>12.4. Image Layouts</td>
<td>566</td>
</tr>
<tr>
<td>12.5. Image Views</td>
<td>570</td>
</tr>
<tr>
<td>12.6. Resource Memory Association</td>
<td>587</td>
</tr>
<tr>
<td>12.7. Resource Sharing Mode</td>
<td>613</td>
</tr>
<tr>
<td>12.8. Memory Aliasing</td>
<td>615</td>
</tr>
<tr>
<td>13. Samplers</td>
<td>618</td>
</tr>
<tr>
<td>13.1. Sampler Y′C_bC_r conversion</td>
<td>628</td>
</tr>
<tr>
<td>14. Resource Descriptors</td>
<td>637</td>
</tr>
<tr>
<td>14.1. Descriptor Types</td>
<td>637</td>
</tr>
<tr>
<td>14.2. Descriptor Sets</td>
<td>640</td>
</tr>
<tr>
<td>14.3. Physical Storage Buffer Access</td>
<td>694</td>
</tr>
<tr>
<td>15. Shader Interfaces</td>
<td>697</td>
</tr>
<tr>
<td>15.1. Shader Input and Output Interfaces</td>
<td>697</td>
</tr>
<tr>
<td>15.2. Vertex Input Interface</td>
<td>701</td>
</tr>
<tr>
<td>15.3. Fragment Output Interface</td>
<td>701</td>
</tr>
<tr>
<td>15.4. Fragment Input Attachment Interface</td>
<td>702</td>
</tr>
<tr>
<td>15.5. Shader Resource Interface</td>
<td>703</td>
</tr>
<tr>
<td>15.6. Built-In Variables</td>
<td>711</td>
</tr>
<tr>
<td>16. Image Operations</td>
<td>736</td>
</tr>
<tr>
<td>16.1. Image Operations Overview</td>
<td>736</td>
</tr>
<tr>
<td>16.2. Conversion Formulas</td>
<td>739</td>
</tr>
<tr>
<td>16.3. Texel Input Operations</td>
<td>741</td>
</tr>
<tr>
<td>16.4. Texel Output Operations</td>
<td>759</td>
</tr>
<tr>
<td>16.5. Normalized Texel Coordinate Operations</td>
<td>760</td>
</tr>
<tr>
<td>16.6. Unnormalized Texel Coordinate Operations</td>
<td>767</td>
</tr>
<tr>
<td>16.7. Integer Texel Coordinate Operations</td>
<td>769</td>
</tr>
<tr>
<td>16.8. Image Sample Operations</td>
<td>769</td>
</tr>
<tr>
<td>16.9. Image Operation Steps</td>
<td>773</td>
</tr>
<tr>
<td>16.10. Image Query Instructions</td>
<td>774</td>
</tr>
<tr>
<td>17. Queries</td>
<td>775</td>
</tr>
<tr>
<td>17.1. Query Pools</td>
<td>775</td>
</tr>
<tr>
<td>17.2. Query Operation</td>
<td>780</td>
</tr>
</tbody>
</table>
23.3. Multiple Invocations of Geometry Shaders ........................................ 996
23.4. Geometry Shader Primitive Ordering .............................................. 996
24. Fixed-Function Vertex Post-Processing ................................................. 997
  24.1. Flat Shading .............................................................................. 997
  24.2. Primitive Clipping ...................................................................... 997
  24.3. Clipping Shader Outputs ............................................................. 999
  24.4. Coordinate Transformations ......................................................... 1000
  24.5. Controlling the Viewport ............................................................. 1001
25. Rasterization ..................................................................................... 1010
  25.1. Discarding Primitives Before Rasterization .................................. 1015
  25.2. Rasterization Order .................................................................... 1016
  25.3. Multisampling ........................................................................... 1016
  25.4. Custom Sample Locations .......................................................... 1021
  25.5. Fragment Shading Rates ............................................................. 1024
  25.6. Sample Shading ......................................................................... 1032
  25.7. Points ......................................................................................... 1033
  25.8. Line Segments ............................................................................ 1034
  25.9. Polygons ..................................................................................... 1043
26. Fragment Operations .......................................................................... 1056
  26.1. Discard Rectangles Test ............................................................... 1057
  26.2. Scissor Test ................................................................................ 1060
  26.3. Sample Mask Test ...................................................................... 1062
  26.4. Fragment Shading ...................................................................... 1062
  26.5. Multisample Coverage ............................................................... 1064
  26.6. Depth and Stencil Operations ..................................................... 1065
  26.7. Depth Bounds Test .................................................................... 1066
  26.8. Stencil Test ................................................................................ 1069
  26.9. Depth Test ................................................................................. 1078
  26.10. Sample Counting ...................................................................... 1082
  26.11. Coverage Reduction .................................................................. 1082
27. The Framebuffer ................................................................................. 1084
  27.1. Blending ..................................................................................... 1084
  27.2. Logical Operations ..................................................................... 1103
  27.3. Color Write Mask ....................................................................... 1106
  27.4. Color Write Enable ..................................................................... 1107
28. Dispatching Commands ....................................................................... 1111
29. Sparse Resources ............................................................................... 1126
  29.1. Sparse Resource Features ......................................................... 1126
  29.2. Sparse Resource API ................................................................. 1128
30. Window System Integration (WSI) ...................................................... 1130
  30.1. WSI Platform ............................................................................ 1130
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.2. WSI Surface</td>
<td>1130</td>
</tr>
<tr>
<td>30.3. Presenting Directly to Display Devices</td>
<td>1132</td>
</tr>
<tr>
<td>30.4. Querying for WSI Support</td>
<td>1158</td>
</tr>
<tr>
<td>30.5. Surface Queries</td>
<td>1159</td>
</tr>
<tr>
<td>30.6. Device Group Queries</td>
<td>1179</td>
</tr>
<tr>
<td>30.7. WSI Swapchain</td>
<td>1184</td>
</tr>
<tr>
<td>30.8. Hdr Metadata</td>
<td>1220</td>
</tr>
<tr>
<td>31. Extending Vulkan</td>
<td>1223</td>
</tr>
<tr>
<td>31.1. Instance and Device Functionality</td>
<td>1223</td>
</tr>
<tr>
<td>31.2. Core Versions</td>
<td>1223</td>
</tr>
<tr>
<td>31.3. Layers</td>
<td>1226</td>
</tr>
<tr>
<td>31.4. Extensions</td>
<td>1229</td>
</tr>
<tr>
<td>31.5. Extension Dependencies</td>
<td>1233</td>
</tr>
<tr>
<td>31.6. Compatibility Guarantees (Informative)</td>
<td>1234</td>
</tr>
<tr>
<td>32. Features</td>
<td>1239</td>
</tr>
<tr>
<td>32.1. Feature Requirements</td>
<td>1303</td>
</tr>
<tr>
<td>33. Limits</td>
<td>1306</td>
</tr>
<tr>
<td>33.1. Limit Requirements</td>
<td>1342</td>
</tr>
<tr>
<td>33.2. Additional Multisampling Capabilities</td>
<td>1357</td>
</tr>
<tr>
<td>34. Formats</td>
<td>1360</td>
</tr>
<tr>
<td>34.1. Format Definition</td>
<td>1360</td>
</tr>
<tr>
<td>34.2. Format Properties</td>
<td>1407</td>
</tr>
<tr>
<td>34.3. Required Format Support</td>
<td>1415</td>
</tr>
<tr>
<td>35. Additional Capabilities</td>
<td>1435</td>
</tr>
<tr>
<td>35.1. Additional Image Capabilities</td>
<td>1435</td>
</tr>
<tr>
<td>35.2. Additional Buffer Capabilities</td>
<td>1453</td>
</tr>
<tr>
<td>35.3. Optional Semaphore Capabilities</td>
<td>1455</td>
</tr>
<tr>
<td>35.4. Optional Fence Capabilities</td>
<td>1459</td>
</tr>
<tr>
<td>35.5. Timestamp Calibration Capabilities</td>
<td>1463</td>
</tr>
<tr>
<td>35.6. Object Refresh Capabilities</td>
<td>1464</td>
</tr>
<tr>
<td>36. Debugging</td>
<td>1466</td>
</tr>
<tr>
<td>36.1. Debug Utilities</td>
<td>1468</td>
</tr>
<tr>
<td>36.2. Fault Handling</td>
<td>1487</td>
</tr>
<tr>
<td>Appendix A: Vulkan Environment for SPIR-V</td>
<td>1495</td>
</tr>
<tr>
<td>Versions and Formats</td>
<td>1495</td>
</tr>
<tr>
<td>Capabilities</td>
<td>1495</td>
</tr>
<tr>
<td>Validation Rules within a Module</td>
<td>1502</td>
</tr>
<tr>
<td>Precision and Operation of SPIR-V Instructions</td>
<td>1516</td>
</tr>
<tr>
<td>Signedness of SPIR-V Image Accesses</td>
<td>1521</td>
</tr>
<tr>
<td>Image Format and Type Matching</td>
<td>1521</td>
</tr>
<tr>
<td>Compatibility Between SPIR-V Image Formats And Vulkan Formats</td>
<td>1523</td>
</tr>
<tr>
<td>Appendix B: Memory Model</td>
<td>1525</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Agent</td>
<td>1525</td>
</tr>
<tr>
<td>Memory Location</td>
<td>1525</td>
</tr>
<tr>
<td>Allocation</td>
<td>1525</td>
</tr>
<tr>
<td>Memory Operation</td>
<td>1525</td>
</tr>
<tr>
<td>Reference</td>
<td>1526</td>
</tr>
<tr>
<td>Program-Order</td>
<td>1526</td>
</tr>
<tr>
<td>Scope</td>
<td>1527</td>
</tr>
<tr>
<td>Atomic Operation</td>
<td>1527</td>
</tr>
<tr>
<td>Scoped Modification Order</td>
<td>1528</td>
</tr>
<tr>
<td>Memory Semantics</td>
<td>1528</td>
</tr>
<tr>
<td>Release Sequence</td>
<td>1530</td>
</tr>
<tr>
<td>Synchronizes-With</td>
<td>1530</td>
</tr>
<tr>
<td>System-Synchronizes-With</td>
<td>1532</td>
</tr>
<tr>
<td>Private vs. Non-Private</td>
<td>1532</td>
</tr>
<tr>
<td>Inter-Thread-Happens-Before</td>
<td>1533</td>
</tr>
<tr>
<td>Happens-Before</td>
<td>1533</td>
</tr>
<tr>
<td>Availability and Visibility</td>
<td>1533</td>
</tr>
<tr>
<td>Availability, Visibility, and Domain Operations</td>
<td>1535</td>
</tr>
<tr>
<td>Availability and Visibility Semantics</td>
<td>1536</td>
</tr>
<tr>
<td>Per-Instruction Availability and Visibility Semantics</td>
<td>1537</td>
</tr>
<tr>
<td>Location-Ordered</td>
<td>1537</td>
</tr>
<tr>
<td>Data Race</td>
<td>1538</td>
</tr>
<tr>
<td>Visible-To</td>
<td>1538</td>
</tr>
<tr>
<td>Acyclicity</td>
<td>1539</td>
</tr>
<tr>
<td>Shader I/O</td>
<td>1540</td>
</tr>
<tr>
<td>Deallocation</td>
<td>1540</td>
</tr>
<tr>
<td>Descriptions (Informative)</td>
<td>1540</td>
</tr>
<tr>
<td>Tessellation Output Ordering</td>
<td>1541</td>
</tr>
<tr>
<td>Appendix C: Compressed Image Formats</td>
<td>1542</td>
</tr>
<tr>
<td>Block-Compressed Image Formats</td>
<td>1543</td>
</tr>
<tr>
<td>ETC Compressed Image Formats</td>
<td>1544</td>
</tr>
<tr>
<td>ASTC Compressed Image Formats</td>
<td>1545</td>
</tr>
<tr>
<td>Appendix D: Core Revisions (Informative)</td>
<td>1548</td>
</tr>
<tr>
<td>Version 1.2</td>
<td>1548</td>
</tr>
<tr>
<td>Version 1.1</td>
<td>1556</td>
</tr>
<tr>
<td>Version 1.0</td>
<td>1567</td>
</tr>
<tr>
<td>Appendix E: Layers &amp; Extensions (Informative)</td>
<td>1581</td>
</tr>
<tr>
<td>List of Current Extensions</td>
<td>1581</td>
</tr>
<tr>
<td>List of Deprecated Extensions</td>
<td>1758</td>
</tr>
<tr>
<td>Appendix F: API Boilerplate</td>
<td>1762</td>
</tr>
</tbody>
</table>
Chapter 1. Preamble

Copyright 2014-2021 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, and which defines the terms 'Scope', 'Compliant Portion', and 'Necessary Patent Claims'. Parties desiring to implement the Specification and make use of Khronos trademarks in relation to that implementation, and receive reciprocal patent license protection under the Khronos Intellectual Property Rights Policy must become Adopters and confirm the implementation as conformant under the process defined by Khronos for this Specification; see https://www.khronos.org/adopters.

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

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

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

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

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

This document contains extensions which are not ratified by Khronos, and as such is not a ratified Specification, though it contains text from (and is a superset of) the ratified Vulkan SC Specification. The ratified versions of the Vulkan SC Specification can be found at https://www.khronos.org/registry/vulkansc/specs/1.0/html/vkspec.html (core only) and https://www.khronos.org/registry/vulkansc/specs/1.0-khr-extensions/html/vkspec.html (core with KHR extensions).

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


The canonical version of the Specification is available in the official Vulkan SC Registry (https://www.khronos.org/registry/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 1.0.12 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.

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

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

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

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

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

*Note*

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

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

Synchronization commands introduce explicit **execution and memory dependencies** between two sets of action commands, where the second set of commands depends on the first set of commands. These dependencies enforce 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 `vkCreate*` and `vkAllocate*` commands, respectively. Once an object is created or allocated, its “structure” is considered to be immutable, though the contents of certain object types is still free to change. Objects are destroyed or freed by `vkDestroy*` and `vkFree*` commands, respectively.

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

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 `VkDeviceObjectReservationCreateInfo` structure provides upper bounds on the simultaneous number of objects of each type that can be allocated during the lifetime of the `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:

- **VkPipelineCache**

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

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

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

  `VkDescriptorSetLayout` objects may be accessed by commands that operate on descriptor sets allocated using that layout, and those descriptor sets must not be updated with `vkUpdateDescriptorSets` after the descriptor set layout has been destroyed. Otherwise, a `VkDescriptorSetLayout` object passed as a parameter to create another object is not further accessed by that object after the duration of the command it is passed into.
The application **must** not destroy any other type of Vulkan object until all uses of that object by the device (such as via command buffer execution) have completed.

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

- VkEvent
- 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, swapchains (**VkSwapchainKHR**), and pool objects (**VkCommandPool**, **VkDescriptorPool**, **VkSemaphoreSciSyncPoolNV**, **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.

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

3.4. Application Binary Interface

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**VkDeviceAddress** represents device buffer address values:

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

Commands that create Vulkan objects are of the form **vkCreate** and take **Vk** CreateInfo structures with the parameters needed to create the object. These Vulkan objects are destroyed with commands of the form **vkDestroy**.
The last in-parameter to each command that creates or destroys a Vulkan object is \texttt{pAllocator}. The \texttt{pAllocator} parameter \textbf{must} be set to \texttt{NULL}. Refer to the Memory Allocation chapter for further details.

Commands that allocate Vulkan objects owned by pool objects are of the form \texttt{vkAllocate*}, and take \texttt{Vk*AllocateInfo} structures. These Vulkan objects are freed with commands of the form \texttt{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 \texttt{vkCmd*}. Each such command \textbf{may} have different restrictions on where it \textbf{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 \textit{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 \texttt{vkGet*} and \texttt{vkEnumerate*}.

Unless otherwise specified for an individual command, the results are \textit{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 \textit{externally synchronized}. This means that the caller \textbf{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 \textbf{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 \textbf{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).

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

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

Parameters of commands that are externally synchronized are listed below.

---

### Externally Synchronized Parameters

- The *instance* parameter in `vkDestroyInstance`
- The *device* parameter in `vkDestroyDevice`
- The *queue* parameter in `vkQueueSubmit`
- The *fence* parameter in `vkQueueSubmit`
- The *queue* parameter in `vkQueueWaitIdle`
- The *memory* parameter in `vkMapMemory`
- The *memory* parameter in `vkUnmapMemory`
- The *buffer* parameter in `vkBindBufferMemory`
- The *image* parameter in `vkBindImageMemory`
- The *fence* parameter in `vkDestroyFence`
- The *semaphore* parameter in `vkDestroySemaphore`
- The *event* parameter in `vkDestroyEvent`
- The *event* parameter in `vkSetEvent`
- The *event* parameter in `vkResetEvent`
- The *buffer* parameter in `vkDestroyBuffer`
- The *bufferView* parameter in `vkDestroyBufferView`
- The *image* parameter in `vkDestroyImage`
- The *imageView* parameter in `vkDestroyImageView`
- The *pipelineCache* parameter in `vkDestroyPipelineCache`
- The *pipeline* parameter in `vkDestroyPipeline`
• The `pipelineLayout` parameter in `vkDestroyPipelineLayout`
• The `sampler` parameter in `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 `vkCmdBindVertexBuffers`
• The `commandBuffer` parameter in `vkCmdDraw`
• The `commandBuffer` parameter in `vkCmdDrawIndexed`
• The `commandBuffer` parameter in `vkCmdDrawIndirect`
• The `commandBuffer` parameter in `vkCmdDispatch`
• The `commandBuffer` parameter in `vkCmdDispatchIndirect`
• The `commandBuffer` parameter in `vkCmdCopyBuffer`
• The `commandBuffer` parameter in `vkCmdCopyImage`
• The `commandBuffer` parameter in `vkCmdBlitImage`
• The `commandBuffer` parameter in `vkCmdCopyBufferToImage`
• The `commandBuffer` parameter in `vkCmdCopyImageToBuffer`
• The `commandBuffer` parameter in `vkCmdUpdateBuffer`
• The `commandBuffer` parameter in `vkCmdFillBuffer`
• The `commandBuffer` parameter in `vkCmdClearColorImage`
• The `commandBuffer` parameter in `vkCmdClearDepthStencilImage`
• The `commandBuffer` parameter in `vkCmdClearAttachments`
• The `commandBuffer` parameter in `vkCmdResolveImage`
• The `commandBuffer` parameter in `vkCmdSetEvent`
• The `commandBuffer` parameter in `vkCmdResetEvent`
• The `commandBuffer` parameter in `vkCmdWaitEvents`
• The `commandBuffer` parameter in `vkCmdPipelineBarrier`
• The `commandBuffer` parameter in `vkCmdBeginQuery`
• The `commandBuffer` parameter in `vkCmdEndQuery`
• The `commandBuffer` parameter in `vkCmdResetQueryPool`
• The `commandBuffer` parameter in `vkCmdWriteTimestamp`
• The `commandBuffer` parameter in `vkCmdCopyQueryPoolResults`
• The `commandBuffer` parameter in `vkCmdPushConstants`
• The `commandBuffer` parameter in `vkCmdBeginRenderPass`
• The `commandBuffer` parameter in `vkCmdNextSubpass`
• The `commandBuffer` parameter in `vkCmdEndRenderPass`
• The `commandBuffer` parameter in `vkCmdExecuteCommands`
• The `commandBuffer` parameter in `vkCmdSetDeviceMask`
• The `commandBuffer` parameter in `vkCmdDispatchBase`
• The `ycbcrConversion` parameter in `vkDestroySamplerYcbcrConversion`
• The `commandBuffer` parameter in `vkCmdDrawIndirectCount`
• The `commandBuffer` parameter in `vkCmdDrawIndexedIndirectCount`
• The `commandBuffer` parameter in `vkCmdBeginRenderPass2`
• The `commandBuffer` parameter in `vkCmdNextSubpass2`
• The `commandBuffer` parameter in `vkCmdEndRenderPass2`
• The `commandPool` parameter in `vkGetCommandPoolMemoryConsumption`
• The `commandBuffer` parameter in `vkGetCommandPoolMemoryConsumption`
• The `surface` parameter in `vkDestroySurfaceKHR`
• The `surface` member of the `pCreateInfo` parameter in `vkCreateSwapchainKHR`
• The swapchain parameter in `vkAcquireNextImageKHR`
• The semaphore parameter in `vkAcquireNextImageKHR`
• The fence parameter in `vkAcquireNextImageKHR`
• The queue parameter in `vkQueuePresentKHR`
• The surface parameter in `vkGetDeviceGroupSurfacePresentModesKHR`
• The surface parameter in `vkGetPhysicalDevicePresentRectanglesKHR`
• The display parameter in `vkCreateDisplayModeKHR`
• The mode parameter in `vkGetDisplayPlaneCapabilitiesKHR`
• The swapchain parameter in `vkGetSwapchainStatusKHR`
• The commandBuffer parameter in `vkCmdSetFragmentShadingRateKHR`
• The commandBuffer parameter in `vkCmdRefreshObjectsKHR`
• The commandBuffer parameter in `vkCmdSetEvent2KHR`
• The commandBuffer parameter in `vkCmdResetEvent2KHR`
• The commandBuffer parameter in `vkCmdWaitEvents2KHR`
• The commandBuffer parameter in `vkCmdPipelineBarrier2KHR`
• The commandBuffer parameter in `vkCmdWriteTimestamp2KHR`
• The queue parameter in `vkQueueSubmit2KHR`
• The fence parameter in `vkQueueSubmit2KHR`
• The commandBuffer parameter in `vkCmdWriteBufferMarker2AMD`
• The commandBuffer parameter in `vkCmdCopyBuffer2KHR`
• The commandBuffer parameter in `vkCmdCopyImage2KHR`
• The commandBuffer parameter in `vkCmdCopyBufferToImage2KHR`
• The commandBuffer parameter in `vkCmdCopyImageToBuffer2KHR`
• The commandBuffer parameter in `vkCmdBlitImage2KHR`
• The commandBuffer parameter in `vkCmdResolveImage2KHR`
• The commandBuffer parameter in `vkCmdSetDiscardRectangleEXT`
• The objectHandle member of the pNameInfo parameter in `vkSetDebugUtilsObjectNameEXT`
• The objectHandle member of the pTagInfo parameter in `vkSetDebugUtilsObjectTagEXT`
• The commandBuffer parameter in `vkCmdBeginDebugUtilsLabelEXT`
• The commandBuffer parameter in `vkCmdEndDebugUtilsLabelEXT`
• The commandBuffer parameter in `vkCmdInsertDebugUtilsLabelEXT`
• The messenger parameter in `vkDestroyDebugUtilsMessengerEXT`
• The commandBuffer parameter in `vkCmdSetSampleLocationsEXT`
• The commandBuffer parameter in `vkCmdSetLineStippleEXT`
The `commandBuffer` parameter in `vkCmdSetFrontFaceEXT`

The `commandBuffer` parameter in `vkCmdSetPrimitiveTopologyEXT`

The `commandBuffer` parameter in `vkCmdSetViewportWithCountEXT`

The `commandBuffer` parameter in `vkCmdSetScissorWithCountEXT`

The `commandBuffer` parameter in `vkCmdBindVertexBuffer2EXT`

The `commandBuffer` parameter in `vkCmdSetDepthTestEnableEXT`

The `commandBuffer` parameter in `vkCmdSetDepthWriteEnableEXT`

The `commandBuffer` parameter in `vkCmdSetDepthCompareOpEXT`

The `commandBuffer` parameter in `vkCmdSetDepthBoundsTestEnableEXT`

The `commandBuffer` parameter in `vkCmdSetStencilTestEnableEXT`

The `commandBuffer` parameter in `vkCmdSetStencilOpEXT`

The `commandBuffer` parameter in `vkCmdSetVertexInputEXT`

The `commandBuffer` parameter in `vkCmdSetPatchControlPointsEXT`

The `commandBuffer` parameter in `vkCmdSetRasterizerDiscardEnableEXT`

The `commandBuffer` parameter in `vkCmdSetDepthBiasEnableEXT`

The `commandBuffer` parameter in `vkCmdSetLogicOpEXT`

The `commandBuffer` parameter in `vkCmdSetPrimitiveRestartEnableEXT`

The `commandBuffer` parameter in `vkCmdSetColorWriteEnableEXT`

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`
- The `dstSet` member of each element of the `pDescriptorWrites` parameter in `vkUpdateDescriptorSets`
- The `dstSet` member of each element of the `pDescriptorCopies` parameter in `vkUpdateDescriptorSets`
- Each element of the `pCommandBuffers` parameter in `vkFreeCommandBuffers`
- Each element of the `pWaitSemaphores` member of the `pPresentInfo` parameter in `vkQueuePresentKHR`
- Each element of the `pSwapchains` member of the `pPresentInfo` parameter in `vkQueuePresentKHR`
- The `surface` member of each element of the `pCreateInfos` parameter in `vkCreateSharedSwapchainsKHR`
In addition, there are some implicit parameters that need to be externally synchronized. For example, all `commandBuffer` parameters that need to be externally synchronized imply that the `commandPool` that was passed in when creating that command buffer also needs to be externally synchronized. The implicit parameters and their associated object are listed below.

**Implicit Externally Synchronized Parameters**

- All `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 `vkCmdBindVertexBuffers`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDraw`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawIndexed`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawIndirect`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `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
- TheVkCommandPool that commandBuffer was allocated from, in vkCmdWaitEvents
- The VkCommandPool that commandBuffer was allocated from, in vkCmdPipelineBarrier
- The VkCommandPool that commandBuffer was allocated from, in vkCmdBeginQuery
- The VkCommandPool that commandBuffer was allocated from, in vkCmdEndQuery
- The VkCommandPool that commandBuffer was allocated from, in vkCmdResetQueryPool
- The VkCommandPool that commandBuffer was allocated from, in vkCmdWriteTimestamp
- The VkCommandPool that commandBuffer was allocated from, in vkCmdCopyQueryPoolResults
- The VkCommandPool that commandBuffer was allocated from, in vkCmdPushConstants
- The VkCommandPool that commandBuffer was allocated from, in vkCmdBeginRenderPass
- The VkCommandPool that commandBuffer was allocated from, in vkCmdNextSubpass
- The VkCommandPool that commandBuffer was allocated from, in vkCmdEndRenderPass
- The VkCommandPool that commandBuffer was allocated from, in vkCmdExecuteCommands
- The VkCommandPool that commandBuffer was allocated from, in vkCmdSetDeviceMask
- The VkCommandPool that commandBuffer was allocated from, in vkCmdDispatchBase
- The VkCommandPool that commandBuffer was allocated from, in vkCmdDrawIndirectCount
- The VkCommandPool that commandBuffer was allocated from, in vkCmdDrawIndexedIndirectCount
- The VkCommandPool that commandBuffer was allocated from, in vkCmdBeginRenderPass2
- The VkCommandPool that commandBuffer was allocated from, in vkCmdNextSubpass2
- The VkCommandPool that commandBuffer was allocated from, in vkCmdEndRenderPass2
- The VkCommandPool that commandBuffer was allocated from, in vkCmdSetFragmentShadingRateKHR
- The VkCommandPool that commandBuffer was allocated from, in vkCmdRefreshObjectsKHR
- The VkCommandPool that commandBuffer was allocated from, in vkCmdSetEvent2KHR
The VkCommandPool that commandBuffer was allocated from, in vkCmdResetEvent2KHR
The VkCommandPool that commandBuffer was allocated from, in vkCmdWaitEvents2KHR
The VkCommandPool that commandBuffer was allocated from, in vkCmdPipelineBarrier2KHR
The VkCommandPool that commandBuffer was allocated from, in vkCmdWriteTimestamp2KHR
The VkCommandPool that commandBuffer was allocated from, in vkCmdWriteBufferMarker2AMD
The VkCommandPool that commandBuffer was allocated from, in vkCmdCopyBuffer2KHR
The VkCommandPool that commandBuffer was allocated from, in vkCmdCopyImage2KHR
The VkCommandPool that commandBuffer was allocated from, in vkCmdCopyBufferToImage2KHR
The VkCommandPool that commandBuffer was allocated from, in vkCmdCopyImageToBuffer2KHR
The VkCommandPool that commandBuffer was allocated from, in vkCmdBlitImage2KHR
The VkCommandPool that commandBuffer was allocated from, in vkCmdResolveImage2KHR
The VkCommandPool that commandBuffer was allocated from, in vkCmdSetDiscardRectangleEXT
The VkCommandPool that commandBuffer was allocated from, in vkCmdBeginDebugUtilsLabelEXT
The VkCommandPool that commandBuffer was allocated from, in vkCmdEndDebugUtilsLabelEXT
The VkCommandPool that commandBuffer was allocated from, in vkCmdInsertDebugUtilsLabelEXT
The VkCommandPool that commandBuffer was allocated from, in vkCmdSetSampleLocationsEXT
The VkCommandPool that commandBuffer was allocated from, in vkCmdSetLineStippleEXT
The VkCommandPool that commandBuffer was allocated from, in vkCmdSetCullModeEXT
The VkCommandPool that commandBuffer was allocated from, in vkCmdSetFrontFaceEXT
The VkCommandPool that commandBuffer was allocated from, in vkCmdSetPrimitiveTopologyEXT
The VkCommandPool that commandBuffer was allocated from, in vkCmdSetViewportWithCountEXT
The VkCommandPool that commandBuffer was allocated from, in vkCmdSetScissorWithCountEXT
The VkCommandPool that commandBuffer was allocated from, in vkCmdBindVertexBuffers2EXT
The VkCommandPool that commandBuffer was allocated from, in vkCmdSetDepthTestEnableEXT
vkCmdSetDepthWriteEnableEXT
- The VkCommandPool that commandBuffer was allocated from, in
vkCmdSetDepthCompareOpEXT
- The VkCommandPool that commandBuffer was allocated from, in
vkCmdSetDepthBoundsTestEnableEXT
- The VkCommandPool that commandBuffer was allocated from, in
vkCmdSetStencilTestEnableEXT
- The VkCommandPool that commandBuffer was allocated from, in
vkCmdSetStencilOpEXT
- The VkCommandPool that commandBuffer was allocated from, in
vkCmdSetVertexInputEXT
- The VkCommandPool that commandBuffer was allocated from, in
vkCmdSetPatchControlPointsEXT
- The VkCommandPool that commandBuffer was allocated from, in
vkCmdSetRasterizerDiscardEnableEXT
- The VkCommandPool that commandBuffer was allocated from, in
vkCmdSetDepthBiasEnableEXT
- The VkCommandPool that commandBuffer was allocated from, in
vkCmdSetLogicOpEXT
- The VkCommandPool that commandBuffer was allocated from, in
vkCmdSetPrimitiveRestartEnableEXT
- The VkCommandPool that commandBuffer was allocated from, in
vkCmdSetColorWriteEnableEXT

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

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. A enumerant is valid if:

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

1

This special value exists only to ensure that C `enum` types are 32 bits in size. It is not part of the API, and **should** not be used by applications.
Any enumerated type returned from a query command or otherwise output from Vulkan to the application must not have a reserved value. Reserved values are values not defined by any extension for that enumerated type.

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

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

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

**Valid Usage for Flags**

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

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

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

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

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

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

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

**Note**
This restriction is due to poorly defined behavior by C compilers given a C enumerant value of `0x80000000`. In some cases adding this enumerant value may
increase the size of the underlying \texttt{VkFlagBits} type, breaking the ABI.

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

\begin{verbatim}
// Provided by VK_KHR_synchronization2
typedef uint64_t VkFlags64;
\end{verbatim}

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

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

- The bit flag is defined as part of the \texttt{Vk*FlagBits2} type, where the bits type is obtained by taking the flag type and replacing the trailing \texttt{Flags2} with \texttt{FlagBits2}. For example, a flag value of type \texttt{VkAccessFlags2KHR} must contain only bit flags defined by \texttt{VkAccessFlagBits2KHR}.

- The flag is allowed in the context in which it is being used. For example, in some cases, certain bit flags or combinations of bit flags are mutually exclusive.

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

\begin{itemize}
\item Note
\end{itemize}

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

\textbf{Valid Usage for Structure Types}

Any parameter that is a structure containing a \texttt{sType} member must have a value of \texttt{sType} which is a valid \texttt{VkStructureType} value matching the type of the structure.

\textbf{Valid Usage for Structure Pointer Chains}

Any parameter that is a structure containing a \texttt{void* pNext} member must have a value of \texttt{pNext} that is either \texttt{NULL}, or is a pointer to a valid extending structure, containing \texttt{sType} and \texttt{pNext} members as described in the Vulkan Documentation and Extensions document in the section “Extension Interactions”. The set of structures connected by \texttt{pNext} pointers is referred to as a \texttt{pNext} chain.

Each structure included in the \texttt{pNext} chain must be defined at runtime by either:

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

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

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

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

The VkBaseInStructure structure is defined as:

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

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

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

The VkBaseOutStructure structure is defined as:

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

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

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

Valid Usage for Nested Structures

The above conditions also apply recursively to members of structures provided as input to a command, either as a direct argument to the command, or themselves a member of another structure.
Specifics on valid usage of each command are covered in their individual sections.

**Valid Usage for Extensions**

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

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

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

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

**Valid Usage for Newer Core Versions**

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

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

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

### 3.8. VkResult Return Codes

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

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

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

```c
// Provided by VK_VERSION_1_0
typedef enum VkResult {
    VK_SUCCESS = 0,
    ...
};
```
VK_NOT_READY = 1,
VK_TIMEOUT = 2,
VK_EVENT_SET = 3,
VK_EVENT_RESET = 4,
VK_INCOMPLETE = 5,
VK_ERROR_OUT_OF_HOST_MEMORY = -1,
VK_ERROR_OUT_OF_DEVICE_MEMORY = -2,
VK_ERROR_INITIALIZATION_FAILED = -3,
VK_ERROR_DEVICE_LOST = -4,
VK_ERROR_MEMORY_MAP_FAILED = -5,
VK_ERROR_LAYER_NOT_PRESENT = -6,
VK_ERROR_EXTENSION_NOT_PRESENT = -7,
VK_ERROR_FEATURE_NOT_PRESENT = -8,
VK_ERROR_INCOMPATIBLE_DRIVER = -9,
VK_ERROR_TOO_MANY_OBJECTS = -10,
VK_ERROR_FORMAT_NOT_SUPPORTED = -11,
VK_ERROR_FRAGMENTED_POOL = -12,
VK_ERROR_UNKNOWN = -13,
// 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 VKKHR_surface
VK_ERROR_SURFACE_LOST_KHR = -1000000000,
// Provided by VKKHR_swapchain
VK_SUBOPTIMAL_KHR = 1000001003,
// Provided by VKKHR_swapchain
VK_ERROR_OUT_OF_DATE_KHR = -1000001004,
// Provided by VK_KHR_display_swapchain
VK_ERROR_INCOMPATIBLE_DISPLAY_KHR = -1000003001,
// Provided by VK_KHR_surface
VK_ERROR_NATIVE_WINDOW_IN_USE_KHR = -1000000001,
// Provided by VK_KHR_swapchain
VK_SUBOPTIMAL_KHR = 1000001003,
// Provided by VK_KHR_swapchain
VK_ERROR_OUT_OF_DATE_KHR = -1000001004,
// Provided by VK_KHR_display_swapchain
VK_ERROR_INCOMPATIBLE_DISPLAY_KHR = -1000003001,
// Provided by VK_KHR_surface
VK_ERROR_NATIVE_WINDOW_IN_USE_KHR = -1000000001,
// Provided by VK_KHR_swapchain
VK_SUBOPTIMAL_KHR = 1000001003,
// Provided by VK_KHR_swapchain
VK_ERROR_OUT_OF_DATE_KHR = -1000001004,
// Provided by VK_KHR_display_swapchain
VK_ERROR_INCOMPATIBLE_DISPLAY_KHR = -1000003001,
// Provided by VK_KHR_surface
VK_ERROR_NATIVE_WINDOW_IN_USE_KHR = -1000000001,
// Provided by VK_KHR_swapchain
VK_SUBOPTIMAL_KHR = 1000001003,
// Provided by VK_KHR_swapchain
VK_ERROR_OUT_OF_DATE_KHR = -1000001004,
// Provided by VK_KHR_display_swapchain
VK_ERROR_INCOMPATIBLE_DISPLAY_KHR = -1000003001,
// Provided by VK_KHR_surface
VK_ERROR_NATIVE_WINDOW_IN_USE_KHR = -1000000001,
// Provided by VK_KHR_swapchain
VK_SUBOPTIMAL_KHR = 1000001003,
// Provided by VK_KHR_swapchain
VK_ERROR_OUT_OF_DATE_KHR = -1000001004,
// Provided by VK_KHR_display_swapchain
VK_ERROR_INCOMPATIBLE_DISPLAY_KHR = -1000003001,
// Provided by VK_KHR_surface
VK_ERROR_NATIVE_WINDOW_IN_USE_KHR = -1000000001,
// Provided by VK_KHR_swapchain
VK_SUBOPTIMAL_KHR = 1000001003,
// Provided by VK_KHR_swapchain
VK_ERROR_OUT_OF_DATE_KHR = -1000001004,
// Provided by VK_KHR_display_swapchain
VK_ERROR_INCOMPATIBLE_DISPLAY_KHR = -1000003001,
// Provided by VK_KHR_surface
VK_ERROR_NATIVE_WINDOW_IN_USE_KHR = -1000000001,
// Provided by VK_KHR_swapchain
VK_SUBOPTIMAL_KHR = 1000001003,
// Provided by VK_KHR_swapchain
VK_ERROR_OUT_OF_DATE_KHR = -1000001004,
// Provided by VK_KHR_display_swapchain
VK_ERROR_INCOMPATIBLE_DISPLAY_KHR = -1000003001,
// Provided by VK_KHR_surface
VK_ERROR_NATIVE_WINDOW_IN_USE_KHR = -1000000001,
// Provided by VK_KHR_swapchain
VK_SUBOPTIMAL_KHR = 1000001003,
// Provided by VK_KHR_swapchain
VK_ERROR_OUT_OF_DATE_KHR = -1000001004,
// Provided by VK_KHR_display_swapchain
VK_ERROR_INCOMPATIBLE_DISPLAY_KHR = -1000003001,
// Provided by VK_KHR_surface
VK_ERROR_NATIVE_WINDOW_IN_USE_KHR = -1000000001,
// Provided by VK_KHR_swapchain
VK_SUBOPTIMAL_KHR = 1000001003,
// Provided by VK_KHR_swapchain
VK_ERROR_OUT_OF_DATE_KHR = -1000001004,
// Provided by VK_KHR_display_swapchain
VK_ERROR_INCOMPATIBLE_DISPLAY_KHR = -1000003001,
// Provided by VK_KHR_surface
VK_ERROR_NATIVE_WINDOW_IN_USE_KHR = -1000000001,
// Provided by VK_KHR_swapchain
VK_SUBOPTIMAL_KHR = 1000001003,
// Provided by VK_KHR_swapchain
VK_ERROR_OUT_OF_DATE_KHR = -1000001004,
// Provided by VK_KHR_display_swapchain
VK_ERROR_INCOMPATIBLE_DISPLAY_KHR = -1000003001,
// Provided by VK_KHR_surface
VK_ERROR_NATIVE_WINDOW_IN_USE_KHR = -1000000001,
// Provided by VK_KHR_swapchain
VK_SUBOPTIMAL_KHR = 1000001003,
// Provided by VK_KHR_swapchain
VK_ERROR_OUT_OF_DATE_KHR = -1000001004,
// Provided by VK_KHR_display_swapchain
VK_ERROR_INCOMPATIBLE_DISPLAY_KHR = -1000003001,
// Provided by VK_KHR_surface
VK_ERROR_NATIVE_WINDOW_IN_USE_KHR = -1000000001,
// Provided by VK_KHR_swapchain
VK_SUBOPTIMAL_KHR = 1000001003,
// Provided by VK_KHR_swapchain
VK_ERROR_OUT_OF_DATE_KHR = -1000001004,
// Provided by VK_KHR_display_swapchain
VK_ERROR_INCOMPATIBLE_DISPLAY_KHR = -1000003001,
// Provided by VK_KHR_surface
VK_ERROR_NATIVE_WINDOW_IN_USE_KHR = -1000000001,
// Provided by VK_KHR_swapchain
VK_SUBOPTIMAL_KHR = 1000001003,
// Provided by VK_KHR_swapchain
VK_ERROR_OUT_OF_DATE_KHR = -1000001004,
// Provided by VK_KHR_display_swapchain
VK_ERROR_INCOMPATIBLE_DISPLAY_KHR = -1000003001,
// Provided by VK_KHR_surface
VK_ERROR_NATIVE_WINDOW_IN_USE_KHR = -1000000001,
// Provided by VK_KHR_swapchain
VK_SUBOPTIMAL_KHR = 1000001003,
// Provided by VK_KHR_swapchain
VK_ERROR_OUT_OF_DATE_KHR = -1000001004,
// Provided by VK_KHR_display_swapchain
VK_ERROR_INCOMPATIBLE_DISPLAY_KHR = -1000003001,
// Provided by VK_KHR_surface
VK_ERROR_NATIVE_WINDOW_IN_USE_KHR = -1000000001,
// Provided by VK_KHR_swapchain
VK_SUBOPTIMAL_KHR = 1000001003,
• **VK_NOT_READY** A fence or query has not yet completed
• **VK_TIMEOUT** A wait operation has not completed in the specified time
• **VK_EVENT_SET** An event is signaled
• **VK_EVENT_RESET** An event is unsignaled
• **VK_INCOMPLETE** A return array was too small for the result
• **VK_SUBOPTIMAL_KHR** A swapchain no longer matches the surface properties exactly, but can still be used to present to the surface successfully.

**Error codes**

• **VK_ERROR_OUT_OF_HOST_MEMORY** A host memory allocation has failed.
• **VK_ERROR_OUT_OF_DEVICE_MEMORY** A device memory allocation has failed.
• **VK_ERROR_INITIALIZATION_FAILED** Initialization of an object could not be completed for implementation-specific reasons.
• **VK_ERROR_DEVICE_LOST** The logical or physical device has been lost. See Lost Device.
• **VK_ERROR_MEMORY_MAP_FAILED** Mapping of a memory object has failed.
• **VK_ERROR_LAYER_NOT_PRESENT** A requested layer is not present or could not be loaded.
• **VK_ERROR_EXTENSION_NOT_PRESENT** A requested extension is not supported.
• **VK_ERROR_FEATURE_NOT_PRESENT** A requested feature is not supported.
• **VK_ERROR_INCOMPATIBLE_DRIVER** The requested version of Vulkan is not supported by the driver or is otherwise incompatible for implementation-specific reasons.
• **VK_ERROR_TOO_MANY_OBJECTS** Too many objects of the type have already been created.
• **VK_ERROR_FORMAT_NOT_SUPPORTED** A requested format is not supported on this device.
• **VK_ERROR_FRAGMENTED_POOL** A pool allocation has failed due to fragmentation of the pool’s memory. This must only be returned if no attempt to allocate host or device memory was made to accommodate the new allocation. This should be returned in preference to **VK_ERROR_OUT_OF_POOL_MEMORY**, but only if the implementation is certain that the pool allocation failure was due to fragmentation.
• **VK_ERROR_SURFACE_LOST_KHR** A surface is no longer available.
• **VK_ERROR_NATIVE_WINDOW_IN_USE_KHR** The requested window is already in use by Vulkan or another API in a manner which prevents it from being used again.
• **VK_ERROR_OUT_OF_DATE_KHR** A surface has changed in such a way that it is no longer compatible with the swapchain, and further presentation requests using the swapchain will fail. Applications must query the new surface properties and recreate their swapchain if they wish to continue presenting to the surface.
• **VK_ERROR_INCOMPATIBLE_DISPLAY_KHR** The display used by a swapchain does not use the same presentable image layout, or is incompatible in a way that prevents sharing an image.
• **VK_ERROR_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 $2^{32}$.

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

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

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

\[ 0^0 = 1. \]
Occasionally, further requirements will be specified. Most single-precision floating-point formats meet these requirements.

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

3.9.2. Floating-Point Format Conversions

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

3.9.3. 16-Bit Floating-Point Numbers

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

3.9.4. Unsigned 11-Bit Floating-Point Numbers

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

3.9.5. Unsigned 10-Bit Floating-Point Numbers

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

3.9.6. General Requirements

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

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

3.10. Fixed-Point Data Conversions

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

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

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

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

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

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

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

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

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

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

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

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

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

where $\text{convertFloatToUint}(r,b)$ returns one of the two unsigned binary integer values with exactly $b$ bits which are closest to the floating-point value $r$. Implementations should round to nearest. If $r$ is equal to an integer, then that integer value must be returned. In particular, if $f$ is equal to 0.0 or 1.0, then $c$ must be assigned 0 or $2^b - 1$, respectively.
The conversion from a floating-point value \( f \) to the corresponding signed normalized fixed-point value \( c \) is performed by clamping \( f \) to the range \([-1, 1]\), then computing

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

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 \(-(2^b - 1)\), 0, or \( 2^b - 1 \), 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` of `VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO`, and thus its `sType` member must equal that when it is passed to the API.

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

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 = 1000094000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO = 1000157000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_INFO = 1000157001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_16BIT_STORAGE_FEATURES = 1000083000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_MEMORY_DEDICATED_REQUIREMENTS = 1000127000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_MEMORY_DEDICATE_ALLOCATE_INFO = 1000127001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_FLAGS_INFO = 1000060000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DEVICE_GROUP_RENDER_PASS_BEGIN_INFO = 1000060003,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DEVICE_GROUP_COMMAND_BUFFER_BEGIN_INFO = 1000060004,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DEVICE_GROUP_SUBMIT_INFO = 1000060005,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_DEVICE_GROUP_INFO = 1000060013,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_DEVICE_GROUP_INFO = 1000060014,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GROUP_PROPERTIES = 1000070000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DEVICE_GROUP_DEVICE_CREATE_INFO = 1000070001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_BUFFER_MEMORY_REQUIREMENTS_INFO_2 = 1000146000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_IMAGE_MEMORY_REQUIREMENTS_INFO_2 = 1000146001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_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,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_FORMAT_INFO_2 = 1000059004,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_QUEUE_FAMILY_PROPERTIES_2 = 1000059005,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PROPERTIES_2 = 1000059006,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_POINT_CLIPPING_PROPERTIES = 1000117000,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_RENDER_PASS_INPUT_ATTACHMENT_ASPECT_CREATE_INFO = 1000117001,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_IMAGE_VIEW_USAGE_CREATE_INFO = 1000117002,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_DOMAIN_ORIGIN_STATE_CREATE_INFO = 1000117003,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO = 1000053000,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_FEATURES = 1000053001,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PROPERTIES = 1000053002,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTERS_FEATURES = 1000120000,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PROTECTED_SUBMIT_INFO = 1000145000,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_FEATURES = 1000145001,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_PROPERTIES = 1000145002,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DEVICE_QUEUE_INFO_2 = 1000145003,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_CREATE_INFO = 1000156000,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_INFO = 1000156001,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_BIND_IMAGE_PLANE_MEMORY_INFO = 1000156002,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_IMAGE_PLANE_MEMORY_REQUIREMENTS_INFO = 1000156003,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_YCBCR_CONVERSION_FEATURES = 1000156004,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_IMAGE_FORMAT_PROPERTIES = 1000156005,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_IMAGE_FORMAT_INFO = 1000071000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXTERNAL_IMAGE_FORMAT_PROPERTIES = 1000071001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_BUFFER_INFO = 1000071002,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXTERNAL_BUFFER_PROPERTIES = 1000071003,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_PROPERTIES = 1000071004,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO = 1000072000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO = 1000072001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO = 1000072002,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_FENCE_INFO = 1000112000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXTERNAL_FENCE_PROPERTIES = 1000112001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXPORT_FENCE_CREATE_INFO = 1000113000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO = 1000077000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SEMAPHORE_INFO = 1000076000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES = 1000076001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_3_PROPERTIES = 1000168000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_SUPPORT = 1000168001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DRAW_PARAMETERS_FEATURES = 1000063000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_1_FEATURES = 49,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_1_PROPERTIES = 50,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_2_FEATURES = 51,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_2_PROPERTIES = 52,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_IMAGE_FORMAT_LIST_CREATE_INFO = 1000147000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_2 = 1000109000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_2 = 1000109001,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_2 = 1000109002,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_SUBPASS_DEPENDENCY_2 = 1000109003,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO_2 = 1000109004,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_SUBPASS_BEGIN_INFO = 1000109005,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_SUBPASS_END_INFO = 1000109006,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_8BIT_STORAGE_FEATURES = 1000177000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DRIVER_PROPERTIES = 1000196000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_INT64_FEATURES = 1000180000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_FLOAT16_INT8_FEATURES = 1000082000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FLOAT_CONTROLS_PROPERTIES = 1000197000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_BINDING_FLAGS_CREATE_INFO = 1000161000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_FEATURES = 1000161001,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_PROPERTIES = 1000161002,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_ALLOCATE_INFO =
1000161003,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_LAYOUT_SUPPORT =
1000161004,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEPTH_STENCIL_RESOLVE_PROPERTIES = 1000199000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_DEPTH_STENCIL_RESOLVE = 1000199001,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SCALAR_BLOCK_LAYOUT_FEATURES = 1000221000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_IMAGE_STENCIL_USAGE_CREATE_INFO = 1000246000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_FILTER_MINMAX_PROPERTIES = 1000130000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_SAMPLER_REDUCTION_MODE_CREATE_INFO = 1000130001,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_MEMORY_MODEL_FEATURES = 1000211000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGELESS_FRAMEBUFFER_FEATURES = 1000108000,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENTS_CREATE_INFO = 1000108001,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENT_IMAGE_INFO = 1000108002,
// Provided by VK_VERSION_1_2
VK_STRUCTURE_TYPE_RENDER_PASS_ATTACHMENT_BEGIN_INFO = 1000108003,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_UNIFORM_BUFFER_STANDARD_LAYOUT_FEATURES = 1000253000,
  // Provided by VK_VERSION_1_2
  VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SUBGROUP_EXTENDED_TYPES_FEATURES = 1000175000,
  // Provided by VK_VERSION_1_2
  VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SEPARATE_DEPTH_STENCIL_LAYOUTS_FEATURES = 1000241000,
  // Provided by VK_VERSION_1_2
  VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_STENCIL_LAYOUT = 1000241001,
  // Provided by VK_VERSION_1_2
  VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_STENCIL_LAYOUT = 1000241002,
  // Provided by VK_VERSION_1_2
  VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_HOST_QUERY_RESET_FEATURES = 1000261000,
  // Provided by VK_VERSION_1_2
  VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_FEATURES = 1000207000,
  // Provided by VK_VERSION_1_2
  VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_PROPERTIES = 1000207001,
  // Provided by VK_VERSION_1_2
  VK_STRUCTURE_TYPE_SEMAPHORE_TYPE_CREATE_INFO = 1000207002,
  // Provided by VK_VERSION_1_2
  VK_STRUCTURE_TYPE_TIMELINE_SEMAPHORE_SUBMIT_INFO = 1000207003,
  // Provided by VK_VERSION_1_2
  VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO = 1000207004,
  // Provided by VK_VERSION_1_2
  VK_STRUCTURE_TYPE_SEMAPHORE_SIGNAL_INFO = 1000207005,
  // Provided by VK_VERSION_1_2
  VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES = 1000257000,
  // Provided by VK_VERSION_1_2
  VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO = 1000244001,
  // Provided by VK_VERSION_1_2
  VK_STRUCTURE_TYPE_BUFFER_OPAQUE_CAPTURE_ADDRESS_CREATE_INFO = 1000257002,
  // Provided by VK_VERSION_1_2
  VK_STRUCTURE_TYPE_MEMORY_OPAQUE_CAPTURE_ADDRESS_ALLOCATE_INFO = 1000257003,
  // Provided by VK_VERSION_1_2
  VK_STRUCTURE_TYPE_DEVICE_MEMORY_OPAQUE_CAPTURE_ADDRESS_INFO = 1000257004,
  // Provided by VKSC_VERSION_1_0
  VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_SC_1_0_FEATURES = 1000298000,
  // Provided by VKSC_VERSION_1_0
  VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_SC_1_0_PROPERTIES = 1000298001,
  // Provided by VKSC_VERSION_1_0
  VK_STRUCTURE_TYPE_DEVICE_OBJECT_RESERVATION_CREATE_INFO = 1000298002,
  // Provided by VKSC_VERSION_1_0
  VK_STRUCTURE_TYPE_COMMAND_POOL_MEMORY_RESERVATION_CREATE_INFO = 1000298003,
  // Provided by VKSC_VERSION_1_0
  VK_STRUCTURE_TYPE_COMMAND_POOL_MEMORY_CONSUMPTION = 1000298004,
  // Provided by VKSC_VERSION_1_0
  VK_STRUCTURE_TYPE_PIPELINE_POOL_SIZE = 1000298005,
  // Provided by VKSC_VERSION_1_0
  VK_STRUCTURE_TYPE_FAULT_DATA = 1000298007,
  // Provided by VKSC_VERSION_1_0
VK_STRUCTURE_TYPE_FAULT_CALLBACK_INFO = 1000298008,
// Provided by VKSC_VERSION_1_0
VK_STRUCTURE_TYPE_PIPELINE_OFFLINE_CREATE_INFO = 1000298010,
// Provided by VK_KHR_swapchain
VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR = 1000001000,
// Provided by VK_KHR_swapchain
VK_STRUCTURE_TYPE_PRESENT_INFO_KHR = 1000001001,
// Provided by VK_VERSION_1_1 with VK_KHR_swapchain
VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_CAPABILITIES_KHR = 1000060007,
// Provided by VK_VERSION_1_1 with VK_KHR_swapchain
VK_STRUCTURE_TYPE_IMAGE_SWAPCHAIN_CREATE_INFO_KHR = 1000060008,
// Provided by VK_VERSION_1_1 with VK_KHR_swapchain
VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_SWAPCHAIN_INFO_KHR = 1000060009,
// Provided by VK_VERSION_1_1 with VK_KHR_swapchain
VK_STRUCTURE_TYPE_ACQUIRE_NEXT_IMAGE_INFO_KHR = 1000060010,
// Provided by VK_VERSION_1_1 with VK_KHR_swapchain
VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_INFO_KHR = 1000060011,
// Provided by VK_VERSION_1_1 with VK_KHR_swapchain
VK_STRUCTURE_TYPE_DEVICE_GROUP_SWAPCHAIN_CREATE_INFO_KHR = 1000060012,
// Provided by VK_KHR_display
VK_STRUCTURE_TYPE_DISPLAY_MODE_CREATE_INFO_KHR = 1000002000,
// Provided by VK_KHR_display
VK_STRUCTURE_TYPE_DISPLAY_SURFACE_CREATE_INFO_KHR = 1000002001,
// Provided by VK_KHR_swapchain
VK_STRUCTURE_TYPE_DISPLAY_PRESENT_INFO_KHR = 1000003000,
// Provided by VK_NV_private_vendor_info
VK_STRUCTURE_TYPE_PRIVATE_VENDOR_INFO_RESERVED_OFFSET_0_NV = 1000051000,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXTURE_COMPRESSION_ASTC_HDR_FEATURES_EXT = 1000066000,
// Provided by VK_EXT_astc_decode_mode
VK_STRUCTURE_TYPE_IMAGE_VIEW_ASTC_DECODE_MODE_EXT = 1000067000,
// Provided by VK_EXT_astc_decode_mode
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ASTC_DECODE_FEATURES_EXT = 1000067001,
// Provided by VK_KHR_external_memory_fd
VK_STRUCTURE_TYPE_IMPORT_MEMORY_FD_INFO_KHR = 1000074000,
// Provided by VK_KHR_external_memory_fd
VK_STRUCTURE_TYPE_MEMORY_FD_PROPERTIES_KHR = 1000074001,
// Provided by VK_KHR_external_memory_fd
VK_STRUCTURE_TYPE_MEMORY_GET_FD_INFO_KHR = 1000074002,
// Provided by VK_KHR_incremental_present
VK_STRUCTURE_TYPE_PRESENT_REGIONS_KHR = 1000084000,
// Provided by VK_EXT_display_surface_counter
VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_EXT = 1000090000,
// Provided by VK_KHR_display_control
VK_STRUCTURE_TYPE_DISPLAY_POWER_INFO_EXT = 1000091000,
VK_STRUCTURE_TYPE_DEVICE_EVENT_INFO_EXT = 1000091001,
// Provided by VK_EXT_display_control
VK_STRUCTURE_TYPE_DISPLAY_EVENT_INFO_EXT = 1000091002,
// Provided by VK_EXT_display_control
VK_STRUCTURE_TYPE_SWAPCHAIN_COUNTER_CREATE_INFO_EXT = 1000091003,
// Provided by VK_EXT_discard_rectangles
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DISCARD_RECTANGLE_PROPERTIES_EXT = 1000099000,
// Provided by VK_EXT_discard_rectangles
VK_STRUCTURE_TYPE_PIPELINE_DISCARD_RECTANGLE_STATE_CREATE_INFO_EXT = 1000099001,
// Provided by VK_EXT_conservative_rasterization
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CONSERVATIVE_RASTERIZATION_PROPERTIES_EXT = 1000101000,
// Provided by VK_EXT_conservative_rasterization
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_CONSERVATIVE_STATE_CREATE_INFO_EXT = 1000101001,
// Provided by VK_EXT_depth_clip_enable
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEPTH_CLIP_ENABLEFEATURES_EXT = 1000102000,
// Provided by VK_EXT_depth_clip_enable
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_DEPTH_CLIP_STATE_CREATE_INFO_EXT = 1000102001,
// Provided by VK_KHR_shared_presentable_image
VK_STRUCTURE_TYPE_SHARED_PRESENT_SURFACE_CAPABILITIES_KHR = 1000111000,
// Provided by VK_KHR_external_fence_fd
VK_STRUCTURE_TYPE_IMPORT_FENCE_FD_INFO_KHR = 1000115000,
// Provided by VK_KHR_external_fence_fd
VK_STRUCTURE_TYPE_FENCE_GET_FD_INFO_KHR = 1000115001,
// Provided by VK_KHR_performance_query
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PERFORMANCE_QUERY_FEATURES_KHR = 1000116000,
// Provided by VK_KHR_performance_query
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PERFORMANCE_QUERY_PROPERTIES_KHR = 1000116001,
// Provided by VK_KHR_performance_query
VK_STRUCTURE_TYPE_QUERY_POOL_PERFORMANCE_CREATE_INFO_KHR = 1000116002,
// Provided by VK_KHR_performance_query
VK_STRUCTURE_TYPE_PERFORMANCE_QUERY_SUBMIT_INFO_KHR = 1000116003,
// Provided by VK_KHR_performance_query
VK_STRUCTURE_TYPE_ACQUIRE_PROFILING_LOCK_INFO_KHR = 1000116004,
// Provided by VK_KHR_performance_query
VK_STRUCTURE_TYPE_PERFORMANCE_COUNTER_KHR = 1000116005,
// Provided by VK_KHR_performance_query
VK_STRUCTURE_TYPE_PERFORMANCE_COUNTER_DESCRIPTION_KHR = 1000116006,
// Provided by VK_KHR_get_surface_capabilities2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SURFACE_INFO_2_KHR = 1000119000,
// Provided by VK_KHR_get_surface_capabilities2
VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_KHR = 1000119001,
// Provided by VK_KHR_get_surface_capabilities2
VK_STRUCTURE_TYPE_SURFACE_FORMAT_2_KHR = 1000119002,
VK_STRUCTURE_TYPE_DISPLAY_PROPERTIES_2_KHR = 1000121000,
    // Provided by VK_KHR_get_display_properties2
VK_STRUCTURE_TYPE_DISPLAY_PLANE_PROPERTIES_2_KHR = 1000121001,
    // Provided by VK_KHR_get_display_properties2
VK_STRUCTURE_TYPE_DISPLAY_MODE_PROPERTIES_2_KHR = 1000121002,
    // Provided by VK_KHR_get_display_properties2
VK_STRUCTURE_TYPE_DISPLAY_PLANE_INFO_2_KHR = 1000121003,
    // Provided by VK_KHR_get_display_properties2
VK_STRUCTURE_TYPE_DISPLAY_PLANE_CAPABILITIES_2_KHR = 1000121004,
    // Provided by VK_KHR_get_display_properties2
VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_NAME_INFO_EXT = 1000128000,
    // Provided by VK_EXT_debug_utils
VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_TAG_INFO_EXT = 1000128001,
    // Provided by VK_EXT_debug_utils
VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT = 1000128002,
    // Provided by VK_EXT_debug_utils
VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CALLBACK_DATA_EXT = 1000128003,
    // Provided by VK_EXT_debug_utils
VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT = 1000128004,
    // Provided by VK_EXT_sample_locations
VK_STRUCTURE_TYPE_SAMPLE_LOCATIONS_INFO_EXT = 1000143000,
    // Provided by VK_EXT_sample_locations
VK_STRUCTURE_TYPE_RENDER_PASS_SAMPLE_LOCATIONS_BEGIN_INFO_EXT = 1000143001,
    // Provided by VK_EXT_sample_locations
VK_STRUCTURE_TYPE_PIPELINE_SAMPLE_LOCATIONS_STATE_CREATE_INFO_EXT = 1000143002,
    // Provided by VK_EXT_sample_locations
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLE_LOCATIONS_PROPERTIES_EXT = 1000143003,
    // Provided by VK_EXT_sample_locations
VK_STRUCTURE_TYPE_MULTISAMPLE_PROPERTIES_EXT = 1000143004,
    // Provided by VK_KHR_blend_operation_advanced
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BLEND_OPERATION_ADVANCED_FEATURES_EXT = 1000148000,
    // Provided by VK_KHR_blend_operation_advanced
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BLEND_OPERATION_ADVANCED_PROPERTIES_EXT = 1000148001,
    // Provided by VK_KHR_blend_operation_advanced
VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_ADVANCED_STATE_CREATE_INFO_EXT = 1000148002,
    // Provided by VK_KHR_image_drm_format_modifier
VK_STRUCTURE_TYPE_DRM_FORMAT_MODIFIER_PROPERTIES_LIST_EXT = 1000158000,
    // Provided by VK_KHR_image_drm_format_modifier
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_DRM_FORMAT_MODIFIER_INFO_EXT = 1000158002,
    // Provided by VK_KHR_image_drm_format_modifier
VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_LIST_CREATE_INFO_EXT = 1000158003,
    // Provided by VK_KHR_image_drm_format_modifier
VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_EXPLICIT_CREATE_INFO_EXT = 1000158004,
    // Provided by VK_KHR_image_drm_format_modifier
VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_PROPERTIES_EXT = 1000158005,
    // Provided by VK_KHR_format_feature_flags2 with VK_KHR_image_drm_format_modifier
VK_STRUCTURE_TYPE_DRM_FORMAT_MODIFIER_PROPERTIES_LIST_2_EXT = 1000158006,
    // Provided by VK_KHR_filter_cubic
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_VIEW_IMAGE_FORMAT_INFO_EXT = 00017000,
// Provided by VK_EXT_filter_cubic
VK_STRUCTURE_TYPE_FILTER_CUBIC_IMAGE_VIEW_IMAGE_FORMAT_PROPERTIES_EXT =
0001700001,
// Provided by VK_EXT_global_priority
VK_STRUCTURE_TYPE_DEVICE_QUEUE_GLOBAL_PRIORITY_CREATE_INFO_EXT = 000174000,
// Provided by VK_EXT_external_memory_host
VK_STRUCTURE_TYPE_IMPORT_MEMORY_HOST_POINTER_INFO_EXT = 000178000,
// Provided by VK_EXT_external_memory_host
VK_STRUCTURE_TYPE_MEMORY_HOST_POINTER_PROPERTIES_EXT = 0001780001,
// Provided by VK_EXT_external_memory_host
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_MEMORY_HOST_PROPERTIES_EXT =
0001780002,
// Provided by VK_KHR_shader_clock
VK_STRUCTURE_TYPE_PHYSICALDEVICE_SHADER_CLOCK_FEATURES_KHR = 000181000,
// Provided by VK_EXT_calibrated_timestamps
VK_STRUCTURE_TYPE_CALIBRATED_TIMESTAMP_INFO_EXT = 000184000,
// Provided by VK_EXT_vertex_attribute_divisor
VK_STRUCTURE_TYPE_PHYSICALDEVICE_VERTEX_ATTRIBUTE_DIVISOR_PROPERTIES_EXT =
000190000,
// Provided by VK_EXT_vertex_attribute_divisor
VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_DIVISOR_STATE_CREATE_INFO_EXT =
0001900001,
// Provided by VK_EXT_vertex_attribute_divisor
VK_STRUCTURE_TYPE_PHYSICALDEVICE_VERTEX_ATTRIBUTE_DIVISOR_FEATURES_EXT =
0001900002,
// Provided by VK_EXT_pci_bus_info
VK_STRUCTURE_TYPE_PHYSICALDEVICE_PCI_BUS_INFO_PROPERTIES_EXT = 000212000,
// Provided by VK_KHR_shader_terminate_invocation
VK_STRUCTURE_TYPE_PHYSICALDEVICE_SHADER_TERMINATE_INVOCATION_FEATURES_KHR =
000215000,
// Provided by VK_EXT_subgroup_size_control
VK_STRUCTURE_TYPE_PHYSICALDEVICE_SUBGROUP_SIZE_CONTROL_PROPERTIES_EXT =
000225000,
// Provided by VK_KHR_fragment_shading_rate
VK_STRUCTURE_TYPE_FRAGMENT_SHADING_RATE_ATTACHMENT_INFO_KHR = 000226000,
// Provided by VK_KHR_fragment_shading_rate
VK_STRUCTURE_TYPE_PIPELINE_FRAGMENT_SHADING_RATE_STATE_CREATE_INFO_KHR =
0002260001,
// Provided by VK_KHR_fragment_shading_rate
VK_STRUCTURE_TYPE_PHYSICALDEVICE_FRAGMENT_SHADING_RATE_PROPERTIES_KHR =
0002260002,
// Provided by VK_KHR_fragment_shading_rate
VK_STRUCTURE_TYPE_PHYSICALDEVICE_FRAGMENT_SHADING_RATE_FEATURES_KHR =
0002260003,
// Provided by VK_EXT_shader_image_atomic_int64
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_IMAGE_ATOMIC_INT64_FEATURES_EXT = 1000234000,
// Provided by VK_EXT_memory_budget
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_BUDGET_PROPERTIES_EXT = 1000237000,
// Provided by VK_EXT_validation_features
VK_STRUCTURE_TYPE_VALIDATION_FEATURES_EXT = 1000247000,
// Provided by VK_EXT_fragment_shader_interlock
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADER_INTERLOCK_FEATURES_EXT = 1000251000,
// Provided by VK_EXT_ycbcr_image_arrays
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_YCBCR_IMAGE_ARRAYS_FEATURES_EXT = 1000252000,
// Provided by VK_EXT_headless_surface
VK_STRUCTURE_TYPE_HEADLESS_SURFACE_CREATE_INFO_EXT = 1000256000,
// Provided by VK_EXT_line_rasterization
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_LINE_RASTERIZATION_FEATURES_EXT = 1000259000,
// Provided by VK_EXT_line_rasterization
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_LINE_STATE_CREATE_INFO_EXT = 1000259001,
// Provided by VK_EXT_line_rasterization
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_LINE_RASTERIZATION_PROPERTIES_EXT = 1000259002,
// Provided by VK_EXT_shader_atomic_float
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_FLOAT_FEATURES_EXT = 1000260000,
// Provided by VK_EXT_index_type_uint8
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INDEX_TYPE_UINT8_FEATURES_EXT = 1000265000,
// Provided by VK_EXT_extended_dynamic_state
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTENDED_DYNAMIC_STATE_FEATURES_EXT = 1000267000,
// Provided by VK_EXT_shader_demote_to_helper_invocation
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DEMOTE_TO_HELPER_INVOCATION_FEATURES_EXT = 1000276000,
// Provided by VK_EXT_texel_buffer_alignment
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_FEATURES_EXT = 1000281000,
// Provided by VK_KHR_object_refresh
VK_STRUCTURE_TYPE_REFRESH_OBJECT_LIST_KHR = 1000308000,
// Provided by VK_KHR_synchronization2
VK_STRUCTURE_TYPE_MEMORY_BARRIER_2_KHR = 1000314000,
VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER_2_KHR = 1000314001,
    // Provided by VK_KHR_synchronization2
VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER_2_KHR = 1000314002,
    // Provided by VK_KHR_synchronization2
VK_STRUCTURE_TYPE_DEPENDENCY_INFO_KHR = 1000314003,
    // Provided by VK_KHR_synchronization2
VK_STRUCTURE_TYPE_SUBMIT_INFO_2_KHR = 1000314004,
    // Provided by VK_KHR_synchronization2
VK_STRUCTURE_TYPE_SEMAPHORE_SUBMIT_INFO_KHR = 1000314005,
    // Provided by VK_KHR_synchronization2
VK_STRUCTURE_TYPE_COMMAND_BUFFER_SUBMIT_INFO_KHR = 1000314006,
    // Provided by VK_KHR_synchronization2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SYNCHRONIZATION_2_FEATURES_KHR = 1000314007,
    // Provided by VK_KHR_synchronization2 with VK_NV_device_diagnostic_checkpoints
VK_STRUCTURE_TYPE_QUEUE_FAMILY_CHECKPOINT_PROPERTIES_2_NV = 1000314008,
    // Provided by VK_KHR_synchronization2 with VK_NV_device_diagnostic_checkpoints
VK_STRUCTURE_TYPE_CHECKPOINT_DATA_2_NV = 1000314009,
    // Provided by VK_EXT_ycbcr_2plane_444_formats
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_YCBCR_2_PLANE_444_FORMATS_FEATURES_EXT = 1000330000,
    // Provided by VK_KHR_copy_commands2
VK_STRUCTURE_TYPE_COPY_BUFFER_INFO_2_KHR = 1000337000,
    // Provided by VK_KHR_copy_commands2
VK_STRUCTURE_TYPE_COPY_IMAGE_INFO_2_KHR = 1000337001,
    // Provided by VK_KHR_copy_commands2
VK_STRUCTURE_TYPE_COPY_BUFFER_TO_IMAGE_INFO_2_KHR = 1000337002,
    // Provided by VK_KHR_copy_commands2
VK_STRUCTURE_TYPE_COPY_IMAGE_TO_BUFFER_INFO_2_KHR = 1000337003,
    // Provided by VK_KHR_copy_commands2
VK_STRUCTURE_TYPE_BLIT_IMAGE_INFO_2_KHR = 1000337004,
    // Provided by VK_KHR_copy_commands2
VK_STRUCTURE_TYPE_RESOLVE_IMAGE_INFO_2_KHR = 1000337005,
    // Provided by VK_KHR_copy_commands2
VK_STRUCTURE_TYPE_BUFFER_COPY_2_KHR = 1000337006,
    // Provided by VK_KHR_copy_commands2
VK_STRUCTURE_TYPE_IMAGE_COPY_2_KHR = 1000337007,
    // Provided by VK_KHR_copy_commands2
VK_STRUCTURE_TYPE_IMAGE_BLIT_2_KHR = 1000337008,
    // Provided by VK_KHR_copy_commands2
VK_STRUCTURE_TYPE_BUFFER_IMAGE_COPY_2_KHR = 1000337009,
    // Provided by VK_KHR_copy_commands2
VK_STRUCTURE_TYPE_IMAGE_RESOLVE_2_KHR = 1000337010,
    // Provided by VK_KHR_copy_commands2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_4444_FORMATS_FEATURES_EXT = 1000340000,
    // Provided by VK_KHR_copy_commands2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_ROBUSTNESS_FEATURES_EXT = 1000335000,
    // Provided by VK_KHR_copy_commands2
VK_STRUCTURE_TYPE_VERTEX_INPUT_BINDING_DESCRIPTION_2_EXT = 1000352001,
/** Provided by VK_EXT_vertex_input_dynamic_state **
VK_STRUCTURE_TYPE_VERTEX_INPUT_ATTRIBUTE_DESCRIPTION_2_EXT = 1000352002,

// Provided by VK_NV_external_sci_sync, VK_NV_external_sci_sync2
VK_STRUCTURE_TYPE_IMPORT_FENCE_SCI_SYNC_INFO_NV = 1000373000,

// Provided by VK_NV_external_sci_sync, VK_NV_external_sci_sync2
VK_STRUCTURE_TYPE_EXPORT_FENCE_SCI_SYNC_INFO_NV = 1000373001,

// Provided by VK_NV_external_sci_sync, VK_NV_external_sci_sync2
VK_STRUCTURE_TYPE_FENCE_GET_SCI_SYNC_INFO_NV = 1000373002,

// Provided by VK_NV_external_sci_sync, VK_NV_external_sci_sync2
VK_STRUCTURE_TYPE_SCI_SYNC_ATTRIBUTES_INFO_NV = 1000373003,

// Provided by VK_NV_external_sci_sync
VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_SCI_SYNC_INFO_NV = 1000373004,

// Provided by VK_NV_external_sci_sync
VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_SCI_SYNC_INFO_NV = 1000373005,

// Provided by VK_NV_external_sci_sync
VK_STRUCTURE_TYPE_SEMAPHORE_GET_SCI_SYNC_INFO_NV = 1000373006,

// Provided by VK_NV_external_sci_sync
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SCI_SYNC_FEATURES_NV = 1000373007,

// Provided by VK_NV_external_memory_sci_buf
VK_STRUCTURE_TYPE_IMPORT_MEMORY_SCI_BUF_INFO_NV = 1000374000,

// Provided by VK_NV_external_memory_sci_buf
VK_STRUCTURE_TYPE_EXPORT_MEMORY_SCI_BUF_INFO_NV = 1000374001,

// Provided by VK_NV_external_memory_sci_buf
VK_STRUCTURE_TYPE_MEMORY_GET_SCI_BUF_INFO_NV = 1000374002,

// Provided by VK_NV_external_memory_sci_buf
VK_STRUCTURE_TYPE_MEMORY_SCI_BUF_PROPERTIES_NV = 1000374003,

// Provided by VK_NV_external_memory_sci_buf
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_MEMORY_SCI_BUF_FEATURES_NV = 1000374004,

// Provided by VK_EXT_extended_dynamic_state2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTENDED_DYNAMIC_STATE_2_FEATURES_EXT = 1000377000,

// Provided by VK_EXT_color_write_enable
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COLOR_WRITE_ENABLE_FEATURES_EXT = 1000381000,

// Provided by VK_EXT_color_write_enable
VK_STRUCTURE_TYPE_PIPELINE_COLOR_WRITE_CREATE_INFO_EXT = 1000381001,

// Provided by VK_NV_external_sci_sync2
VK_STRUCTURE_TYPE_SEMAPHORE_SCI_SYNC_POOL_CREATE_INFO_NV = 1000489000,

// Provided by VK_NV_external_sci_sync2
VK_STRUCTURE_TYPE_SEMAPHORE_SCI_SYNC_CREATE_INFO_NV = 1000489001,

// Provided by VK_NV_external_sci_sync2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SCI_SYNC_2_FEATURES_NV = 1000489002,

// Provided by VKSC_VERSION_1_0 with VK_NV_external_sci_sync2
VK_STRUCTURE_TYPE_DEVICE_SEMAPHORE_SCI_SYNC_POOL_RESERVATION_CREATE_INFO_NV = 1000489003,

// Provided by VK_NV_external_memory_sci_buf
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_MEMORY_SCI_BUF_FEATURES_NV = 1000374004,
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
An example of a typo alias is from the type `VkSurfaceCounterFlagBitsEXT`, introduced by the `VK_EXT_display_surface_counter` extension. The enumerant `VK_SURFACE_COUNTER_VBLANK_EXT` was initially defined as part of this type. Once the naming inconsistency was noticed, it was renamed to `VK_SURFACE_COUNTER_VBLANK_BIT_EXT`, and the old name aliased to the correct name.
Chapter 4. Initialization

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

### 4.1. Command Function Pointers

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

**Note**

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

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

```c
// Provided by VK_VERSION_1_0
PFN_vkVoidFunction vkGetInstanceProcAddr(
    VkInstance instance,  // instance is the instance that the function pointer will be compatible with, or NULL for commands not dependent on any instance.
    const char* pName,   // pName is the name of the command to obtain.
);
```

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

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

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

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

<table>
<thead>
<tr>
<th>instance</th>
<th>pName</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>*1 invalid non-NULL instance</td>
<td>NULL</td>
<td>undefined</td>
</tr>
<tr>
<td>*1 NULL</td>
<td>vkGetInstanceProcAddr</td>
<td>fp$^5$</td>
</tr>
<tr>
<td>*1 vkGetInstanceProcAddr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$^5$
<table>
<thead>
<tr>
<th>instance</th>
<th>pName</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td><em>global command</em></td>
<td><em>fp</em></td>
</tr>
<tr>
<td>instance</td>
<td><em>core dispatchable command</em></td>
<td><em>fp</em>³</td>
</tr>
<tr>
<td>instance</td>
<td>enabled instance extension dispatchable command for <em>instance</em></td>
<td><em>fp</em>³</td>
</tr>
<tr>
<td>instance</td>
<td>available device extension² dispatchable command for <em>instance</em></td>
<td><em>fp</em>³</td>
</tr>
<tr>
<td>any other case, not covered above</td>
<td></td>
<td>NULL</td>
</tr>
</tbody>
</table>

1

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

2

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 with the command:

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

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

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

<table>
<thead>
<tr>
<th>device</th>
<th>pName</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>*1</td>
<td>undefined</td>
</tr>
<tr>
<td>invalid device</td>
<td>*1</td>
<td>undefined</td>
</tr>
<tr>
<td>device</td>
<td>NULL</td>
<td>undefined</td>
</tr>
<tr>
<td>device</td>
<td>core device-level dispatchable command²</td>
<td>fp³</td>
</tr>
<tr>
<td>device</td>
<td>enabled extension device-level dispatchable command²</td>
<td>fp³</td>
</tr>
<tr>
<td>any other case, not covered above</td>
<td></td>
<td>NULL</td>
</tr>
</tbody>
</table>

1

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

2

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

3

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);
```

### 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
VK_DEFINE_HANDLE(VkInstance)
```

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;
    const VkApplicationInfo* pApplicationInfo;
    uint32_t enabledLayerCount;
    const char* const* ppEnabledLayerNames;
    uint32_t enabledExtensionCount;
    const char* const* ppEnabledExtensionNames;
} VkInstanceCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `pApplicationInfo` is `NULL` or a pointer to a `VkApplicationInfo` structure. If not `NULL`, this information helps implementations recognize behavior inherent to classes of applications. `VkApplicationInfo` is defined in detail below.
• **enabledLayerCount** is the number of global layers to enable.

• **ppEnabledLayerNames** is a pointer to an array of enabledLayerCount null-terminated UTF-8 strings containing the names of layers to enable for the created instance. 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.

To capture events that occur while creating or destroying an instance, an application can link a VkDebugUtilsMessengerCreateInfoEXT structure to the pNext element of the VkInstanceCreateInfo structure given to vkCreateInstance. This callback is only valid for the duration of the vkCreateInstance and the vkDestroyInstance call. Use vkCreateDebugUtilsMessengerEXT to create persistent callback objects.

### Valid Usage

- VUID-VkInstanceCreateInfo-pNext-04926
  If the pNext chain of VkInstanceCreateInfo includes a VkDebugUtilsMessengerCreateInfoEXT structure, the list of enabled extensions in ppEnabledExtensionNames must contain VK_EXT_debug_utils

### Valid Usage (Implicit)

- VUID-VkInstanceCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO

- VUID-VkInstanceCreateInfo-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 VkDebugUtilsMessengerCreateInfoEXT or VkValidationFeaturesEXT

- VUID-VkInstanceCreateInfo-sType-unique
  The sType value of each struct in the pNext chain must be unique, with the exception of structures of type VkDebugUtilsMessengerCreateInfoEXT

- 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.

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

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

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

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

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

### Valid Usage

- **VUID-VkValidationFeaturesEXT-pEnabledValidationFeatures-02967**
  If the `pEnabledValidationFeatures` array contains `VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_RESERVE_BINDING_SLOT_EXT`, then it must also contain `VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_EXT`.

- **VUID-VkValidationFeaturesEXT-pEnabledValidationFeatures-02968**
  If the `pEnabledValidationFeatures` array contains `VK_VALIDATION_FEATURE_ENABLE_DEBUG_PRINTF_EXT`, then it must not contain `VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_EXT`. 

---

60
Valid Usage (Implicit)

- VUID-VkValidationFeaturesEXT-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_VALIDATION_FEATURES_EXT

- VUID-VkValidationFeaturesEXT-pEnabledValidationFeatures-parameter
  
  If enabledValidationFeatureCount is not 0, pEnabledValidationFeatures must be a valid pointer to an array of enabledValidationFeatureCount valid VkValidationFeatureEnableEXT values

- VUID-VkValidationFeaturesEXT-pDisabledValidationFeatures-parameter
  
  If disabledValidationFeatureCount is not 0, pDisabledValidationFeatures must be a valid pointer to an array of disabledValidationFeatureCount valid VkValidationFeatureDisableEXT values

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

```c
// Provided by VK_EXT_validation_features
typedef enum VkValidationFeatureEnableEXT {
    VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_EXT = 0,
    VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_reserve_BINDING_SLOT_EXT = 1,
    VK_VALIDATION_FEATURE_ENABLE_BEST_PRACTICES_EXT = 2,
    VK_VALIDATION_FEATURE_ENABLE_DEBUG_PRINTF_EXT = 3,
    VK_VALIDATION_FEATURE_ENABLE_SYNCHRONIZATION_VALIDATION_EXT = 4,
} VkValidationFeatureEnableEXT;
```

- **VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_EXT** specifies that GPU-assisted validation is enabled. Activating this feature instruments shader programs to generate additional diagnostic data. This feature is disabled by default.

- **VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_reserve_BINDING_SLOT_EXT** specifies that the validation layers reserve a descriptor set binding slot for their own use. The layer reports a value for VkPhysicalDeviceLimits::maxBoundDescriptorSets that is one less than the value reported by the device. If the device supports the binding of only one descriptor set, the validation layer does not perform GPU-assisted validation. This feature is disabled by default.

- **VK_VALIDATION_FEATURE_Enable_BEST_PRACTICES_EXT** specifies that Vulkan best-practices validation is enabled. Activating this feature enables the output of warnings related to common misuse of the API, but which are not explicitly prohibited by the specification. This feature is disabled by default.

- **VK_VALIDATION_FEATURE_ENABLE_DEBUG_PRINTF_EXT** specifies that the layers will process debugPrintfEXT operations in shaders and send the resulting output to the debug callback. This feature is disabled by default.

- **VK_VALIDATION_FEATURE_ENABLE_SYNCHRONIZATION_VALIDATION_EXT** specifies that Vulkan synchronization validation is enabled. This feature reports resource access conflicts due to missing or incorrect synchronization operations between actions (Draw, Copy, Dispatch, Blit) reading or writing the same regions of memory. This feature is disabled by default.
Possible values of elements of the `VkValidationFeaturesEXT::pDisabledValidationFeatures` array, specifying validation features to be disabled, are:

```c
// Provided by VK_EXT_validation_features
typedef enum VkValidationFeatureDisableEXT {
    VK_VALIDATION_FEATURE_DISABLE_ALL_EXT = 0,
    VK_VALIDATION_FEATURE_DISABLE_SHADERS_EXT = 1,
    VK_VALIDATION_FEATURE_DISABLE_THREAD_SAFETY_EXT = 2,
    VK_VALIDATION_FEATURE_DISABLE_API_PARAMETERS_EXT = 3,
    VK_VALIDATION_FEATURE_DISABLE_OBJECT_LIFETIMES_EXT = 4,
    VK_VALIDATION_FEATURE_DISABLE_CORE_CHECKS_EXT = 5,
    VK_VALIDATION_FEATURE_DISABLE_UNIQUE_HANDLES_EXT = 6,
    VK_VALIDATION_FEATURE_DISABLE_SHADER_VALIDATION_CACHE_EXT = 7,
} VkValidationFeatureDisableEXT;
```

- **VK_VALIDATION_FEATURE_DISABLE_ALL_EXT** specifies that all validation checks are disabled.
- **VK_VALIDATION_FEATURE_DISABLE_SHADERS_EXT** specifies that shader validation is disabled. This feature is enabled by default.
- **VK_VALIDATION_FEATURE_DISABLE_THREAD_SAFETY_EXT** specifies that thread safety validation is disabled. This feature is enabled by default.
- **VK_VALIDATION_FEATURE_DISABLE_API_PARAMETERS_EXT** specifies that stateless parameter validation is disabled. This feature is enabled by default.
- **VK_VALIDATION_FEATURE_DISABLE_OBJECT_LIFETIMES_EXT** specifies that object lifetime validation is disabled. This feature is enabled by default.
- **VK_VALIDATION_FEATURE_DISABLE_CORE_CHECKS_EXT** specifies that core validation checks are disabled. This feature is enabled by default. If this feature is disabled, the shader validation and GPU-assisted validation features are also disabled.
- **VK_VALIDATION_FEATURE_DISABLE_UNIQUE_HANDLES_EXT** specifies that protection against duplicate non-dispatchable object handles is disabled. This feature is enabled by default.
- **VK_VALIDATION_FEATURE_DISABLE_SHADER_VALIDATION_CACHE_EXT** specifies that there will be no caching of shader validation results and every shader will be validated on every application execution. Shader validation caching is enabled by default.

**Note**

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

The `VkApplicationInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkApplicationInfo {
    VkStructureType sType;
```
const void* pNext;
const char* pApplicationName;
uint32_t applicationVersion;
const char* pEngineName;
uint32_t engineVersion;
uint32_t apiVersion;
}
VkApplicationInfo;

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

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.

Implicit layers must be disabled if they do not support a version at least as high as **apiVersion**. See the “Architecture of the Vulkan Loader Interfaces” document for additional information.

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).

To provide application parameters at instance creation time, an application can link one or more VkApplicationParametersEXT structures to the **pNext** chain of the VkApplicationInfo structure.
If `VkApplicationParametersEXT::vendorID` does not correspond to an ICD that is currently available, or if `VkApplicationParametersEXT::deviceID` is not 0 and does not correspond to a physical device that is available on the system, `vkCreateInstance` will fail and return `VK_ERROR_INCOMPATIBLE_DRIVER`. If `VkApplicationParametersEXT::deviceID` is 0, the application parameter applies to all physical devices supported by the ICD identified by `VkApplicationParametersEXT::vendorID`.

If `VkApplicationParametersEXT::key` is not a valid implementation-defined application parameter key for the instance being created with `vendorID`, or if `value` is not a valid value for the specified `key`, `vkCreateInstance` will fail and return `VK_ERROR_INITIALIZATION_FAILED`.

For any implementation-defined application parameter `key` that exists but is not set by the application, the implementation-specific default value is used.

### 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`

- **VUID-VkApplicationInfo-key-05093**
  
  The `key` value of each `VkApplicationParametersEXT` structure in the `VkApplicationInfo::pNext` chain **must** be unique for each `vendorID` and `deviceID` pairing

### Valid Usage (Implicit)

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

- **VUID-VkApplicationInfo-pNext-pNext**
  
  `pNext` **must** be `NULL` or a pointer to a valid instance of `VkApplicationParametersEXT`

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

- **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

The `VkApplicationParametersEXT` structure is defined as:

```c
// Provided by VK_EXT_application_parameters
typedef struct VkApplicationParametersEXT {
    VkStructureType     sType;
    const void*         pNext;
    uint32_t            vendorID;
    uint32_t            deviceID;
} VkApplicationParametersEXT;
```
uint32_t key;
uint64_t value;
} VkApplicationParametersEXT;

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **vendorID** is the `VkPhysicalDeviceProperties::vendorID` of the ICD that the application parameter is applied to.
- **deviceID** is 0 or the `VkPhysicalDeviceProperties::deviceID` of the physical device that the application parameter is applied to.
- **key** is a 32-bit vendor-specific enumerant identifying the application parameter that is being set.
- **value** is the 64-bit value that is being set for the application parameter specified by **key**.

### Valid Usage (Implicit)

- VUID-VkApplicationParametersEXT-sType-sType
  
  **sType** must be `VK_STRUCTURE_TYPE_APPLICATION_PARAMETERS_EXT`

To destroy an instance, call:

```c
// Provided by VK_VERSION_1_0
void vk_DestroyInstance(
    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 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 vendor identified by vendorID is the entity responsible for the most salient characteristics of the underlying implementation of the VkPhysicalDevice being queried.

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

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

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

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

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

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

Note
Khronos vendor IDs may be allocated by vendors at any time. Only the latest canonical versions of this Specification, of the corresponding vk.xml API Registry, and of the corresponding vulkan_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
// Provided by VK_VERSION_1_1
void vkGetPhysicalDeviceProperties2(
    VkPhysicalDevice physicalDevice,
    VkPhysicalDeviceProperties2* pProperties);
```

- **physicalDevice** is the handle to the physical device whose properties will be queried.
- **pProperties** is a pointer to a `VkPhysicalDeviceProperties2` structure in which properties are returned.
Each structure in `pProperties` and its `pNext` chain contains members corresponding to implementation-dependent properties, behaviors, or limits. `vkGetPhysicalDeviceProperties2` fills in each member to specify the corresponding value for the implementation.

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceProperties2-physicalDevice-parameter
  - `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceProperties2-pProperties-parameter
  - `pProperties` must be a valid pointer to a `VkPhysicalDeviceProperties2` structure

The `VkPhysicalDeviceProperties2` structure is defined as:

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

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

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

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceProperties2-sType-sType
  - `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2`
- VUID-VkPhysicalDeviceProperties2-pNext-pNext
  - Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of
    - `VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT`,
    - `VkPhysicalDeviceConservativeRasterizationPropertiesEXT`,
    - `VkPhysicalDeviceCustomBorderColorPropertiesEXT`,
    - `VkPhysicalDeviceDepthStencilResolveProperties`,
    - `VkPhysicalDeviceDescriptorIndexingProperties`,
    - `VkPhysicalDeviceDiscardRectanglePropertiesEXT`,
    - `VkPhysicalDeviceDriverProperties`,
    - `VkPhysicalDeviceExternalMemoryHostPropertiesEXT`,
    - `VkPhysicalDeviceFloatControlsProperties`,
    - `VkPhysicalDeviceFragmentShadingRatePropertiesKHR`,
    - `VkPhysicalDeviceIDProperties`,
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 the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **deviceUUID** is an array of `VK_UUID_SIZE uint8_t` values representing a universally unique identifier for the device.
- **driverUUID** is an array of `VK_UUID_SIZE uint8_t` values representing a universally unique identifier for the driver build in use by the device.
• **deviceLUID** is an array of `VK_LUID_SIZE uint8_t` values representing a locally unique identifier for the device.

• **deviceNodeMask** is a `uint32_t` bitfield identifying the node within a linked device adapter corresponding to the device.

• **deviceLUIDValid** is a boolean value that will be `VK_TRUE` if `deviceLUID` contains a valid LUID and `deviceNodeMask` contains a valid node mask, and `VK_FALSE` if they do not.

• **subgroupSize** is the default number of invocations in each subgroup. `subgroupSize` is at least 1 if any of the physical device's queues support `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT`. `subgroupSize` is a power-of-two.

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

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

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

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

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

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

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

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

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

If the `VkPhysicalDeviceVulkan11Properties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.
These properties correspond to Vulkan 1.1 functionality.

The members of `VkPhysicalDeviceVulkan11Properties` have the same values as the corresponding members of `VkPhysicalDeviceIDProperties`, `VkPhysicalDeviceSubgroupProperties`, `VkPhysicalDevicePointClippingProperties`, `VkPhysicalDeviceMultiviewProperties`, `VkPhysicalDeviceProtectedMemoryProperties`, and `VkPhysicalDeviceMaintenance3Properties`.

**Valid Usage (Implicit)**

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

The `VkPhysicalDeviceVulkan12Properties` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceVulkan12Properties {
    VkStructureType sType;
    void* pNext;
    VkDriverId driverID;
    char driverName[VK_MAX_DRIVER_NAME_SIZE];
    char driverInfo[VK_MAX_DRIVER_INFO_SIZE];
    VkConformanceVersion conformanceVersion;
    VkShaderFloatControlsIndependence denormBehaviorIndependence;
    VkShaderFloatControlsIndependence roundingModeIndependence;
    VkBool32 shaderSignedZeroInfNanPreserveFloat16;
    VkBool32 shaderSignedZeroInfNanPreserveFloat32;
    VkBool32 shaderSignedZeroInfNanPreserveFloat64;
    VkBool32 shaderDenormPreserveFloat16;
    VkBool32 shaderDenormPreserveFloat32;
    VkBool32 shaderDenormPreserveFloat64;
    VkBool32 shaderDenormFlushToZeroFloat16;
    VkBool32 shaderDenormFlushToZeroFloat32;
    VkBool32 shaderDenormFlushToZeroFloat64;
    VkBool32 shaderRoundingModeRTEFloat16;
    VkBool32 shaderRoundingModeRTEFloat32;
    VkBool32 shaderRoundingModeRTEFloat64;
    VkBool32 shaderRoundingModeRTZFloat16;
    VkBool32 shaderRoundingModeRTZFloat32;
    VkBool32 shaderRoundingModeRTZFloat64;
    uint32_t maxUpdateAfterBindDescriptorsInAllPools;
    VkBool32 shaderUniformBufferArrayNonUniformIndexingNative;
    VkBool32 shaderSampledImageArrayNonUniformIndexingNative;
    VkBool32 shaderStorageBufferArrayNonUniformIndexingNative;
    VkBool32 shaderStorageImageArrayNonUniformIndexingNative;
} VkPhysicalDeviceVulkan12Properties;
```
• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• driverID is a unique identifier for the driver of the physical device.
• driverName is an array of VK_MAX_DRIVER_NAME_SIZE char containing a null-terminated UTF-8 string which is the name of the driver.
• driverInfo is an array of VK_MAX_DRIVER_INFO_SIZE char containing a null-terminated UTF-8 string with additional information about the driver.
• conformanceVersion is the version of the Vulkan conformance test this driver is conformant against (see VkConformanceVersion).
• `denormBehaviorIndependence` is a `VkShaderFloatControlsIndependence` value indicating whether, and how, denorm behavior can be set independently for different bit widths.

• `roundingModeIndependence` is a `VkShaderFloatControlsIndependence` value indicating whether, and how, rounding modes can be set independently for different bit widths.

• `shaderSignedZeroInfNanPreserveFloat16` is a boolean value indicating whether sign of a zero, Nans and ±∞ can be preserved in 16-bit floating-point computations. It also indicates whether the `SignedZeroInfNanPreserve` execution mode can be used for 16-bit floating-point types.

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

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

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

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

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

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

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

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

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

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

• `shaderRoundingModeRTEFloat64` is a boolean value indicating whether an implementation supports the round-to-nearest-even rounding mode for 64-bit floating-point arithmetic and conversion instructions. It also indicates whether the `RoundingModeRTE` execution mode can be used for 64-bit floating-point types.
• **shaderRoundingModeRTZFloat16** is a boolean value indicating whether an implementation supports the round-towards-zero rounding mode for 16-bit floating-point arithmetic and conversion instructions. It also indicates whether the RoundingModeRTZ execution mode can be used for 16-bit floating-point types.

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

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

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

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

- **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 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`.

- **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.

- **maxDescriptorSetUpdateAfterBindResources** is similar to **maxDescriptorSetResources** but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set.
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, and 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 maxDescriptorSetLayoutAttachments;
    uint32_t maxQueryFaultCount;
    uint32_t maxCallbackFaultCount;
    uint32_t maxCommandPoolCommandBuffers;
    VkDeviceSize maxCommandBufferSize;
} VkPhysicalDeviceVulkanSC10Properties;
```

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

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

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

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

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

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

- External memory handle types compatibility
- External semaphore handle types compatibility
- External fence handle types compatibility

If deviceLUIDValid is VK_FALSE, the values of deviceUUID and deviceNodeMask are undefined. If deviceLUIDValid is VK_TRUE and Vulkan is running on the Windows operating system, the contents of deviceUUID can be cast to an LUID object and must be equal to the locally unique identifier of a DXGIAdapter1 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 can not 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.

```
#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.

```
#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 the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **driverID** is a unique identifier for the driver of the physical device.
- **driverName** is an array of VK_MAX_DRIVER_NAME_SIZE char containing a null-terminated UTF-8 string
which is the name of the driver.

- **driverInfo** is an array of `VK_MAX_DRIVER_INFO_SIZE` char containing a null-terminated UTF-8 string with additional information about the driver.

- **conformanceVersion** is the version of the Vulkan conformance test this driver is conformant against (see `VkConformanceVersion`).

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

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

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

---

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceDriverProperties-sType-sType

  * **sType** **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DRIVER_PROPERTIES`

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

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

Khronos driver IDs may be allocated by vendors at any time. There may be multiple driver IDs for the same vendor, representing different drivers (for e.g. different platforms, proprietary or open source, etc.). Only the latest canonical versions of this Specification, of the corresponding `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`.

```c
#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`.

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

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

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

- `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.

The `VkPhysicalDevicePCIBusInfoPropertiesEXT` structure is defined as:

```c
// Provided by VK_EXT_pci_bus_info
typedef struct VkPhysicalDevicePCIBusInfoPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    uint32_t pciDomain;
} VkPhysicalDevicePCIBusInfoPropertiesEXT;
```
VkPhysicalDevicePCIBusInfoPropertiesEXT

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

If the VkPhysicalDevicePCIBusInfoPropertiesEXT 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 PCI bus information of a physical device.

Valid Usage (Implicit)

- VUID-VkPhysicalDevicePCIBusInfoPropertiesEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PCI_BUS_INFO_PROPERTIES_EXT

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

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

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

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

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

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

- VUID-vkGetPhysicalDeviceQueueFamilyProperties-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 `vkCmdWriteTimestamp2KHR` or `vkCmdWriteTimestamp`. The valid range for the count is 36..64 bits, or a value of 0, indicating no support for timestamps. Bits outside the valid range are guaranteed to be zeros.

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

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

Possible values of `minImageTransferGranularity` are:

- `(0,0,0)` 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 \(\text{VkOffset3D}\) parameter must be integer multiples of \(A_x, A_y,\) and \(A_z\), respectively.
  - width of a \(\text{VkExtent3D}\) parameter must be an integer multiple of \(A_x\), or else \(x + \text{width}\) must equal the width of the image subresource corresponding to the parameter.
  - height of a \(\text{VkExtent3D}\) parameter must be an integer multiple of \(A_y\), or else \(y + \text{height}\) must equal the height of the image subresource corresponding to the parameter.
  - depth of a \(\text{VkExtent3D}\) parameter must be an integer multiple of \(A_z\), or else \(z + \text{depth}\) must equal the depth of the image subresource corresponding to the parameter.
  - If the format of the image corresponding to the parameters is one of the block-compressed formats then for the purposes of the above calculations the granularity must be scaled up by the compressed texel block dimensions.

Queues supporting graphics and/or compute operations must report \((1,1,1)\) in \(\text{minImageTransferGranularity}\), 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 \(\text{minImageTransferGranularity}\) 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 \(\text{VkQueueFamilyProperties::queueFlags}\) indicating capabilities of queues in a queue family are:

```c
#include <limits.h>

typedef enum VkQueueFlagBits {
    VK_QUEUE_GRAPHICS_BIT = 0x00000001,
    VK_QUEUE_COMPUTE_BIT = 0x00000002,
    VK_QUEUE_TRANSFER_BIT = 0x00000004,
    VK_QUEUE_PROTECTED_BIT = 0x00000010,
    VK_QUEUE_SPARSE_BINDING_BIT = 0x00000020
} VkQueueFlagBits;
```

- \(\text{VK_QUEUE_GRAPHICS_BIT}\) specifies that queues in this queue family support graphics operations.
- \(\text{VK_QUEUE_COMPUTE_BIT}\) specifies that queues in this queue family support compute operations.
- \(\text{VK_QUEUE_TRANSFER_BIT}\) specifies that queues in this queue family support transfer operations.
- \(\text{VK_QUEUE_SPARSE_BINDING_BIT}\) 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 protected memory physical device feature is supported, then at least one queue family of at least one physical device exposed by the implementation **must** support graphics operations, compute operations, and protected memory operations.

**Note**

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

For further details see Queues.

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

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

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

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

• **physicalDevice** is the handle to the physical device whose properties will be queried.

• **pQueueFamilyPropertyCount** is a pointer to an integer related to the number of queue families available or queried, as described in **vkGetPhysicalDeviceQueueFamilyProperties**.

• **pQueueFamilyProperties** is either **NULL** or a pointer to an array of **VkQueueFamilyProperties2** structures.

**vkGetPhysicalDeviceQueueFamilyProperties2** behaves similarly to **vkGetPhysicalDeviceQueueFamilyProperties**, with the ability to return extended information in a **pNext** chain of output structures.
Valid Usage (Implicit)

- **VUID-vkGetPhysicalDeviceQueueFamilyProperties2-physicalDevice-parameter**
  - `physicalDevice` **must** be a valid `VkPhysicalDevice` handle

- **VUID-vkGetPhysicalDeviceQueueFamilyProperties2-pQueueFamilyPropertyCount-parameter**
  - `pQueueFamilyPropertyCount` **must** be a valid pointer to a `uint32_t` value

- **VUID-vkGetPhysicalDeviceQueueFamilyProperties2-pQueueFamilyProperties-parameter**
  - If the value referenced by `pQueueFamilyPropertyCount` is not 0, and `pQueueFamilyProperties` 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 the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `queueFamilyProperties` is a `VkQueueFamilyProperties` structure which is populated with the same values as in `vkGetPhysicalDeviceQueueFamilyProperties`.

Valid Usage (Implicit)

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

- **VUID-VkQueueFamilyProperties2-pNext-pNext**
  - `pNext` **must** be NULL or a pointer to a valid instance of `VkQueueFamilyCheckpointProperties2NV`

- **VUID-VkQueueFamilyProperties2-sType-unique**
  - The `sType` value of each struct in the `pNext` chain **must** be unique

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

```c
// Provided by VK_KHR_performance_query
VkResult vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR(
    VkPhysicalDevice physicalDevice,
    uint32_t queueFamilyIndex,
    VkQueueFamilyProperties2Properties2,
    void* pNext
);
```
uint32_t* pCounterCount,
VkPerformanceCounterKHR* pCounters,
VkPerformanceCounterDescriptionKHR* pCounterDescriptions);

- **physicalDevice** is the handle to the physical device whose queue family performance query counter properties will be queried.
- **queueFamilyIndex** is the index into the queue family of the physical device we want to get properties for.
- **pCounterCount** is a pointer to an integer related to the number of counters available or queried, as described below.
- **pCounters** is either NULL or a pointer to an array of VkPerformanceCounterKHR structures.
- **pCounterDescriptions** is either NULL or a pointer to an array of VkPerformanceCounterDescriptionKHR structures.

If **pCounters** is NULL and **pCounterDescriptions** is NULL, then the number of counters available is returned in **pCounterCount**. Otherwise, **pCounterCount** must point to a variable set by the user to the number of elements in the **pCounters**, **pCounterDescriptions**, or both arrays and on return the variable is overwritten with the number of structures actually written out. If **pCounterCount** is less than the number of counters available, at most **pCounterCount** structures will be written, and **VK_INCOMPLETE** will be returned instead of **VK_SUCCESS**, to indicate that not all the available counters were returned.

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

**Valid Usage (Implicit)**

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

**Success**
- VK_SUCCESS
- VK_INCOMPLETE

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_INITIALIZATION_FAILED

The `VkPerformanceCounterKHR` structure is defined as:

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

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

**Valid Usage (Implicit)**

- VUID-VkPerformanceCounterKHR-sType-sType
  - **sType** must be `VK_STRUCTURE_TYPE_PERFORMANCE_COUNTER_KHR`
- VUID-VkPerformanceCounterKHR-pNext-pNext
  - **pNext** must be NULL

Performance counters have an associated unit. This unit describes how to interpret the performance counter result.
The performance counter unit types which may be returned in `VkPerformanceCounterKHR::unit` are:

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

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

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

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

```c
typedef enum VkPerformanceCounterScopeKHR {
    VK_PERFORMANCE_COUNTER_SCOPE_COMMAND_BUFFER_KHR = 0,
    VK_PERFORMANCE_COUNTER_SCOPE_RENDER_PASS_KHR = 1,
    VK_PERFORMANCE_COUNTER_SCOPE_COMMAND_KHR = 2,
} VkPerformanceCounterScopeKHR;
```
VK_QUERY_SCOPE_COMMAND_BUFFER_KHR = VK_PERFORMANCE_COUNTER_SCOPE_COMMAND_BUFFER_KHR,
VK_QUERY_SCOPE_RENDER_PASS_KHR = VK_PERFORMANCE_COUNTER_SCOPE_RENDER_PASS_KHR,
VK_QUERY_SCOPE_COMMAND_KHR = VK_PERFORMANCE_COUNTER_SCOPE_COMMAND_KHR,
} VkPerformanceCounterScopeKHR;

• VK_PERFORMANCE_COUNTER_SCOPE_COMMAND_BUFFER_KHR - the performance counter scope is a single complete command buffer.
• VK_PERFORMANCE_COUNTER_SCOPE_RENDER_PASS_KHR - the performance counter scope is zero or more complete render passes. The performance query containing the performance counter must begin and end outside a render pass instance.
• VK_PERFORMANCE_COUNTER_SCOPE_COMMAND_KHR - the performance counter scope is zero or more commands.

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

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

```c
// Provided by VK_KHR_performance_query
typedef enum VkPerformanceCounterStorageKHR {
    VK_PERFORMANCE_COUNTER_STORAGE_INT32_KHR = 0,
    VK_PERFORMANCE_COUNTER_STORAGE_INT64_KHR = 1,
    VK_PERFORMANCE_COUNTER_STORAGE_UINT32_KHR = 2,
    VK_PERFORMANCE_COUNTER_STORAGE_UINT64_KHR = 3,
    VK_PERFORMANCE_COUNTER_STORAGE_FLOAT32_KHR = 4,
    VK_PERFORMANCE_COUNTER_STORAGE_FLOAT64_KHR = 5,
} VkPerformanceCounterStorageKHR;
```

• VK_PERFORMANCE_COUNTER_STORAGE_INT32_KHR - the performance counter storage is a 32-bit signed integer.
• VK_PERFORMANCE_COUNTER_STORAGE_INT64_KHR - the performance counter storage is a 64-bit signed integer.
• VK_PERFORMANCE_COUNTER_STORAGE_UINT32_KHR - the performance counter storage is a 32-bit unsigned integer.
• VK_PERFORMANCE_COUNTER_STORAGE_UINT64_KHR - the performance counter storage is a 64-bit unsigned integer.
• VK_PERFORMANCE_COUNTER_STORAGE_FLOAT32_KHR - the performance counter storage is a 32-bit floating-point.
• VK_PERFORMANCE_COUNTER_STORAGE_FLOAT64_KHR - the performance counter storage is a 64-bit floating-point.

The VkPerformanceCounterDescriptionKHR structure is defined as:
typedef struct VkPerformanceCounterDescriptionKHR {
    VkStructureType sType;
    void* pNext;
    VkPerformanceCounterDescriptionFlagsKHR flags;
    char name[VK_MAX_DESCRIPTION_SIZE];
    char category[VK_MAX_DESCRIPTION_SIZE];
    char description[VK_MAX_DESCRIPTION_SIZE];
} VkPerformanceCounterDescriptionKHR;

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

### Valid Usage (Implicit)

- VUID-VkPerformanceCounterDescriptionKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_PERFORMANCE_COUNTER_DESCRIPTION_KHR
- VUID-VkPerformanceCounterDescriptionKHR-pNext-pNext
  pNext must be NULL

Bits which can be set in VkPerformanceCounterDescriptionKHR::flags to specify usage behavior for a performance counter are:

- **VK_PERFORMANCE_COUNTER_DESCRIPTION_PERFORMANCE_IMPACTING_BIT_KHR** specifies that recording the counter may have a noticeable performance impact.
• **VK_PERFORMANCE_COUNTER_DESCRIPTION_CONCURRENTLY_IMPACTED_BIT_KHR** specifies that concurrently recording the counter while other submitted command buffers are running may impact the accuracy of the recording.

```c
// Provided by VK_KHR_performance_query
typedefVkFlags VkPerformanceCounterDescriptionFlagsKHR;
```

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

### 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,

```
uint32_t* pPhysicalDeviceGroupCount,
VkPhysicalDeviceGroupProperties* pPhysicalDeviceGroupProperties);

• instance is a handle to a Vulkan instance previously created with vkCreateInstance.

• pPhysicalDeviceGroupCount is a pointer to an integer related to the number of device groups available or queried, as described below.

• pPhysicalDeviceGroupProperties is either NULL or a pointer to an array of VkPhysicalDeviceGroupProperties structures.

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

Every physical device must be in exactly one device group.

Valid Usage (Implicit)

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

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

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

Return Codes

Success

• VK_SUCCESS

• VK_INCOMPLETE

Failure

• VK_ERROR_OUT_OF_HOST_MEMORY

• VK_ERROR_OUT_OF_DEVICE_MEMORY

• VK_ERROR_INITIALIZATION_FAILED
The `VkPhysicalDeviceGroupProperties` structure is defined as:

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

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

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceGroupProperties-sType-sType**
  - `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GROUP_PROPERTIES`
- **VUID-VkPhysicalDeviceGroupProperties-pNext-pNext**
  - `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`.

To provide _application parameters_ at device creation time, an application _can_ link one or more `VkApplicationParametersEXT` structures to the `pNext` chain of the `VkDeviceCreateInfo` structure.

If the `VkApplicationParametersEXT::vendorID` and `VkApplicationParametersEXT::deviceID` values do...
not match the `VkPhysicalDeviceProperties::vendorID` and `VkPhysicalDeviceProperties::deviceID` of `physicalDevice`, `vkCreateDevice` must return `VK_ERROR_INITIALIZATION_FAILED`.

If `VkApplicationParametersEXT::key` is not a valid implementation-defined application parameter key for the device being created, or if `value` is not a valid value for the specified `key`, `vkCreateDevice` will fail and return `VK_ERROR_INITIALIZATION_FAILED`.

For any implementation-defined application parameter `key` that exists but is not set by the application, the implementation-specific default value is used.

---

**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-key-05092**
  
  The key value of each `VkApplicationParametersEXT` structure in the `VkDeviceCreateInfo::pNext` chain must be unique

- **VUID-vkCreateDevice-deviceMemoryRequestCount-05095**
  
  The sum of `deviceMemoryRequestCount` over all `VkDeviceObjectReservationCreateInfo` structures must be less than or equal to `VkPhysicalDeviceLimits::maxMemoryAllocationCount`

- **VUID-vkCreateDevice-samplerRequestCount-05096**
  
  The sum of `samplerRequestCount` over all `VkDeviceObjectReservationCreateInfo` structures 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:

```c
// 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* const* ppEnabledLayerNames;
    uint32_t enabledExtensionCount;
    const char* const* ppEnabledExtensionNames;
    const VkPhysicalDeviceFeatures* pEnabledFeatures;
} VkDeviceCreateInfo;
```

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

Valid Usage

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

- 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, VkPhysicalDeviceDescriptorIndexingFeatures, VkPhysicalDeviceScalarBlockLayoutFeatures, VkPhysicalDeviceImagelessFramebufferFeatures, VkPhysicalDeviceUniformBufferStandardLayoutFeatures, VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures, VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures, VkPhysicalDeviceHostQueryResetFeatures, VkPhysicalDeviceTimelineSemaphoreFeatures, VkPhysicalDeviceBufferDeviceAddressFeatures, or VkPhysicalDeviceVulkanMemoryModelFeatures structure.

- VUID-VkDeviceCreateInfo-shadingRateImage-04478
  If shadingRateImage is enabled, pipelineFragmentShadingRate must not be enabled.

- VUID-VkDeviceCreateInfo-shadingRateImage-04479
  If shadingRateImage is enabled, primitiveFragmentShadingRate must not be enabled.

- VUID-VkDeviceCreateInfo-shadingRateImage-04480
  If shadingRateImage is enabled, attachmentFragmentShadingRate must not be enabled.

- VUID-VkDeviceCreateInfo-fragmentDensityMap-04481
  If fragmentDensityMap is enabled, pipelineFragmentShadingRate must not be enabled.

- VUID-VkDeviceCreateInfo-fragmentDensityMap-04482
  If fragmentDensityMap is enabled, primitiveFragmentShadingRate must not be enabled.
• VUID-VkDeviceCreateInfo-fragmentDensityMap-04483
   If `fragmentDensityMap` is enabled, `attachmentFragmentShadingRate` must not be enabled

• VUID-VkDeviceCreateInfo-None-04896
   If `sparseImageInt64Atomics` is enabled, `shaderImageInt64Atomics` must be enabled

• VUID-VkDeviceCreateInfo-None-04897
   If `sparseImageFloat32Atomics` is enabled, `shaderImageFloat32Atomics` must be enabled

• VUID-VkDeviceCreateInfo-None-04898
   If `sparseImageFloat32AtomicAdd` is enabled, `shaderImageFloat32AtomicAdd` must be enabled

---

Valid Usage (Implicit)

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

• VUID-VkDeviceCreateInfo-pNext-pNext
   Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkApplicationParametersEXT`, `VkDeviceGroupDeviceCreateInfo`, `VkFaultCallbackInfo`, `VkPhysicalDevice4444FormatsFeaturesEXT`, `VkPhysicalDeviceASTCDecodeFeaturesEXT`, `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT`, `VkPhysicalDeviceBufferDeviceAddressFeatures`, `VkPhysicalDeviceColorWriteEnableFeaturesEXT`, `VkPhysicalDeviceCustomBorderColorFeaturesEXT`, `VkPhysicalDeviceDepthClipEnableFeaturesEXT`, `VkPhysicalDeviceDescriptorIndexingFeatures`, `VkPhysicalDeviceExtendedDynamicState2FeaturesEXT`, `VkPhysicalDeviceExtendedDynamicStateFeaturesEXT`, `VkPhysicalDeviceExternalMemoryScibuffeaturesNV`, `VkPhysicalDeviceExternalSciSync2FeaturesNV`, `VkPhysicalDeviceExternalSciSyncFeaturesNV`, `VkPhysicalDeviceFeatures2`, `VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT`, `VkPhysicalDeviceFragmentShadingRateFeaturesKHR`, `VkPhysicalDeviceHostQueryResetFeatures`, `VkPhysicalDeviceImageRobustnessFeaturesEXT`, `VkPhysicalDeviceImagelessFramebufferFeatures`, `VkPhysicalDeviceIndexTypeUint8FeaturesEXT`, `VkPhysicalDeviceLineRasterizationFeaturesEXT`, `VkPhysicalDeviceMultiviewFeatures`, `VkPhysicalDevicePerformanceQueryFeaturesKHR`, `VkPhysicalDeviceProtectedMemoryFeatures`, `VkPhysicalDeviceRobustness2FeaturesEXT`, `VkPhysicalDeviceSamplerYcbcrConversionFeatures`, `VkPhysicalDeviceScalarBlockLayoutFeatures`, `VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures`, `VkPhysicalDeviceShaderAtomicFloatFeaturesEXT`, `VkPhysicalDeviceShaderAtomicInt64Features`

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

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

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

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

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

- VUID-VkDeviceCreateInfo-pEnabledFeatures-parameter
  If `pEnabledFeatures` is not `NULL`, `pEnabledFeatures` must be a valid pointer to a valid `VkPhysicalDeviceFeatures` structure

- VUID-VkDeviceCreateInfo-queueCreateInfoCount-arraylength
  `queueCreateInfoCount` must be greater than 0

// Provided by VK_VERSION_1_0

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

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

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

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

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

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

### Valid Usage

- VUID-VkDeviceGroupDeviceCreateInfo-pPhysicalDevices-00375 Each element of pPhysicalDevices must be unique
- VUID-VkDeviceGroupDeviceCreateInfo-pPhysicalDevices-00376 All elements of pPhysicalDevices must be in the same device group as enumerated by 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.

// Provided by VKSC_VERSION_1_0

typedef struct VkDeviceObjectReservationCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t pipelineCacheCreateInfoCount;
    const VkPipelineCacheCreateInfo* pPipelineCreateInfos;
    uint32_t pipelinePoolSizeCount;
    const VkPipelinePoolSize* pPipelinePoolSizes;
    uint32_t semaphoreRequestCount;
    uint32_t commandBufferRequestCount;
    uint32_t fenceRequestCount;
    uint32_t deviceMemoryRequestCount;
    uint32_t bufferRequestCount;
    uint32_t imageViewRequestCount;
    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 the type of 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.
• imageViewRequestCount 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 of 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, and maxTimestampQueriesPerPool 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

If the pNext chain of VkDeviceObjectReservationCreateInfo includes a VkPerformanceQueryReservationInfoKHR structure, then the structure indicates upper bounds on the number of performance queries that can exist at the same time in a query pool.

The VkPerformanceQueryReservationInfoKHR structure is defined as:

```c
// Provided by VKSC_VERSION_1_0 with VK_KHR_performance_query
typedef struct VkPerformanceQueryReservationInfoKHR {
    VkStructureType sType;
    const void* pNext;
    uint32_t maxPerformanceQueriesPerPool;
} VkPerformanceQueryReservationInfoKHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- maxPerformanceQueriesPerPool is the requested maximum number of VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR queries that can exist at the same time in a single query pool.

If the VkDeviceObjectReservationCreateInfo::pNext chain does not include this structure, then maxPerformanceQueriesPerPool defaults to 0.

Multiple VkPerformanceQueryReservationInfoKHR structures can be chained together. The maximum value from all instances of maxPerformanceQueriesPerPool will be reserved.
If the `pNext` chain of `VkDeviceObjectReservationCreateInfo` includes a `VkDeviceSemaphoreSciSyncPoolReservationCreateInfoNV` structure, then the structure indicates the maximum number of `VkSemaphoreSciSyncPoolNV` objects that can exist at the same time.

The `VkDeviceSemaphoreSciSyncPoolReservationCreateInfoNV` structure is defined as:

```c
// Provided by VKSC_VERSION_1_0 with VK_NV_external_sci_sync2
typedef struct VkDeviceSemaphoreSciSyncPoolReservationCreateInfoNV {
    VkStructureType sType;
    const void* pNext;
    uint32_t semaphoreSciSyncPoolRequestCount;
} VkDeviceSemaphoreSciSyncPoolReservationCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `semaphoreSciSyncPoolRequestCount` is the requested maximum number of `VkSemaphoreSciSyncPoolNV` objects that can exist at the same time.

If the `VkDeviceObjectReservationCreateInfo::pNext` chain does not include this structure, then `semaphoreSciSyncPoolRequestCount` defaults to 0.

Multiple `VkDeviceSemaphoreSciSyncPoolReservationCreateInfoNV` structures can be chained together. The sum of the `semaphoreSciSyncPoolRequestCount` values from all instances of `VkDeviceSemaphoreSciSyncPoolReservationCreateInfoNV` will be reserved.

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;
} VkPipelinePoolSize;
```
uint32_t poolEntryCount;
} VkPipelinePoolSize;

• sType is the type of 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 commands that return a VkResult may return VK_ERROR_DEVICE_LOST. 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 or vkAcquireNextImageKHR with a maximum timeout, and vkGetQueryPoolResults with the VK_QUERY_RESULT_WAIT_BIT bit set in flags) must return in finite time even in the case of a lost device, and return either VK_SUCCESS or VK_ERROR_DEVICE_LOST. For any command that may return VK_ERROR_DEVICE_LOST, for the purpose of determining whether a command buffer is in the pending state, or whether resources are considered in-use by the device, a return value of VK_ERROR_DEVICE_LOST is equivalent to VK_SUCCESS.

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

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

5.2.4. Device Destruction

To destroy a device, call:

```c
// Provided by VK_VERSION_1_0
define 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`
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:

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

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

### Valid Usage

- **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 protected memory feature is not enabled, the `VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT` bit of `flags` must not be set
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` or a pointer to a valid instance of `VkDeviceQueueGlobalPriorityCreateInfoEXT`.

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

- **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` to specify usage behavior of the 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`.

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

The `VkDeviceQueueGlobalPriorityCreateInfoEXT` structure is defined as:
// Provided by VK_EXT_global_priority

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

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

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

### Valid Usage (Implicit)

- **VUID-VkDeviceQueueGlobalPriorityCreateInfoEXT-sType-sType**
  - **sType** must be **VK_STRUCTURE_TYPE_DEVICE_QUEUE_GLOBAL_PRIORITY_CREATE_INFO_EXT**
- **VUID-VkDeviceQueueGlobalPriorityCreateInfoEXT-globalPriority-parameter**
  - **globalPriority** must be a valid **VkQueueGlobalPriorityEXT** value

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

// Provided by VK_EXT_global_priority

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

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

- **VK_QUEUEGLOBAL_PRIORITY_LOW_EXT** is below the system default. Useful for non-interactive tasks.
- **VK_QUEUEGLOBAL_PRIORITY_MEDIUM_EXT** is the system default priority.
- **VK_QUEUEGLOBAL_PRIORITY_HIGH_EXT** is above the system default.
- **VK_QUEUEGLOBAL_PRIORITY_REALTIME_EXT** is the highest priority. Useful for critical tasks.

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

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

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

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

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

To retrieve a handle to a VkQueue object, call:

```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 `VkDeviceQueueCreateFlags` 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 the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure. The pNext chain of VkDeviceQueueInfo2 can be used to provide additional device queue parameters to vkGetDeviceQueue2.

• flags is a VkDeviceQueueCreateInfo value indicating the flags used to create the device queue.
• queueFamilyIndex is the index of the queue family to which the queue belongs.
• queueIndex is the index within this queue family of the queue to retrieve.

The queue returned by vkGetDeviceQueue2 must have the same flags value from this structure as that used at device creation time in a VkDeviceQueueCreateInfo structure. If no matching flags were specified at device creation time, then the handle returned in pQueue must be NULL.

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 VkDeviceQueueCreateInfo 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 `vkQueueSubmit2KHR` or `vkQueueSubmit`. Queue submission commands define a set of *queue operations* to be executed by the underlying physical device, including synchronization with semaphores and fences.

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

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

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

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

```cpp
// 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

\texttt{vkBeginCommandBuffer} changes the state of a command buffer from the initial state to the \textit{recording state}. Once a command buffer is in the recording state, \texttt{vkCmd*} commands \textbf{can} be used to record to the command buffer.

Executable

\texttt{vkEndCommandBuffer} ends the recording of a command buffer, and moves it from the recording state to the \textit{executable state}. Executable command buffers \textbf{can} be submitted, reset, or recorded to another command buffer.

Pending

\textit{Queue submission} of a command buffer changes the state of a command buffer from the executable state to the \textit{pending state}. Whilst in the pending state, applications \textbf{must} not attempt to modify the command buffer in any way - as the device \textbf{may} be processing the commands recorded to it. Once execution of a command buffer completes, the command buffer either reverts back to the \textit{executable state}, or if it was recorded with \texttt{VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT}, it moves to the \textit{invalid state}. A \textit{synchronization} command \textbf{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 \textit{invalid state}. Command buffers in the invalid state \textbf{can} only be reset or freed.

Any given command that operates on a command buffer has its own requirements on what state a command buffer \textbf{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 \textit{initial state}. Resetting occurs as a result of \texttt{vkResetCommandBuffer} or \texttt{vkResetCommandPool}, or as part of \texttt{vkBeginCommandBuffer} (which additionally puts the command buffer in the \textit{recording state}).

\textbf{Secondary command buffers} \textbf{can} be recorded to a primary command buffer via

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{lifecycle_of_a_command_buffer.png}
\caption{Lifecycle of a command buffer}
\end{figure}
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`. 

128
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 the type of this structure.
• `pNext` is NULL or a pointer to a structure extending this structure.

• `flags` is a bitmask of `VkCommandPoolCreateFlagBits` indicating usage behavior for the pool and command buffers allocated from it.

• `queueFamilyIndex` designates a queue family as described in section `Queue Family Properties`. All command buffers allocated from this command pool must be submitted on queues from the same queue family.

### Valid Usage

- **VUID-VkCommandPoolCreateInfo-flags-02860**
  If the protected memory 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` to specify usage behavior for a command pool are:

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

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

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

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

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

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

The `pNext` chain of `VkCommandPoolCreateInfo` must include a `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 `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;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **commandPoolReservedSize** is the number of bytes to be allocated for all command buffer data recorded into this pool.
- **commandPoolMaxCommandBuffers** is the maximum number of command buffers that can be allocated from this command pool.

The number of command buffers reserved using `commandPoolMaxCommandBuffers` is permanently counted against the total number of command buffers requested via `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 `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 `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.

**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 **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 `vkGetCommandPoolMemoryConsumption` will be sufficient to record the same command buffers again.

### Valid Usage

- **VUID-VkCommandPoolMemoryReservationCreateInfo-commandPoolReservedSize-05003**
  
  `commandPoolReservedSize` **must** be greater than 0

- **VUID-VkCommandPoolMemoryReservationCreateInfo-commandPoolMaxCommandBuffers-05004**
  
  `commandPoolMaxCommandBuffers` **must** be greater than 0

- **VUID-VkCommandPoolMemoryReservationCreateInfo-commandPoolMaxCommandBuffers-05090**
  
  `commandPoolMaxCommandBuffers` **must** be less than or equal to `VkPhysicalDeviceVulkanSC10Properties::maxCommandPoolCommandBuffers`

- **VUID-VkCommandPoolMemoryReservationCreateInfo-commandPoolMaxCommandBuffers-05074**
  
  The number of command buffers reserved by all command pools plus `commandPoolMaxCommandBuffers` **must** be less than or equal to the total number of command buffers requested via `VkDeviceObjectReservationCreateInfo::commandBufferRequestCount`

### Valid Usage (Implicit)

- **VUID-VkCommandPoolMemoryReservationCreateInfo-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_COMMAND_POOL_MEMORY_RESERVATION_CREATE_INFO`

To get memory usage information for a command pool object, call:
```c
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;
} VkCommandPoolMemoryConsumption;
```
 VkDeviceSize commandPoolAllocated;
VkDeviceSize commandPoolReservedSize;
VkDeviceSize commandBufferAllocated;
} VkCommandPoolMemoryConsumption;

- **sType** is the type of 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

- VUID-vkResetCommandPool-flags-05005
  flags must not contain VK_COMMAND_POOL_RESET_RELEASE_RESOURCES_BIT

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-zerobitmask
  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 to control the reset operation are:

```c
// 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;
```
\texttt{VkCommandPoolResetFlags} is a bitmask type for setting a mask of zero or more \texttt{VkCommandPoolResetFlagBits}. Command pools cannot be destroyed or trimmed \cite{SCID-4}. If \texttt{VkPhysicalDeviceVulkanSC10Properties::deviceDestroyFreesMemory} is \texttt{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);
```

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

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

\textit{Note}

Filling \texttt{pCommandBuffers} with \texttt{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 \texttt{VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations} is \texttt{VK_TRUE}, \texttt{vkAllocateCommandBuffers} must not return \texttt{VK_ERROR_OUT_OF_HOST_MEMORY}.

### Valid Usage (Implicit)

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

• VUID-vkAllocateCommandBuffers-pAllocateInfo::commandBufferCount-arraylength
  pAllocateInfo->commandBufferCount must be greater than 0

Host Synchronization

• Host access to pAllocateInfo->commandPool must be externally synchronized

Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkCommandBufferAllocateInfo structure is defined as:

```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 the type of 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`.

138
Valid Usage

- VUID-vkResetCommandBuffer-commandBuffer-00045
  
  `commandBuffer` must not be in the pending state

- VUID-vkResetCommandBuffer-commandBuffer-00046
  
  `commandBuffer` must have been allocated from a pool that was created with the `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT`

- VUID-vkResetCommandBuffer-commandPoolResetCommandBuffer-05135
  
  `commandPoolResetCommandBuffer` must be supported

Valid Usage (Implicit)

- VUID-vkResetCommandBuffer-commandBuffer-parameter
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkResetCommandBuffer-flags-parameter
  
  `flags` must be a valid combination of `VkCommandBufferResetFlagBits` values

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

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

Return Codes

Success

- `VK_SUCCESS`

Failure

- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

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

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

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

Valid Usage

• VUID-VkCommandBufferBeginInfo-flags-00053
  If flags contains VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT, the renderPass member of pInheritanceInfo must be a valid VkRenderPass

• VUID-VkCommandBufferBeginInfo-flags-00054
  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

• 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 must be a valid VkFramebuffer that is compatible with the renderPass member of pInheritanceInfo and must not have been created with a VkFramebufferCreateInfo::flags value that includes VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT

Valid Usage (Implicit)

• VUID-VkCommandBufferBeginInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO

• VUID-VkCommandBufferBeginInfo-pNext-pNext
  pNext must be NULL or a pointer to a valid instance of
VkDeviceGroupCommandBufferBeginInfo

- VUID-VkCommandBufferBeginInfo-sType-unique
  The sType value of each struct in the pNext chain must be unique
- VUID-VkCommandBufferBeginInfo-flags-parameter
  flags must be a valid combination of VkCommandBufferUsageFlagBits values

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

```c
// Provided by VK_VERSION_1_0
typedef enum VkCommandBufferUsageFlagBits {
    VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT = 0x00000001,
    VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT = 0x00000002,
    VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT = 0x00000004,
}VkCommandBufferUsageFlagBits;
```

- **VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT** specifies that each recording of the command buffer will only be submitted once, and the command buffer will be reset and recorded again between each submission.

- **VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT** specifies that a secondary command buffer is considered to be entirely inside a render pass. If this is a primary command buffer, then this bit is ignored.

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

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

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

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

```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 the type of this structure.

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

• **renderPass** is a **VkRenderPass** object defining which render passes the **VkCommandBuffer** will be compatible with and **can** be executed within.

• **subpass** is the index of the subpass within the render pass instance that the **VkCommandBuffer** will be executed within.

• **framebuffer** can refer to the **VkFramebuffer** object that the **VkCommandBuffer** will be rendering to if it is executed within a render pass instance. It **can be** **VK_NULL_HANDLE** if the framebuffer is not known.

---

**Note**

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

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

• **queryFlags** specifies the query flags that **can** be used by an active occlusion query in the primary command buffer when this secondary command buffer is executed. If this value includes the **VK_QUERY_CONTROL_PRECISE_BIT** bit, then the active query **can** return boolean results or actual sample counts. If this bit is not set, then the active query **must** not use the **VK_QUERY_CONTROL_PRECISE_BIT** bit.

• **pipelineStatistics** is a bitmask of **VkQueryPipelineStatisticFlagBits** specifying the set of pipeline statistics that **can** be counted by an active query in the primary command buffer when this secondary command buffer is executed. If this value includes a given bit, then this command buffer **can** be executed whether the primary command buffer has a pipeline statistics query active that includes this bit or not. If this value excludes a given bit, then the active pipeline statistics query **must** not be from a query pool that counts that statistic.

If the **VkCommandBuffer** will not be executed within a render pass instance, **renderPass, subpass**, and **framebuffer** are ignored.

---

**Valid Usage**

• **VUID-VkCommandBufferInheritanceInfo-occlusionQueryEnable-00056**  
  If the **inherited queries** feature is not enabled, **occlusionQueryEnable** **must** be **VK_FALSE**

• **VUID-VkCommandBufferInheritanceInfo-queryFlags-00057**  
  If the **inherited queries** feature is enabled, **queryFlags** **must** be a valid combination of **VkQueryControlFlagBits** values
If the inherited queries feature is not enabled, queryFlags must be 0.

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

If the pipeline statistics queries feature is not enabled, pipelineStatistics must be 0.

Valid Usage (Implicit)

- VUID-VkCommandBufferInheritanceInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_INFO.

- VUID-VkCommandBufferInheritanceInfo-pNext-pNext
  pNext must be NULL.

- VUID-VkCommandBufferInheritanceInfo-commonparent
  Both of framebuffer, and renderPass that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same VkDevice.

Note

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

If a command buffer is in the invalid, or executable state, and the command buffer was allocated from a command pool with the VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT flag set, then vkBeginCommandBuffer implicitly resets the command buffer, behaving as if vkResetCommandBuffer had been called with VK_COMMAND_BUFFER_RESET_RELEASE_RESOURCES_BIT not set. After the implicit reset, commandBuffer is moved to the recording state.

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

To complete recording of a command buffer, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkEndCommandBuffer( VkCommandBuffer commandBuffer);
```

- commandBuffer is the command buffer to complete recording.

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

If the memory used for command recording exceeded the amount reserved by 
VkCommandPoolMemoryReservationCreateInfo::commandPoolReservedSize in the command pool
that commandBuffer was allocated from, then vkEndCommandBuffer must return
VK_ERROR_OUT_OF_DEVICE_MEMORY.

Note
Implementations can also use Fault Handling and in particular the
VK_FAULT_TYPE_COMMAND_BUFFER_FULL fault to report vkCmd* recording errors earlier
than vkEndCommandBuffer.

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

• VUID-vkEndCommandBuffer-commandBuffer-01815
  If commandBuffer is a secondary command buffer, there must not be an outstanding
  vkCmdBeginDebugUtilsLabelEXT command recorded to commandBuffer that has not
  previously been ended by a call to vkCmdEndDebugUtilsLabelEXT

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
6.5. Command Buffer Submission

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

To submit command buffers to a queue, call:

```cpp
// Provided by VK_KHR_synchronization2
VkResult vkQueueSubmit2KHR(
    VkQueue queue,
    uint32_t submitCount,
    const VkSubmitInfo2KHR* 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 `VkSubmitInfo2KHR` 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.

`vkQueueSubmit2KHR` is a queue submission command, with each batch defined by an element of `pSubmits`.

Semaphore operations submitted with `vkQueueSubmit2KHR` have additional ordering constraints compared to other submission commands, with dependencies involving previous and subsequent queue operations. Information about these additional constraints can be found in the semaphore section of the synchronization chapter.

If any command buffer submitted to this queue is in the executable state, it is moved to the pending state. Once execution of all submissions of a command buffer complete, it moves from the pending state, back to the executable state. If a command buffer was recorded with the
VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT flag, it instead moves back to the invalid state.

If vkQueueSubmit2KHR fails, it may return VK_ERROR_OUT_OF_HOST_MEMORY or VK_ERROR_OUT_OF_DEVICE_MEMORY. If it does, the implementation must ensure that the state and contents of any resources or synchronization primitives referenced by the submitted command buffers and any semaphores referenced by pSubmits is unaffected by the call or its failure. If vkQueueSubmit2KHR fails in such a way that the implementation is unable to make that guarantee, the implementation must return VK_ERROR_DEVICE_LOST. See Lost Device.

If VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations is VK_TRUE, vkQueueSubmit2KHR must not return VK_ERROR_OUT_OF_HOST_MEMORY.

Valid Usage

- VUID-vkQueueSubmit2KHR-fence-04894
  If fence is not VK_NULL_HANDLE, fence must be unsignaled

- VUID-vkQueueSubmit2KHR-fence-04895
  If fence is not VK_NULL_HANDLE, fence must not be associated with any other queue command that has not yet completed execution on that queue

- VUID-vkQueueSubmit2KHR-synchronization2-03866
  The synchronization2 feature must be enabled

- VUID-vkQueueSubmit2KHR-commandBuffer-03867
  If a command recorded into the commandBuffer member of any element of the pCommandBufferInfos member of any element of pSubmits referenced an VkEvent, that event must not be referenced by a command that has been submitted to another queue and is still in the pending state

- VUID-vkQueueSubmit2KHR-semaphore-03868
  The semaphore member of any binary semaphore element of the pSignalSemaphoreInfos member of any element of pSubmits must be unsignaled when the semaphore signal operation it defines is executed on the device

- VUID-vkQueueSubmit2KHR-stageMask-03869
  The stageMask member of any element of the pSignalSemaphoreInfos member of any element of pSubmits must only include pipeline stages that are supported by the queue family which queue belongs to

- VUID-vkQueueSubmit2KHR-stageMask-03870
  The stageMask member of any element of the pWaitSemaphoreInfos member of any element of pSubmits must only include pipeline stages that are supported by the queue family which queue belongs to

- VUID-vkQueueSubmit2KHR-semaphore-03871
  When a semaphore wait operation for a binary semaphore is executed, as defined by the semaphore member of any element of the pWaitSemaphoreInfos member of any element of pSubmits, there must be no other queues waiting on the same semaphore

- VUID-vkQueueSubmit2KHR-semaphore-03872
  The semaphore member of any element of the pWaitSemaphoreInfos member of any element of pSubmits must be semaphores that are signaled, or have semaphore signal operations
previouly submitted for execution

- **VUID-vkQueueSubmit2KHR-commandBuffer-03874**
  The `commandBuffer` member of any element of the `pCommandBufferInfos` member of any element of `pSubmits` **must** be in the pending or executable state

- **VUID-vkQueueSubmit2KHR-commandBuffer-03875**
  If a command recorded into the `commandBuffer` member of any element of the `pCommandBufferInfos` member of any element of `pSubmits` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT`, it **must** not be in the pending state

- **VUID-vkQueueSubmit2KHR-commandBuffer-03876**
  Any secondary command buffers recorded into the `commandBuffer` member of any element of the `pCommandBufferInfos` member of any element of `pSubmits` **must** be in the pending or executable state

- **VUID-vkQueueSubmit2KHR-commandBuffer-03877**
  If any secondary command buffers recorded into the `commandBuffer` member of any element of the `pCommandBufferInfos` member of any element of `pSubmits` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT`, it **must** not be in the pending state

- **VUID-vkQueueSubmit2KHR-commandBuffer-03878**
  The `commandBuffer` member of any element of the `pCommandBufferInfos` member of any element of `pSubmits` **must** have been allocated from a `VkCommandPool` that was created for the same queue family `queue` belongs to

- **VUID-vkQueueSubmit2KHR-commandBuffer-03879**
  If a command recorded into the `commandBuffer` member of any element of the `pCommandBufferInfos` member of any element of `pSubmits` includes a Queue Family Transfer Acquire Operation, there **must** exist a previously submitted Queue Family Transfer Release Operation on a queue in the queue family identified by the acquire operation, with parameters matching the acquire operation as defined in the definition of such acquire operations, and which happens before the acquire operation

- **VUID-vkQueueSubmit2KHR-commandBuffer-03880**
  If a command recorded into the `commandBuffer` member of any element of the `pCommandBufferInfos` member of any element of `pSubmits` was a `vkCmdBeginQuery` whose `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, the profiling lock **must** have been held continuously on the `VkDevice` that `queue` was retrieved from, throughout recording of those command buffers

- **VUID-vkQueueSubmit2KHR-queue-06447**
  If `queue` was not created with `VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT`, the `flags` member of any element of `pSubmits` **must** not include `VK_SUBMIT_PROTECTED_BIT_KHR`

---

**Valid Usage (Implicit)**

- **VUID-vkQueueSubmit2KHR-queue-parameter**
  `queue` **must** be a valid `VkQueue` handle

- **VUID-vkQueueSubmit2KHR-pSubmits-parameter**
If `submitCount` is not 0, `pSubmits` must be a valid pointer to an array of `submitCount` valid `VkSubmitInfo2KHR` structures.

- VUID-vkQueueSubmit2KHR-fence-parameter
  If `fence` is not `VK_NULL_HANDLE`, `fence` must be a valid `VkFence` handle.

- VUID-vkQueueSubmit2KHR-commonparent
  Both of `fence`, and `queue` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`.

### Host Synchronization

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>Any</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 `VkSubmitInfo2KHR` structure is defined as:

```c
// Provided by VK_KHR_synchronization2
typedef struct VkSubmitInfo2KHR {
    VkStructureType sType;
    const void* pNext;
    VkSubmitFlagsKHR flags;
    uint32_t waitSemaphoreInfoCount;
    const VkSemaphoreSubmitInfoKHR* pWaitSemaphoreInfos;
    uint32_t commandBufferInfoCount;
    const VkCommandBufferSubmitInfoKHR* pCommandBufferInfos;
    uint32_t signalSemaphoreInfoCount;
    const VkSemaphoreSubmitInfoKHR* pSignalSemaphoreInfos;
} VkSubmitInfo2KHR;
```
VkSubmitInfo2KHR;

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `flags` is a bitmask of `VkSubmitFlagBitsKHR`.
- `waitSemaphoreInfoCount` is the number of elements in `pWaitSemaphoreInfos`.
- `pWaitSemaphoreInfos` is a pointer to an array of `VkSemaphoreSubmitInfoKHR` structures defining semaphore wait operations.
- `commandBufferInfoCount` is the number of elements in `pCommandBufferInfos` and the number of command buffers to execute in the batch.
- `pCommandBufferInfos` is a pointer to an array of `VkCommandBufferSubmitInfoKHR` structures describing command buffers to execute in the batch.
- `signalSemaphoreInfoCount` is the number of elements in `pSignalSemaphoreInfos`.
- `pSignalSemaphoreInfos` is a pointer to an array of `VkSemaphoreSubmitInfoKHR` describing semaphore signal operations.

### Valid Usage

- **VUID-VkSubmitInfo2KHR-flags-03886**
  If `flags` includes `VK_SUBMIT_PROTECTED_BIT_KHR`, all elements of `pCommandBuffers` must be protected command buffers

- **VUID-VkSubmitInfo2KHR-flags-03887**
  If `flags` does not include `VK_SUBMIT_PROTECTED_BIT_KHR`, each element of `pCommandBuffers` must not be a protected command buffer

### Valid Usage (Implicit)

- **VUID-VkSubmitInfo2KHR-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_SUBMIT_INFO_2_KHR`

- **VUID-VkSubmitInfo2KHR-pNext-pNext**
  `pNext` must be NULL or a pointer to a valid instance of `VkPerformanceQuerySubmitInfoKHR`

- **VUID-VkSubmitInfo2KHR-sType-unique**
  The `sType` value of each struct in the `pNext` chain must be unique

- **VUID-VkSubmitInfo2KHR-flags-parameter**
  `flags` must be a valid combination of `VkSubmitFlagBitsKHR` values

- **VUID-VkSubmitInfo2KHR-pWaitSemaphoreInfos-parameter**
  If `waitSemaphoreInfoCount` is not 0, `pWaitSemaphoreInfos` must be a valid pointer to an array of `VkSemaphoreSubmitInfoKHR` structures

- **VUID-VkSubmitInfo2KHR-pCommandBufferInfos-parameter**
If `commandBufferInfoCount` is not 0, `pCommandBufferInfos` must be a valid pointer to an array of `commandBufferInfoCount` valid `VkCommandBufferSubmitInfoKHR` structures.

- **VUID-VkSubmitInfo2KHR-pSignalSemaphoreInfos-parameter**
  If `signalSemaphoreInfoCount` is not 0, `pSignalSemaphoreInfos` must be a valid pointer to an array of `signalSemaphoreInfoCount` valid `VkSemaphoreSubmitInfoKHR` structures.

Bits which can be set in `VkSubmitInfo2KHR::flags` to specify submission behavior are:

```c
// Provided by VK_KHR_synchronization2
typedef enum VkSubmitFlagBitsKHR {
    VK_SUBMIT_PROTECTED_BIT_KHR = 0x00000001,
} VkSubmitFlagBitsKHR;
```

- **VK_SUBMIT_PROTECTED_BIT_KHR** specifies that this batch is a protected submission.

```c
// Provided by VK_KHR_synchronization2
typedef VkFlags VkSubmitFlagsKHR;
```

`VkSubmitFlagsKHR` is a bitmask type for setting a mask of zero or more `VkSubmitFlagBitsKHR`.

The `VkSemaphoreSubmitInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_synchronization2
typedef struct VkSemaphoreSubmitInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkSemaphore semaphore;
    uint64_t value;
    VkPipelineStageFlags2KHR stageMask;
    uint32_t deviceIndex;
} VkSemaphoreSubmitInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **semaphore** is a `VkSemaphore` affected by this operation.
- **value** is ignored.
- **stageMask** is a `VkPipelineStageFlags2KHR` mask of pipeline stages which limit the first synchronization scope of a semaphore signal operation, or second synchronization scope of a semaphore wait operation as described in the **semaphore wait operation** and **semaphore signal operation** sections of the synchronization chapter.
- **deviceIndex** is the index of the device within a device group that executes the semaphore wait or signal operation.
Whether this structure defines a semaphore wait or signal operation is defined by how it is used.

### Valid Usage

- **VUID-VkSemaphoreSubmitInfoKHR-stageMask-03929**
  If the **geometry shaders** feature is not enabled, `stageMask` **must** not contain `VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR`.

- **VUID-VkSemaphoreSubmitInfoKHR-stageMask-03930**
  If the **tessellation shaders** feature is not enabled, `stageMask` **must** not contain `VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR` or `VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR`.

- **VUID-VkSemaphoreSubmitInfoKHR-device-03888**
  If the **device** that semaphore was created on is not a device group, `deviceIndex` **must** be 0.

- **VUID-VkSemaphoreSubmitInfoKHR-device-03889**
  If the **device** that semaphore was created on is a device group, `deviceIndex` **must** be a valid device index.

- **VUID-VkSemaphoreSubmitInfoKHR-semaphore-05094**
  If semaphore has a payload of `NvSciSyncObj`, `value` **must** be calculated by application via `NvSciSync APIs`.

### Valid Usage (Implicit)

- **VUID-VkSemaphoreSubmitInfoKHR-sType-sType**
  `sType` **must** be `VK_STRUCTURE_TYPE_SEMAPHORE_SUBMIT_INFO_KHR`.

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

- **VUID-VkSemaphoreSubmitInfoKHR-semaphore-parameter**
  `semaphore` **must** be a valid `VkSemaphore` handle.

- **VUID-VkSemaphoreSubmitInfoKHR-stageMask-parameter**
  `stageMask` **must** be a valid combination of `VkPipelineStageFlagBits2KHR` values.

The `VkCommandBufferSubmitInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_synchronization2
typedef struct VkCommandBufferSubmitInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkCommandBuffer commandBuffer;
    uint32_t deviceMask;
} VkCommandBufferSubmitInfoKHR;
```

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

• `commandBuffer` is a `VkCommandBuffer` to be submitted for execution.

• `deviceMask` is a bitmask indicating which devices in a device group execute the command buffer. A `deviceMask` of `0` is equivalent to setting all bits corresponding to valid devices in the group to `1`.

## Valid Usage

- VUID-VkCommandBufferSubmitInfoKHR-commandBuffer-03890
  `commandBuffer` must not have been allocated with `VK_COMMAND_BUFFER_LEVEL_SECONDARY`

- VUID-VkCommandBufferSubmitInfoKHR-deviceMask-03891
  If `deviceMask` is not `0`, it must be a valid device mask

## Valid Usage (Implicit)

- VUID-VkCommandBufferSubmitInfoKHR-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_COMMAND_BUFFER_SUBMIT_INFO_KHR`

- VUID-VkCommandBufferSubmitInfoKHR-pNext-pNext
  `pNext` must be `NULL`

- VUID-VkCommandBufferSubmitInfoKHR-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

To submit command buffers to a queue, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkQueueSubmit(
    VkQueue queue,                       // The queue that the command buffers will be submitted to.
    uint32_t submitCount,               // The number of elements in the `pSubmits` array.
    const VkSubmitInfo* pSubmits,       // A pointer to an array of `VkSubmitInfo` structures, each specifying a command buffer submission batch.
    VkFence fence                        // 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.
);
```

• `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
Details on the interaction of `pWaitDstStageMask` with synchronization are described in the semaphore wait operation section of the synchronization chapter.

The order that batches appear in `pSubmits` is used to determine submission order, and thus all the implicit ordering guarantees that respect it. Other than these implicit ordering guarantees and any explicit synchronization primitives, these batches may overlap or otherwise execute out of order.

If any command buffer submitted to this queue is in the executable state, it is moved to the pending state. Once execution of all submissions of a command buffer complete, it moves from the pending state, back to the executable state. If a command buffer was recorded with the `VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT` flag, it instead moves to the invalid state.

If `vkQueueSubmit` fails, it may return `VK_ERROR_OUT_OF_HOST_MEMORY` or `VK_ERROR_OUT_OF_DEVICE_MEMORY`. If it does, the implementation must ensure that the state and contents of any resources or synchronization primitives referenced by the submitted command buffers and any semaphores referenced by `pSubmits` is unaffected by the call or its failure. If `vkQueueSubmit` fails in such a way that the implementation is unable to make that guarantee, the implementation must return `VK_ERROR_DEVICE_LOST`. See Lost Device.

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

### Valid Usage

- **VUID-vkQueueSubmit-fence-00063**
  If `fence` is not `VK_NULL_HANDLE`, `fence` must be unsignaled

- **VUID-vkQueueSubmit-fence-00064**
  If `fence` is not `VK_NULL_HANDLE`, `fence` must not be associated with any other queue command that has not yet completed execution on that queue

- **VUID-vkQueueSubmit-pCommandBuffers-00065**
  Any calls to `vkCmdSetEvent`, `vkCmdResetEvent` or `vkCmdWaitEvents` that have been recorded into any of the command buffer elements of the `pCommandBuffers` member of any element of `pSubmits`, must not reference any `VkEvent` that is referenced by any of those commands in a command buffer that has been submitted to another queue and is still in the pending state

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

- **VUID-vkQueueSubmit-pSignalSemaphores-00067**
  Each binary semaphore element of the `pSignalSemaphores` member of any element of `pSubmits` must be unsignaled when the semaphore signal operation it defines is executed on the device

- **VUID-vkQueueSubmit-pWaitSemaphores-00068**
  When a semaphore wait operation referring to a binary semaphore defined by any
element of the pWaitSemaphores member of any element of pSubmits executes on queue, there must be no other queues waiting on the same semaphore

- **VUID-vkQueueSubmit-pWaitSemaphores-00069**
  All elements of the pWaitSemaphores member of all elements of pSubmits must be semaphores that are signaled, or have semaphore signal operations previously submitted for execution

- **VUID-vkQueueSubmit-pWaitSemaphores-03238**
  All elements of the pWaitSemaphores member of all elements of pSubmits created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_BINARY must reference a semaphore signal operation that has been submitted for execution and any semaphore signal operations on which it depends (if any) must have also been submitted for execution

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

- **VUID-vkQueueSubmit-pCommandBuffers-00071**
  If any element of the pCommandBuffers member of any element of pSubmits was not recorded with the VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT, it must not be in the pending state

- **VUID-vkQueueSubmit-pCommandBuffers-00072**
  Any secondary command buffers recorded into any element of the pCommandBuffers member of any element of pSubmits must be in the pending or executable state

- **VUID-vkQueueSubmit-pCommandBuffers-00073**
  If any secondary command buffers recorded into any element of the pCommandBuffers member of any element of pSubmits was not recorded with the VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT, it must not be in the pending state

- **VUID-vkQueueSubmit-pCommandBuffers-00074**
  Each element of the pCommandBuffers member of each element of pSubmits must have been allocated from a VkCommandPool that was created for the same queue family queue belongs to

- **VUID-vkQueueSubmit-pSubmits-02207**
  If any element of pSubmits->pCommandBuffers includes a Queue Family Transfer Acquire Operation, there must exist a previously submitted Queue Family Transfer Release Operation on a queue in the queue family identified by the acquire operation, with parameters matching the acquire operation as defined in the definition of such acquire operations, and which happens-before the acquire operation

- **VUID-vkQueueSubmit-pCommandBuffers-03220**
  If a command recorded into any element of pCommandBuffers was a vkCmdBeginQuery whose queryPool was created with a queryType of VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR, the profiling lock must have been held continuously on the VkDevice that queue was retrieved from, throughout recording of those command buffers

- **VUID-vkQueueSubmit-pSubmits-02808**
  Any resource created with VK_SHARING_MODE_EXCLUSIVE that is read by an operation specified by pSubmits must not be owned by any queue family other than the one which queue belongs to, at the time it is executed
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 `VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT`, there must be no element of `pSubmits` that includes an `VkProtectedSubmitInfo` structure in its `pNext` chain with `protectedSubmit` equal to `VK_TRUE`.

Valid Usage (Implicit)

- `queue` must be a valid `VkQueue` handle.
- `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>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>Any</td>
</tr>
</tbody>
</table>

Return Codes

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
The `VkSubmitInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSubmitInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t waitSemaphoreCount;
    const VkSemaphore* pWaitSemaphores;
    const VkPipelineStageFlags* pWaitDstStageMask;
    uint32_t commandBufferCount;
    const VkCommandBuffer* pCommandBuffers;
    uint32_t signalSemaphoreCount;
    const VkSemaphore* pSignalSemaphores;
} VkSubmitInfo;
```

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

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

**Valid Usage**

- VUID-VkSubmitInfo-pWaitDstStageMask-04090
  If the geometry shaders feature is not enabled, `pWaitDstStageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
If the tessellation shaders feature is not enabled, `pWaitDstStageMask` must not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`.

If the synchronization2 feature is not enabled, `pWaitDstStageMask` must not be `0`.

Each element of `pCommandBuffers` must not have been allocated with `VK_COMMAND_BUFFER_LEVEL_SECONDARY`.

Each element of `pWaitDstStageMask` must not include `VK_PIPELINE_STAGE_HOST_BIT`.

If any element of `pWaitSemaphores` or `pSignalSemaphores` was created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE`, then the `pNext` chain must include a `VkTimelineSemaphoreSubmitInfo` structure.

If the `pNext` chain of this structure includes a `VkTimelineSemaphoreSubmitInfo` structure and any element of `pWaitSemaphores` was created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE`, then its `waitSemaphoreValueCount` member must equal `waitSemaphoreCount`.

If the `pNext` chain of this structure includes a `VkTimelineSemaphoreSubmitInfo` structure and any element of `pSignalSemaphores` was created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE`, then its `signalSemaphoreValueCount` member must equal `signalSemaphoreCount`.

For each element of `pSignalSemaphores` created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` the corresponding element of `VkTimelineSemaphoreSubmitInfo::pSignalSemaphoreValues` must have a value greater than the current value of the semaphore when the semaphore signal operation is executed.

For each element of `pWaitSemaphores` created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` the corresponding element of `VkTimelineSemaphoreSubmitInfo::pWaitSemaphoreValues` must have a value which does not differ from the current value of the semaphore or the value of any outstanding semaphore wait or signal operation on that semaphore by more than `maxTimelineSemaphoreValueDifference`.

For each element of `pSignalSemaphores` created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` the corresponding element of `VkTimelineSemaphoreSubmitInfo::pSignalSemaphoreValues` must have a value which does not differ from the current value of the semaphore or the value of any outstanding semaphore wait or signal operation on that semaphore by more than `maxTimelineSemaphoreValueDifference`.
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`, `VkPerformanceQuerySubmitInfoKHR`, `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:
typedef struct VkTimelineSemaphoreSubmitInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t waitSemaphoreValueCount;
    const uint64_t* pWaitSemaphoreValues;
    uint32_t signalSemaphoreValueCount;
    const uint64_t* pSignalSemaphoreValues;
} VkTimelineSemaphoreSubmitInfo;

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• waitSemaphoreValueCount is the number of semaphore wait values specified in pWaitSemaphoreValues.
• pWaitSemaphoreValues is a pointer to an array of waitSemaphoreValueCount values for the corresponding semaphores in VkSubmitInfo::pWaitSemaphores to wait for.
• signalSemaphoreValueCount is the number of semaphore signal values specified in pSignalSemaphoreValues.
• pSignalSemaphoreValues is a pointer to an array 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.

If the semaphore in VkSubmitInfo::pWaitSemaphores or VkSubmitInfo::pSignalSemaphores corresponding to an entry in pWaitSemaphoreValues or pSignalSemaphoreValues respectively was created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_TIMELINE, and has NvSciSyncObj as the payload, the value in the pWaitSemaphoreValues or pSignalSemaphoreValues entry must be calculated by application via NvSciSync APIs.

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 \texttt{pNext} chain of \texttt{VkSubmitInfo} includes a \texttt{VkProtectedSubmitInfo} structure, then the structure indicates whether the batch is protected. The \texttt{VkProtectedSubmitInfo} structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkProtectedSubmitInfo {
    VkStructureType sType;
    const void* pNext;
    VkBool32 protectedSubmit;
} VkProtectedSubmitInfo;
```

- \texttt{protectedSubmit} specifies whether the batch is protected. If \texttt{protectedSubmit} is \texttt{VK_TRUE}, the batch is protected. If \texttt{protectedSubmit} is \texttt{VK_FALSE}, the batch is unprotected. If the \texttt{VkSubmitInfo::pNext} chain does not include this structure, the batch is unprotected.

### Valid Usage (Implicit)

- \texttt{VUID-VkProtectedSubmitInfo-sType-sType}
  \texttt{sType} must be \texttt{VK_STRUCTURE_TYPE_PROTECTED_SUBMIT_INFO}

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

The \texttt{VkDeviceGroupSubmitInfo} structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkDeviceGroupSubmitInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t waitSemaphoreCount;
    const uint32_t* pWaitSemaphoreDeviceIndices;
    uint32_t commandBufferCount;
    const uint32_t* pCommandBufferDeviceMasks;
    uint32_t signalSemaphoreCount;
    const uint32_t* pSignalSemaphoreDeviceIndices;
} VkDeviceGroupSubmitInfo;
```

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

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

```c
// Provided by VK_KHR_performance_query
typedef struct VkPerformanceQuerySubmitInfoKHR {
    VkStructureType sType;
    const void* pNext;
    uint32_t counterPassIndex;
} VkPerformanceQuerySubmitInfoKHR;
```

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

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

**Valid Usage**

- VUID-VkPerformanceQuerySubmitInfoKHR-counterPassIndex-03221
  counterPassIndex must be less than the number of counter passes required by any queries within the batch. The required number of counter passes for a performance query is obtained by calling `vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR`.

**Valid Usage (Implicit)**

- VUID-VkPerformanceQuerySubmitInfoKHR-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PERFORMANCE_QUERY_SUBMIT_INFO_KHR`.

### 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.

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

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

### 6.7. Secondary Command Buffer Execution

A secondary command buffer must not be directly submitted to a queue. Instead, secondary command buffers are recorded to execute as part of a primary command buffer with the command:

```
// Provided by VK_VERSION_1_0
void vkCmdExecuteCommands(
    VkCommandBuffer commandBuffer,
    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 \texttt{pCommandBuffers} array.
- \texttt{pCommandBuffers} is a pointer to an array of \texttt{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 \texttt{pCommandBuffers} was not recorded with the \texttt{VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT} flag, and it was recorded into any other primary command buffer which is currently in the executable or recording state, that primary command buffer becomes invalid.

### Valid Usage

- **VUID-vkCmdExecuteCommands-pCommandBuffers-00088**
  Each element of \texttt{pCommandBuffers} must have been allocated with a \texttt{level} of \texttt{VK_COMMAND_BUFFER_LEVEL_SECONDARY}

- **VUID-vkCmdExecuteCommands-pCommandBuffers-00089**
  Each element of \texttt{pCommandBuffers} must be in the pending or executable state

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

- **VUID-vkCmdExecuteCommands-pCommandBuffers-00092**
  If any element of \texttt{pCommandBuffers} was not recorded with the \texttt{VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT} flag, it must not have already been
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`.

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

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

If `vkCmdExecuteCommands` is being called within a render pass instance begun with `vkCmdBeginRenderPass`, its `contents` parameter must have been set to `VK_SUBPASS_CONTENTS_SECONDARYCOMMANDBUFFERS`.

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.

If `vkCmdExecuteCommands` is being called within a render pass instance, that render pass instance must have been begun with the `contents` parameter of `vkCmdBeginRenderPass` set to `VK_SUBPASS_CONTENTS_SECONDARYCOMMANDBUFFERS`.

If `vkCmdExecuteCommands` is being called within a render pass instance, each element of `pCommandBuffers` must have been recorded with `VkCommandBufferInheritanceInfo::subpass` set to the index of the subpass which the given command buffer will be executed in.

If `vkCmdExecuteCommands` is being called within a render pass instance, the render passes specified in the `pBeginInfo->pInheritanceInfo->renderPass` members of the `vkBeginCommandBuffer` commands used to begin recording each element of `pCommandBuffers` must be compatible with the current render pass.

If `vkCmdExecuteCommands` is being called within a render pass instance, the render passes specified in the `pBeginInfo->pInheritanceInfo->renderPass` members of the `vkBeginCommandBuffer` commands used to begin recording 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.
**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

---

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 inherited queries feature is not enabled, `commandBuffer` must not have any queries active

- VUID-vkCmdExecuteCommands-commandBuffer-00102
  
  If `commandBuffer` has a `VK_QUERY_TYPE_OCCLUSION` query active, then each element of `pCommandBuffers` must have been recorded with `VkCommandBufferInheritanceInfo::occlusionQueryEnable` set to `VK_TRUE`

- VUID-vkCmdExecuteCommands-commandBuffer-00103
  
  If `commandBuffer` has a `VK_QUERY_TYPE_OCCLUSION` query active, then each element of `pCommandBuffers` must have been recorded with `VkCommandBufferInheritanceInfo::queryFlags` having all bits set that are set for the query

- VUID-vkCmdExecuteCommands-commandBuffer-00104
  
  If `commandBuffer` has a `VK_QUERY_TYPE_PIPELINE_STATISTICS` query active, then each element of `pCommandBuffers` must have been recorded with `VkCommandBufferInheritanceInfo::pipelineStatistics` having all bits set that are set in the `VkQueryPool` the query uses

- VUID-vkCmdExecuteCommands-pCommandBuffers-00105
  
  Each element of `pCommandBuffers` must not begin any query types that are active in `commandBuffer`

- VUID-vkCmdExecuteCommands-commandBuffer-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
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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Transfer, Graphics, Compute</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 (excluding the count of each) can be set to different values on each physical device (only when set as dynamic state), and each physical device will render using its local copy of the state. Other state is shared between physical devices, such that all physical devices use the most recently set values for the state. However, when recording an action command that uses a piece of state, the most recent command that set that state must have included all physical devices that execute the action command in its current device mask.

The command buffer's device mask is orthogonal to the `pNext` chain of `VkCommandBufferBeginInfo`. Commands only execute on a physical device if the device index is set in both device masks.

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

The `VkDeviceGroupCommandBufferBeginInfo` structure is defined as:
typedef struct VkDeviceGroupCommandBufferBeginInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t deviceMask;
} VkDeviceGroupCommandBufferBeginInfo;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **deviceMask** is the initial value of the command buffer’s device mask.

The initial device mask also acts as an upper bound on the set of devices that can ever be in the device mask in the command buffer.

If this structure is not present, the initial value of a command buffer’s device mask is set to include all physical devices in the logical device when the command buffer begins recording.

### Valid Usage

- VUID-VkDeviceGroupCommandBufferBeginInfo-deviceMask-00106
  deviceMask must be a valid device mask value
- VUID-VkDeviceGroupCommandBufferBeginInfo-deviceMask-00107
  deviceMask must not be zero

### Valid Usage (Implicit)

- VUID-VkDeviceGroupCommandBufferBeginInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_DEVICE_GROUP_COMMAND_BUFFER_BEGIN_INFO

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

```c
// Provided by VK_VERSION_1_1
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
The `deviceMask` member of the `VkDeviceGroupRenderPassBeginInfo` structure passed to the command beginning the corresponding render pass instance.

### Valid Usage

- VUID-vkCmdSetDeviceMask-deviceMask-00108
  
  `deviceMask` **must** be a valid device mask value

- VUID-vkCmdSetDeviceMask-deviceMask-00109
  
  `deviceMask` **must** not be zero

- VUID-vkCmdSetDeviceMask-deviceMask-00110
  
  `deviceMask` **must** not include any set bits that were not in the `VkDeviceGroupCommandBufferBeginInfo::deviceMask` value when the command buffer began recording

- VUID-vkCmdSetDeviceMask-deviceMask-00111
  
  If `vkCmdSetDeviceMask` is called inside a render pass instance, `deviceMask` **must** not include any set bits that were not in the `VkDeviceGroupRenderPassBeginInfo::deviceMask` value when the render pass instance began recording

### Valid Usage (Implicit)

- VUID-vkCmdSetDeviceMask-commandBuffer-parameter
  
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetDeviceMask-commandBuffer-recording
  
  `commandBuffer` **must** be in the recording state

- VUID-vkCmdSetDeviceMask-commandBuffer-cmdpool
  
  The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, compute, or transfer operations

### Host Synchronization

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

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Secondary</td>
<td>Both</td>
<td>Graphics Compute Transfer</td>
</tr>
</tbody>
</table>
Chapter 7. Synchronization and Cache Control

Synchronization of access to resources is primarily the responsibility of the application in Vulkan. The order of execution of commands with respect to the host and other commands on the device has few implicit guarantees, and needs to be explicitly specified. Memory caches and other optimizations are also explicitly managed, requiring that the flow of data through the system is largely under application control.

Whilst some implicit guarantees exist between commands, five explicit synchronization mechanisms are exposed by Vulkan:

Fences
Fences can be used to communicate to the host that execution of some task on the device has completed.

Semaphores
Semaphores can be used to control resource access across multiple queues.

Events
Events provide a fine-grained synchronization primitive which can be signaled either within a command buffer or by the host, and can be waited upon within a command buffer or queried on the host.

Pipeline Barriers
Pipeline barriers also provide synchronization control within a command buffer, but at a single point, rather than with separate signal and wait operations.

Render Passes
Render passes provide a useful synchronization framework for most rendering tasks, built upon the concepts in this chapter. Many cases that would otherwise need an application to use other synchronization primitives can be expressed more efficiently as part of a render pass.

7.1. Execution and Memory Dependencies

An operation is an arbitrary amount of work to be executed on the host, a device, or an external entity such as a presentation engine. Synchronization commands introduce explicit execution dependencies, and memory dependencies between two sets of operations defined by the command’s two synchronization scopes.

The synchronization scopes define which other operations a synchronization command is able to create execution dependencies with. Any type of operation that is not in a synchronization command’s synchronization scopes will not be included in the resulting dependency. For example, for many synchronization commands, the synchronization scopes can be limited to just operations executing in specific pipeline stages, which allows other pipeline stages to be excluded from a dependency. Other scoping options are possible, depending on the particular command.
An execution dependency is a guarantee that for two sets of operations, the first set must happen-before the second set. If an operation happens-before another operation, then the first operation must complete before the second operation is initiated. More precisely:

- Let \( A \) and \( B \) be separate sets of operations.
- Let \( S \) be a synchronization command.
- Let \( A_s \) and \( B_s \) be the synchronization scopes of \( S \).
- Let \( A' \) be the intersection of sets \( A \) and \( A_s \).
- Let \( B' \) be the intersection of sets \( B \) and \( B_s \).
- Submitting \( A, S \) and \( B \) for execution, in that order, will result in execution dependency \( E \) between \( A' \) and \( B' \).
- Execution dependency \( E \) guarantees that \( A' \) happens-before \( B' \).

An execution dependency chain is a sequence of execution dependencies that form a happens-before relation between the first dependency's \( A' \) and the final dependency's \( B' \). For each consecutive pair of execution dependencies, a chain exists if the intersection of \( B_s \) in the first dependency and \( A_s \) in the second dependency is not an empty set. The formation of a single execution dependency from an execution dependency chain can be described by substituting the following in the description of execution dependencies:

- Let \( S \) be a set of synchronization commands that generate an execution dependency chain.
- Let \( A_s \) be the first synchronization scope of the first command in \( S \).
- Let \( B_s \) be the second synchronization scope of the last command in \( S \).

Execution dependencies alone are not sufficient to guarantee that values resulting from writes in one set of operations can be read from another set of operations.

Three additional types of 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 A' is made available. Any type of access that is in a memory dependency's second access scope and occurs in B' has any available writes made visible to it. Any type of operation that is not in a synchronization command's access scopes will not be included in the resulting dependency.

A memory dependency enforces availability and visibility of memory accesses and execution order between two sets of operations. Adding to the description of execution dependency chains:

- Let \( a \) be the set of memory accesses performed by A'.
- Let \( b \) be the set of memory accesses performed by B'.
- Let \( a_s \) be the first access scope of the first command in S.
- Let \( b_s \) be the second access scope of the last command in S.
- Let \( a' \) be the intersection of sets \( a \) and \( a_s \).
- Let \( b' \) be the intersection of sets \( b \) and \( b_s \).
- Submitting A, S and B for execution, in that order, will result in a memory dependency \( m \) between A' and B'.
- Memory dependency \( m \) guarantees that:
  - Memory writes in \( a' \) are made available.
  - Available memory writes, including those from \( a' \), are made visible to \( b' \).

**Note**

Execution and memory dependencies are used to solve data hazards, i.e. to ensure that read and write operations occur in a well-defined order. Write-after-read hazards can be solved with just an execution dependency, but read-after-write and write-after-write hazards need appropriate memory dependencies to be included between them. If an application does not include dependencies to solve these hazards, the results and execution orders of memory accesses are undefined.

### 7.1.1. Image Layout Transitions

Image subresources can be transitioned from one layout to another as part of a memory dependency (e.g. by using an image memory barrier). When a layout transition is specified in a memory dependency, it happens-after the availability operations in the memory dependency, and happens-before the visibility operations. Image layout transitions may perform read and write accesses on all memory bound to the image subresource range, so applications must ensure that all memory writes have been made available before a layout transition is executed. Available memory is automatically made visible to a layout transition, and writes performed by a layout transition are automatically made available.
Layout transitions always apply to a particular image subresource range, and specify both an old layout and new layout. The old layout must either be VK_IMAGE_LAYOUT_UNDEFINED, or match the current layout of the image subresource range. If the old layout matches the current layout of the image subresource range, the transition preserves the contents of that range. If the old layout is VK_IMAGE_LAYOUT_UNDEFINED, the contents of that range may be discarded.

As image layout transitions may perform read and write accesses on the memory bound to the image, if the image subresource affected by the layout transition is bound to peer memory for any device in the current device mask then the memory heap the bound memory comes from must support the VK_PEER_MEMORY_FEATURE_GENERIC_SRC_BIT and VK_PEER_MEMORY_FEATURE_GENERIC_DST_BIT capabilities as returned by vkGetDeviceGroupPeerMemoryFeatures.

Note

Applications must ensure that layout transitions happen-after all operations accessing the image with the old layout, and happen-before any operations that will access the image with the new layout. Layout transitions are potentially read/write operations, so not defining appropriate memory dependencies to guarantee this will result in a data race.

Image layout transitions interact with memory aliasing.

Layout transitions that are performed via image memory barriers execute in their entirety in submission order, relative to other image layout transitions submitted to the same queue, including those performed by render passes. In effect there is an implicit execution dependency from each such layout transition to all layout transitions previously submitted to the same queue.

The image layout of each image subresource of a depth/stencil image created with VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT is dependent on the last sample locations used to render to the image subresource as a depth/stencil attachment, thus when the image member of an image memory barrier is an image created with this flag the application can chain a VkSampleLocationsInfoEXT structure to the pNext chain of VkImageMemoryBarrier2KHR or VkImageMemoryBarrier to specify the sample locations to use during any image layout transition.

If the VkSampleLocationsInfoEXT structure does not match the sample location state last used to render to the image subresource range specified by subresourceRange, or if no VkSampleLocationsInfoEXT structure is present, then the contents of the given image subresource range becomes undefined as if oldLayout would equal VK_IMAGE_LAYOUT_UNDEFINED.

7.1.2. Pipeline Stages

The work performed by an action or synchronization command consists of multiple operations, which are performed as a sequence of logically independent steps known as pipeline stages. The exact pipeline stages executed depend on the particular command that is used, and current command buffer state when the command was recorded. Drawing commands, dispatching commands, copy commands, clear commands, and synchronization commands all execute in different sets of pipeline stages. Synchronization commands do not execute in a defined pipeline stage.
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 VkPipelineStageFlags2KHR mask, specifying stages of execution, are:

```c
// Provided by VK_KHR_synchronization2
// Flag bits for VkPipelineStageFlagBits2KHR
typedef VkFlags64 VkPipelineStageFlagBits2KHR;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_NONE_KHR = 0ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_TOP_OF_PIPE_BIT_KHR = 0x00000001ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR = 0x00000002ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR = 0x00000004ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT_KHR = 0x00000008ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR = 0x00000010ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR = 0x00000020ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR = 0x00000040ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR = 0x00000080ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR = 0x00000100ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR = 0x00000200ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR = 0x00000400ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_COMPUTE_SHADER_BIT_KHR = 0x00000800ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR = 0x00001000ULL;
```
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_BOTTOM_OF_PIPE_BIT_KHR = 0x00002000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_HOST_BIT_KHR = 0x00004000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR = 0x00008000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR = 0x00010000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_COPY_BIT_KHR = 0x100000000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR = 0x200000000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_BLIT_BIT_KHR = 0x400000000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR = 0x800000000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR = 0x1000000000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR = 0x2000000000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_PRE_RASTERIZATION_SHADERS_BIT_KHR = 0x4000000000ULL;
// Provided by VK_KHR_synchronization2 with VK_EXT_transform_feedback
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT = 0x01000000ULL;
// Provided by VK_KHR_synchronization2 with VK_EXT_conditional_rendering
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT = 0x00040000ULL;
// Provided by VK_KHR_synchronization2 with VK_NV_device_generated_commands
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_COMMAND_PREPROCESS_BIT_NV = 0x00020000ULL;
// Provided by VK_KHR_fragment_shading_rate with VK_KHR_synchronization2
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR = 0x00400000ULL;
// Provided by VK_KHR_synchronization2 with VK_NV_shading_rate_image
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV = 0x00400000ULL;
// Provided by VK_KHR_acceleration_structure with VK_KHR_synchronization2
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR = 0x02000000ULL;
// Provided by VK_KHR_ray_tracing_pipeline with VK_KHR_synchronization2
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_RAY_TRACING_SHADER_BIT_KHR = 0x00200000ULL;
// Provided by VK_KHR_synchronization2 with VK_NV_ray_tracing
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_NV = 0x00200000ULL;
static const VkPipelineStageFlagBits2KHR
VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT = 0x00800000ULL;
// Provided by VK_KHR_synchronization2 with VK_NV_mesh_shader
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_TASK_SHADER_BIT_NV =
0x00080000ULL;
// Provided by VK_KHR_synchronization2 with VK_NV_mesh_shader
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_MESH_SHADER_BIT_NV =
0x00100000ULL;

• VK_PIPELINE_STAGE_2_NONE_KHR specifies no stages of execution.
• VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR specifies the stage of the pipeline where indirect
  command parameters are consumed.
• VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR specifies the stage of the pipeline where index buffers
  are consumed.
• VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR specifies the stage of the pipeline where
  vertex buffers are consumed.
• VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR is equivalent to the logical OR of:
  ◦ VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR
  ◦ VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR
• VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT_KHR specifies the vertex shader stage.
• VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR specifies the tessellation control
  shader stage.
• VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR specifies the tessellation
  evaluation shader stage.
• VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR specifies the geometry shader stage.
• VK_PIPELINE_STAGE_2_PRE_RASTERIZATION_SHADERS_BIT_KHR is equivalent to specifying all supported
  pre-rasterization shader stages:
  ◦ VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT_KHR
  ◦ VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR
  ◦ VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR
  ◦ VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR
• VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR specifies the fragment shader stage.
• VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR specifies the stage of the pipeline where
  early fragment tests (depth and stencil tests before fragment shading) are performed. This stage
  also includes subpass load operations for framebuffer attachments with a depth/stencil format.
• VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR specifies the stage of the pipeline where late
  fragment tests (depth and stencil tests after fragment shading) are performed. This stage also
  includes subpass store operations for framebuffer attachments with a depth/stencil format.
• VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR specifies the stage of the pipeline after
  blending where the final color values are output from the pipeline. This stage also includes
subpass load and store operations and multisample resolve operations for framebuffer attachments with a color or depth/stencil format.

- **VK_PIPELINE_STAGE_2_COMPUTE_SHADER_BIT_KHR** specifies the compute shader stage.
- **VK_PIPELINE_STAGE_2_HOST_BIT_KHR** specifies a pseudo-stage indicating execution on the host of reads/writes of device memory. This stage is not invoked by any commands recorded in a command buffer.
- **VK_PIPELINE_STAGE_2_COPY_BIT_KHR** specifies the execution of all copy commands, including `vkCmdCopyQueryPoolResults`.
- **VK_PIPELINE_STAGE_2_BLIT_BIT_KHR** specifies the execution of `vkCmdBlitImage`.
- **VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR** specifies the execution of `vkCmdResolveImage`.
- **VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR** specifies the execution of clear commands, with the exception of `vkCmdClearAttachments`.
- **VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR** is equivalent to specifying all of:
  - **VK_PIPELINE_STAGE_2_COPY_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_BLIT_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR**
- **VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR** specifies the execution of all graphics pipeline stages, and is equivalent to the logical OR of:
  - **VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR**
- **VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR** specifies all operations performed by all commands supported on the queue it is used with.
- **VK_PIPELINE_STAGE_2_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR** specifies the stage of the pipeline where the fragment shading rate attachment is read to determine the fragment shading rate for portions of a rasterized primitive.
- **VK_PIPELINE_STAGE_2_TOP_OF_PIPE_BIT_KHR** is equivalent to **VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR** with `VkAccessFlags2KHR` set to 0 when specified in the second synchronization scope, but equivalent to **VK_PIPELINE_STAGE_2_NONE_KHR** in the first scope.
VK_PIPELINE_STAGE_2_BOTTOM_OF_PIPE_BIT_KHR is equivalent to VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR with VkAccessFlags2KHR set to 0 when specified in the first synchronization scope, but equivalent to VK_PIPELINE_STAGE_2_NONE_KHR in the second scope.

**Note**
The TOP and BOTTOM pipeline stages are deprecated, and applications should prefer VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR and VK_PIPELINE_STAGE_2_NONE_KHR.

**Note**
The VkPipelineStageFlags2KHR bitmask goes beyond the 31 individual bit flags allowable within a C99 enum, which is how VkPipelineStageFlagBits is defined. The first 31 values are common to both, and are interchangeable.

VkPipelineStageFlags2KHR is a bitmask type for setting a mask of zero or more VkPipelineStageFlagBits2KHR flags:

```c
// Provided by VK_KHR_synchronization2
typedef VkFlags64 VkPipelineStageFlags2KHR;
```

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,
    VK_PIPELINE_STAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR = 0x00400000,
    // Provided by VK_KHR_fragment_shading_rate
    // Provided by VK_KHR_synchronization2
    VK_PIPELINE_STAGE_NONE_KHR = 0,
} VkPipelineStageFlagBits;
```
These values all have the same meaning as the equivalently named values for VkPipelineStageFlags2KHR.

- **VK_PIPELINE_STAGE_NONE_KHR** specifies no stages of execution.
- **VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT** specifies the stage of the pipeline where VkDrawIndirect*/ VkDispatchIndirect*/ VkTraceRaysIndirect* data structures are consumed.
- **VK_PIPELINE_STAGE_VERTEX_INPUT_BIT** specifies the stage of the pipeline where vertex and index buffers are consumed.
- **VK_PIPELINE_STAGE_VERTEX_SHADER_BIT** specifies the vertex shader stage.
- **VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT** specifies the tessellation control shader stage.
- **VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT** specifies the tessellation evaluation shader stage.
- **VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT** specifies the geometry shader stage.
- **VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT** specifies the fragment shader stage.
- **VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT** specifies the stage of the pipeline where early fragment tests (depth and stencil tests before fragment shading) are performed. This stage also includes subpass load operations for framebuffer attachments with a depth/stencil format.
- **VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT** specifies the stage of the pipeline where late fragment tests (depth and stencil tests after fragment shading) are performed. This stage also includes subpass store operations for framebuffer attachments with a depth/stencil format.
- **VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT** specifies the stage of the pipeline after blending where the final color values are output from the pipeline. This stage also includes subpass load and store operations and multisample resolve operations for framebuffer attachments with a color or depth/stencil format.
- **VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT** specifies the execution of a compute shader.
- **VK_PIPELINE_STAGE_TRANSFER_BIT** specifies the following commands:
  - All copy commands, including vkCmdCopyQueryPoolResults
  - vkCmdBlitImage2KHR and vkCmdBlitImage
  - vkCmdResolveImage2KHR and vkCmdResolveImage
  - All clear commands, with the exception of vkCmdClearAttachments
- **VK_PIPELINE_STAGE_HOST_BIT** specifies a pseudo-stage indicating execution on the host of reads/writes of device memory. This stage is not invoked by any commands recorded in a command buffer.
- **VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT** specifies the execution of all graphics pipeline stages, and is equivalent to the logical OR of:
  - **VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT**
  - **VK_PIPELINE_STAGE_VERTEX_INPUT_BIT**
  - **VK_PIPELINE_STAGE_VERTEX_SHADER_BIT**
• **VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT**
• **VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT**
• **VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT**
• **VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT**
• **VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT**
• **VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT**
• **VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT**
• **VK_PIPELINE_STAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR**

• **VK_PIPELINE_STAGE_ALL_COMMANDS_BIT** specifies all operations performed by all commands supported on the queue it is used with.

• **VK_PIPELINE_STAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR** specifies the stage of the pipeline where the fragment shading rate attachment is read to determine the fragment shading rate for portions of a rasterized primitive.

• **VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT** is equivalent to **VK_PIPELINE_STAGE_ALL_COMMANDS_BIT** with **VkAccessFlags** set to 0 when specified in the second synchronization scope, but specifies no stage of execution when specified in the first scope.

• **VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT** is equivalent to **VK_PIPELINE_STAGE_ALL_COMMANDS_BIT** with **VkAccessFlags** set to 0 when specified in the first synchronization scope, but specifies no stage of execution when specified in the second scope.

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

**VkPipelineStageFlags** is a bitmask type for setting a mask of zero or more **VkPipelineStageFlagBits**.

If a synchronization command includes a source stage mask, its first **synchronization scope** only includes execution of the pipeline stages specified in that mask, and its first **access scope** only includes memory accesses performed by pipeline stages specified in that mask.

If a synchronization command includes a destination stage mask, its second **synchronization scope** only includes execution of the pipeline stages specified in that mask, and its second **access scope** only includes memory access performed by pipeline stages specified in that mask.

**Note**
Including a particular pipeline stage in the first **synchronization scope** of a command implicitly includes logically earlier pipeline stages in the synchronization scope. Similarly, the second **synchronization scope** includes logically later pipeline stages.

However, note that **access scopes** are not affected in this way - only the precise stages specified are considered part of each access scope.

Certain pipeline stages are only available on queues that support a particular set of operations. The
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_NONE_KHR</td>
<td>None required</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</td>
<td>None required</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
<td>VK_QUEUE_COMPUTE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TRANSFER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT VK_QUEUE_COMPUTE_BIT or VK_QUEUE_TRANSFER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
<td>None required</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_HOST_BIT</td>
<td>None required</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_COMMANDS_BIT</td>
<td>None required</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR</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 $A_s$ for that command.

Similarly, initiation of a logically earlier pipeline stage must not happen-after initiation of a logically later pipeline stage. Including any given stage in the destination stage mask for a particular synchronization command also implies that any logically later stages are included in $B_s$ 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_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR
- VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR
- 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_SHADING_RATE_ATTACHMENT_BIT_KHR
- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT

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

- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT

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

- VK_PIPELINE_STAGE_TRANSFER_BIT

For host operations, only one pipeline stage occurs, so no order is guaranteed:
7.1.3. Access Types

Memory in Vulkan can be accessed from within shader invocations and via some fixed-function stages of the pipeline. The access type is a function of the descriptor type used, or how a fixed-function stage accesses memory.

Some synchronization commands take sets of access types as parameters to define the access scopes of a memory dependency. If a synchronization command includes a source access mask, its first access scope only includes accesses via the access types specified in that mask. Similarly, if a synchronization command includes a destination access mask, its second access scope only includes accesses via the access types specified in that mask.

Bits which can be set in the srcAccessMask and dstAccessMask members of VkMemoryBarrier2KHR, VkImageMemoryBarrier2KHR, and VkBufferMemoryBarrier2KHR, specifying access behavior, are:

```
// Provided by VK_KHR_synchronization2
// Flag bits for VkAccessFlagBits2KHR
typedef VkFlags64 VkAccessFlagBits2KHR;
static const VkAccessFlagBits2KHR VK_ACCESS_2_NONE_KHR = 0ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT_KHR = 0x00000001ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_INDEX_READ_BIT_KHR = 0x00000002ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT_KHR = 0x00000004ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_UNIFORM_READ_BIT_KHR = 0x00000008ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT_KHR = 0x00000010ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_SHADER_READ_BIT_KHR = 0x00000020ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_SHADER_WRITE_BIT_KHR = 0x00000040ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR = 0x00000080ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT_KHR = 0x00000100ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR = 0x00000200ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT_KHR = 0x00000400ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_TRANSFER_READ_BIT_KHR = 0x00000800ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_TRANSFER_WRITE_BIT_KHR = 0x00001000ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_HOST_READ_BIT_KHR = 0x00002000ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_HOST_WRITE_BIT_KHR = 0x00004000ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_MEMORY_READ_BIT_KHR = 0x00008000ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_MEMORY_WRITE_BIT_KHR = 0x00010000ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_SHADER_SAMPLED_READ_BIT_KHR = 0x100000000ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR = 0x200000000ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT_KHR = 185
```
0x400000000ULL;
// Provided by VK_KHR_synchronization2 with VK_EXT_transform_feedback
static const VkAccessFlagBits2KHR VK_ACCESS_2_TRANSFORM_FEEDBACK_WRITE_BIT_EXT =
0x02000000ULL;
// Provided by VK_KHR_synchronization2 with VK_EXT_transform_feedback
static const VkAccessFlagBits2KHR VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT =
0x04000000ULL;
// Provided by VK_KHR_synchronization2 with VK_EXT_transform_feedback
static const VkAccessFlagBits2KHR VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT =
0x08000000ULL;
// Provided by VK_KHR_synchronization2 with VK_EXT_conditional_rendering
static const VkAccessFlagBits2KHR VK_ACCESS_2_CONDITIONAL_RENDERING_READ_BIT_EXT =
0x00100000ULL;
// Provided by VK_KHR_synchronization2 with VK_NV_device_generated_commands
static const VkAccessFlagBits2KHR VK_ACCESS_2_COMMAND_PREPROCESS_READ_BIT_NV =
0x00020000ULL;
// Provided by VK_KHR_synchronization2 with VK_NV_device_generated_commands
static const VkAccessFlagBits2KHR VK_ACCESS_2_COMMAND_PREPROCESS_WRITE_BIT_NV =
0x00040000ULL;
// Provided by VK_KHR_fragment_shading_rate with VK_KHR_synchronization2
static const VkAccessFlagBits2KHR VK_ACCESS_2_FRAGMENT_SHADING_RATE_ATTACHMENT_READ_BIT_KHR =
0x00800000ULL;
// Provided by VK_KHR_synchronization2 with VK_NV_shading_rate_image
static const VkAccessFlagBits2KHR VK_ACCESS_2_SHADING_RATE_IMAGE_READ_BIT_NV =
0x00800000ULL;
// Provided by VK_KHR_acceleration_structure with VK_KHR_synchronization2
static const VkAccessFlagBits2KHR VK_ACCESS_2_ACCELERATION_STRUCTURE_READ_BIT_KHR =
0x00200000ULL;
// Provided by VK_KHR_acceleration_structure with VK_KHR_synchronization2
static const VkAccessFlagBits2KHR VK_ACCESS_2_ACCELERATION_STRUCTURE_WRITE_BIT_KHR =
0x00400000ULL;
// Provided by VK_KHR_synchronization2 with VK_NV_ray_tracing
static const VkAccessFlagBits2KHR VK_ACCESS_2_ACCELERATION_STRUCTURE_READ_BIT_NV =
0x00200000ULL;
// Provided by VK_KHR_synchronization2 with VK_NV_ray_tracing
static const VkAccessFlagBits2KHR VK_ACCESS_2_ACCELERATION_STRUCTURE_WRITE_BIT_NV =
0x00400000ULL;
// Provided by VK_KHR_synchronization2 with VK_EXT_fragment_density_map
static const VkAccessFlagBits2KHR VK_ACCESS_2_FRAGMENT_DENSITY_MAP_READ_BIT_EXT =
0x01000000ULL;
// Provided by VK_KHR_synchronization2 with VK_EXT_blend_operation_advanced
static const VkAccessFlagBits2KHR VK_ACCESS_2_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT =
0x00080000ULL;

• VK_ACCESS_2_NONE_KHR specifies no accesses.
• VK_ACCESS_2_MEMORY_READ_BIT_KHR specifies all read accesses. It is always valid in any access
mask, and is treated as equivalent to setting all READ access flags that are valid where it is used.
• VK_ACCESS_2_MEMORY_WRITE_BIT_KHR specifies all write accesses. It is always valid in any access
mask, and is treated as equivalent to setting all WRITE access flags that are valid where it is used.
• **VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT_KHR** specifies read access to command data read from indirect buffers as part of an indirect drawing or dispatch command. Such access occurs in the **VK_PIPELINE_STAGE_2_DRAW_INDIVIDUAL_COMMAND_BIT_KHR** pipeline stage.

• **VK_ACCESS_2_INDEX_READ_BIT_KHR** specifies read access to an index buffer as part of an indexed drawing command, bound by **vkCmdBindIndexBuffer**. Such access occurs in the **VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR** pipeline stage.

• **VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT_KHR** specifies read access to a vertex buffer as part of a drawing command, bound by **vkCmdBindVertexBuffers**. Such access occurs in the **VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR** pipeline stage.

• **VK_ACCESS_2_UNIFORM_READ_BIT_KHR** specifies read access to a uniform buffer in any shader pipeline stage.

• **VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT_KHR** specifies read access to an input attachment within a render pass during fragment shading. Such access occurs in the **VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR** pipeline stage.

• **VK_ACCESS_2_SHADER_SAMPLED_READ_BIT_KHR** specifies read access to a uniform texel buffer or sampled image in any shader pipeline stage.

• **VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR** specifies read access to a storage buffer, physical storage buffer, storage texel buffer, or storage image in any shader pipeline stage.

• **VK_ACCESS_2_SHADER_READ_BIT_KHR** is equivalent to the logical OR of:
  - **VK_ACCESS_2_UNIFORM_READ_BIT_KHR**
  - **VK_ACCESS_2_SHADER_SAMPLED_READ_BIT_KHR**
  - **VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR**

• **VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT_KHR** specifies write access to a storage buffer, physical storage buffer, storage texel buffer, or storage image in any shader pipeline stage.

• **VK_ACCESS_2_SHADER_WRITE_BIT_KHR** is equivalent to **VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT_KHR**.

• **VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR** specifies read access to a color attachment, such as via blending, logic operations, or via certain subpass load operations. It does not include advanced blend operations. Such access occurs in the **VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR** pipeline stage.

• **VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT_KHR** specifies write access to a color, resolve, or depth/stencil resolve attachment during a render pass or via certain subpass load and store operations. Such access occurs in the **VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR** pipeline stage.

• **VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR** specifies read access to a depth/stencil attachment, via depth or stencil operations or via certain subpass load operations. Such access occurs in the **VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR** or **VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR** pipeline stages.

• **VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT_KHR** specifies write access to a depth/stencil attachment, via depth or stencil operations or via certain subpass load and store operations. Such access occurs in the **VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR** or **VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR** pipeline stages.
• **VK_ACCESS_2_TRANSFER_READ_BIT_KHR** specifies read access to an image or buffer in a copy operation. Such access occurs in the **VK_PIPELINE_STAGE_2_COPY_BIT_KHR**, **VK_PIPELINE_STAGE_2_BLIT_BIT_KHR**, or **VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR** pipeline stages.

• **VK_ACCESS_2_TRANSFER_WRITE_BIT_KHR** specifies write access to an image or buffer in a clear or copy operation. Such access occurs in the **VK_PIPELINE_STAGE_2_COPY_BIT_KHR**, **VK_PIPELINE_STAGE_2_BLIT_BIT_KHR**, or **VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR** pipeline stages.

• **VK_ACCESS_2_HOST_READ_BIT_KHR** specifies read access by a host operation. Accesses of this type are not performed through a resource, but directly on memory. Such access occurs in the **VK_PIPELINE_STAGE_2_HOST_BIT_KHR** pipeline stage.

• **VK_ACCESS_2_HOST_WRITE_BIT_KHR** specifies write access by a host operation. Accesses of this type are not performed through a resource, but directly on memory. Such access occurs in the **VK_PIPELINE_STAGE_2_HOST_BIT_KHR** pipeline stage.

• **VK_ACCESS_2_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT** specifies read access to color attachments, including advanced blend operations. Such access occurs in the **VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR** pipeline stage.

• **VK_ACCESS_2_FRAGMENT_SHADING_RATE_ATTACHMENT_READ_BIT_KHR** specifies read access to a fragment shading rate attachment during rasterization. Such access occurs in the **VK_PIPELINE_STAGE_2_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR** pipeline stage.

**Note**

In situations where an application wishes to select all access types for a given set of pipeline stages, **VK_ACCESS_2_MEMORY_READ_BIT_KHR** or **VK_ACCESS_2_MEMORY_WRITE_BIT_KHR** can be used. This is particularly useful when specifying stages that only have a single access type.

**Note**

The **VkAccessFlags2KHR** bitmask goes beyond the 31 individual bit flags allowable within a C99 enum, which is how **VkAccessFlagBits** is defined. The first 31 values are common to both, and are interchangeable.

**VkAccessFlags2KHR** is a bitmask type for setting a mask of zero or more **VkAccessFlagBits2KHR**:

```c
typedef VkFlags64 VkAccessFlags2KHR;
```

Bits which can be set in the **srcAccessMask** and **dstAccessMask** members of **VkSubpassDependency**, **VkSubpassDependency2**, **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,
    // ... other access flags...
};
```
These values all have the same meaning as the equivalently named values for `VkAccessFlags2KHR`.

- **VK_ACCESS_NONE_KHR** specifies no accesses.
- **VK_ACCESS_MEMORY_READ_BIT** specifies all read accesses. It is always valid in any access mask, and is treated as equivalent to setting all `READ` access flags that are valid where it is used.
- **VK_ACCESS_MEMORY_WRITE_BIT** specifies all write accesses. It is always valid in any access mask, and is treated as equivalent to setting all `WRITE` access flags that are valid where it is used.
- **VK_ACCESS_INDIRECT_COMMAND_READ_BIT** specifies read access to indirect command data read as part of an indirect drawing or dispatching command. Such access occurs in the `VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT` pipeline stage.
- **VK_ACCESS_INDEX_READ_BIT** specifies read access to an index buffer as part of an indexed drawing command, bound by `vkCmdBindIndexBuffer`. Such access occurs in the `VK_PIPELINE_STAGE_VERTEX_INPUT_BIT` pipeline stage.
- **VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT** specifies read access to a vertex buffer as part of a drawing command, bound by `vkCmdBindVertexBuffers`. Such access occurs in the `VK_PIPELINE_STAGE_VERTEX_INPUT_BIT` pipeline stage.
- **VK_ACCESS_UNIFORM_READ_BIT** specifies read access to a uniform buffer in any shader pipeline stage.
- **VK_ACCESS_INPUT_ATTACHMENT_READ_BIT** specifies read access to an input attachment within a render pass during fragment shading. Such access occurs in the `VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT` pipeline stage.
- **VK_ACCESS_SHADER_READ_BIT** specifies read access to a uniform buffer, uniform texel buffer, sampled image, storage buffer, physical storage buffer, storage texel buffer, or storage image in...
any shader pipeline stage.

- **VK_ACCESS_SHADER_WRITE_BIT** specifies write access to a storage buffer, physical storage buffer, storage texel buffer, or storage image in any shader pipeline stage.

- **VK_ACCESS_COLOR_ATTACHMENT_READ_BIT** specifies read access to a color attachment, such as via blending, logic operations, or via certain subpass load operations. It does not include advanced blend 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 subpass load and store operations. Such access occurs in the **VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT** pipeline stage.

- **VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT** specifies read access to a depth/stencil attachment, via depth or stencil operations or via certain subpass load operations. Such access occurs in the **VK_PIPELINE_STAGE_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 via certain subpass 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. Such access occurs in the **VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR** pipeline stage.

- **VK_ACCESS_TRANSFER_WRITE_BIT** specifies write access to an image or buffer in a clear or copy operation. Such access occurs in the **VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR** pipeline stage.

- **VK_ACCESS_HOST_READ_BIT** specifies read access by a host operation. Accesses of this type are not performed through a resource, but directly on memory. Such access occurs in the **VK_PIPELINE_STAGE_HOST_BIT** pipeline stage.

- **VK_ACCESS_HOST_WRITE_BIT** specifies write access by a host operation. Accesses of this type are not performed through a resource, but directly on memory. Such access occurs in the **VK_PIPELINE_STAGE_HOST_BIT** pipeline stage.

- **VK_ACCESS_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT** specifies read access to color attachments, including advanced blend operations. Such access occurs in the **VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT** pipeline stage.

- **VK_ACCESS_FRAGMENT_SHADING_RATE_ATTACHMENT_READ_BIT_KHR** specifies read access to a fragment shading rate attachment during rasterization. Such access occurs in the **VK_PIPELINE_STAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR** 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 5. Supported access types**

---

190
<table>
<thead>
<tr>
<th>Access flag</th>
<th>Supported pipeline stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_ACCESS_INDIRECT_COMMAND_READ_BIT</td>
<td>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_INDEX_READ_BIT</td>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT</td>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_UNIFORM_READ_BIT</td>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT, VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT, VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, or VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_READ_BIT</td>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT, VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT, VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, or VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT, VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT, VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, or VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_INPUT_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT, or VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT, or VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_READ_BIT</td>
<td>VK_PIPELINE_STAGE_TRANSFER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_TRANSFER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_HOST_READ_BIT</td>
<td>VK_PIPELINE_STAGE_HOST_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_HOST_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_HOST_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_READ_BIT</td>
<td>Any</td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_WRITE_BIT</td>
<td>Any</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT</td>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_FRAGMENT_SHADING_RATE_ATTACHMENT_READ_BIT_KHR</td>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR</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 \((x, y, \text{layer})\) coordinate in the first synchronization scope corresponds to a region containing all samples at the same coordinate in the second synchronization scope. In other words, it is a pixel granularity dependency. If N equals M, then a framebuffer region containing a single \((x, y, \text{layer}, \text{sample})\) 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` and `vkQueueSubmit2KHR`
commands are executed on the host, for a single queue, from first to last.

2. The order in which VkSubmitInfo structures are specified in the pSubmits parameter of vkQueueSubmit, or in which VkSubmitInfo2KHR structures are specified in the pSubmits parameter of vkQueueSubmit2KHR, from lowest index to highest.

3. The order in which command buffers are specified in the pCommandBuffers member of VkSubmitInfo or VkSubmitInfo2KHR 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.

Query commands, the order of primitives passing through the graphics pipeline and image layout transitions as part of an image memory barrier provide additional guarantees based on submission order.

Execution of pipeline stages within a given command also has a loose ordering, dependent only on a single command.

Signal operation order is a fundamental ordering in Vulkan, giving meaning to the order in which semaphore and fence signal operations occur when submitted to a single queue. The signal operation order for queue operations is determined as follows:

1. The initial order is determined by the order in which vkQueueSubmit and vkQueueSubmit2KHR commands are executed on the host, for a single queue, from first to last.

2. The order in which VkSubmitInfo structures are specified in the pSubmits parameter of vkQueueSubmit, or in which VkSubmitInfo2KHR structures are specified in the pSubmits parameter of vkQueueSubmit2KHR, from lowest index to highest.

3. The fence signal operation defined by the fence parameter of a vkQueueSubmit, or vkQueueSubmit2KHR, command is ordered after all semaphore signal operations defined by that command.

Semaphore signal operations defined by a single VkSubmitInfo, or VkSubmitInfo2KHR, 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:
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.

- **VUID-vkCreateFence-pNext-05106**
  If the `pNext` chain of `VkFenceCreateInfo` includes `VkExportFenceSciSyncInfoNV`, then `VkFenceCreateInfo::flags` must not include `VK_FENCE_CREATE_SIGNALED_BIT`.

### 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:

```c
// Provided by VK_VERSION_1_0
typedef struct VkFenceCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkFenceCreateFlags flags;
} VkFenceCreateInfo;
```

• `sType` is the type of this structure.
• `pNext` is NULL or a pointer to a structure extending this structure.
• `flags` is a bitmask of VkFenceCreateFlagBits specifying the initial state and behavior of the fence.

Valid Usage (Implicit)

• VUID-VkFenceCreateInfo-sType-sType
  `sType` must be VK_STRUCTURE_TYPE_FENCE_CREATE_INFO

• VUID-VkFenceCreateInfo-pNext-pNext
  Each `pNext` member of any structure (including this one) in the `pNext` chain must be either NULL or a pointer to a valid instance of VkExportFenceCreateInfo or VkExportFenceSciSyncInfoNV

• 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

```c
// Provided by VK_VERSION_1_0
typedef enum VkFenceCreateFlagBits {
    VK_FENCE_CREATE_SIGNALED_BIT = 0x00000001,
} VkFenceCreateFlagBits;
```

• `VK_FENCE_CREATE_SIGNALED_BIT` specifies that the fence object is created in the signaled state.
Otherwise, it is created in the unsignaled state.

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

`VkFenceCreateFlags` is a bitmask type for setting a mask of zero or more `VkFenceCreateFlagBits`.

To create a fence whose payload can be exported to external handles, add a `VkExportFenceCreateInfo` structure to the `pNext` chain of the `VkFenceCreateInfo` structure. The `VkExportFenceCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkExportFenceCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkExternalFenceHandleTypeFlags handleTypes;
} VkExportFenceCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `handleTypes` is a bitmask of `VkExternalFenceHandleTypeFlagBits` specifying one or more fence handle types the application can export from the resulting fence. The application can request multiple handle types for the same fence.

**Valid Usage**

- VUID-VkExportFenceCreateInfo-handleTypes-01446
  The bits in `handleTypes` must be supported and compatible, as reported by `VkExternalFenceProperties`

- VUID-VkExportFenceCreateInfo-pNext-05107
  If the `pNext` chain includes a `VkExportFenceSciSyncInfoNV` structure, `VkPhysicalDeviceExternalSciSyncFeaturesNV::sciSyncFence` and `VkPhysicalDeviceExternalSciSyncFeaturesNV::sciSyncExport`, or `VkPhysicalDeviceExternalSciSync2FeaturesNV::sciSyncFence` and `VkPhysicalDeviceExternalSciSync2FeaturesNV::sciSyncExport` must be enabled

**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 export a POSIX file descriptor representing the payload of a fence, call:

```c
// Provided by VK_KHR_external_fence_fd
VkResult vkGetFenceFdKHR(
    VkDevice device,
    const VkFenceGetFdInfoKHR* pGetFdInfo,
    int* pFd);
```

- `device` is the logical device that created the fence being exported.
- `pGetFdInfo` is a pointer to a `VkFenceGetFdInfoKHR` structure containing parameters of the export operation.
- `pFd` will return the file descriptor representing the fence payload.

Each call to `vkGetFenceFdKHR` must create a new file descriptor and transfer ownership of it to the application. To avoid leaking resources, the application must release ownership of the file descriptor when it is no longer needed.

**Note**

Ownership can be released in many ways. For example, the application can call `close()` on the file descriptor, or transfer ownership back to Vulkan by using the file descriptor to import a fence payload.

If `pGetFdInfo->handleType` is `VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT` and the fence is signaled at the time `vkGetFenceFdKHR` is called, `pFd` may return the value `-1` instead of a valid file descriptor.

Where supported by the operating system, the implementation must set the file descriptor to be closed automatically when an `execve` system call is made.

Exporting a file descriptor from a fence may have side effects depending on the transference of the specified handle type, as described in Importing Fence State.

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

**Valid Usage (Implicit)**

- VUID-vkGetFenceFdKHR-device-parameter
  `device` must be a valid `VkDevice` handle
- VUID-vkGetFenceFdKHR-pGetFdInfo-parameter
  `pGetFdInfo` must be a valid pointer to a valid `VkFenceGetFdInfoKHR` structure
- VUID-vkGetFenceFdKHR-pFd-parameter
  `pFd` must be a valid pointer to an `int` value
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_TOO_MANY_OBJECTS
• VK_ERROR_OUT_OF_HOST_MEMORY

The `VkFenceGetFdInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_external_fence_fd
typedef struct VkFenceGetFdInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkFence fence;
    VkExternalFenceHandleTypeFlagBits handleType;
} VkFenceGetFdInfoKHR;
```

• `sType` is the type of this structure.
• `pNext` is `NULL` or a pointer to a structure extending this structure.
• `fence` is the fence from which state will be exported.
• `handleType` is a `VkExternalFenceHandleTypeFlagBits` value specifying the type of handle requested.

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

Valid Usage

• VUID-VkFenceGetFdInfoKHR-handleType-01453
  `handleType` must have been included in `VkExportFenceCreateInfo::handleTypes` when `fence`'s current payload was created

• VUID-VkFenceGetFdInfoKHR-handleType-01454
  If `handleType` refers to a handle type with copy payload transference semantics, `fence` must be signaled, or have an associated fence signal operation pending execution

• VUID-VkFenceGetFdInfoKHR-fence-01455
  `fence` must not currently have its payload replaced by an imported payload as described below in Importing Fence Payloads unless that imported payload's handle type was included in `VkExternalFenceProperties::exportFromImportedHandleTypes` for `handleType`

• VUID-VkFenceGetFdInfoKHR-handleType-01456
  `handleType` must be defined as a POSIX file descriptor handle
Valid Usage (Implicit)

- VUID-VkFenceGetFdInfoKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_FENCE_GET_FD_INFO_KHR

- VUID-VkFenceGetFdInfoKHR-pNext-pNext
  pNext must be NULL

- VUID-VkFenceGetFdInfoKHR-fence-parameter
  fence must be a valid VkFence handle

- VUID-VkFenceGetFdInfoKHR-handleType-parameter
  handleType must be a valid VkExternalFenceHandleTypeFlagBits value

To specify additional attributes of NvSciSync handles exported from a fence, add a VkExportFenceSciSyncInfoNV structure to the pNext chain of the VkFenceCreateInfo structure. The VkExportFenceSciSyncInfoNV structure is defined as:

```c
// Provided by VK_NV_external_sci_sync, VK_NV_external_sci_sync2
typedef struct VkExportFenceSciSyncInfoNV {
    VkStructureType sType;
    const void* pNext;
    NvSciSyncAttrList pAttributes;
} VkExportFenceSciSyncInfoNV;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **pAttributes** is an opaque NvSciSyncAttrList describing the attributes of the NvSciSync object that will be exported.

If VkExportFenceCreateInfo is not present in the same pNext chain, this structure is ignored. If VkExportFenceCreateInfo is present in the pNext chain of VkFenceCreateInfo with a NvSciSync handleType, but either VkExportFenceSciSyncInfoNV is not present in the pNext chain, or if it is present but pAttributes is set to NULL, vkCreateFence will return VK_ERROR_INITIALIZATION_FAILED.

The pAttributes must be a reconciled NvSciSyncAttrList. Before exporting the NvSciSync handles, applications must use the vkGetPhysicalDeviceSciSyncAttributesNV command to get the unreconciled NvSciSyncAttrList and then use the NvSciSync API to reconcile it.

Valid Usage

- VUID-VkExportFenceSciSyncInfoNV-pAttributes-05108
  pAttributes must be a reconciled NvSciSyncAttrList
Valid Usage (Implicit)

- **VUID-VkExportFenceSciSyncInfoNV-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_EXPORT_FENCE_SCI_SYNC_INFO_NV`

To obtain the implementation-specific NvSciSync attributes in an unreconciled NvSciSyncAttrList, call:

```c
// Provided by VK_NV_external_sci_sync, VK_NV_external_sci_sync2
VkResult vkGetPhysicalDeviceSciSyncAttributesNV(
    VkPhysicalDevice physicalDevice,
    const VkSciSyncAttributesInfoNV* pSciSyncAttributesInfo,
    NvSciSyncAttrList pAttributes);
```

- `physicalDevice` is the handle to the physical device that will be used to determine the attributes.
- `pSciSyncAttributesInfo` is a pointer to a `VkSciSyncAttributesInfoNV` structure containing information about how the attributes are to be filled.
- `pAttributes` is an opaque NvSciSyncAttrList in which the implementation will set the requested attributes.

On success, `pAttributes` will contain an unreconciled NvSciSyncAttrList whose private attributes and some public attributes are filled in by the implementation. If the attributes of `physicalDevice` could not be obtained, `VK_ERROR_INITIALIZATION_FAILED` is returned.

Valid Usage

  
  If `pSciSyncAttributesInfo->primitiveType` is `VK_SCI_SYNC_PRIMITIVE_TYPE_FENCE_NV` then
  
  `VkPhysicalDeviceExternalSciSyncFeaturesNV::sciSyncFence` or
  
  `VkPhysicalDeviceExternalSciSync2FeaturesNV::sciSyncFence` **must** be enabled

  
  If `pSciSyncAttributesInfo->primitiveType` is `VK_SCI_SYNC_PRIMITIVE_TYPE_SEMAPHORE_NV` then
  
  `VkPhysicalDeviceExternalSciSyncFeaturesNV::sciSyncSemaphore` or
  
  `VkPhysicalDeviceExternalSciSync2FeaturesNV::sciSyncSemaphore2` **must** be enabled

- **VUID-vkGetPhysicalDeviceSciSyncAttributesNV-pAttributes-05111**
  
  `pAttributes` **must** be a valid NvSciSyncAttrList and **must not** be NULL

Valid Usage (Implicit)

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

  
  `pSciSyncAttributesInfo` **must** be a valid pointer to a valid `VkSciSyncAttributesInfoNV`
Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_INITIALIZATION_FAILED

The `VkSciSyncAttributesInfoNV` structure is defined as:

```
// Provided by VK_NV_external_sci_sync, VK_NV_external_sci_sync2
typedef struct VkSciSyncAttributesInfoNV {
    VkStructureType  sType;
    const void*     pNext;
    VkSciSyncClientTypeNV  clientType;
    VkSciSyncPrimitiveTypeNV  primitiveType;
} VkSciSyncAttributesInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `clientType` is the permission type of client.
- `primitiveType` is the synchronization primitive type.

NvSciSync disallows multi-signalers, therefore clients **must** specify their permission types as one of signaler, waiter or signaler_waiter. In addition, NvSciSync requires clients to specify which primitive type is to be used in synchronization, hence clients also need to provide the primitive type (VkFence or VkSemaphore) that will be used.

**Valid Usage (Implicit)**

- VUID-VkSciSyncAttributesInfoNV-sType-sType `sType` must be `VK_STRUCTURE_TYPE_SCI_SYNC_ATTRIBUTES_INFO_NV`
- VUID-VkSciSyncAttributesInfoNV-pNext-pNext `pNext` must be NULL
- VUID-VkSciSyncAttributesInfoNV-clientType-parameter `clientType` must be a valid `VkSciSyncClientTypeNV` value
- VUID-VkSciSyncAttributesInfoNV-primitiveType-parameter `primitiveType` must be a valid `VkSciSyncPrimitiveTypeNV` value

The `VkSciSyncClientTypeNV` enum is defined as:
typedef enum VkSciSyncClientTypeNV {
    VK_SCI_SYNC_CLIENT_TYPE_SIGNALER_NV = 0,
    VK_SCI_SYNC_CLIENT_TYPE_WAITER_NV = 1,
    VK_SCI_SYNC_CLIENT_TYPE_SIGNALER_WAITER_NV = 2,
} VkSciSyncClientTypeNV;

- **VK_SCI_SYNC_CLIENT_TYPE_SIGNALER_NV** specifies the permission of the client as signaler. It indicates that the client can only signal the created fence or semaphore and disallows waiting on it.

- **VK_SCI_SYNC_CLIENT_TYPE_WAITER_NV** specifies the permission of the client as waiter. It indicates that the client can only wait on the imported fence or semaphore and disallows signalling it. This type of permission is only used when the client imports NvSciSync handles, and export is not allowed.

- **VK_SCI_SYNC_CLIENT_TYPE_SIGNALER_WAITER_NV** specifies the permission of client as both signaler and waiter. It indicates that the client can signal and wait on the created fence or semaphore.

The **VkSciSyncPrimitiveTypeNV** enum is defined as:

typedef enum VkSciSyncPrimitiveTypeNV {
    VK_SCI_SYNC_PRIMITIVE_TYPE_FENCE_NV = 0,
    VK_SCI_SYNC_PRIMITIVE_TYPE_SEMAPHORE_NV = 1,
} VkSciSyncPrimitiveTypeNV;

- **VK_SCI_SYNC_PRIMITIVE_TYPE_FENCE_NV** specifies that the synchronization primitive type the client will create is a VkFence.

- **VK_SCI_SYNC_PRIMITIVE_TYPE_SEMAPHORE_NV** specifies that the synchronization primitive type the client will create is a VkSemaphore.

To export a NvSciSyncFence handle representing the payload of a fence, call:

```c
// Provided by VK_NV_external_sci_sync, VK_NV_external_sci_sync2
VkResult vkGetFenceSciSyncFenceNV(
    VkDevice      device,
    const VkFenceGetSciSyncInfoNV* pGetSciSyncHandleInfo,
    void*          pHandle);
```

- **device** is the logical device that created the fence being exported.

- **pGetSciSyncHandleInfo** is a pointer to a VkFenceGetSciSyncInfoNV structure containing parameters of the export operation.

- **pHandle** is a pointer to a NvSciSyncFence which will contain the fence payload on return.

Each call to vkGetFenceSciSyncFenceNV will duplicate the underlying NvSciSyncFence handle and
transfer the ownership of the *NvSciSyncFence* handle to the application. To avoid leaking resources, the application **must** release the ownership of the *NvSciSyncFence* handle when it is no longer needed.

### Valid Usage

- VUID-vkGetFenceSciSyncFenceNV-pGetSciSyncHandleInfo-05112
  
  `pGetSciSyncHandleInfo->handleType` **must** be `VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_FENCE_BIT_NV`

- VUID-vkGetFenceSciSyncFenceNV-sciSyncFence-05113
  
  `VkPhysicalDeviceExternalSciSyncFeaturesNV::sciSyncFence` or `VkPhysicalDeviceExternalSciSync2FeaturesNV::sciSyncFence` **must** be enabled

### Valid Usage (Implicit)

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

- VUID-vkGetFenceSciSyncFenceNV-pGetSciSyncHandleInfo-parameter
  
  `pGetSciSyncHandleInfo` **must** be a valid pointer to a valid `VkFenceGetSciSyncInfoNV` structure

- VUID-vkGetFenceSciSyncFenceNV-pHandle-parameter
  
  `pHandle` **must** be a pointer value

### Return Codes

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_INVALID_EXTERNAL_HANDLE`
- `VK_ERROR_NOT_PERMITTED_EXT`

To export a *NvSciSyncObj* handle representing the payload of a fence, call:

```c
// Provided by VK_NV_external_sci_sync, VK_NV_external_sci_sync2
VkResult vkGetFenceSciSyncObjNV(
    VkDevice device,
    const VkFenceGetSciSyncInfoNV* pGetSciSyncHandleInfo,
    void* pHandle);
```

- `device` is the logical device that created the fence being exported.
- `pGetSciSyncHandleInfo` is a pointer to a `VkFenceGetSciSyncInfoNV` structure containing
parameters of the export operation.

- `pHandle` will return the `NvSciSyncObj` handle representing the fence payload.

Each call to `vkGetFenceSciSyncObjNV` will duplicate the underlying `NvSciSyncObj` handle and transfer the ownership of the `NvSciSyncObj` handle to the application. To avoid leaking resources, the application must release the ownership of the `NvSciSyncObj` handle when it is no longer needed.

### Valid Usage

- VUID-vkGetFenceSciSyncObjNV-pGetSciSyncHandleInfo-05114
  - `pGetSciSyncHandleInfo->handleType` must be `VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_OBJ_BIT_NV`

- VUID-vkGetFenceSciSyncObjNV-sciSyncFence-05115
  - `VkPhysicalDeviceExternalSciSyncFeaturesNV::sciSyncFence` or
  - `VkPhysicalDeviceExternalSciSync2FeaturesNV::sciSyncFence` must be enabled

### Valid Usage (Implicit)

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

- VUID-vkGetFenceSciSyncObjNV-pGetSciSyncHandleInfo-parameter
  - `pGetSciSyncHandleInfo` must be a valid pointer to a valid `VkFenceGetSciSyncInfoNV` structure

- VUID-vkGetFenceSciSyncObjNV-pHandle-parameter
  - `pHandle` must be a pointer value

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_INVALID_EXTERNAL_HANDLE`
- `VK_ERROR_NOT_PERMITTED_EXT`

The `VkFenceGetSciSyncInfoNV` structure is defined as:

```c
// Provided by VK_NV_external_sci_sync, VK_NV_external_sci_sync2
typedef struct VkFenceGetSciSyncInfoNV {
    VkStructureType   sType;
    const void*       pNext;
    VkFence           fence;
    VkExternalFenceHandleTypeFlagBits handleType;
} VkFenceGetSciSyncInfoNV;
```
sType is the type of this structure.

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

fence is the fence from which state will be exported.

handleType is the type of NvSciSync handle (NvSciSyncObj or NvSciSyncFence) representing the fence payload that will be exported.

If handleType is VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_OBJ_BIT_NV, a NvSciSyncObj will be exported. If handleType is VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_FENCE_BIT_NV, a NvSciSyncFence will be exported.

Valid Usage (Implicit)

- VUID-VkFenceGetSciSyncInfoNV-sType-sType
  sType must be VK_STRUCTURE_TYPE_FENCE_GET_SCI_SYNC_INFO_NV

- VUID-VkFenceGetSciSyncInfoNV-pNext-pNext
  pNext must be NULL

- VUID-VkFenceGetSciSyncInfoNV-fence-parameter
  fence must be a valid VkFence handle

- VUID-VkFenceGetSciSyncInfoNV-handleType-parameter
  handleType must be a valid VkExternalFenceHandleTypeFlagBits value

To destroy a fence, call:

```c
void vkDestroyFence(
  VkDevice device,  // Provided by VK_VERSION_1_0
  VkFence fence,  // device,
  const VkAllocationCallbacks* pAllocator);  // fence,
```

- 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 fence is signaled.</td>
</tr>
<tr>
<td>VK_NOT_READY</td>
<td>The fence specified by fence is unsignaled.</td>
</tr>
<tr>
<td>VK_ERRORDEVICE_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 \texttt{VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations} is VK_TRUE, \texttt{vkGetFenceStatus} must not return VK_ERROR_OUT_OF_HOST_MEMORY.

### Valid Usage (Implicit)

- **VUID-vkGetFenceStatus-device-parameter**
  \texttt{device} must be a valid \texttt{VkDevice} handle
- **VUID-vkGetFenceStatus-fence-parameter**
  \texttt{fence} must be a valid \texttt{VkFence} handle
- **VUID-vkGetFenceStatus-fence-parent**
  \texttt{fence} must have been created, allocated, or retrieved from \texttt{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 \textbf{vkResetFences}(VkDevice device, uint32_t fenceCount, const \texttt{VkFence}* pFences);
```

- **device** is the logical device that owns the fences.
- **fenceCount** is the number of fences to reset.
- **pFences** is a pointer to an array of fence handles to reset.

If any member of \texttt{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 \textbf{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 fenceCount valid \texttt{VkFence} handles

- VUID-vkResetFences-fenceCount-arraylength
  fenceCount must be greater than 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 queue submission command, it defines a memory dependency on the batches that were submitted as part of that command, and defines a fence signal operation which sets the fence to the signaled state.

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

The second synchronization scope only includes the fence signal operation.
The first **access scope** includes all memory access performed by the device.

The second **access scope** is empty.

To wait for one or more fences to enter the signaled state on the host, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkWaitForFences(
    VkDevice device,
    uint32_t fenceCount,
    const VkFence* pFences,
    VkBool32 waitAll,
    uint64_t timeout);
```

- **device** is the logical device that owns the fences.
- **fenceCount** is the number of fences to wait on.
- **pFences** is a pointer to an array of **fenceCount** fence handles.
- **waitAll** is the condition that must be satisfied to successfully unblock the wait. If **waitAll** is **VK_TRUE**, then the condition is that all fences in **pFences** are signaled. Otherwise, the condition is that at least one fence in **pFences** is signaled.
- **timeout** is the timeout period in units of nanoseconds. **timeout** is adjusted to the closest value allowed by the implementation-dependent timeout accuracy, which may be substantially longer than one nanosecond, and may be longer than the requested period.

If the condition is satisfied when **vkWaitForFences** is called, then **vkWaitForFences** returns immediately. If the condition is not satisfied at the time **vkWaitForFences** is called, then **vkWaitForFences** will block and wait until the condition is satisfied or the **timeout** has expired, whichever is sooner.

If **timeout** is zero, then **vkWaitForFences** does not wait, but simply returns the current state of the fences. **VK_TIMEOUT** will be returned in this case if the condition is not satisfied, even though no actual wait was performed.

If the condition is satisfied before the **timeout** has expired, **vkWaitForFences** returns **VK_SUCCESS**. Otherwise, **vkWaitForFences** returns **VK_TIMEOUT** after the **timeout** has expired.

If device loss occurs (see **Lost Device**) before the timeout has expired, **vkWaitForFences** must return in finite time with either **VK_SUCCESS** or **VK_ERROR_DEVICE_LOST**.

**Note**

While we guarantee that **vkWaitForFences** must return in finite time, no guarantees are made that it returns immediately upon device loss. However, the client can reasonably expect that the delay will be on the order of seconds and that calling **vkWaitForFences** will not result in a permanently (or seemingly permanently) dead process.

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. Alternate Methods to Signal Fences

Besides submitting a fence to a queue as part of a queue submission command, a fence may also be signaled when a particular event occurs on a device or display.
To create a fence that will be signaled when an event occurs on a device, call:

```c
// Provided by VK_EXT_display_control
VkResult vkRegisterDeviceEventEXT(
    VkDevice device,
    const VkDeviceEventInfoEXT* pDeviceEventInfo,
    const VkAllocationCallbacks* pAllocator,
    VkFence* pFence);
```

- `device` is a logical device on which the event may occur.
- `pDeviceEventInfo` is a pointer to a `VkDeviceEventInfoEXT` structure describing the event of interest to the application.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pFence` is a pointer to a handle in which the resulting fence object is returned.

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

### Valid Usage (Implicit)

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

- `VUID-vkRegisterDeviceEventEXT-pDeviceEventInfo-parameter`  
  `pDeviceEventInfo` must be a valid pointer to a valid `VkDeviceEventInfoEXT` structure

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

- `VUID-vkRegisterDeviceEventEXT-pFence-parameter`  
  `pFence` must be a valid pointer to a `VkFence` handle

### Return Codes

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkDeviceEventInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_display_control
typedef struct VkDeviceEventInfoEXT {
    VkStructureType sType;
    const void* pNext;
};
```
VkDeviceEventTypeEXT  deviceEvent;
} VkDeviceEventInfoEXT;

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `device` is a `VkDeviceEventTypeEXT` value specifying when the fence will be signaled.

**Valid Usage (Implicit)**

- VUID-VkDeviceEventInfoEXT-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_DEVICE_EVENT_INFO_EXT`
- VUID-VkDeviceEventInfoEXT-pNext-pNext
  `pNext` must be `NULL`
- VUID-VkDeviceEventInfoEXT-deviceEvent-parameter
  `deviceEvent` must be a valid `VkDeviceEventTypeEXT` value

Possible values of `VkDeviceEventInfoEXT::device`, specifying when a fence will be signaled, are:

```c
// Provided by VK_EXT_display_control
typedef enum VkDeviceEventTypeEXT {
   VKDEVICEEVENTTYPE_DISPLAY_HOTPLUG_EXT = 0,
} VkDeviceEventTypeEXT;
```

- `VKDEVICEEVENTTYPE_DISPLAY_HOTPLUG_EXT` specifies that the fence is signaled when a display is plugged into or unplugged from the specified device. Applications can use this notification to determine when they need to re-enumerate the available displays on a device.

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

```c
// Provided by VK_EXT_display_control
VkResult vkRegisterDisplayEventEXT(
    VkDevice device,            // Logical device
    VkDisplayKHR display,       // Display on which the event may occur
    const VkDisplayEventInfoEXT* pDisplayEventInfo,    // Pointer to a VkDisplayEventInfoEXT structure describing the event of interest to the application
    const VkAllocationCallbacks* pAllocator,           // Controls host memory allocation as described in the Memory Allocation chapter.
    VkFence* pFence);
```

- `device` is a logical device associated with `display`
- `display` is the display on which the event may occur.
- `pDisplayEventInfo` is a pointer to a `VkDisplayEventInfoEXT` structure describing the event of interest to the application.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
• \texttt{pFence} is a pointer to a handle in which the resulting fence object is returned.

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

\section*{Valid Usage (Implicit)}

- VUID-vkRegisterDisplayEventEXT-device-parameter \texttt{device} \textbf{must} be a valid \texttt{VkDevice} handle
- VUID-vkRegisterDisplayEventEXT-display-parameter \texttt{display} \textbf{must} be a valid \texttt{VkDisplayKHR} handle
- VUID-vkRegisterDisplayEventEXT-pDisplayEventInfo-parameter \texttt{pDisplayEventInfo} \textbf{must} be a valid pointer to a valid \texttt{VkDisplayEventInfoEXT} structure
- VUID-vkRegisterDisplayEventEXT-pAllocator-null \texttt{pAllocator} \textbf{must} be \texttt{NULL}
- VUID-vkRegisterDisplayEventEXT-pFence-parameter \texttt{pFence} \textbf{must} be a valid pointer to a \texttt{VkFence} handle
- VUID-vkRegisterDisplayEventEXT-commonparent Both of \texttt{device}, and \texttt{display} \textbf{must} have been created, allocated, or retrieved from the same \texttt{VkPhysicalDevice}

\section*{Return Codes}

\begin{itemize}
  \item \textbf{Success}
    \begin{itemize}
      \item \texttt{VK_SUCCESS}
    \end{itemize}
  \item \textbf{Failure}
    \begin{itemize}
      \item \texttt{VK_ERROR_OUT_OF_HOST_MEMORY}
    \end{itemize}
\end{itemize}

The \texttt{VkDisplayEventInfoEXT} structure is defined as:

\begin{verbatim}
// Provided by VK_EXT_display_control
typedef struct VkDisplayEventInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkDisplayEventTypeEXT displayEvent;
} VkDisplayEventInfoEXT;
\end{verbatim}

- \texttt{sType} is the type of this structure.
- \texttt{pNext} is \texttt{NULL} or a pointer to a structure extending this structure.
- \texttt{displayEvent} is a \texttt{VkDisplayEventTypeEXT} specifying when the fence will be signaled.
Valid Usage (Implicit)

- **VUID-VkDisplayEventInfoEXT-sType-sType**: `sType` must be `VK_STRUCTURE_TYPE_DISPLAY_EVENT_INFO_EXT`
- **VUID-VkDisplayEventInfoEXT-pNext-pNext**: `pNext` must be `NULL`
- **VUID-VkDisplayEventInfoEXT-displayEvent-parameter**: `displayEvent` must be a valid `VkDisplayEventTypeEXT` value

Possible values of `VkDisplayEventInfoEXT::displayEvent`, specifying when a fence will be signaled, are:

```cpp
// Provided by VK_EXT_display_control
typedef enum VkDisplayEventTypeEXT {
    VK_DISPLAY_EVENT_TYPE_FIRST_PIXEL_OUT_EXT = 0,
} VkDisplayEventTypeEXT;
```

- `VK_DISPLAY_EVENT_TYPE_FIRST_PIXEL_OUT_EXT` specifies that the fence is signaled when the first pixel of the next display refresh cycle leaves the display engine for the display.

### 7.3.2. Importing Fence Payloads

Applications can import a fence payload into an existing fence using an external fence handle. The effects of the import operation will be either temporary or permanent, as specified by the application. If the import is temporary, the fence will be restored to its permanent state the next time that fence is passed to `vkResetFences`.

**Note**

Restoring a fence to its prior permanent payload is a distinct operation from resetting a fence payload. See `vkResetFences` for more detail.

Performing a subsequent temporary import on a fence before resetting it has no effect on this requirement; the next unsignal of the fence must still restore its last permanent state. A permanent payload import behaves as if the target fence was destroyed, and a new fence was created with the same handle but the imported payload. Because importing a fence payload temporarily or permanently detaches the existing payload from a fence, similar usage restrictions to those applied to `vkDestroyFence` are applied to any command that imports a fence payload. Which of these import types is used is referred to as the import operation's permanence. Each handle type supports either one or both types of permanence.

The implementation must perform the import operation by either referencing or copying the payload referred to by the specified external fence handle, depending on the handle’s type. The import method used is referred to as the handle type’s transference. When using handle types with reference transference, importing a payload to a fence adds the fence to the set of all fences sharing that payload. This set includes the fence from which the payload was exported. Fence signaling,
waiting, and resetting operations performed on any fence in the set must behave as if the set were a single fence. Importing a payload using handle types with copy transference creates a duplicate copy of the payload at the time of import, but makes no further reference to it. Fence signaling, waiting, and resetting operations performed on the target of copy imports must not affect any other fence or payload.

Export operations have the same transference as the specified handle type's import operations. Additionally, exporting a fence payload to a handle with copy transference has the same side effects on the source fence’s payload as executing a fence reset operation. If the fence was using a temporarily imported payload, the fence’s prior permanent payload will be restored.

Export operations have the same transference as the specified handle type's import operations. Additionally, exporting a fence payload to a handle with copy transference has the same side effects on the source fence’s payload as executing a fence reset operation. If the fence was using a temporarily imported payload, the fence’s prior permanent payload will be restored.

External synchronization allows implementations to modify an object's internal state, i.e. payload, without internal synchronization. However, for fences sharing a payload across processes, satisfying the external synchronization requirements of VkFence parameters as if all fences in the set were the same object is sometimes infeasible. Satisfying valid usage constraints on the state of a fence would similarly require impractical coordination or levels of trust between processes. Therefore, these constraints only apply to a specific fence handle, not to its payload. For distinct fence objects which share a payload:

- If multiple commands which queue a signal operation, or which unsignal a fence, are called concurrently, behavior will be as if the commands were called in an arbitrary sequential order.
- If a queue submission command is called with a fence that is sharing a payload, and the payload is already associated with another queue command that has not yet completed execution, either one or both of the commands will cause the fence to become signaled when they complete execution.
- If a fence payload is reset while it is associated with a queue command that has not yet completed execution, the payload will become unsignaled, but may become signaled again when the command completes execution.
- In the preceding cases, any of the devices associated with the fences sharing the payload may be lost, or any of the queue submission or fence reset commands may return VK_ERROR_INITIALIZATION_FAILED.

Other than these non-deterministic results, behavior is well defined. In particular:

- The implementation must not crash or enter an internally inconsistent state where future valid Vulkan commands might cause undefined results,
- Timeouts on future wait commands on fences sharing the payload must be effective.

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. Passing a fence to `vkAcquireNextImageKHR` is equivalent to temporarily importing a fence payload to that fence.

**Note**

Because the exportable handle types of an imported fence correspond to its current imported payload, and `vkAcquireNextImageKHR` behaves the same as a temporary import operation for which the source fence is opaque to the application, applications have no way of determining whether any external handle types can be exported from a fence in this state. Therefore, applications must not attempt to export handles from fences using a temporarily imported payload from `vkAcquireNextImageKHR`.

When importing a fence payload, it is the responsibility of the application to ensure the external handles meet all valid usage requirements. However, implementations must perform sufficient validation of external handles to ensure that the operation results in a valid fence which will not cause program termination, device loss, queue stalls, host thread stalls, or corruption of other resources when used as allowed according to its import parameters. If the external handle provided does not meet these requirements, the implementation must fail the fence payload import operation with the error code `VK_ERROR_INVALID_EXTERNAL_HANDLE`.

To import a fence payload from a POSIX file descriptor, call:

```c
// Provided by VK_KHR_external_fence_fd
VkResult vkImportFenceFdKHR(
    VkDevice device,
    const VkImportFenceFdInfoKHR* pImportFenceFdInfo);
```

- `device` is the logical device that created the fence.
- `pImportFenceFdInfo` is a pointer to a `VkImportFenceFdInfoKHR` structure specifying the fence and import parameters.

Importing a fence payload from a file descriptor transfers ownership of the file descriptor from the application to the Vulkan implementation. The application must not perform any operations on the file descriptor after a successful import.

Applications can import the same fence payload into multiple instances of Vulkan, into the same instance from which it was exported, and multiple times into a given Vulkan instance.

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

Valid Usage

- VUID-vkImportFenceFdKHR-fence-01463
  fence must not be associated with any queue command that has not yet completed execution on that queue

Valid Usage (Implicit)

- VUID-vkImportFenceFdKHR-device-parameter
device must be a valid VkDevice handle
- VUID-vkImportFenceFdKHR-pImportFenceFdInfo-parameter
  pImportFenceFdInfo must be a valid pointer to a valid VkImportFenceFdInfoKHR structure

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_INVALID_EXTERNAL_HANDLE

The VkImportFenceFdInfoKHR structure is defined as:

```c
// Provided by VK_KHR_external_fence_fd
typedef struct VkImportFenceFdInfoKHR {
    VkStructureType sType;  
    const void* pNext;     
    VkFence fence;         
    VkFenceImportFlags flags; 
    VkExternalFenceHandleTypeFlagBits handleType;  
    int fd;                
} VkImportFenceFdInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **fence** is the fence into which the payload will be imported.
- **flags** is a bitmask of VkFenceImportFlagBits specifying additional parameters for the fence payload import operation.
- **handleType** is a VkExternalFenceHandleTypeFlagBits value specifying the type of fd.
• *fd* is the external handle to import.

The handle types supported by *handleType* are:

<table>
<thead>
<tr>
<th>Handle Type</th>
<th>Transference</th>
<th>Permanence Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_FD_BIT</td>
<td>Reference</td>
<td>Temporary, Permanent</td>
</tr>
<tr>
<td>VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT</td>
<td>Copy</td>
<td>Temporary</td>
</tr>
</tbody>
</table>

**Valid Usage**

- VUID-VkImportFenceFdInfoKHR-handleType-01464
  *handleType* must be a value included in the *Handle Types Supported by VkImportFenceFdInfoKHR* table

- VUID-VkImportFenceFdInfoKHR-fd-01541
  *fd* must obey any requirements listed for *handleType* in *external fence handle types compatibility*

If *handleType* is VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT, the special value -1 for *fd* is treated like a valid sync file descriptor referring to an object that has already signaled. The import operation will succeed and the *VkFence* will have a temporarily imported payload as if a valid file descriptor had been provided.

**Note**

This special behavior for importing an invalid sync file descriptor allows easier interoperability with other system APIs which use the convention that an invalid sync file descriptor represents work that has already completed and does not need to be waited for. It is consistent with the option for implementations to return a -1 file descriptor when exporting a VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT from a *VkFence* which is signaled.

**Valid Usage (Implicit)**

- VUID-VkImportFenceFdInfoKHR-sType-sType
  *sType* must be VK_STRUCTURE_TYPE_IMPORT_FENCE_FD_INFO_KHR

- VUID-VkImportFenceFdInfoKHR-pNext-pNext
  *pNext* must be NULL

- VUID-VkImportFenceFdInfoKHR-fence-parameter
  *fence* must be a valid *VkFence* handle

- VUID-VkImportFenceFdInfoKHR-flags-parameter
  *flags* must be a valid combination of *VkFenceImportFlagBits* values

- VUID-VkImportFenceFdInfoKHR-handleType-parameter
**Host Synchronization**

- Host access to fence must be externally synchronized

To import a fence payload from a NvSciSyncFence handle, call:

```c
// Provided by VK_NV_external_sci_sync, VK_NV_external_sci_sync2
VkResult vkImportFenceSciSyncFenceNV(
    VkDevice device,
    const VkImportFenceSciSyncInfoNV* pImportFenceSciSyncInfo);
```

- device is the logical device that created the fence.
- pImportFenceSciSyncInfo is a pointer to a VkImportFenceSciSyncInfoNV structure containing parameters of the import operation

Importing a fence payload from NvSciSyncFence does not transfer ownership of the handle to the Vulkan implementation. Vulkan will make a copy of NvSciSyncFence when importing it. The application must release ownership using the NvSciSync API when the handle is no longer needed.

**Valid Usage**

- VUID-vkImportFenceSciSyncFenceNV-sciSyncImport-05140
  
  VkPhysicalDeviceExternalSciSyncFeaturesNV::sciSyncImport and VkPhysicalDeviceExternalSciSyncFeaturesNV::sciSyncFence, or
  
  VkPhysicalDeviceExternalSciSync2FeaturesNV::sciSyncImport and VkPhysicalDeviceExternalSciSync2FeaturesNV::sciSyncFence must be enabled

- VUID-vkImportFenceSciSyncFenceNV-fence-05141
  
  fence must not be associated with any queue command that has not yet completed execution on that queue

- VUID-vkImportFenceSciSyncFenceNV-pImportFenceSciSyncInfo-05142
  
  pImportFenceSciSyncInfo->handleType must be VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_FENCE_BIT_NV

**Valid Usage (Implicit)**

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

- VUID-vkImportFenceSciSyncFenceNV-pImportFenceSciSyncInfo-parameter
  
  pImportFenceSciSyncInfo must be a valid pointer to a valid VkImportFenceSciSyncInfoNV structure
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_INVALID_EXTERNAL_HANDLE
• VK_ERROR_NOT_PERMITTED_EXT

To import a fence payload from a NvSciSyncObj handle, call:

```
// Provided by VK_NV_external_sci_sync, VK_NV_external_sci_sync2
VkResult vkImportFenceSciSyncObjNV(
    VkDevice device,
    const VkImportFenceSciSyncInfoNV* pImportFenceSciSyncInfo);
```

• `device` is the logical device that created the fence.
• `pImportFenceSciSyncInfo` is a pointer to a `VkImportFenceSciSyncInfoNV` structure containing parameters of the import operation.

Importing a fence payload from a NvSciSyncObj does not transfer ownership of the handle to the Vulkan implementation. Vulkan will make a new reference to the NvSciSyncObj object when importing it. The application **must** release ownership using the NvSciSync API when the handle is no longer needed.

The application **must** not import the same NvSciSyncObj with signaler access permissions into multiple instances of VkFence, and **must** not import into the same instance from which it was exported.

Valid Usage

• VUID-vkImportFenceSciSyncObjNV-sciSyncImport-05143
  `VkPhysicalDeviceExternalSciSyncFeaturesNV::sciSyncImport` and
  `VkPhysicalDeviceExternalSciSyncFeaturesNV::sciSyncFence`, or
  `VkPhysicalDeviceExternalSciSync2FeaturesNV::sciSyncImport` and
  `VkPhysicalDeviceExternalSciSync2FeaturesNV::sciSyncFence` **must** be enabled

• VUID-vkImportFenceSciSyncObjNV-fence-05144
  `fence` **must** not be associated with any queue command that has not yet completed execution on that queue

• VUID-vkImportFenceSciSyncObjNV-pImportFenceSciSyncInfo-05145
  `pImportFenceSciSyncInfo->handleType` **must** be
  `VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_OBJ_BIT_NV`
Valid Usage (Implicit)

- VUID-vkImportFenceSciSyncObjNV-device-parameter
  
  **device** must be a valid **VkDevice** handle

- VUID-vkImportFenceSciSyncObjNV-pImportFenceSciSyncInfo-parameter
  
  **pImportFenceSciSyncInfo** must be a valid pointer to a valid **VkImportFenceSciSyncInfoNV** structure

Return Codes

**Success**

- **VK_SUCCESS**

**Failure**

- **VK_ERROR_INVALID_EXTERNAL_HANDLE**
- **VK_ERROR_NOT_PERMITTED_EXT**

The **VkImportFenceSciSyncInfoNV** structure is defined as:

```c
// Provided by VK_NV_external_sci_sync, VK_NV_external_sci_sync2
typedef struct VkImportFenceSciSyncInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkFence fence;
    VkExternalFenceHandleTypeFlagBits handleType;
    void* handle;
} VkImportFenceSciSyncInfoNV;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **fence** is the fence into which the state will be imported.
- **handleType** specifies the type of **handle**.
- **handle** is the external handle to import.

The handle types supported by **handleType** are:

**Table 8. Handle Types Supported by VkImportFenceSciSyncInfoNV**

<table>
<thead>
<tr>
<th>Handle Type</th>
<th>Transference</th>
<th>Permanence Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_OBJ_BIT_NV</td>
<td>Reference</td>
<td>Permanent</td>
</tr>
<tr>
<td>Handle Type</td>
<td>Transference</td>
<td>Permanence Supported</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>--------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_FENCE_BIT_NV</td>
<td>Copy</td>
<td>Temporary</td>
</tr>
</tbody>
</table>

### Valid Usage (Implicit)

- VUID-VkImportFenceSciSyncInfoNV-sType-sType
  - `sType` must be `VK_STRUCTURE_TYPE_IMPORT_FENCE_SCI_SYNC_INFO_NV`
- VUID-VkImportFenceSciSyncInfoNV-pNext-pNext
  - `pNext` must be `NULL`
- VUID-VkImportFenceSciSyncInfoNV-fence-parameter
  - `fence` must be a valid `VkFence` handle
- VUID-VkImportFenceSciSyncInfoNV-handleType-parameter
  - `handleType` must be a valid `VkExternalFenceHandleTypeFlagBits` value
- VUID-VkImportFenceSciSyncInfoNV-handle-parameter
  - `handle` must be a pointer value

### Host Synchronization

- Host access to `fence` must be externally synchronized

Bits which **can** be set in

- `VkImportFenceFdInfoKHR::flags`

specifying additional parameters of a fence import operation are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkFenceImportFlagBits {
    VK_FENCE_IMPORT_TEMPORARY_BIT = 0x00000001,
} VkFenceImportFlagBits;
```

- `VK_FENCE_IMPORT_TEMPORARY_BIT` specifies that the fence payload will be imported only temporarily, as described in Importing Fence Payloads, regardless of the permanence of `handleType`.

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

`VkFenceImportFlags` is a bitmask type for setting a mask of zero or more `VkFenceImportFlagBits`. 
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:

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

To create a semaphore, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateSemaphore(
    VkDevice device,
    const VkSemaphoreCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkSemaphore* pSemaphore);
```

- device is the logical device that creates the semaphore.
- pCreateInfo is a pointer to a VkSemaphoreCreateInfo structure containing information about how the semaphore is to be created.
- pAllocator controls host memory allocation as described in the Memory Allocation chapter.
- pSemaphore is a pointer to a handle in which the resulting semaphore object is returned.

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 the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- flags is reserved for future use.

**Valid Usage**

- VUID-VkSemaphoreCreateInfo-pNext-05118
If the pNext chain includes VkExportSemaphoreSciSyncInfoNV, it must also include
VkSemaphoreTypeCreateInfo with a VkSemaphoreTypeCreateInfo::semaphoreType of

VK_SEMAPHORE_TYPE_TIMELINE

- VUID-VkSemaphoreCreateInfo-pNext-05146
  If the pNext chain includes VkSemaphoreSciSyncCreateInfoNV, it must also include VkSemaphoreTypeCreateInfo with a VkSemaphoreTypeCreateInfo::semaphoreType of VK_SEMAPHORE_TYPE_TIMELINE

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, VkExportSemaphoreSciSyncInfoNV, VkSemaphoreSciSyncCreateInfoNV, 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

// 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:

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

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- semaphoreType is a VkSemaphoreType value specifying the type of the semaphore.
- initialValue is the initial payload value if semaphoreType is VK_SEMAPHORE_TYPE_TIMELINE.

To create a semaphore of a specific type, add a VkSemaphoreTypeCreateInfo structure to the VkSemaphoreCreateInfo::pNext chain.
If no `VkSemaphoreTypeCreateInfo` structure is included in the `pNext` chain of `VkSemaphoreCreateInfo`, then the created semaphore will have a default `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_BINARY`.

If `VkSemaphoreSciSyncCreateInfoNV` structure is included in the `pNext` chain of `VkSemaphoreTypeCreateInfo`, `initialValue` is ignored.

### 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

- VUID-VkSemaphoreTypeCreateInfo-pNext-05119
  If the `pNext` chain includes `VkExportSemaphoreSciSyncInfoNV`, `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 `initialValue` field of `VkSemaphoreTypeCreateInfo`.

To create a semaphore whose payload can be exported to external handles, add a `VkExportSemaphoreCreateInfo` structure to the `pNext` chain of the `VkSemaphoreCreateInfo` structure. The `VkExportSemaphoreCreateInfo` structure is defined as:
typedef struct VkExportSemaphoreCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkExternalSemaphoreHandleTypeFlags handleTypes;
} VkExportSemaphoreCreateInfo;

• **sType** is the type of this structure.

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

• **handleTypes** is a bitmask of **VkExternalSemaphoreHandleTypeFlagBits** specifying one or more semaphore handle types the application **can** export from the resulting semaphore. The application **can** request multiple handle types for the same semaphore.

### Valid Usage

- **VUID-VkExportSemaphoreCreateInfo-handleTypes-01124**
  The bits in **handleTypes** must be supported and compatible, as reported by **VkExternalSemaphoreProperties**

- **VUID-VkExportSemaphoreCreateInfo-pNext-05120**
  If the **pNext** chain includes a **VkExportSemaphoreSciSyncInfoNV** structure, **VkPhysicalDeviceExternalSciSyncFeaturesNV::sciSyncSemapore** and **VkPhysicalDeviceExternalSciSyncFeaturesNV::sciSyncExport** must be enabled

### 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 export a POSIX file descriptor representing the payload of a semaphore, call:

```c
// Provided by VK_KHR_external_semaphore_fd
VkResult vkGetSemaphoreFdKHR(VkDevice device, const VkSemaphoreGetFdInfoKHR* pGetFdInfo, int* pFd);
```

• **device** is the logical device that created the semaphore being exported.

• **pGetFdInfo** is a pointer to a **VkSemaphoreGetFdInfoKHR** structure containing parameters of the export operation.
• `pFd` will return the file descriptor representing the semaphore payload.

Each call to `vkGetSemaphoreFdKHR` must create a new file descriptor and transfer ownership of it to the application. To avoid leaking resources, the application must release ownership of the file descriptor when it is no longer needed.

```
Note
Ownership can be released in many ways. For example, the application can call `close()` on the file descriptor, or transfer ownership back to Vulkan by using the file descriptor to import a semaphore payload.
```

Where supported by the operating system, the implementation must set the file descriptor to be closed automatically when an `execve` system call is made.

Exporting a file descriptor from a semaphore may have side effects depending on the transference of the specified handle type, as described in Importing Semaphore State.

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

### Valid Usage (Implicit)

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

- VUID-vkGetSemaphoreFdKHR-pGetFdInfo-parameter
  
  `pGetFdInfo` must be a valid pointer to a valid `VkSemaphoreGetFdInfoKHR` structure

- VUID-vkGetSemaphoreFdKHR-pFd-parameter
  
  `pFd` must be a valid pointer to an int value

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkSemaphoreGetFdInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_external_semaphore_fd
typedef struct VkSemaphoreGetFdInfoKHR {
    VkStructureType            sType;
    const void*                 pNext;
    VkSemaphore                semaphore;
} VkSemaphoreGetFdInfoKHR;
```
sType is the type of this structure.

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

semaphore is the semaphore from which state will be exported.

handleType is a VkExternalSemaphoreHandleTypeFlagBits value specifying the type of handle requested.

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

Valid Usage

• VUID-VkSemaphoreGetFdInfoKHR-handleType-01132
  handleType must have been included in VkExportSemaphoreCreateInfo::handleTypes when semaphore's current payload was created

• VUID-VkSemaphoreGetFdInfoKHR-semaphore-01133
  semaphore must not currently have its payload replaced by an imported payload as described below in Importing Semaphore Payloads unless that imported payload's handle type was included in VkExternalSemaphoreProperties::exportFromImportedHandleTypes for handleType

• VUID-VkSemaphoreGetFdInfoKHR-handleType-01134
  If handleType refers to a handle type with copy payload transference semantics, as defined below in Importing Semaphore Payloads, there must be no queue waiting on semaphore

• VUID-VkSemaphoreGetFdInfoKHR-handleType-01135
  If handleType refers to a handle type with copy payload transference semantics, semaphore must be signaled, or have an associated semaphore signal operation pending execution

• VUID-VkSemaphoreGetFdInfoKHR-handleType-01136
  handleType must be defined as a POSIX file descriptor handle

• VUID-VkSemaphoreGetFdInfoKHR-handleType-03253
  If handleType refers to a handle type with copy payload transference semantics, semaphore must have been created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_BINARY

• VUID-VkSemaphoreGetFdInfoKHR-handleType-03254
  If handleType refers to a handle type with copy payload transference semantics, semaphore must have an associated semaphore signal operation that has been submitted for execution and any semaphore signal operations on which it depends (if any) must have also been submitted for execution

Valid Usage (Implicit)

• VUID-VkSemaphoreGetFdInfoKHR-sType-sType
sType must be VK_STRUCTURE_TYPE_SEMAPHORE_GET_FD_INFO_KHR

- VUID-VkSemaphoreGetFdInfoKHR-pNext-pNext
  pNext must be NULL

- VUID-VkSemaphoreGetFdInfoKHR-semaphore-parameter
  semaphore must be a valid VkSemaphore handle

- VUID-VkSemaphoreGetFdInfoKHR-handleType-parameter
  handleType must be a valid VkExternalSemaphoreHandleTypeFlagBits value

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

```c
// Provided by VK_NV_external_sci_sync
typedef struct VkExportSemaphoreSciSyncInfoNV {
    VkStructureType sType;
    const void* pNext;
    NvSciSyncAttrList pAttributes;
} VkExportSemaphoreSciSyncInfoNV;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- pAttributes is an opaque NvSciSyncAttrList describing the attributes of the NvSciSync object that will be exported.

If VkExportSemaphoreCreateInfo is not present in the same pNext chain, this structure is ignored. If VkExportSemaphoreCreateInfo is present in the pNext chain of VkSemaphoreCreateInfo with a NvSciSync handleType, but either VkExportSemaphoreSciSyncInfoNV is not present in the pNext chain, or if it is but pAttributes is set to NULL, vkCreateSemaphore will return VK_ERROR_INITIALIZATION_FAILED.

The pAttributes must be a reconciled NvSciSyncAttrList. Before exporting a NvSciSync handle, the application must use the vkGetPhysicalDeviceSciSyncAttributesNV command to obtain the unreconciled NvSciSyncAttrList and then use the NvSciSync API to reconcile it.

### Valid Usage

- VUID-VkExportSemaphoreSciSyncInfoNV-pAttributes-05121
  pAttributes must be a reconciled NvSciSyncAttrList

### Valid Usage (Implicit)

- VUID-VkExportSemaphoreSciSyncInfoNV-sType-sType
  sType must be VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_SCI_SYNC_INFO_NV
To export a NvSciSyncObj handle representing the payload of a semaphore, call:

```c
// Provided by VK_NV_external_sci_sync
VkResult vkGetSemaphoreSciSyncObjNV(
    VkDevice device,
    const VkSemaphoreGetSciSyncInfoNV* pGetSciSyncInfo,
    void* pHandle);
```

- `device` is the logical device that created the semaphore being exported.
- `pGetSciSyncInfo` is a pointer to a `VkSemaphoreGetSciSyncInfoNV` structure containing parameters of the export operation.
- `pHandle` will return the NvSciSyncObj representing the semaphore payload.

Each call to `vkGetSemaphoreSciSyncObjNV` will duplicate the underlying NvSciSyncObj and transfer the ownership of the NvSciSyncObj handle to the application. To avoid leaking resources, the application must release ownership of the NvSciSyncObj when it is no longer needed.

### Valid Usage
- VUID-vkGetSemaphoreSciSyncObjNV-sciSyncSemaphore-05147
  `VkPhysicalDeviceExternalSciSyncFeaturesNV::sciSyncSemaphore` must be enabled

### Valid Usage (Implicit)
- VUID-vkGetSemaphoreSciSyncObjNV-device-parameter
  `device` must be a valid `VkDevice` handle
  `pGetSciSyncInfo` must be a valid pointer to a valid `VkSemaphoreGetSciSyncInfoNV` structure
- VUID-vkGetSemaphoreSciSyncObjNV-pHandle-parameter
  `pHandle` must be a pointer value

### Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_INVALID_EXTERNAL_HANDLE
- VK_ERROR_NOT_PERMITTED_EXT

The `VkSemaphoreGetSciSyncInfoNV` structure is defined as:
typedef struct VkSemaphoreGetSciSyncInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkSemaphore semaphore;
    VkExternalSemaphoreHandleTypeFlagBits handleType;
} VkSemaphoreGetSciSyncInfoNV;

• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **semaphore** is the semaphore from which state will be exported.
• **handleType** is the type of NvSciSync handle (**NvSciSyncObj**) representing the semaphore that will be exported.

### Valid Usage

- **VUID-VkSemaphoreGetSciSyncInfoNV-handleType-05122**
  handleType must be **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SCI_SYNC_OBJ_BIT_NV**

- **VUID-VkSemaphoreGetSciSyncInfoNV-semaphore-05123**
  semaphore must have been created with a **VkSemaphoreType** of **VK_SEMAPHORE_TYPE_TIMELINE**

- **VUID-VkSemaphoreGetSciSyncInfoNV-semaphore-05129**
  semaphore must have been created with **VkExportSemaphoreSciSyncInfoNV** included in the **pNext** chain of **VkSemaphoreCreateInfo**, or previously imported by **vkImportSemaphoreSciSyncObjNV**

### Valid Usage (Implicit)

- **VUID-VkSemaphoreGetSciSyncInfoNV-sType-sType**
  sType must be **VK_STRUCTURE_TYPE_SEMAPHORE_GET_SCI_SYNC_INFO_NV**

- **VUID-VkSemaphoreGetSciSyncInfoNV-pNext-pNext**
  pNext must be **NULL**

- **VUID-VkSemaphoreGetSciSyncInfoNV-semaphore-parameter**
  semaphore must be a valid **VkSemaphore** handle

- **VUID-VkSemaphoreGetSciSyncInfoNV-handleType-parameter**
  handleType must be a valid **VkExternalSemaphoreHandleTypeFlagBits** value

The **VkSemaphoreSciSyncCreateInfoNV** structure is defined as:

```c
// Provided by VK_NV_external_sci_sync2
typedef struct VkSemaphoreSciSyncCreateInfoNV {
    VkStructureType sType;
```
const void* pNext;
VkSemaphoreSciSyncPoolNV semaphorePool;
const NvSciSyncFence* pFence;
} VkSemaphoreSciSyncCreateInfoNV;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **semaphorePool** is a `VkSemaphoreSciSyncPoolNV` handle.
- **pFence** is a pointer to a `NvSciSyncFence`.

When `VkSemaphoreSciSyncCreateInfoNV` is included in `VkSemaphoreCreateInfo`::pNext chain, the semaphore is created from the `VkSemaphoreSciSyncPoolNV` handle that represents a `NvSciSyncObj` with one or more primitives. The `VkSemaphoreSciSyncCreateInfoNV`::pFence parameter provides the information to select the corresponding primitive represented by this semaphore. When a `NvSciSyncObj` with signaler permissions is imported to `VkSemaphoreSciSyncPoolNV`, it only supports one primitive and `VkSemaphoreSciSyncCreateInfoNV`::pFence must be in the cleared state.

### Valid Usage

- **VUID-VkSemaphoreSciSyncCreateInfoNV-sciSyncSemaphore2-05148**
  The `VkPhysicalDeviceExternalSciSync2FeaturesNV`::sciSyncSemaphore2 feature must be enabled

### Valid Usage (Implicit)

- **VUID-VkSemaphoreSciSyncCreateInfoNV-sType-sType**
  sType must be `VK_STRUCTURE_TYPE_SEMAPHORE_SCI_SYNC_CREATE_INFO_NV`
- **VUID-VkSemaphoreSciSyncCreateInfoNV-semaphorePool-parameter**
  semaphorePool must be a valid `VkSemaphoreSciSyncPoolNV` handle
- **VUID-VkSemaphoreSciSyncCreateInfoNV-pFence-parameter**
  pFence must be a valid pointer to a valid `NvSciSyncFence` value

To destroy a semaphore, call:

```c
// Provided by VK_VERSION_1_0
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.

If semaphore was created with VkSemaphoreSciSyncCreateInfoNV present in the VkSemaphoreCreateInfo::pNext chain, semaphore can be destroyed immediately after all batches that refer to it are submitted. Otherwise, all submitted batches that refer to semaphore must have completed execution before it can be destroyed.

Valid Usage

- VUID-vkDestroySemaphore-semaphore-05149
  If semaphore was not created with VkSemaphoreSciSyncCreateInfoNV present in the VkSemaphoreCreateInfo::pNext chain when it was created, 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 SciSync Pools

A semaphore SciSync pool is used to represent a NvSciSyncObj with one or more primitives.

Semaphore SciSync pools are represented by VkSemaphoreSciSyncPoolNV handles:

```c
// Provided by VK_NV_external_sci_sync2
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSemaphoreSciSyncPoolNV)
```

To import a NvSciSyncObj with multiple primitives, use vkCreateSemaphoreSciSyncPoolNV to reserve a semaphore pool to map the multiple semaphores allocated by NvSciSyncObj. Then create a VkSemaphore from the semaphore pool using the index provided by the NvSciSyncFence when chaining the VkSemaphoreSciSyncCreateInfoNV structure to VkSemaphoreCreateInfo.
To create a `VkSemaphoreSciSyncPoolNV`, call:

```c
// Provided by VK_NV_external_sci_sync2
VkResult vkCreateSemaphoreSciSyncPoolNV(
    VkDevice device,
    const VkSemaphoreSciSyncPoolCreateInfoNV* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkSemaphoreSciSyncPoolNV* pSemaphorePool);
```

- `device` is the logical device that creates the semaphore pool.
- `pCreateInfo` is a pointer to a `VkSemaphoreSciSyncPoolCreateInfoNV` structure containing information about the semaphore SciSync pool being created.
- `pAllocator` controls host memory allocation as described in the `Memory Allocation` chapter.
- `pSemaphorePool` is a pointer to a handle in which the resulting semaphore pool object is returned.

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

### Valid Usage

- **VUID-vkCreateSemaphoreSciSyncPoolNV-device-05150**
  The number of semaphore pools currently allocated from `device` plus 1 must be less than or equal to the total number of semaphore pools requested via `VkDeviceSemaphoreSciSyncPoolReservationCreateInfoNV::semaphoreSciSyncPoolRequestCount` specified when `device` was created.

- **VUID-vkCreateSemaphoreSciSyncPoolNV-sciSyncSemaphore2-05151**
  The `VkPhysicalDeviceExternalSciSync2FeaturesNV::sciSyncSemaphore2` feature must be enabled.

### Valid Usage (Implicit)

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

- **VUID-vkCreateSemaphoreSciSyncPoolNV-pCreateInfo-parameter**
  `pCreateInfo` must be a valid pointer to a valid `VkSemaphoreSciSyncPoolCreateInfoNV` structure.

- **VUID-vkCreateSemaphoreSciSyncPoolNV-pAllocator-null**
  `pAllocator` must be `NULL`.

- **VUID-vkCreateSemaphoreSciSyncPoolNV-pSemaphorePool-parameter**
  `pSemaphorePool` must be a valid pointer to a `VkSemaphoreSciSyncPoolNV` handle.
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_INITIALIZATION_FAILED
• VK_ERROR_OUT_OF_HOST_MEMORY

The `VkSemaphoreSciSyncPoolCreateInfoNV` structure is defined as:

```c
// Provided by VK_NV_external_sci_sync2
typedef struct VkSemaphoreSciSyncPoolCreateInfoNV {
    VkStructureType sType;
    const void* pNext;
    NvSciSyncObj handle;
} VkSemaphoreSciSyncPoolCreateInfoNV;
```

• `sType` is the type of this structure.
• `pNext` is NULL or a pointer to a structure extending this structure.
• `handle` is an external `NvSciSyncObj` to import.

During `vkCreateSemaphoreSciSyncPoolNV`, the external `NvSciSyncObj` is imported to `VkSemaphoreSciSyncPoolNV`. The import does not transfer the ownership of the `NvSciSyncObj` to the implementation, but will increment the reference count of that object. The application must delete other references of the original `NvSciSyncObj` using `NvSciSync APIs` when it is no longer needed.

Applications must not import the same `NvSciSyncObj` with signaler access permissions to multiple instances of `VkSemaphoreSciSyncPoolNV`.

Valid Usage

• VUID-VkSemaphoreSciSyncPoolCreateInfoNV-handle-05152
  handle must a valid `NvSciSyncObj`

Valid Usage (Implicit)

• VUID-VkSemaphoreSciSyncPoolCreateInfoNV-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_SEMAPHORE_SCI_SYNC_POOL_CREATE_INFO_NV`
• VUID-VkSemaphoreSciSyncPoolCreateInfoNV-pNext-pNext
  `pNext` must be NULL

Semaphore SciSync pools cannot be freed [SCID-4]. If `VkPhysicalDeviceVulkanSC10Properties`
::deviceDestroyFreesMemory is VK_TRUE, the memory is returned to the system and the reference to the NvSciSyncObj that was imported is released when the device is destroyed.

### 7.4.2. Semaphore Signaling

When a batch is submitted to a queue via a queue submission, and it includes semaphores to be signaled, it defines a memory dependency on the batch, and defines semaphore signal operations which set the semaphores to the signaled state.

In case of semaphores created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_TIMELINE the semaphore is considered signaled with respect to the counter value set to be signaled as specified in VkTimelineSemaphoreSubmitInfo or VkSemaphoreSignalInfo.

The first synchronization scope includes every command submitted in the same batch. In the case of vkQueueSubmit2KHR, the first synchronization scope is limited to the pipeline stage specified by VkSemaphoreSubmitInfoKHR::stageMask. Semaphore signal operations that are defined by vkQueueSubmit or vkQueueSubmit2KHR 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.

### 7.4.3. Semaphore Waiting

When a batch is submitted to a queue via a queue submission, and it includes semaphores to be waited on, it defines a memory dependency between prior semaphore signal operations and the batch, and defines semaphore wait operations.

Such semaphore wait operations set the semaphores created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_BINARY to the unsignaled state. In case of semaphores created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_TIMELINE a prior semaphore signal operation defines a memory dependency with a semaphore wait operation if the value the semaphore is signaled with is greater than or equal to the value the semaphore is waited with, thus the semaphore will continue to be considered signaled with respect to the counter value waited on as specified in VkTimelineSemaphoreSubmitInfo.

The first synchronization scope includes all semaphore signal operations that operate on semaphores waited on in the same batch, and that happen-before the wait completes.

The second synchronization scope includes every command submitted in the same batch. In the case of vkQueueSubmit, the second synchronization scope is limited to operations on the pipeline stages determined by the destination stage mask specified by the corresponding element of pWaitDstStageMask. In the case of vkQueueSubmit2KHR, the second synchronization scope is limited to the pipeline stage specified by VkSemaphoreSubmitInfoKHR::stageMask. Also, in the case of either vkQueueSubmit2KHR or vkQueueSubmit, the second synchronization scope additionally includes
all commands that occur later in submission order.

The first access scope is empty.

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

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

Note
 Unlike timeline semaphores, fences or events, the act of waiting for a binary semaphore also unsignals that semaphore. Applications must ensure that between two such wait operations, the semaphore is signaled again, with execution dependencies used to ensure these occur in order. Binary semaphore waits and signals should thus occur in discrete 1:1 pairs.

Note
 A common scenario for using pWaitDstStageMask with values other than VK_PIPELINE_STAGE_ALL_COMMANDS_BIT is when synchronizing a window system presentation operation against subsequent command buffers which render the next frame. In this case, a presentation image must not be overwritten until the presentation operation completes, but other pipeline stages can execute without waiting. A mask of VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT prevents subsequent color attachment writes from executing until the semaphore signals. Some implementations may be able to execute transfer operations and/or pre-rasterization work before the semaphore is signaled.

If an image layout transition needs to be performed on a presentable image before it is used in a framebuffer, that can be performed as the first operation submitted to the queue after acquiring the image, and should not prevent other work from overlapping with the presentation operation. For example, a VkImageMemoryBarrier could use:

• srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
• srcAccessMask = 0
• dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
• dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_READ_BIT | VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT.
• oldLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR
• newLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL

Alternatively, oldLayout can be VK_IMAGE_LAYOUT_UNDEFINED, if the image's contents need not be preserved.

This barrier accomplishes a dependency chain between previous presentation operations and subsequent color attachment output operations, with the layout transition performed in between, and does not introduce a dependency between
previous work and any pre-rasterization shader stages. More precisely, the semaphore signals after the presentation operation completes, the semaphore wait stalls the `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` stage, and there is a dependency from that same stage to itself with the layout transition performed in between.

### 7.4.4. 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.

### 7.4.5. Host Operations on Semaphores

In addition to semaphore signal operations and semaphore wait operations submitted to device queues, timeline semaphores support the following host operations:

- Query the current counter value of the semaphore using the `vkGetSemaphoreCounterValue` command.
- Wait for a set of semaphores to reach particular counter values using the `vkWaitSemaphores` command.
- Signal the semaphore with a particular counter value from the host using the `vkSignalSemaphore` command.

To query the current counter value of a semaphore created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` from the host, call:

```c
// Provided by VK_VERSION_1_2
VkResult vkGetSemaphoreCounterValue(
    VkDevice device,
    VkSemaphore semaphore,
    uint64_t* pValue);
```

- **device** is the logical device that owns the semaphore.
- **semaphore** is the handle of the semaphore to query.
- **pValue** is a pointer to a 64-bit integer value in which the current counter value of the semaphore is returned.
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`.

### 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:
typedef struct VkSemaphoreWaitInfo {
    VkStructureType sType;
    const void* pNext;
    VkSemaphoreWaitFlags flags;
    uint32_t semaphoreCount;
    const VkSemaphore* pSemaphores;
    const uint64_t* pValues;
} VkSemaphoreWaitInfo;

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **flags** is a bitmask of **VkSemaphoreWaitFlagBits** specifying additional parameters for the semaphore wait operation.
- **semaphoreCount** is the number of semaphores to wait on.
- **pSemaphores** is a pointer to an array of **semaphoreCount** semaphore handles to wait on.
- **pValues** is a pointer to an array of **semaphoreCount** timeline semaphore values.

### Valid Usage

- **VUID-VkSemaphoreWaitInfo-pSemaphores-03256**
  All of the elements of **pSemaphores** must reference a semaphore that was created with a **VkSemaphoreType** of **VK_SEMAPHORE_TYPE_TIMELINE**

- **VUID-VkSemaphoreWaitInfo-pSemaphores-05124**
  If any of the semaphores in **pSemaphores** have **NvSciSyncObj** as payload, application must calculate the corresponding timeline semaphore values in **pValues** by calling **NvSciSync APIs**.

### Valid Usage (Implicit)

- **VUID-VkSemaphoreWaitInfo-sType-sType**
  **sType** must be **VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO**

- **VUID-VkSemaphoreWaitInfo-pNext-pNext**
  **pNext** must be **NULL**

- **VUID-VkSemaphoreWaitInfo-flags-parameter**
  **flags** must be a valid combination of **VkSemaphoreWaitFlagBits** values

- **VUID-VkSemaphoreWaitInfo-pSemaphores-parameter**
  **pSemaphores** must be a valid pointer to an array of **semaphoreCount** valid **VkSemaphore** handles

- **VUID-VkSemaphoreWaitInfo-pValues-parameter**
  **pValues** must be a valid pointer to an array of **semaphoreCount uint64_t** values
• VUID-VkSemaphoreWaitInfo-semaphoreCount-arraylength

semaphoreCount must be greater than 0

Bits which can be set in VkSemaphoreWaitInfo::flags, specifying additional parameters of a semaphore wait operation, are:

```c
// Provided by VK_VERSION_1_2
typedef enum VkSemaphoreWaitFlagBits {
    VK_SEMAPHORE_WAIT_ANY_BIT = 0x00000001,
} VkSemaphoreWaitFlagBits;
```

- **VK_SEMAPHORE_WAIT_ANY_BIT** specifies that the semaphore wait condition is that at least one of the semaphores in VkSemaphoreWaitInfo::pSemaphores has reached the value specified by the corresponding element of VkSemaphoreWaitInfo::pValues. If VK_SEMAPHORE_WAIT_ANY_BIT is not set, the semaphore wait condition is that all of the semaphores in VkSemaphoreWaitInfo::pSemaphores have reached the value specified by the corresponding element of VkSemaphoreWaitInfo::pValues.

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

VkSemaphoreWaitFlags is a bitmask type for setting a mask of zero or more VkSemaphoreWaitFlagBits.

To signal a semaphore created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_TIMELINE with a particular counter value, on the host, call:

```c
// Provided by VK_VERSION_1_2
VkResult vkSignalSemaphore(
    VkDevice device,
    const VkSemaphoreSignalInfo* pSignalInfo);
```

- **device** is the logical device that owns the semaphore.
- **pSignalInfo** is a pointer to a VkSemaphoreSignalInfo structure containing information about the signal operation.

When vkSignalSemaphore is executed on the host, it defines and immediately executes a semaphore signal operation which sets the timeline semaphore to the given value.

The first synchronization scope is defined by the host execution model, but includes execution of vkSignalSemaphore on the host and anything that happened-before it.

The second synchronization scope is empty.

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 the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **semaphore** is the handle of the semaphore to signal.
- **value** is the value to signal.

Valid Usage

- VUID-VkSemaphoreSignalInfo-semaphore-03257
  
  **semaphore** must have been created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE`

- VUID-VkSemaphoreSignalInfo-value-03258
  
  **value** must have a value greater than the current value of the semaphore

- VUID-VkSemaphoreSignalInfo-value-03259
  
  **value** must be less than the value of any pending semaphore signal operations

- VUID-VkSemaphoreSignalInfo-value-03260
  
  **value** must have a value which does not differ from the current value of the semaphore
or the value of any outstanding semaphore wait or signal operation on semaphore by more than maxTimelineSemaphoreValueDifference

- VUID-VkSemaphoreSignalInfo-semaphores-05125
  If semaphores has NvSciSyncObj as payload, application must calculate the corresponding timeline semaphore value in value by calling NvSciSync APIs.

## Valid Usage (Implicit)

- VUID-VkSemaphoreSignalInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_SEMAPHORE_SIGNAL_INFO

- VUID-VkSemaphoreSignalInfo-pNext-pNext
  pNext must be NULL

- VUID-VkSemaphoreSignalInfo-semaphore-parameter
  semaphore must be a valid VkSemaphore handle

### 7.4.6. Importing Semaphore Payloads

Applications can import a semaphore payload into an existing semaphore using an external semaphore handle. The effects of the import operation will be either temporary or permanent, as specified by the application. If the import is temporary, the implementation must restore the semaphore to its prior permanent state after submitting the next semaphore wait operation. Performing a subsequent temporary import on a semaphore before performing a semaphore wait has no effect on this requirement; the next wait submitted on the semaphore must still restore its last permanent state. A permanent payload import behaves as if the target semaphore was destroyed, and a new semaphore was created with the same handle but the imported payload. Because importing a semaphore payload temporarily or permanently detaches the existing payload from a semaphore, similar usage restrictions to those applied to vkDestroySemaphore are applied to any command that imports a semaphore payload. Which of these import types is used is referred to as the import operation's permanence. Each handle type supports either one or both types of permanence.

The implementation must perform the import operation by either referencing or copying the payload referred to by the specified external semaphore handle, depending on the handle's type. The import method used is referred to as the handle type's transference. When using handle types with reference transference, importing a payload to a semaphore adds the semaphore to the set of all semaphores sharing that payload. This set includes the semaphore from which the payload was exported. Semaphore signaling and waiting operations performed on any semaphore in the set must behave as if the set were a single semaphore. Importing a payload using handle types with copy transference creates a duplicate copy of the payload at the time of import, but makes no further reference to it. Semaphore signaling and waiting operations performed on the target of copy imports must not affect any other semaphore or payload.

Export operations have the same transference as the specified handle type's import operations. Additionally, exporting a semaphore payload to a handle with copy transference has the same side effects on the source semaphore's payload as executing a semaphore wait operation. If the
Semaphore was using a temporarily imported payload, the semaphore's prior permanent payload will be restored.

**Note**
The permanence and transference of handle types can be found in:

- Handle Types Supported by `VkImportSemaphoreFdInfoKHR`

*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. Passing a semaphore to `vkAcquireNextImageKHR` is equivalent to temporarily importing a semaphore payload to that semaphore.
Because the exportable handle types of an imported semaphore correspond to its current imported payload, and `vkAcquireNextImageKHR` behaves the same as a temporary import operation for which the source semaphore is opaque to the application, applications have no way of determining whether any external handle types can be exported from a semaphore in this state. Therefore, applications must not attempt to export external handles from semaphores using a temporarily imported payload from `vkAcquireNextImageKHR`.

When importing a semaphore payload, it is the responsibility of the application to ensure the external handles meet all valid usage requirements. However, implementations must perform sufficient validation of external handles to ensure that the operation results in a valid semaphore which will not cause program termination, device loss, queue stalls, or corruption of other resources when used as allowed according to its import parameters, and excepting those side effects allowed for violations of the valid semaphore state for wait operations rules. If the external handle provided does not meet these requirements, the implementation must fail the semaphore payload import operation with the error code `VK_ERROR_INVALID_EXTERNAL_HANDLE`.

In addition, when importing a semaphore payload that is not compatible with the payload type corresponding to the `VkSemaphoreType` the semaphore was created with, the implementation may fail the semaphore payload import operation with the error code `VK_ERROR_INVALID_EXTERNAL_HANDLE`.

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`.

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

```c
// Provided by VK_KHR_external_semaphore_fd
VkResult vkImportSemaphoreFdKHR(
    VkDevice device,
    const VkImportSemaphoreFdInfoKHR* pImportSemaphoreFdInfo);
```

- `device` is the logical device that created the semaphore.
- `pImportSemaphoreFdInfo` is a pointer to a `VkImportSemaphoreFdInfoKHR` structure specifying the semaphore and import parameters.

Importing a semaphore payload from a file descriptor transfers ownership of the file descriptor from the application to the Vulkan implementation. The application must not perform any operations on the file descriptor after a successful import.

Applications can import the same semaphore payload into multiple instances of Vulkan, into the
same instance from which it was exported, and multiple times into a given Vulkan instance.

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

### Valid Usage

- **semaphore must** not be associated with any queue command that has not yet completed execution on that queue

### Valid Usage (Implicit)

- **device must** be a valid `VkDevice` handle
- **pImportSemaphoreFdInfo must** be a valid pointer to a valid `VkImportSemaphoreFdInfoKHR` structure

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_INVALID_EXTERNAL_HANDLE`

The `VkImportSemaphoreFdInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_external_semaphore_fd
typedef struct VkImportSemaphoreFdInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkSemaphore semaphore;
    VkSemaphoreImportFlags flags;
    VkExternalSemaphoreHandleTypeFlagBits handleType;
    int fd;
} VkImportSemaphoreFdInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `semaphore` is the semaphore into which the payload will be imported.
• **flags** is a bitmask of `VkSemaphoreImportFlagBits` specifying additional parameters for the semaphore payload import operation.

• **handleType** is a `VkExternalSemaphoreHandleTypeFlagBits` value specifying the type of `fd`.

• **fd** is the external handle to import.

The handle types supported by **handleType** are:

Table 9. Handle Types Supported by `VkImportSemaphoreFdInfoKHR`

<table>
<thead>
<tr>
<th>Handle Type</th>
<th>Transference</th>
<th>Permanence Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT</code></td>
<td>Reference</td>
<td>Temporary, Permanent</td>
</tr>
<tr>
<td><code>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT</code></td>
<td>Copy</td>
<td>Temporary</td>
</tr>
</tbody>
</table>

**Valid Usage**

• **VUID-VkImportSemaphoreFdInfoKHR-handleType-01143**
  
  **handleType** must be a value included in the Handle Types Supported by `VkImportSemaphoreFdInfoKHR` table

• **VUID-VkImportSemaphoreFdInfoKHR-fd-01544**
  
  **fd** must obey any requirements listed for **handleType** in external semaphore handle types compatibility

• **VUID-VkImportSemaphoreFdInfoKHR-handleType-03263**
  
  If **handleType** is `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT`, the `VkSemaphoreCreateInfo::flags` field must match that of the semaphore from which `fd` was exported

• **VUID-VkImportSemaphoreFdInfoKHR-handleType-03264**
  
  If **handleType** is `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT`, the `VkSemaphoreTypeCreateInfo::semaphoreType` field must match that of the semaphore from which `fd` was exported

• **VUID-VkImportSemaphoreFdInfoKHR-flags-03323**
  
  If **flags** contains `VK_SEMAPHORE_IMPORT_TEMPORARY_BIT`, the `VkSemaphoreTypeCreateInfo::semaphoreType` field of the semaphore from which `fd` was exported must not be `VK_SEMAPHORE_TYPE_TIMELINE`

If **handleType** is `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT`, the special value -1 for `fd` is treated like a valid sync file descriptor referring to an object that has already signaled. The import operation will succeed and the `VkSemaphore` will have a temporarily imported payload as if a valid file descriptor had been provided.

**Note**

This special behavior for importing an invalid sync file descriptor allows easier interoperability with other system APIs which use the convention that an invalid sync file descriptor represents work that has already completed and does not need...
to be waited for. It is consistent with the option for implementations to return a -1 file descriptor when exporting a \texttt{VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT} from a \texttt{VkSemaphore} which is signaled.

### Valid Usage (Implicit)

- \textit{VUID-VkImportSemaphoreFdInfoKHR-sType-sType} \texttt{sType} \textbf{must} be \texttt{VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_FD_INFO_KHR}
- \textit{VUID-VkImportSemaphoreFdInfoKHR-pNext-pNext} \texttt{pNext} \textbf{must} be \texttt{NULL}
- \textit{VUID-VkImportSemaphoreFdInfoKHR-semaphore-parameter} \texttt{semaphore} \textbf{must} be a valid \texttt{VkSemaphore} handle
- \textit{VUID-VkImportSemaphoreFdInfoKHR-flags-parameter} \texttt{flags} \textbf{must} be a valid combination of \texttt{VkSemaphoreImportFlagBits} values
- \textit{VUID-VkImportSemaphoreFdInfoKHR-handleType-parameter} \texttt{handleType} \textbf{must} be a valid \texttt{VkExternalSemaphoreHandleTypeFlagBits} value

### Host Synchronization

- Host access to \texttt{semaphore} \textbf{must} be externally synchronized

To import a semaphore payload from a \texttt{NvSciSyncObj}, call:

```
// Provided by VK_NV_external_sci_sync
VkResult vkImportSemaphoreSciSyncObjNV(
    VkDevice device,
    const VkImportSemaphoreSciSyncInfoNV* pImportSemaphoreSciSyncInfo);
```

- \texttt{device} is the logical device that created the semaphore.
- \texttt{pImportSemaphoreSciSyncInfo} is a pointer to a \texttt{VkImportSemaphoreSciSyncInfoNV} structure containing parameters of the import operation

Importing a semaphore payload from \texttt{NvSciSyncObj} does not transfer ownership of the handle to the Vulkan implementation. When importing \texttt{NvSciSyncObj}, Vulkan will make a new reference to that object, the application \textbf{must} release its ownership using \texttt{NvSciSync APIs} when that ownership is no longer needed.

Application \textbf{must} not import the same \texttt{NvSciSyncObj} with signaler access permissions into multiple instances of VkSemaphore, and \textbf{must} not import into the same instance from which it was exported.
Valid Usage

• VUID-vkImportSemaphoreSciSyncObjNV-sciSyncImport-05155
  VkPhysicalDeviceExternalSciSyncFeaturesNV::sciSyncImport
  and
  VkPhysicalDeviceExternalSciSyncFeaturesNV::sciSyncSemaphore must be enabled

Valid Usage (Implicit)

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

• VUID-vkImportSemaphoreSciSyncObjNV-pImportSemaphoreSciSyncInfo-parameter
  pImportSemaphoreSciSyncInfo must be a valid pointer to a valid
  VkImportSemaphoreSciSyncInfoNV structure

Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_INVALID_EXTERNAL_HANDLE
• VK_ERROR_NOT_PERMITTED_EXT
• VK_ERROR_OUT_OF_HOST_MEMORY

The VkImportSemaphoreSciSyncInfoNV structure is defined as:

```c
// Provided by VK_NV_external_sci_sync
typedef struct VkImportSemaphoreSciSyncInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkSemaphore semaphore;
    VkExternalSemaphoreHandleTypeFlagBits handleType;
    void* handle;
} VkImportSemaphoreSciSyncInfoNV;
```

• `sType` is the type of this structure.
• `pNext` is NULL or a pointer to a structure extending this structure.
• `semaphore` is the semaphore into which the payload will be imported.
• `handleType` specifies the type of `handle`.
• `handle` is the external handle to import.

The handle types supported by `handleType` are:
### Table 10. Handle Types Supported by VkImportSemaphoreSciSyncInfoNV

<table>
<thead>
<tr>
<th>Handle Type</th>
<th>Transference</th>
<th>Permanence Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SCI_SYNC_OBJECT_BIT_NV</td>
<td>Reference</td>
<td>Permanent</td>
</tr>
</tbody>
</table>

#### Valid Usage

- VUID-VkImportSemaphoreSciSyncInfoNV-handleType-05126
  
  `handleType` must be a value included in the Handle Types Supported by VkImportSemaphoreSciSyncInfoNV table

- VUID-VkImportSemaphoreSciSyncInfoNV-semaphore-05127
  
  `semaphore` must have been created with a `VkSemaphoreType` of VK_SEMAPHORE_TYPE_TIMELINE

- VUID-VkImportSemaphoreSciSyncInfoNV-semaphore-05128
  
  `semaphore` must not be associated with any queue command that has not yet completed execution on that queue

#### Valid Usage (Implicit)

- VUID-VkImportSemaphoreSciSyncInfoNV-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_SCI_SYNC_INFO_NV`

- VUID-VkImportSemaphoreSciSyncInfoNV-pNext-pNext
  
  `pNext` must be NULL

- VUID-VkImportSemaphoreSciSyncInfoNV-semaphore-parameter
  
  `semaphore` must be a valid `VkSemaphore` handle

- VUID-VkImportSemaphoreSciSyncInfoNV-handleType-parameter
  
  `handleType` must be a valid `VkExternalSemaphoreHandleTypeFlagBits` value

- VUID-VkImportSemaphoreSciSyncInfoNV-handle-parameter
  
  `handle` must be a pointer value

#### Host Synchronization

- Host access to `semaphore` must be externally synchronized

**Bits which can be set in**

- `VkImportSemaphoreFdInfoKHR::flags`

**Specifying additional parameters of a semaphore import operation are:**

// Provided by VK_VERSION_1_1
typedef enum VkSemaphoreImportFlagBits {
    VK_SEMAPHORE_IMPORT_TEMPORARY_BIT = 0x00000001,
} VkSemaphoreImportFlagBits;

These bits have the following meanings:

- **VK_SEMAPHORE_IMPORT_TEMPORARY_BIT** specifies that the semaphore payload will be imported only temporarily, as described in Importing Semaphore Payloads, regardless of the permanence of handleType.

// Provided by VK_VERSION_1_1
typedef VkFlags VkSemaphoreImportFlags;

VkSemaphoreImportFlags is a bitmask type for setting a mask of zero or more VkSemaphoreImportFlagBits.

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:

// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkEvent)

To create an event, call:

// 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 the type of this structure.

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

• **flags** is a bitmask of **VkEventCreateFlagBits** defining additional creation parameters.

---

### Valid Usage (Implicit)

- **VUID-VkEventCreateInfo-sType-sType**
  - **sType** must be **VK_STRUCTURE_TYPE_EVENT_CREATE_INFO**

- **VUID-VkEventCreateInfo-pNext-pNext**
  - **pNext** must be **NULL**

- **VUID-VkEventCreateInfo-flags-parameter**
  - **flags** must be a valid combination of **VkEventCreateFlagBits** values

---

```cpp
// Provided by VK_VERSION_1_0
typedef enum VkEventCreateFlagBits {
    // Provided by VK_KHR_synchronization2
    VK_EVENT_CREATE_DEVICE_ONLY_BIT_KHR = 0x00000001,
} VkEventCreateFlagBits;
```

• **VK_EVENT_CREATE_DEVICE_ONLY_BIT_KHR** specifies that host event commands will not be used with this event.

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

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

To destroy an event, call:

```cpp
// Provided by VK_VERSION_1_0
void vkDestroyEvent(
    VkDevice device,       
    VkEvent event,         
    const VkAllocationCallbacks* pAllocator);
```

• **device** is the logical device that destroys the event.

• **event** is the handle of the event to destroy.

• **pAllocator** controls host memory allocation as described in the **Memory Allocation** chapter.

---

### Valid Usage

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

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:

```c
// 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 event is signaled.</td>
</tr>
<tr>
<td>VK_EVENT_RESET</td>
<td>The event specified by event 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

- VUID-vkGetEventStatus-event-03940
  - `event` must not have been created with `VK_EVENT_CREATE_DEVICE_ONLY_BIT_KHR`

### 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.

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

### Valid Usage

- VUID-vkSetEvent-event-03941
  event must not have been created with VK_EVENT_CREATE_DEVICE_ONLY_BIT_KHR

### Valid Usage (Implicit)

- VUID-vkSetEvent-device-parameter
device must be a valid VkDevice handle
- VUID-vkSetEvent-event-parameter
event must be a valid VkEvent handle
- VUID-vkSetEvent-event-parent
event must have been created, allocated, or retrieved from device

### Host Synchronization

- Host access to event must be externally synchronized

### Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

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

```c
// Provided by VK_VERSION_1_0
VkResult vkResetEvent(
    VkDevice device,
    VkEvent event);
```

- device is the logical device that owns the event.
- event is the event to reset.
When `vkResetEvent` is executed on the host, it defines an event unsignal operation which resets the event to the unsignaled state.

If `event` is already in the unsignaled state when `vkResetEvent` is executed, then `vkResetEvent` has no effect, and no event unsignal operation occurs.

### Valid Usage

- **VUID-vkResetEvent-event-03821**
  There must be an execution dependency between `vkResetEvent` and the execution of any `vkCmdWaitEvents` that includes `event` in its `pEvents` parameter

- **VUID-vkResetEvent-event-03822**
  There must be an execution dependency between `vkResetEvent` and the execution of any `vkCmdWaitEvents2KHR` that includes `event` in its `pEvents` parameter

- **VUID-vkResetEvent-event-03823**
  `event` must not have been created with `VK_EVENT_CREATE_DEVICE_ONLY_BIT_KHR`

### Valid Usage (Implicit)

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

- **VUID-vkResetEvent-event-parameter**
  `event` must be a valid `VkEvent` handle

- **VUID-vkResetEvent-event-parent**
  `event` must have been created, allocated, or retrieved from `device`

### Host Synchronization

- Host access to `event` must be externally synchronized

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The state of an event can also be updated on the device by commands inserted in command buffers.

To signal an event from a device, call:
```c
void vkCmdSetEvent2KHR(
    VkCommandBuffer commandBuffer,
    VkEvent event,
    const VkDependencyInfoKHR* pDependencyInfo);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `event` is the event that will be signaled.
- `pDependencyInfo` is a pointer to a `VkDependencyInfoKHR` structure defining the first scopes of this operation.

When `vkCmdSetEvent2KHR` is submitted to a queue, it defines the first half of memory dependencies defined by `pDependencyInfo`, as well as an event signal operation which sets the event to the signaled state. A memory dependency is defined between the event signal operation and commands that occur earlier in submission order.

The first synchronization scope and access scope are defined by the union of all the memory dependencies defined by `pDependencyInfo`, and are applied to all operations that occur earlier in submission order. Queue family ownership transfers and image layout transitions defined by `pDependencyInfo` are also included in the first scopes.

The second synchronization scope includes only the event signal operation, and any queue family ownership transfers and image layout transitions defined by `pDependencyInfo` are also included in the first scopes.

The second access scope includes only queue family ownership transfers and image layout transitions.

Future `vkCmdWaitEvents2KHR` commands rely on all values of each element in `pDependencyInfo` matching exactly with those used to signal the corresponding event. `vkCmdWaitEvents` must not be used to wait on the result of a signal operation defined by `vkCmdSetEvent2KHR`.

**Note**

The extra information provided by `vkCmdSetEvent2KHR` compared to `vkCmdSetEvent` allows implementations to more efficiently schedule the operations required to satisfy the requested dependencies. With `vkCmdSetEvent`, the full dependency information is not known until `vkCmdWaitEvents` is recorded, forcing implementations to insert the required operations at that point and not before.

If `event` is already in the signaled state when `vkCmdSetEvent2KHR` is executed on the device, then `vkCmdSetEvent2KHR` has no effect, no event signal operation occurs, and no dependency is generated.

**Valid Usage**

- VUID-vkCmdSetEvent2KHR-synchronization2-03824
  The synchronization2 feature must be enabled
• VUID-vkCmdSetEvent2KHR-dependencyFlags-03825
  The dependencyFlags member of pDependencyInfo must be 0

• VUID-vkCmdSetEvent2KHR-commandBuffer-03826
  The current device mask of commandBuffer must include exactly one physical device

• VUID-vkCmdSetEvent2KHR-srcStageMask-03827
  The srcStageMask member of any element of the pMemoryBarriers, pBufferMemoryBarriers, or pImageMemoryBarriers members of pDependencyInfo must only include pipeline stages valid for the queue family that was used to create the command pool that commandBuffer was allocated from

• VUID-vkCmdSetEvent2KHR-dstStageMask-03828
  The dstStageMask member of any element of the pMemoryBarriers, pBufferMemoryBarriers, or pImageMemoryBarriers members of pDependencyInfo must only include pipeline stages valid for the queue family that was used to create the command pool that commandBuffer was allocated from

Valid Usage (Implicit)

• VUID-vkCmdSetEvent2KHR-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

• VUID-vkCmdSetEvent2KHR-event-parameter
  event must be a valid VkEvent handle

• VUID-vkCmdSetEvent2KHR-pDependencyInfo-parameter
  pDependencyInfo must be a valid pointer to a valid VkDependencyInfoKHR structure

• VUID-vkCmdSetEvent2KHR-commandBuffer-recording
  commandBuffer must be in the recording state

• VUID-vkCmdSetEvent2KHR-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations

• VUID-vkCmdSetEvent2KHR-renderpass
  This command must only be called outside of a render pass instance

• VUID-vkCmdSetEvent2KHR-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
The `VkDependencyInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_synchronization2
typedef struct VkDependencyInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkDependencyFlags dependencyFlags;
    uint32_t memoryBarrierCount;
    const VkMemoryBarrier2KHR* pMemoryBarriers;
    uint32_t bufferMemoryBarrierCount;
    const VkBufferMemoryBarrier2KHR* pBufferMemoryBarriers;
    uint32_t imageMemoryBarrierCount;
    const VkImageMemoryBarrier2KHR* pImageMemoryBarriers;
} VkDependencyInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **dependencyFlags** is a bitmask of `VkDependencyFlagBits` specifying how execution and memory dependencies are formed.
- **memoryBarrierCount** is the length of the `pMemoryBarriers` array.
- **pMemoryBarriers** is a pointer to an array of `VkMemoryBarrier2KHR` structures defining memory dependencies between any memory accesses.
- **bufferMemoryBarrierCount** is the length of the `pBufferMemoryBarriers` array.
- **pBufferMemoryBarriers** is a pointer to an array of `VkBufferMemoryBarrier2KHR` structures defining memory dependencies between buffer ranges.
- **imageMemoryBarrierCount** is the length of the `pImageMemoryBarriers` array.
- **pImageMemoryBarriers** is a pointer to an array of `VkImageMemoryBarrier2KHR` structures defining memory dependencies between image subresources.

This structure defines a set of memory dependencies, as well as queue family transfer operations and image layout transitions.

Each member of `pMemoryBarriers`, `pBufferMemoryBarriers`, and `pImageMemoryBarriers` defines a separate memory dependency.
Valid Usage (Implicit)

- VUID-VkDependencyInfoKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_DEPENDENCY_INFO_KHR

- VUID-VkDependencyInfoKHR-pNext-pNext
  pNext must be NULL

- VUID-VkDependencyInfoKHR-dependencyFlags-parameter
  dependencyFlags must be a valid combination of VkDependencyFlagBits values

- VUID-VkDependencyInfoKHR-pMemoryBarriers-parameter
  If memoryBarrierCount is not 0, pMemoryBarriers must be a valid pointer to an array of memoryBarrierCount valid VkMemoryBarrier2KHR structures

- VUID-VkDependencyInfoKHR-pBufferMemoryBarriers-parameter
  If bufferMemoryBarrierCount is not 0, pBufferMemoryBarriers must be a valid pointer to an array of bufferMemoryBarrierCount valid VkBufferMemoryBarrier2KHR structures

- VUID-VkDependencyInfoKHR-pImageMemoryBarriers-parameter
  If imageMemoryBarrierCount is not 0, pImageMemoryBarriers must be a valid pointer to an array of imageMemoryBarrierCount valid VkImageMemoryBarrier2KHR structures

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

```c
// Provided by VK_VERSION_1_0
void vkCmdSetEvent(
    VkCommandBuffer commandBuffer,
    VkEvent event,
    VkPipelineStageFlags stageMask);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `event` is the event that will be signaled.
- `stageMask` specifies the source stage mask used to determine the first synchronization scope.

`vkCmdSetEvent` behaves identically to `vkCmdSetEvent2KHR`, except that it does not define an access scope, and must only be used with `vkCmdWaitEvents`, not `vkCmdWaitEvents2KHR`.

Valid Usage

- VUID-vkCmdSetEvent-stageMask-04090
  If the geometry shaders feature is not enabled, stageMask must not contain VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT

- VUID-vkCmdSetEvent-stageMask-04091
  If the tessellation shaders feature is not enabled, stageMask must not contain VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT or VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
If the synchronization feature is not enabled, `stageMask` must not be 0.

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.

`stageMask` must not include `VK_PIPELINE_STAGE_HOST_BIT`.

`commandBuffer`’s current device mask must include exactly one physical device.

### Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle.
- `event` must be a valid `VkEvent` handle.
- `stageMask` must be a valid combination of `VkPipelineStageFlagBits` values.
- `commandBuffer` must be in the recording state.
- The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations.
- This command must only be called outside of a render pass instance.
- Both of `commandBuffer`, and `event` must have been created, allocated, or retrieved from the same `VkDevice`.

### Host Synchronization

- Host access to `commandBuffer` must be externally synchronized.
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized.
To unsignal the event from a device, call:

```c
// Provided by VK_KHR_synchronization2
void vkCmdResetEvent2KHR(
    VkCommandBuffer commandBuffer,
    VkEvent event,
    VkPipelineStageFlags2KHR stageMask);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `event` is the event that will be unsignaled.
- `stageMask` is a `VkPipelineStageFlags2KHR` mask of pipeline stages used to determine the first synchronization scope.

When `vkCmdResetEvent2KHR` is submitted to a queue, it defines an execution dependency on commands that were submitted before it, and defines an event unsignal operation which resets the event to the unsignaled state.

The first synchronization scope includes all commands that occur earlier in submission order. The synchronization scope is limited to operations by `stageMask` or stages that are logically earlier than `stageMask`.

The second synchronization scope includes only the event unsignal operation.

If `event` is already in the unsignaled state when `vkCmdResetEvent2KHR` is executed on the device, then this command has no effect, no event unsignal operation occurs, and no execution dependency is generated.

### Valid Usage

- **VUID-vkCmdResetEvent2KHR-stageMask-03929**
  If the geometry shaders feature is not enabled, `stageMask` must not contain `VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR`.

- **VUID-vkCmdResetEvent2KHR-stageMask-03930**
  If the tessellation shaders feature is not enabled, `stageMask` must not contain `VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR` or `VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR`.

- **VUID-vkCmdResetEvent2KHR-synchronization2-03829**
  The synchronization2 feature must be enabled.
• VUID-vkCmdResetEvent2KHR-stageMask-03830
  stageMask must not include `VK_PIPELINE_STAGE_2_HOST_BIT_KHR`

• VUID-vkCmdResetEvent2KHR-event-03831
  There must be an execution dependency between `vkCmdResetEvent2KHR` and the execution of any `vkCmdWaitEvents` that includes `event` in its `pEvents` parameter

• VUID-vkCmdResetEvent2KHR-event-03832
  There must be an execution dependency between `vkCmdResetEvent2KHR` and the execution of any `vkCmdWaitEvents2KHR` that includes `event` in its `pEvents` parameter

• VUID-vkCmdResetEvent2KHR-commandBuffer-03833
  `commandBuffer`'s current device mask must include exactly one physical device

Valid Usage (Implicit)

• VUID-vkCmdResetEvent2KHR-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

• VUID-vkCmdResetEvent2KHR-event-parameter
  `event` must be a valid `VkEvent` handle

• VUID-vkCmdResetEvent2KHR-stageMask-parameter
  `stageMask` must be a valid combination of `VkPipelineStageFlagBits2KHR` values

• VUID-vkCmdResetEvent2KHR-commandBuffer-recording
  `commandBuffer` must be in the recording state

• VUID-vkCmdResetEvent2KHR-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations

• VUID-vkCmdResetEvent2KHR-renderpass
  This command must only be called outside of a render pass instance

• VUID-vkCmdResetEvent2KHR-commonparent
  Both of `commandBuffer`, and `event` must have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

• Host access to `commandBuffer` must be externally synchronized

• Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized
Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
</tr>
</tbody>
</table>

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

```c
// Provided by VK_VERSION_1_0
definition void vkCmdResetEvent(
    VkCommandBuffer commandBuffer,
    VkEvent event,
    VkPipelineStageFlags stageMask);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `event` is the event that will be unsignaled.
- `stageMask` is a bitmask of `VkPipelineStageFlagBits` specifying the source stage mask used to determine when the `event` is unsignaled.

`vkCmdResetEvent` behaves identically to `vkCmdResetEvent2KHR`.

Valid Usage

- VUID-vkCmdResetEvent-stageMask-04090
  If the `geometry shaders` feature is not enabled, `stageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`

- VUID-vkCmdResetEvent-stageMask-04091
  If the `tessellation shaders` feature is not enabled, `stageMask` must not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`

- VUID-vkCmdResetEvent-stageMask-03937
  If the `synchronization2` feature is not enabled, `stageMask` must not be 0

- VUID-vkCmdResetEvent-stageMask-06458
  Any pipeline stage included in `stageMask` must be supported by the capabilities of the queue family specified by the `queueFamilyIndex` member of the `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
There must be an execution dependency between `vkCmdResetEvent` and the execution of any `vkCmdWaitEvents2KHR` that includes `event` in its `pEvents` parameter.

`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

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
</tr>
</tbody>
</table>

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

```
// Provided by VK_KHR_synchronization2
```
void vkCmdWaitEvents2KHR(
    VkCommandBuffer commandBuffer,
    uint32_t eventCount,
    const VkEvent* pEvents,
    const VkDependencyInfoKHR* pDependencyInfos);

- commandBuffer is the command buffer into which the command is recorded.
- eventCount is the length of the pEvents array.
- pEvents is a pointer to an array of eventCount events to wait on.
- pDependencyInfos is a pointer to an array of eventCount VkDependencyInfoKHR structures, defining the second synchronization scope.

When vkCmdWaitEvents2KHR is submitted to a queue, it inserts memory dependencies according to the elements of pDependencyInfos and each corresponding element of pEvents. vkCmdWaitEvents2KHR must not be used to wait on event signal operations occurring on other queues, or signal operations executed by vkCmdSetEvent.

The first synchronization scope and access scope of each memory dependency defined by any element i of pDependencyInfos are applied to operations that occurred earlier in submission order than the last event signal operation on element i of pEvents.

Signal operations for an event at index i are only included if:

- The event was signaled by a vkCmdSetEvent2KHR command that occurred earlier in submission order with a dependencyInfo parameter exactly equal to the element of pDependencyInfos at index i; or
- The event was created without VK_EVENT_CREATE_DEVICE_ONLY_BIT_KHR, and the first synchronization scope defined by the element of pDependencyInfos at index i only includes host operations (VK_PIPELINE_STAGE_2_HOST_BIT_KHR).

The second synchronization scope and access scope of each memory dependency defined by any element i of pDependencyInfos are applied to operations that occurred later in submission order than vkCmdWaitEvents2KHR.

**Note**

vkCmdWaitEvents2KHR is used with vkCmdSetEvent2KHR to define a memory dependency between two sets of action commands, roughly in the same way as pipeline barriers, but split into two commands such that work between the two may execute unhindered.

**Note**

Applications should be careful to avoid race conditions when using events. There is no direct ordering guarantee between vkCmdSetEvent2KHR and vkCmdResetEvent2KHR, vkCmdResetEvent, or vkCmdSetEvent. Another execution dependency (e.g. a pipeline barrier or semaphore with VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR) is needed to prevent such a race.
**Valid Usage**

- **VUID-vkCmdWaitEvents2KHR-synchronization2-03836**
  The `synchronization2` feature **must** be enabled

- **VUID-vkCmdWaitEvents2KHR-pEvents-03837**
  Members of `pEvents` **must** not have been signaled by `vkCmdSetEvent`

- **VUID-vkCmdWaitEvents2KHR-pEvents-03838**
  For any element `i` of `pEvents`, if that event is signaled by `vkCmdSetEvent2KHR`, that command’s `dependencyInfo` parameter **must** be exactly equal to the `ith` element of `pDependencyInfos`

- **VUID-vkCmdWaitEvents2KHR-pEvents-03839**
  For any element `i` of `pEvents`, if barriers in the `ith` element of `pDependencyInfos` **must** include only host operations in their first `synchronization` scope

- **VUID-vkCmdWaitEvents2KHR-pEvents-03840**
  For any element `i` of `pEvents`, if barriers in the `ith` element of `pDependencyInfos` include only host operations, the `ith` element of `pEvents` **must** be signaled before `vkCmdWaitEvents2KHR` is executed

- **VUID-vkCmdWaitEvents2KHR-pEvents-03841**
  For any element `i` of `pEvents`, if barriers in the `ith` element of `pDependencyInfos` do not include host operations, the `ith` element of `pEvents` **must** be signaled by a corresponding `vkCmdSetEvent2KHR` that occurred earlier in `submission` order

- **VUID-vkCmdWaitEvents2KHR-srcStageMask-03842**
  The `srcStageMask` member of any element of the `pMemoryBarriers`, `pBufferMemoryBarriers`, or `pImageMemoryBarriers` members of `pDependencyInfos` **must** either include only pipeline stages valid for the queue family that was used to create the command pool that `commandBuffer` was allocated from, or include only `VK_PIPELINE_STAGE_2_HOST_BIT_KHR`

- **VUID-vkCmdWaitEvents2KHR-dstStageMask-03843**
  The `dstStageMask` member of any element of the `pMemoryBarriers`, `pBufferMemoryBarriers`, or `pImageMemoryBarriers` members of `pDependencyInfos` **must** only include pipeline stages valid for the queue family that was used to create the command pool that `commandBuffer` was allocated from

- **VUID-vkCmdWaitEvents2KHR-dependencyFlags-03844**
  The `dependencyFlags` member of any element of `pDependencyInfo` **must** be 0

- **VUID-vkCmdWaitEvents2KHR-pEvents-03845**
  If `pEvents` includes one or more events that will be signaled by `vkSetEvent` after `commandBuffer` has been submitted to a queue, then `vkCmdWaitEvents2KHR` **must** not be called inside a `render pass instance`

- **VUID-vkCmdWaitEvents2KHR-commandBuffer-03846**
  `commandBuffer`’s current device mask **must** include exactly one physical device
Valid Usage (Implicit)

- VUID-vkCmdWaitEvents2KHR-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdWaitEvents2KHR-pEvents-parameter
  pEvents must be a valid pointer to an array of eventCount valid VkEvent handles

- VUID-vkCmdWaitEvents2KHR-pDependencyInfos-parameter
  pDependencyInfos must be a valid pointer to an array of eventCount valid VkDependencyInfoKHR structures

- VUID-vkCmdWaitEvents2KHR-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdWaitEvents2KHR-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations

- VUID-vkCmdWaitEvents2KHR-eventCount-arraylength
  eventCount must be greater than 0

- VUID-vkCmdWaitEvents2KHR-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
</tr>
</tbody>
</table>

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

```c
// Provided by VK_VERSION_1_0
do
void vkCmdWaitEvents(
    VkCommandBuffer
    uint32_t
    const VkEvent*
    VkPipelineStageFlags
    VkPipelineStageFlags
        commandBuffer,
        eventCount,
        pEvents,
        srcStageMask,
        dstStageMask,
```
uint32_t memoryBarrierCount,
const VkMemoryBarrier* pMemoryBarriers,
uint32_t bufferMemoryBarrierCount,
const VkBufferMemoryBarrier* pBufferMemoryBarriers,
uint32_t imageMemoryBarrierCount,
const VkImageMemoryBarrier* pImageMemoryBarriers);

• commandBuffer is the command buffer into which the command is recorded.
• eventCount is the length of the pEvents array.
• pEvents is a pointer to an array of event object handles to wait on.
• srcStageMask is a bitmask of VkPipelineStageFlagBits specifying the source stage mask.
• dstStageMask is a bitmask of VkPipelineStageFlagBits specifying the destination stage mask.
• memoryBarrierCount is the length of the pMemoryBarriers array.
• pMemoryBarriers is a pointer to an array of VkMemoryBarrier structures.
• bufferMemoryBarrierCount is the length of the pBufferMemoryBarriers array.
• pBufferMemoryBarriers is a pointer to an array of VkBufferMemoryBarrier structures.
• imageMemoryBarrierCount is the length of the pImageMemoryBarriers array.
• pImageMemoryBarriers is a pointer to an array of VkImageMemoryBarrier structures.

vkCmdWaitEvents is largely similar to vkCmdWaitEvents2KHR, but can only wait on signal operations defined by vkCmdSetEvent. As vkCmdSetEvent does not define any access scopes, vkCmdWaitEvents defines the first access scope for each event signal operation in addition to its own access scopes.

Note
Since vkCmdSetEvent does not have any dependency information beyond a stage mask, implementations do not have the same opportunity to perform availability and visibility operations or image layout transitions in advance as they do with vkCmdSetEvent2KHR and vkCmdWaitEvents2KHR.

When vkCmdWaitEvents is submitted to a queue, it defines a memory dependency between prior event signal operations on the same queue or the host, and subsequent commands. vkCmdWaitEvents must not be used to wait on event signal operations occurring on other queues.

The first synchronization scope only includes event signal operations that operate on members of pEvents, and the operations that happened-before the event signal operations. Event signal operations performed by vkCmdSetEvent that occur earlier in submission order are included in the first synchronization scope, if the logically latest pipeline stage in their stageMask parameter is logically earlier than or equal to the logically latest pipeline stage in srcStageMask. Event signal operations performed by vkSetEvent are only included in the first synchronization scope if VK_PIPELINE_STAGE_HOST_BIT is included in srcStageMask.

The second synchronization scope includes all commands that occur later in submission order. The second synchronization scope is limited to operations on the pipeline stages determined by the destination stage mask specified by dstStageMask.
The first access scope is limited to accesses in the pipeline stages determined by the source stage mask specified by srcStageMask. Within that, the first access scope only includes the first access scopes defined by elements of the pMemoryBarriers, pBufferMemoryBarriers and pImageMemoryBarriers arrays, which each define a set of memory barriers. If no memory barriers are specified, then the first access scope includes no accesses.

The second access scope is limited to accesses in the pipeline stages determined by the destination stage mask specified by dstStageMask. Within that, the second access scope only includes the second access scopes defined by elements of the pMemoryBarriers, pBufferMemoryBarriers and pImageMemoryBarriers arrays, which each define a set of memory barriers. If no memory barriers are specified, then the second access scope includes no accesses.

**Valid Usage**

- VUID-vkCmdWaitEvents-srcStageMask-04090
  If the geometry shaders feature is not enabled, srcStageMask must not contain VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT

- VUID-vkCmdWaitEvents-srcStageMask-04091
  If the tessellation shaders feature is not enabled, srcStageMask must not contain VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT or VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT

- VUID-vkCmdWaitEvents-srcStageMask-03937
  If the synchronization2 feature is not enabled, srcStageMask must not be 0

- VUID-vkCmdWaitEvents-dstStageMask-04090
  If the geometry shaders feature is not enabled, dstStageMask must not contain VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT

- VUID-vkCmdWaitEvents-dstStageMask-04091
  If the tessellation shaders feature is not enabled, dstStageMask must not contain VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT or VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT

- VUID-vkCmdWaitEvents-dstStageMask-03937
  If the synchronization2 feature is not enabled, 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 dstQueueFamilyIndex 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-pEvents-01163
  If pEvents includes one or more events that will be signaled by vkSetEvent after commandBuffer has been submitted to a queue, then vkCmdWaitEvents must not be called inside a render pass instance

- VUID-vkCmdWaitEvents-srcQueueFamilyIndex-02803
  The srcQueueFamilyIndex and dstQueueFamilyIndex members of any element of pBufferMemoryBarriers or pImageMemoryBarriers must be equal
• VUID-vkCmdWaitEvents-commandBuffer-01167
  commandBuffer's current device mask must include exactly one physical device

• VUID-vkCmdWaitEvents-pEvents-03847
  Elements of pEvents must not have been signaled by vkCmdSetEvent2KHR

Valid Usage (Implicit)

• VUID-vkCmdWaitEvents-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

• VUID-vkCmdWaitEvents-pEvents-parameter
  pEvents must be a valid pointer to an array of eventCount valid VkEvent handles

• VUID-vkCmdWaitEvents-srcStageMask-parameter
  srcStageMask must be a valid combination of VkPipelineStageFlagBits values

• VUID-vkCmdWaitEvents-dstStageMask-parameter
  dstStageMask must be a valid combination of VkPipelineStageFlagBits values

• VUID-vkCmdWaitEvents-pMemoryBarriers-parameter
  If memoryBarrierCount is not 0, pMemoryBarriers must be a valid pointer to an array of
  memoryBarrierCount valid VkMemoryBarrier structures

• VUID-vkCmdWaitEvents-pBufferMemoryBarriers-parameter
  If bufferMemoryBarrierCount is not 0, pBufferMemoryBarriers must be a valid pointer to an
  array of bufferMemoryBarrierCount valid VkBufferMemoryBarrier structures

• VUID-vkCmdWaitEvents-pImageMemoryBarriers-parameter
  If imageMemoryBarrierCount is not 0, pImageMemoryBarriers must be a valid pointer to an
  array of imageMemoryBarrierCount valid VkImageMemoryBarrier structures

• VUID-vkCmdWaitEvents-commandBuffer-recording
  commandBuffer must be in the recording state

• VUID-vkCmdWaitEvents-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics, 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
7.6. Pipeline Barriers

To record a pipeline barrier, call:

```c
// Provided by VK_KHR_synchronization2
void vkCmdPipelineBarrier2KHR(
    VkCommandBuffer commandBuffer,
    const VkDependencyInfoKHR* pDependencyInfo);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `pDependencyInfo` is a pointer to a `VkDependencyInfoKHR` structure defining the scopes of this operation.

When `vkCmdPipelineBarrier2KHR` is submitted to a queue, it defines memory dependencies between commands that were submitted before it, and those submitted after it.

The first synchronization scope and access scope of each memory dependency defined by `pDependencyInfo` are applied to operations that occurred earlier in submission order.

The second synchronization scope and access scope of each memory dependency defined by `pDependencyInfo` are applied to operations that occurred later in submission order.

If `vkCmdPipelineBarrier2KHR` is recorded within a render pass instance, the synchronization scopes are limited to operations within the same subpass.

### Valid Usage

- **VUID-vkCmdPipelineBarrier2KHR-pDependencies-02285**
  If `vkCmdPipelineBarrier2KHR` is called within a render pass instance, the render pass must have been created with at least one `VkSubpassDependency` instance in `VkRenderPassCreateInfo::pDependencies` that expresses a dependency from the current subpass to itself, with synchronization scopes and access scopes that are all supersets of the scopes defined in this command.

- **VUID-vkCmdPipelineBarrier2KHR-bufferMemoryBarrierCount-01178**
  If `vkCmdPipelineBarrier2KHR` is called within a render pass instance, it must not include any buffer memory barriers.

- **VUID-vkCmdPipelineBarrier2KHR-image-04073**
  If `vkCmdPipelineBarrier2KHR` is called within a render pass instance, the image member of
any image memory barrier included in this command must be an attachment used in the current subpass both as an input attachment, and as either a color or depth/stencil attachment

- **VUID-vkCmdPipelineBarrier2KHR-oldLayout-01181**
  If `vkCmdPipelineBarrier2KHR` 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-vkCmdPipelineBarrier2KHR-srcQueueFamilyIndex-01182**
  If `vkCmdPipelineBarrier2KHR` is called within a render pass instance, the `srcQueueFamilyIndex` and `dstQueueFamilyIndex` members of any image memory barrier included in this command must be equal

- **VUID-vkCmdPipelineBarrier2KHR-dependencyFlags-01186**
  If `vkCmdPipelineBarrier2KHR` is called outside of a render pass instance, `VK_DEPENDENCY_VIEW_LOCAL_BIT` must not be included in the dependency flags

- **VUID-vkCmdPipelineBarrier2KHR-synchronization2-03848**
  The `synchronization2` feature must be enabled

- **VUID-vkCmdPipelineBarrier2KHR-srcStageMask-03849**
  The `srcStageMask` member of any element of the `pMemoryBarriers`, `pBufferMemoryBarriers`, or `pImageMemoryBarriers` members of `pDependencyInfo` must only include pipeline stages valid for the queue family that was used to create the command pool that `commandBuffer` was allocated from

- **VUID-vkCmdPipelineBarrier2KHR-dstStageMask-03850**
  The `dstStageMask` member of any element of the `pMemoryBarriers`, `pBufferMemoryBarriers`, or `pImageMemoryBarriers` members of `pDependencyInfo` must only include pipeline stages valid for the queue family that was used to create the command pool that `commandBuffer` was allocated from

---

**Valid Usage (Implicit)**

- **VUID-vkCmdPipelineBarrier2KHR-commandBuffer-parameter**
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdPipelineBarrier2KHR-pDependencyInfo-parameter**
  `pDependencyInfo` must be a valid pointer to a valid `VkDependencyInfoKHR` structure

- **VUID-vkCmdPipelineBarrier2KHR-commandBuffer-recording**
  `commandBuffer` must be in the recording state

- **VUID-vkCmdPipelineBarrier2KHR-commandBuffer-cmdpool**
  The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations

---

**Host Synchronization**

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
</tr>
</tbody>
</table>

To record a pipeline barrier, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdPipelineBarrier(
    VkCommandBuffer commandBuffer,
    VkPipelineStageFlags srcStageMask,
    VkPipelineStageFlags dstStageMask,
    VkDependencyFlags dependencyFlags,
    uint32_t memoryBarrierCount,
    const VkMemoryBarrier* pMemoryBarriers,
    uint32_t bufferMemoryBarrierCount,
    const VkBufferMemoryBarrier* pBufferMemoryBarriers,
    uint32_t imageMemoryBarrierCount,
    const VkImageMemoryBarrier* pImageMemoryBarriers);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `srcStageMask` is a bitmask of VkPipelineStageFlagBits specifying the source stages.
- `dstStageMask` is a bitmask of VkPipelineStageFlagBits specifying the destination stages.
- `dependencyFlags` is a bitmask of VkDependencyFlagBits specifying how execution and memory dependencies are formed.
- `memoryBarrierCount` is the length of the `pMemoryBarriers` array.
- `pMemoryBarriers` is a pointer to an array of VkMemoryBarrier structures.
- `bufferMemoryBarrierCount` is the length of the ` pBufferMemoryBarriers` array.
- ` pBufferMemoryBarriers` is a pointer to an array of VkBufferMemoryBarrier structures.
- `imageMemoryBarrierCount` is the length of the ` pImageMemoryBarriers` array.
- ` pImageMemoryBarriers` is a pointer to an array of VkImageMemoryBarrier structures.

`vkCmdPipelineBarrier` operates almost identically to `vkCmdPipelineBarrier2KHR`, except that the scopes and barriers are defined as direct parameters rather than being defined by an VkDependencyInfoKHR.

When `vkCmdPipelineBarrier` is submitted to a queue, it defines a memory dependency between
commands that were submitted before it, and those submitted after it.

If \texttt{vkCmdPipelineBarrier} was recorded outside a render pass instance, the first \textit{synchronization scope} includes all commands that occur earlier in \textit{submission order}. If \texttt{vkCmdPipelineBarrier} was recorded inside a render pass instance, the first synchronization scope includes only commands that occur earlier in \textit{submission order} within the same subpass. In either case, the first synchronization scope is limited to operations on the pipeline stages determined by the \textit{source stage mask} specified by \texttt{srcStageMask}.

If \texttt{vkCmdPipelineBarrier} was recorded outside a render pass instance, the second \textit{synchronization scope} includes all commands that occur later in \textit{submission order}. If \texttt{vkCmdPipelineBarrier} was recorded inside a render pass instance, the second synchronization scope includes only commands that occur later in \textit{submission order} within the same subpass. In either case, the second synchronization scope is limited to operations on the pipeline stages determined by the \textit{destination stage mask} specified by \texttt{dstStageMask}.

The first \textit{access scope} is limited to accesses in the pipeline stages determined by the \textit{source stage mask} specified by \texttt{srcStageMask}. Within that, the first access scope only includes the first access scopes defined by elements of the \texttt{pMemoryBarriers}, \texttt{pBufferMemoryBarriers} and \texttt{pImageMemoryBarriers} arrays, which each define a set of \textit{memory barriers}. If no memory barriers are specified, then the first access scope includes no accesses.

The second \textit{access scope} is limited to accesses in the pipeline stages determined by the \textit{destination stage mask} specified by \texttt{dstStageMask}. Within that, the second access scope only includes the second access scopes defined by elements of the \texttt{pMemoryBarriers}, \texttt{pBufferMemoryBarriers} and \texttt{pImageMemoryBarriers} arrays, which each define a set of \textit{memory barriers}. If no memory barriers are specified, then the second access scope includes no accesses.

If \texttt{dependencyFlags} includes \texttt{VK_DEPENDENCY_BY_REGION_BIT}, then any dependency between \textit{framebuffer-space} pipeline stages is \textit{framebuffer-local} - otherwise it is \textit{framebuffer-global}.

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
\textbf{Valid Usage} \\
\hline
\begin{itemize}
\item VUID-vkCmdPipelineBarrier-srcStageMask-04090
If the \texttt{geometry shaders} feature is not enabled, \texttt{srcStageMask} \textbf{must} not contain \texttt{VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT}

\item VUID-vkCmdPipelineBarrier-srcStageMask-04091
If the \texttt{tessellation shaders} feature is not enabled, \texttt{srcStageMask} \textbf{must} not contain \texttt{VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT} or \texttt{VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT}

\item VUID-vkCmdPipelineBarrier-srcStageMask-03937
If the \texttt{synchronization2} feature is not enabled, \texttt{srcStageMask} \textbf{must} not be \texttt{0}

\item VUID-vkCmdPipelineBarrier-dstStageMask-04090
If the \texttt{geometry shaders} feature is not enabled, \texttt{dstStageMask} \textbf{must} not contain \texttt{VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT}

\item VUID-vkCmdPipelineBarrier-dstStageMask-04091
If the \texttt{tessellation shaders} feature is not enabled, \texttt{dstStageMask} \textbf{must} not contain \texttt{VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT}
\end{itemize}
\hline
\end{tabular}
\end{table}
VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
or
VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT

- VUID-vkCmdPipelineBarrier-dstStageMask-03937
  If the synchronization feature is not enabled, dstStageMask must not be 0

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

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

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

- VUID-vkCmdPipelineBarrier-pBufferMemoryBarriers-02818
  For any element of pBufferMemoryBarriers, if its 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-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-pDependencies-02285
  If vkCmdPipelineBarrier is called within a render pass instance, the render pass must have been created with at least one VkSubpassDependency instance inVkRenderPassCreateInfo::pDependencies that expresses a dependency from the current subpass to itself, with
synchronization scopes and access scopes that are all supersets of the scopes defined in this command

- VUID-vkCmdPipelineBarrier-bufferMemoryBarrierCount-01178
  If `vkCmdPipelineBarrier` is called within a render pass instance, it must not include any buffer memory barriers

- VUID-vkCmdPipelineBarrier-image-04073
  If `vkCmdPipelineBarrier` is called within a render pass instance, the image member of any image memory barrier included in this command must be an attachment used in the current subpass both as an input attachment, and as either a color or depth/stencil attachment

- VUID-vkCmdPipelineBarrier-oldLayout-01181
  If `vkCmdPipelineBarrier` is called within a render pass instance, the oldLayout and newLayout members of any image memory barrier included in this command must be equal

- VUID-vkCmdPipelineBarrier-srcQueueFamilyIndex-01182
  If `vkCmdPipelineBarrier` is called within a render pass instance, the srcQueueFamilyIndex and dstQueueFamilyIndex members of any image memory barrier included in this command must be equal

- VUID-vkCmdPipelineBarrier-dependencyFlags-01186
  If `vkCmdPipelineBarrier` is called outside of a render pass instance, VK_DEPENDENCY_VIEW_LOCAL_BIT must not be included in the dependency flags

- VUID-vkCmdPipelineBarrier-srcStageMask-06461
  Any pipeline stage included in srcStageMask must be supported by the capabilities of the queue family specified by the queueFamilyIndex member of the VkCommandPoolCreateInfo structure that was used to create the VkCommandPool that commandBuffer was allocated from, as specified in the table of supported pipeline stages

- VUID-vkCmdPipelineBarrier-dstStageMask-06462
  Any pipeline stage included in dstStageMask must be supported by the capabilities of the queue family specified by the queueFamilyIndex member of the VkCommandPoolCreateInfo structure that was used to create the VkCommandPool that commandBuffer was allocated from, as specified in the table of supported pipeline stages

Valid Usage (Implicit)

- VUID-vkCmdPipelineBarrier-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

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

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

- VUID-vkCmdPipelineBarrier-dependencyFlags-parameter
  dependencyFlags must be a valid combination of VkDependencyFlagBits values
• VUID-vkCmdPipelineBarrier-pMemoryBarriers-parameter
If `memoryBarrierCount` is not 0, `pMemoryBarriers` must be a valid pointer to an array of `memoryBarrierCount` valid `VkMemoryBarrier` structures

• VUID-vkCmdPipelineBarrier-pBufferMemoryBarriers-parameter
If `bufferMemoryBarrierCount` is not 0, `pBufferMemoryBarriers` must be a valid pointer to an array of `bufferMemoryBarrierCount` valid `VkBufferMemoryBarrier` structures

• VUID-vkCmdPipelineBarrier-pImageMemoryBarriers-parameter
If `imageMemoryBarrierCount` is not 0, `pImageMemoryBarriers` must be a valid pointer to an array of `imageMemoryBarrierCount` valid `VkImageMemoryBarrier` structures

• VUID-vkCmdPipelineBarrier-commandBuffer-recording
`commandBuffer` must be in the recording state

• VUID-vkCmdPipelineBarrier-commandBuffer-cmdpool
The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations

---

**Host Synchronization**

• Host access to `commandBuffer` must be externally synchronized

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

---

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
</tr>
</tbody>
</table>

Bits which can be set in `vkCmdPipelineBarrier::dependencyFlags`, specifying how execution and memory dependencies are formed, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkDependencyFlagBits {
    // Provided by VK_VERSION_1_0
    VK_DEPENDENCY_BY_REGION_BIT = 0x00000001,
    // Provided by VK_VERSION_1_1
    VK_DEPENDENCY_DEVICE_GROUP_BIT = 0x00000004,
    // Provided by VK_VERSION_1_1
    VK_DEPENDENCY_VIEW_LOCAL_BIT = 0x00000002,
} VkDependencyFlagBits;
```

• `VK_DEPENDENCY_BY_REGION_BIT` specifies that dependencies will be framebuffer-local.
• `VK_DEPENDENCY_VIEW_LOCAL_BIT` specifies that a subpass has more than one view.
• `VK_DEPENDENCY_DEVICE_GROUP_BIT` specifies that dependencies are non-device-local.

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

`VkDependencyFlags` is a bitmask type for setting a mask of zero or more `VkDependencyFlagBits`.

### 7.6.1. Subpass Self-dependency

If `vkCmdPipelineBarrier` or `vkCmdPipelineBarrier2KHR` is called inside a render pass instance, the following restrictions apply. For a given subpass to allow a pipeline barrier, the render pass **must** declare a self-dependency from that subpass to itself. That is, there **must** exist a subpass dependency with `srcSubpass` and `dstSubpass` both equal to that subpass index. More than one self-dependency **can** be declared for each subpass.

Self-dependencies **must** only include pipeline stage bits that are graphics stages. If any of the stages in `srcStageMask` are framebuffer-space stages, `dstStageMask` **must** only contain framebuffer-space stages. This means that pseudo-stages like `VK_PIPELINE_STAGE_ALL_COMMANDS_BIT` which include the execution of both framebuffer-space stages and non-framebuffer-space stages **must** not be used.

If the source and destination stage masks both include framebuffer-space stages, then `dependencyFlags` **must** include `VK_DEPENDENCY_BY_REGION_BIT`. If the subpass has more than one view, then `dependencyFlags` **must** include `VK_DEPENDENCY_VIEW_LOCAL_BIT`.

Each of the synchronization scopes and access scopes of a `vkCmdPipelineBarrier2KHR` or `vkCmdPipelineBarrier` command inside a render pass instance **must** be a subset of the scopes of one of the self-dependencies for the current subpass.

If the self-dependency has `VK_DEPENDENCY_BY_REGION_BIT` or `VK_DEPENDENCY_VIEW_LOCAL_BIT` set, then so **must** the pipeline barrier. Pipeline barriers within a render pass instance **must** not include buffer memory barriers. Image memory barriers **must** only specify image subresources that are used as attachments within the subpass, and **must** not define an image layout transition or queue family ownership transfer.

### 7.7. Memory Barriers

Memory barriers are used to explicitly control access to buffer and image subresource ranges. Memory barriers are used to transfer ownership between queue families, change image layouts, and define availability and visibility operations. They explicitly define the access types and buffer and image subresource ranges that are included in the access scopes of a memory dependency that is created by a synchronization command that includes them.

#### 7.7.1. Global Memory Barriers

Global memory barriers apply to memory accesses involving all memory objects that exist at the time of its execution.
The `VkMemoryBarrier2KHR` structure is defined as:

```c
// Provided by VK_KHR_synchronization2
typedef struct VkMemoryBarrier2KHR {
    VkStructureType sType;
    const void* pNext;
    VkPipelineStageFlags2KHR srcStageMask;
    VkAccessFlags2KHR srcAccessMask;
    VkPipelineStageFlags2KHR dstStageMask;
    VkAccessFlags2KHR dstAccessMask;
} VkMemoryBarrier2KHR;
```

• **sType** is the type of this structure.

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

• **srcStageMask** is a `VkPipelineStageFlags2KHR` mask of pipeline stages to be included in the **first synchronization scope**.

• **srcAccessMask** is a `VkAccessFlags2KHR` mask of access flags to be included in the **first access scope**.

• **dstStageMask** is a `VkPipelineStageFlags2KHR` mask of pipeline stages to be included in the **second synchronization scope**.

• **dstAccessMask** is a `VkAccessFlags2KHR` mask of access flags to be included in the **second access scope**.

This structure defines a memory dependency affecting all device memory.

The first synchronization scope and access scope described by this structure include only operations and memory accesses specified by **srcStageMask** and **srcAccessMask**.

The second synchronization scope and access scope described by this structure include only operations and memory accesses specified by **dstStageMask** and **dstAccessMask**.

### Valid Usage

- **VUID-VkMemoryBarrier2KHR-srcStageMask-03929**
  If the geometry shaders feature is not enabled, **srcStageMask** must not contain
  `VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR`

- **VUID-VkMemoryBarrier2KHR-srcStageMask-03930**
  If the tessellation shaders feature is not enabled, **srcStageMask** must not contain
  `VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR`
  or `VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR`

- **VUID-VkMemoryBarrier2KHR-srcAccessMask-03900**
  If **srcAccessMask** includes `VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT_KHR`, **srcStageMask** must include
  `VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`
• VUID-VkMemoryBarrier2KHR-srcAccessMask-03901
  If `srcAccessMask` includes `VK_ACCESS_2_INDEX_READ_BIT_KHR`, `srcStageMask` must include
  `VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR`,
  `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`

• VUID-VkMemoryBarrier2KHR-srcAccessMask-03902
  If `srcAccessMask` includes `VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT_KHR`, `srcStageMask` must
  include `VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR`,
  `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`

• VUID-VkMemoryBarrier2KHR-srcAccessMask-03903
  If `srcAccessMask` includes `VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT_KHR`, `srcStageMask` must
  include `VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR`, `VK_PIPELINE_STAGE_2_SUBPASS_SHADING_BIT_HUAWEI`,
  `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`

• VUID-VkMemoryBarrier2KHR-srcAccessMask-03904
  If `srcAccessMask` includes `VK_ACCESS_2_UNIFORM_READ_BIT_KHR`, `srcStageMask` must include
  `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or
  one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages

• VUID-VkMemoryBarrier2KHR-srcAccessMask-03905
  If `srcAccessMask` includes `VK_ACCESS_2_SHADER_SAMPLED_READ_BIT_KHR`, `srcStageMask` must include
  `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or
  one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages

• VUID-VkMemoryBarrier2KHR-srcAccessMask-03906
  If `srcAccessMask` includes `VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR`, `srcStageMask` must include
  `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or
  one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages

• VUID-VkMemoryBarrier2KHR-srcAccessMask-03907
  If `srcAccessMask` includes `VK_ACCESS_2_SHADER_WRITE_BIT_KHR`, `srcStageMask` must include
  `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or
  one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages

• VUID-VkMemoryBarrier2KHR-srcAccessMask-03908
  If `srcAccessMask` includes `VK_ACCESS_2_INDEX_READ_BIT_KHR`, `srcStageMask` must include
  `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or
  one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages

• VUID-VkMemoryBarrier2KHR-srcAccessMask-03909
  If `srcAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR`, `srcStageMask` must include
  `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or
  one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages

• VUID-VkMemoryBarrier2KHR-srcAccessMask-03910
  If `srcAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR`, `srcStageMask` must
• VUID-VkMemoryBarrier2KHR-srcAccessMask-03911
If srcAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT_KHR, srcStageMask must include
VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkMemoryBarrier2KHR-srcAccessMask-03912
If srcAccessMask includes VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR, srcStageMask must include
VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkMemoryBarrier2KHR-srcAccessMask-03913
If srcAccessMask includes VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT_KHR, srcStageMask must include
VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkMemoryBarrier2KHR-srcAccessMask-03914
If srcAccessMask includes VK_ACCESS_2_TRANSFER_READ_BIT_KHR, srcStageMask must include
VK_PIPELINE_STAGE_2_COPY_BIT_KHR, VK_PIPELINE_STAGE_2_BLIT_BIT_KHR, VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR, VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkMemoryBarrier2KHR-srcAccessMask-03915
If srcAccessMask includes VK_ACCESS_2_TRANSFER_WRITE_BIT_KHR, srcStageMask must include
VK_PIPELINE_STAGE_2_COPY_BIT_KHR, VK_PIPELINE_STAGE_2_BLIT_BIT_KHR, VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR, VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR, VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkMemoryBarrier2KHR-srcAccessMask-03916
If srcAccessMask includes VK_ACCESS_2_HOST_READ_BIT_KHR, srcStageMask must include
VK_PIPELINE_STAGE_2_HOST_BIT_KHR

• VUID-VkMemoryBarrier2KHR-srcAccessMask-03917
If srcAccessMask includes VK_ACCESS_2_HOST_WRITE_BIT_KHR, srcStageMask must include
VK_PIPELINE_STAGE_2_HOST_BIT_KHR

• VUID-VkMemoryBarrier2KHR-srcAccessMask-03926
If srcAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT, srcStageMask must include
VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkMemoryBarrier2KHR-dstStageMask-03929
If the geometry shaders feature is not enabled, dstStageMask must not contain
VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR

• VUID-VkMemoryBarrier2KHR-dstStageMask-03930
If the tessellation shaders feature is not enabled, \( \text{dstStageMask} \) must not contain

\[
\text{VK_PIPELINE_STAGE_2_TESSellation\_CONTROL\_SHADER\_BIT\_KHR} \\
\text{or} \\
\text{VK_PIPELINE_STAGE_2_TESSellation\_EVALUATION\_SHADER\_BIT\_KHR}
\]

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03900
If \( \text{dstAccessMask} \) includes \( \text{VK\_ACCESS\_2\_INDIRECT\_COMMAND\_READ\_BIT\_KHR} \), \( \text{dstStageMask} \) must include

\[
\text{VK\_PIPELINE\_STAGE\_2\_DRAW\_INDIRECT\_BIT\_KHR}, \\
\text{VK\_PIPELINE\_STAGE\_2\_ACCELERATION\_STRUCTURE\_BUILD\_BIT\_KHR}, \\
\text{VK\_PIPELINE\_STAGE\_2\_ALL\_GRAPHICS\_BIT\_KHR}, \text{or} \ \text{VK\_PIPELINE\_STAGE\_2\_ALL\_COMMANDS\_BIT\_KHR}
\]

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03901
If \( \text{dstAccessMask} \) includes \( \text{VK\_ACCESS\_2\_INDEX\_READ\_BIT\_KHR} \), \( \text{dstStageMask} \) must include

\[
\text{VK\_PIPELINE\_STAGE\_2\_INDEX\_INPUT\_BIT\_KHR}, \\
\text{VK\_PIPELINE\_STAGE\_2\_VERTEX\_INPUT\_BIT\_KHR}, \\
\text{VK\_PIPELINE\_STAGE\_2\_ALL\_GRAPHICS\_BIT\_KHR}, \text{or} \ \text{VK\_PIPELINE\_STAGE\_2\_ALL\_COMMANDS\_BIT\_KHR}
\]

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03902
If \( \text{dstAccessMask} \) includes \( \text{VK\_ACCESS\_2\_VERTEX\_ATTRIBUTE\_READ\_BIT\_KHR} \), \( \text{dstStageMask} \) must include

\[
\text{VK\_PIPELINE\_STAGE\_2\_VERTEX\_ATTRIBUTE\_INPUT\_BIT\_KHR}, \\
\text{VK\_PIPELINE\_STAGE\_2\_VERTEX\_INPUT\_BIT\_KHR}, \\
\text{VK\_PIPELINE\_STAGE\_2\_ALL\_GRAPHICS\_BIT\_KHR}, \text{or} \ \text{VK\_PIPELINE\_STAGE\_2\_ALL\_COMMANDS\_BIT\_KHR}
\]

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03903
If \( \text{dstAccessMask} \) includes \( \text{VK\_ACCESS\_2\_INPUT\_ATTACHMENT\_READ\_BIT\_KHR} \), \( \text{dstStageMask} \) must include

\[
\text{VK\_PIPELINE\_STAGE\_2\_FRAGMENT\_SHADER\_BIT\_KHR}, \\
\text{VK\_PIPELINE\_STAGE\_2\_SUBPASS\_SHADING\_BIT\_HUAWEI}, \\
\text{VK\_PIPELINE\_STAGE\_2\_ALL\_GRAPHICS\_BIT\_KHR}, \text{or} \ \text{VK\_PIPELINE\_STAGE\_2\_ALL\_COMMANDS\_BIT\_KHR}
\]

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03904
If \( \text{dstAccessMask} \) includes \( \text{VK\_ACCESS\_2\_UNIFORM\_READ\_BIT\_KHR} \), \( \text{dstStageMask} \) must include

\[
\text{VK\_PIPELINE\_STAGE\_2\_ALL\_GRAPHICS\_BIT\_KHR}, \ \text{VK\_PIPELINE\_STAGE\_2\_ALL\_COMMANDS\_BIT\_KHR}, \text{or} \text{ one of the } \text{VK\_PIPELINE\_STAGE\_*_SHADER\_BIT} \text{ stages}
\]

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03905
If \( \text{dstAccessMask} \) includes \( \text{VK\_ACCESS\_2\_SHADER\_SAMPLED\_READ\_BIT\_KHR} \), \( \text{dstStageMask} \) must include

\[
\text{VK\_PIPELINE\_STAGE\_2\_ALL\_GRAPHICS\_BIT\_KHR}, \text{or} \text{ one of the } \text{VK\_PIPELINE\_STAGE\_*_SHADER\_BIT} \text{ stages}
\]

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03906
If \( \text{dstAccessMask} \) includes \( \text{VK\_ACCESS\_2\_SHADER\_STORAGE\_READ\_BIT\_KHR} \), \( \text{dstStageMask} \) must include

\[
\text{VK\_PIPELINE\_STAGE\_2\_ALL\_GRAPHICS\_BIT\_KHR}, \text{or} \text{ one of the } \text{VK\_PIPELINE\_STAGE\_*_SHADER\_BIT} \text{ stages}
\]

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03907
If \( \text{dstAccessMask} \) includes \( \text{VK\_ACCESS\_2\_SHADER\_STORAGE\_WRITE\_BIT\_KHR} \), \( \text{dstStageMask} \) must include

\[
\text{VK\_PIPELINE\_STAGE\_2\_ALL\_GRAPHICS\_BIT\_KHR}, \text{or} \text{ one of the } \text{VK\_PIPELINE\_STAGE\_*_SHADER\_BIT} \text{ stages}
\]

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03908
If \( \text{dstAccessMask} \) includes \( \text{VK\_ACCESS\_2\_SHADER\_READ\_BIT\_KHR} \), \( \text{dstStageMask} \) must include

\[
\text{VK\_PIPELINE\_STAGE\_2\_ALL\_GRAPHICS\_BIT\_KHR}, \text{ or} \ \text{VK\_PIPELINE\_STAGE\_2\_ALL\_COMMANDS\_BIT\_KHR}
\]
VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or one of the
VK_PIPELINE_STAGE_*_SHADER_BIT stages

- **VUID-VkMemoryBarrier2KHR-dstAccessMask-03909**
  If dstAccessMask includes VK_ACCESS_2_SHADER_WRITE_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or
  one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- **VUID-VkMemoryBarrier2KHR-dstAccessMask-03910**
  If dstAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR, dstStageMask must
  include VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR

- **VUID-VkMemoryBarrier2KHR-dstAccessMask-03911**
  If dstAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT_KHR, dstStageMask must
  include VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR

- **VUID-VkMemoryBarrier2KHR-dstAccessMask-03912**
  If dstAccessMask includes VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR, dstStageMask must
  include VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- **VUID-VkMemoryBarrier2KHR-dstAccessMask-03913**
  If dstAccessMask includes VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT_KHR, dstStageMask must
  include VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- **VUID-VkMemoryBarrier2KHR-dstAccessMask-03914**
  If dstAccessMask includes VK_ACCESS_2_TRANSFER_READ_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_COPY_BIT_KHR, VK_PIPELINE_STAGE_2_BLIT_BIT_KHR, VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR,
  VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR, or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- **VUID-VkMemoryBarrier2KHR-dstAccessMask-03915**
  If dstAccessMask includes VK_ACCESS_2_TRANSFER_WRITE_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_COPY_BIT_KHR, VK_PIPELINE_STAGE_2_BLIT_BIT_KHR, VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR,
  VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR, or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- **VUID-VkMemoryBarrier2KHR-dstAccessMask-03916**
  If dstAccessMask includes VK_ACCESS_2_HOST_READ_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_HOST_BIT_KHR

- **VUID-VkMemoryBarrier2KHR-dstAccessMask-03917**
  If dstAccessMask includes VK_ACCESS_2_HOST_WRITE_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_HOST_BIT_KHR

- **VUID-VkMemoryBarrier2KHR-dstAccessMask-03926**
If `dstAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR` or `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR` or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

Valid Usage (Implicit)

- `VUID-VkMemoryBarrier2KHR-sType-sType` - `sType` must be `VK_STRUCTURE_TYPE_MEMORY_BARRIER_2_KHR`
- `VUID-VkMemoryBarrier2KHR-srcStageMask-parameter` - `srcStageMask` must be a valid combination of `VkPipelineStageFlagBits2KHR` values
- `VUID-VkMemoryBarrier2KHR-srcAccessMask-parameter` - `srcAccessMask` must be a valid combination of `VkAccessFlagBits2KHR` values
- `VUID-VkMemoryBarrier2KHR-dstStageMask-parameter` - `dstStageMask` must be a valid combination of `VkPipelineStageFlagBits2KHR` values
- `VUID-VkMemoryBarrier2KHR-dstAccessMask-parameter` - `dstAccessMask` must be a valid combination of `VkAccessFlagBits2KHR` values

The `VkMemoryBarrier` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkMemoryBarrier {
    VkStructureType sType;
    const void* pNext;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
} VkMemoryBarrier;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcAccessMask` is a bitmask of `VkAccessFlagBits` specifying a source access mask.
- `dstAccessMask` is a bitmask of `VkAccessFlagBits` specifying a destination access mask.

The first access scope is limited to access types in the source access mask specified by `srcAccessMask`.

The second access scope is limited to access types in the destination access mask specified by `dstAccessMask`.

Valid Usage (Implicit)

- `VUID-VkMemoryBarrier-sType-sType` - `sType` must be `VK_STRUCTURE_TYPE_MEMORY_BARRIER`
- `VUID-VkMemoryBarrier-pNext-pNext` - `pNext` must be `NULL`
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 `VkBufferMemoryBarrier2KHR` structure is defined as:

```c
// Provided by VK_KHR_synchronization2
typedef struct VkBufferMemoryBarrier2KHR {
    VkStructureType sType;
    const void* pNext;
    VkPipelineStageFlags2KHR srcStageMask;
    VkAccessFlags2KHR srcAccessMask;
    VkPipelineStageFlags2KHR dstStageMask;
    VkAccessFlags2KHR dstAccessMask;
    uint32_t srcQueueFamilyIndex;
    uint32_t dstQueueFamilyIndex;
    VkBuffer buffer;
    VkDeviceSize offset;
    VkDeviceSize size;
} VkBufferMemoryBarrier2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcStageMask` is a `VkPipelineStageFlags2KHR` mask of pipeline stages to be included in the first synchronization scope.
- `srcAccessMask` is a `VkAccessFlags2KHR` mask of access flags to be included in the first access scope.
- `dstStageMask` is a `VkPipelineStageFlags2KHR` mask of pipeline stages to be included in the second synchronization scope.
- `dstAccessMask` is a `VkAccessFlags2KHR` mask of access flags to be included in the second access scope.
- `srcQueueFamilyIndex` is the source queue family for a queue family ownership transfer.
- `dstQueueFamilyIndex` is the destination queue family for a queue family ownership transfer.
- `buffer` is a handle to the buffer whose backing memory is affected by the barrier.
- `offset` is an offset in bytes into the backing memory for `buffer`; this is relative to the base offset
as bound to the buffer (see `vkBindBufferMemory`).

- **size** is a size in bytes of the affected area of backing memory for `buffer`, or `VK_WHOLE_SIZE` to use the range from `offset` to the end of the buffer.

This structure defines a **memory dependency** limited to a range of a buffer, and can define a **queue family transfer operation** for that range.

The first **synchronization scope** and **access scope** described by this structure include only operations and memory accesses specified by `srcStageMask` and `srcAccessMask`.

The second **synchronization scope** and **access scope** described by this structure include only operations and memory accesses specified by `dstStageMask` and `dstAccessMask`.

Both **access scopes** are limited to only memory accesses to `buffer` in the range defined by `offset` and `size`.

If `buffer` was created with `VK_SHARING_MODE_EXCLUSIVE`, and `srcQueueFamilyIndex` is not equal to `dstQueueFamilyIndex`, this memory barrier defines a **queue family transfer operation**. When executed on a queue in the family identified by `srcQueueFamilyIndex`, this barrier defines a **queue family release operation** for the specified buffer range, and the second synchronization and access scopes do not synchronize operations on that queue. When executed on a queue in the family identified by `dstQueueFamilyIndex`, this barrier defines a **queue family acquire operation** for the specified buffer range, and the first synchronization and access scopes do not synchronize operations on that queue.

A **queue family transfer operation** is also defined if the values are not equal, and either is one of the special queue family values reserved for external memory ownership transfers, as described in **Queue Family Ownership Transfer**. A **queue family release operation** is defined when `dstQueueFamilyIndex` is one of those values, and a **queue family acquire operation** is defined when `srcQueueFamilyIndex` is one of those values.

### Valid Usage

- **VUID-VkBufferMemoryBarrier2KHR-srcStageMask-03929**
  If the **geometry shaders** feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR`.

- **VUID-VkBufferMemoryBarrier2KHR-srcStageMask-03930**
  If the **tessellation shaders** feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR` or `VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR`.

- **VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03900**
  If `srcAccessMask` includes `VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT_KHR`, `srcStageMask` **must** include `VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

- **VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03901**
  If `srcAccessMask` includes `VK_ACCESS_2_INDEX_READ_BIT_KHR`, `srcStageMask` **must** include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.  

---

---
VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR, VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03902
  If srcAccessMask includes VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT_KHR, srcStageMask must include VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR, VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03903
  If srcAccessMask includes VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT_KHR, srcStageMask must include VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR, VK_PIPELINE_STAGE_2_SUBPASS_SHADING_BIT_HUAWEI, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03904
  If srcAccessMask includes VK_ACCESS_2_UNIFORM_READ_BIT_KHR, srcStageMask must include VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03905
  If srcAccessMask includes VK_ACCESS_2_SHADER_SAMPLED_READ_BIT_KHR, srcStageMask must include VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03906
  If srcAccessMask includes VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR, srcStageMask must include VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03907
  If srcAccessMask includes VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT_KHR, srcStageMask must include VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03908
  If srcAccessMask includes VK_ACCESS_2_SHADER_READ_BIT_KHR, srcStageMask must include VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03909
  If srcAccessMask includes VK_ACCESS_2_SHADER_WRITE_BIT_KHR, srcStageMask must include VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03910
  If srcAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR, srcStageMask must include VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR
If `srcAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_TRANSFER_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COPY_BIT_KHR`, `VK_PIPELINE_STAGE_2_BLIT_BIT_KHR`, `VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_TRANSFER_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COPY_BIT_KHR`, `VK_PIPELINE_STAGE_2_BLIT_BIT_KHR`, `VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR`, `VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_HOST_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_HOST_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_HOST_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_HOST_BIT_KHR`.

If the geometry shaders feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR`.

If the tessellation shaders feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR` or `VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR`.
• VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03900
  If dstAccessMask includes VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR,
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03901
  If dstAccessMask includes VK_ACCESS_2_INDEX_READ_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR, VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03902
  If dstAccessMask includes VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR, VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03903
  If dstAccessMask includes VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR,
  VK_PIPELINE_STAGE_2_SUBPASS_SHADING_BIT_HUAWEI,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03904
  If dstAccessMask includes VK_ACCESS_2_UNIFORM_READ_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03905
  If dstAccessMask includes VK_ACCESS_2_SHADER_SAMPLED_READ_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03906
  If dstAccessMask includes VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03907
  If dstAccessMask includes VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03908
  If dstAccessMask includes VK_ACCESS_2_SHADER_READ_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR,
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03909
If $\text{dstAccessMask}$ includes `VK_ACCESS_2_SHADER_WRITE_BIT_KHR`, $\text{dstStageMask}$ must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_2_SHADER_BIT` stages

- **VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03910**  
  If $\text{dstAccessMask}$ includes `VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR`, $\text{dstStageMask}$ must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR` or `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

- **VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03911**  
  If $\text{dstAccessMask}$ includes `VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT_KHR`, $\text{dstStageMask}$ must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR` or `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

- **VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03912**  
  If $\text{dstAccessMask}$ includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR`, $\text{dstStageMask}$ must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

- **VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03913**  
  If $\text{dstAccessMask}$ includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT_KHR`, $\text{dstStageMask}$ must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

- **VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03914**  
  If $\text{dstAccessMask}$ includes `VK_ACCESS_2_TRANSFER_READ_BIT_KHR`, $\text{dstStageMask}$ must include `VK_PIPELINE_STAGE_2_COPY_BIT_KHR`, `VK_PIPELINE_STAGE_2_BLIT_BIT_KHR`, `VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

- **VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03915**  
  If $\text{dstAccessMask}$ includes `VK_ACCESS_2_TRANSFER_WRITE_BIT_KHR`, $\text{dstStageMask}$ must include `VK_PIPELINE_STAGE_2_COPY_BIT_KHR`, `VK_PIPELINE_STAGE_2_BLIT_BIT_KHR`, `VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

- **VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03916**  
  If $\text{dstAccessMask}$ includes `VK_ACCESS_2_HOST_READ_BIT_KHR`, $\text{dstStageMask}$ must include `VK_PIPELINE_STAGE_2_HOST_BIT_KHR`.

- **VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03917**  
  If $\text{dstAccessMask}$ includes `VK_ACCESS_2_HOST_WRITE_BIT_KHR`, $\text{dstStageMask}$ must include `VK_PIPELINE_STAGE_2_HOST_BIT_KHR`.

- **VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03926**  
  If $\text{dstAccessMask}$ includes `VK_ACCESS_2_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT`, $\text{dstStageMask}$ must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR` or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.
• VUID-VkBufferMemoryBarrier2KHR-offset-01187
  offset must be less than the size of buffer

• VUID-VkBufferMemoryBarrier2KHR-size-01188
  If size is not equal to VK_WHOLE_SIZE, size must be greater than 0

• VUID-VkBufferMemoryBarrier2KHR-size-01189
  If size is not equal to VK_WHOLE_SIZE, size must be less than or equal to than the size of buffer minus offset

• VUID-VkBufferMemoryBarrier2KHR-buffer-01931
  If buffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

• VUID-VkBufferMemoryBarrier2KHR-srcQueueFamilyIndex-04087
  If srcQueueFamilyIndex is not equal to dstQueueFamilyIndex, at least one must not be a special queue family reserved for external memory ownership transfers, as described in Queue Family Ownership Transfer

• VUID-VkBufferMemoryBarrier2KHR-buffer-04088
  If buffer was created with a sharing mode of VK_SHARING_MODE_CONCURRENT, srcQueueFamilyIndex and dstQueueFamilyIndex are not equal, and one of srcQueueFamilyIndex and dstQueueFamilyIndex is one of the special queue family values reserved for external memory transfers, the other must be VK_QUEUE_FAMILY_IGNORED

• VUID-VkBufferMemoryBarrier2KHR-buffer-04089
  If buffer was created with a sharing mode of VK_SHARING_MODE_EXCLUSIVE, and srcQueueFamilyIndex and dstQueueFamilyIndex are not equal, srcQueueFamilyIndex and dstQueueFamilyIndex must both be valid queue families, or one of the special queue family values reserved for external memory transfers, as described in Queue Family Ownership Transfer

• VUID-VkBufferMemoryBarrier2KHR-srcStageMask-03851
  If either srcStageMask or dstStageMask includes VK_PIPELINE_STAGE_2_HOST_BIT_KHR, srcQueueFamilyIndex and dstQueueFamilyIndex must be equal

---

### Valid Usage (Implicit)

• VUID-VkBufferMemoryBarrier2KHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER_2_KHR

• VUID-VkBufferMemoryBarrier2KHR-pNext-pNext
  pNext must be NULL

• VUID-VkBufferMemoryBarrier2KHR-srcStageMask-parameter
  srcStageMask must be a valid combination of VkPipelineStageFlagBits2KHR values

• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-parameter
  srcAccessMask must be a valid combination of VkAccessFlagBits2KHR values

• VUID-VkBufferMemoryBarrier2KHR-dstStageMask-parameter
  dstStageMask must be a valid combination of VkPipelineStageFlagBits2KHR values

• VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-parameter
**dstAccessMask** must be a valid combination of *VkAccessFlagBits2KHR* values

- VUID-VkBufferMemoryBarrier2KHR-buffer-parameter
  **buffer** must be a valid *VkBuffer* handle

The *VkBufferMemoryBarrier* structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkBufferMemoryBarrier {
    VkStructureType sType;
    const void* pNext;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    uint32_t srcQueueFamilyIndex;
    uint32_t dstQueueFamilyIndex;
    VkBuffer buffer;
    VkDeviceSize offset;
    VkDeviceSize size;
} VkBufferMemoryBarrier;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **srcAccessMask** is a bitmask of *VkAccessFlagBits* specifying a source access mask.
- **dstAccessMask** is a bitmask of *VkAccessFlagBits* specifying a destination access mask.
- **srcQueueFamilyIndex** is the source queue family for a queue family ownership transfer.
- **dstQueueFamilyIndex** is the destination queue family for a queue family ownership transfer.
- **buffer** is a handle to the buffer whose backing memory is affected by the barrier.
- **offset** is an offset in bytes into the backing memory for **buffer**; this is relative to the base offset as bound to the buffer (see *vkBindBufferMemory*).
- **size** is a size in bytes of the affected area of backing memory for **buffer**, or *VK_WHOLE_SIZE* to use the range from **offset** to the end of the buffer.

The first access scope is limited to access to memory through the specified buffer range, via access types in the source access mask specified by **srcAccessMask**. If **srcAccessMask** includes *VK_ACCESS_HOST_WRITE_BIT*, memory writes performed by that access type are also made visible, as that access type is not performed through a resource.

The second access scope is limited to access to memory through the specified buffer range, via access types in the destination access mask specified by **dstAccessMask**. If **dstAccessMask** includes *VK_ACCESS_HOST_WRITE_BIT* or *VK_ACCESS_HOST_READ_BIT*, available memory writes are also made visible to accesses of those types, as those access types are not performed through a resource.

If **srcQueueFamilyIndex** is not equal to **dstQueueFamilyIndex**, and **srcQueueFamilyIndex** is equal to the current queue family, then the memory barrier defines a queue family release operation for the specified buffer range, and the second access scope includes no access, as if **dstAccessMask** was 0.
If $\text{dstQueueFamilyIndex}$ is not equal to $\text{srcQueueFamilyIndex}$, and $\text{dstQueueFamilyIndex}$ 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 $\text{srcAccessMask}$ was 0.

### Valid Usage

- VUID-VkBufferMemoryBarrier-offset-01187
  
  **offset must** be less than the size of **buffer**

- VUID-VkBufferMemoryBarrier-size-01188
  
  If **size** is not equal to **VK_WHOLE_SIZE**, **size must** be greater than 0

- VUID-VkBufferMemoryBarrier-size-01189
  
  If **size** is not equal to **VK_WHOLE_SIZE**, **size must** be less than or equal to than the size of **buffer** minus **offset**

- VUID-VkBufferMemoryBarrier-buffer-01931
  
  If **buffer** is non-sparse then it **must** be bound completely and contiguously to a single **VkDeviceMemory** object

- VUID-VkBufferMemoryBarrier-srcQueueFamilyIndex-04087
  
  If **srcQueueFamilyIndex** is not equal to **dstQueueFamilyIndex**, at least one **must** not be a special queue family reserved for external memory ownership transfers, as described in Queue Family Ownership Transfer

- VUID-VkBufferMemoryBarrier-buffer-04088
  
  If **buffer** was created with a sharing mode of **VK_SHARING_MODE_CONCURRENT**, **srcQueueFamilyIndex** and **dstQueueFamilyIndex** are not equal, and one of **srcQueueFamilyIndex** and **dstQueueFamilyIndex** is one of the special queue family values reserved for external memory transfers, the other **must** be **VK_QUEUE_FAMILY_IGNORED**

- VUID-VkBufferMemoryBarrier-buffer-04089
  
  If **buffer** was created with a sharing mode of **VK_SHARING_MODE_EXCLUSIVE**, and **srcQueueFamilyIndex** and **dstQueueFamilyIndex** are not equal, **srcQueueFamilyIndex** and **dstQueueFamilyIndex** **must** both be valid queue families, or one of the special queue family values reserved for external memory transfers, as described in Queue Family Ownership Transfer

- VUID-VkBufferMemoryBarrier-synchronization2-03853
  
  If the **synchronization2** feature is not enabled, and **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**

### 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.

#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 VkImageMemoryBarrier2KHR structure is defined as:

```c
// Provided by VK_KHR_synchronization2
typedef struct VkImageMemoryBarrier2KHR {
    VkStructureType sType;
    const void* pNext;
    VkPipelineStageFlags2KHR srcStageMask;
    VkAccessFlags2KHR srcAccessMask;
    VkPipelineStageFlags2KHR dstStageMask;
    VkAccessFlags2KHR dstAccessMask;
    VkImageLayout oldLayout;
    VkImageLayout newLayout;
    uint32_t srcQueueFamilyIndex;
    uint32_t dstQueueFamilyIndex;
    VkImage image;
    VkImageSubresourceRange subresourceRange;
} VkImageMemoryBarrier2KHR;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `srcStageMask` is a VkPipelineStageFlags2KHR mask of pipeline stages to be included in the first synchronization scope.
- `srcAccessMask` is a VkAccessFlags2KHR mask of access flags to be included in the first access scope.
- `dstStageMask` is a VkPipelineStageFlags2KHR mask of pipeline stages to be included in the second synchronization scope.
- `dstAccessMask` is a VkAccessFlags2KHR mask of access flags to be included in the second access scope.
• **oldLayout** is the old layout in an image layout transition.
• **newLayout** is the new layout in an image layout transition.
• **srcQueueFamilyIndex** is the source queue family for a queue family ownership transfer.
• **dstQueueFamilyIndex** is the destination queue family for a queue family ownership transfer.
• **image** is a handle to the image affected by this barrier.
• **subresourceRange** describes the image subresource range within image that is affected by this barrier.

This structure defines a memory dependency limited to an image subresource range, and can define a queue family transfer operation and image layout transition for that subresource range.

The first synchronization scope and access scope described by this structure include only operations and memory accesses specified by **srcStageMask** and **srcAccessMask**.

The second synchronization scope and access scope described by this structure include only operations and memory accesses specified by **dstStageMask** and **dstAccessMask**.

Both access scopes are limited to only memory accesses to **image** in the subresource range defined by **subresourceRange**.

If **image** was created with **VK_SHARING_MODE_EXCLUSIVE**, and **srcQueueFamilyIndex** is not equal to **dstQueueFamilyIndex**, this memory barrier defines a queue family transfer operation. When executed on a queue in the family identified by **srcQueueFamilyIndex**, this barrier defines a queue family release operation for the specified image subresource range, and the second synchronization and access scopes do not synchronize operations on that queue. When executed on a queue in the family identified by **dstQueueFamilyIndex**, this barrier defines a queue family acquire operation for the specified image subresource range, and the first synchronization and access scopes do not synchronize operations on that queue.

A queue family transfer operation is also defined if the values are not equal, and either is one of the special queue family values reserved for external memory ownership transfers, as described in Queue Family Ownership Transfer. A queue family release operation is defined when **dstQueueFamilyIndex** is one of those values, and a queue family acquire operation is defined when **srcQueueFamilyIndex** is one of those values.

If **oldLayout** is not equal to **newLayout**, then the memory barrier defines an image layout transition for the specified image subresource range. If this memory barrier defines a queue family transfer operation, the layout transition is only executed once between the queues.

---

**Note**

When the old and new layout are equal, the layout values are ignored - data is preserved no matter what values are specified, or what layout the image is currently in.

If **image** has a multi-planar format and the image is disjoint, then including **VK_IMAGE_ASPECT_COLOR_BIT** in the aspectMask member of **subresourceRange** is equivalent to including **VK_IMAGE_ASPECT_PLANE_0_BIT**, **VK_IMAGE_ASPECT_PLANE_1_BIT**, and (for three-plane formats only)
Valid Usage

• VUID-VkImageMemoryBarrier2KHR-srcStageMask-03929
  If the geometry shaders feature is not enabled, srcStageMask must not contain
  VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR

• VUID-VkImageMemoryBarrier2KHR-srcStageMask-03930
  If the tessellation shaders feature is not enabled, srcStageMask must not contain
  VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR
  or
  VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03900
  If srcAccessMask includes VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR,
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03901
  If srcAccessMask includes VK_ACCESS_2_INDEX_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR,
  VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03902
  If srcAccessMask includes VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR,
  VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03903
  If srcAccessMask includes VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR,
  VK_PIPELINE_STAGE_2_SUBPASS_SHADING_BIT_HUAWEI,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03904
  If srcAccessMask includes VK_ACCESS_2_UNIFORM_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or
  one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03905
  If srcAccessMask includes VK_ACCESS_2_SHADER_SAMPLED_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03906
  If srcAccessMask includes VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages
If `srcAccessMask` includes `VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_TRANSFER_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COPY_BIT_KHR`, `VK_PIPELINE_STAGE_2_BLIT_BIT_KHR`, `VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_TRANSFER_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COPY_BIT_KHR`, `VK_PIPELINE_STAGE_2_BLIT_BIT_KHR`, `VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR`, `VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.
• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03916
  If `srcAccessMask` includes `VK_ACCESS_2_HOST_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_HOST_BIT_KHR`.

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03917
  If `srcAccessMask` includes `VK_ACCESS_2_HOST_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_HOST_BIT_KHR`.

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03926
  If `srcAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

• VUID-VkImageMemoryBarrier2KHR-dstStageMask-03929
  If the geometry shaders feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR`.

• VUID-VkImageMemoryBarrier2KHR-dstStageMask-03930
  If the tessellation shaders feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR` or `VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR`.

• VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03900
  If `dstAccessMask` includes `VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT_KHR`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

• VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03901
  If `dstAccessMask` includes `VK_ACCESS_2_INDEX_READ_BIT_KHR`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

• VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03902
  If `dstAccessMask` includes `VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT_KHR`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

• VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03903
  If `dstAccessMask` includes `VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT_KHR`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

• VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03904
  If `dstAccessMask` includes `VK_ACCESS_2_UNIFORM_READ_BIT_KHR`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_2_SHADER_BIT` stages.

• VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03905
  If `dstAccessMask` includes `VK_ACCESS_2_SHADER_SAMPLED_READ_BIT_KHR`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`. 
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- **VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03906**
  If dstAccessMask includes VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR, dstStageMask must include VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- **VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03907**
  If dstAccessMask includes VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT_KHR, dstStageMask must include VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- **VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03908**
  If dstAccessMask includes VK_ACCESS_2_SHADER_READ_BIT_KHR, dstStageMask must include VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- **VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03909**
  If dstAccessMask includes VK_ACCESS_2_SHADER_WRITE_BIT_KHR, dstStageMask must include VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- **VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03910**
  If dstAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR, dstStageMask must include VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- **VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03911**
  If dstAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT_KHR, dstStageMask must include VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- **VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03912**
  If dstAccessMask includes VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR, dstStageMask must include VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- **VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03913**
  If dstAccessMask includes VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT_KHR, dstStageMask must include VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- **VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03914**
  If dstAccessMask includes VK_ACCESS_2_TRANSFER_READ_BIT_KHR, dstStageMask must include VK_PIPELINE_STAGE_2_COPY_BIT_KHR, VK_PIPELINE_STAGE_2_BLIT_BIT_KHR, VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03915
  If dstAccessMask includes VK_ACCESS_2_TRANSFER_WRITE_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_COPY_BIT_KHR, VK_PIPELINE_STAGE_2_BLIT_BIT_KHR,
  VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR, VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR,
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03916
  If dstAccessMask includes VK_ACCESS_2_HOST_READ_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_HOST_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03917
  If dstAccessMask includes VK_ACCESS_2_HOST_WRITE_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_HOST_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03926
  If dstAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT,
  dstStageMask must include VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-subresourceRange-01486
  subresourceRange.baseMipLevel must be less than the mipLevels specified in
  VkImageCreateInfo when image was created

- VUID-VkImageMemoryBarrier2KHR-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-VkImageMemoryBarrier2KHR-subresourceRange-01488
  subresourceRange.baseArrayLayer must be less than the arrayLayers specified in
  VkImageCreateInfo when image was created

- VUID-VkImageMemoryBarrier2KHR-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-VkImageMemoryBarrier2KHR-image-01932
  If image is non-sparse then it must be bound completely and contiguously to a single
  VkDeviceMemory object

- VUID-VkImageMemoryBarrier2KHR-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-VkImageMemoryBarrier2KHR-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
• VUID-VkImageMemoryBarrier2KHR-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_ATTACHMENT_OPTIMAL then image must have been created with VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT

• VUID-VkImageMemoryBarrier2KHR-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_DEPTH_STENCIL_READ_ONLY_OPTIMAL then image must have been created with VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT

• VUID-VkImageMemoryBarrier2KHR-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_SHADER_READ_ONLY_OPTIMAL then image must have been created with VK_IMAGE_USAGE_SAMPLED_BIT or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT

• VUID-VkImageMemoryBarrier2KHR-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_SRC_OPTIMAL then image must have been created with VK_IMAGE_USAGE_TRANSFER_SRC_BIT

• VUID-VkImageMemoryBarrier2KHR-oldLayout-01214
  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-VkImageMemoryBarrier2KHR-oldLayout-01215
  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-VkImageMemoryBarrier2KHR-oldLayout-01216
  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-VkImageMemoryBarrier2KHR-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-VkImageMemoryBarrier2KHR-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-VkImageMemoryBarrier2KHR-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-VkImageMemoryBarrier2KHR-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-VkImageMemoryBarrier2KHR-srcQueueFamilyIndex-03938
  If srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define an image layout transition, and oldLayout or newLayout is VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL_KHR, image must have been created with VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT or VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT

- VUID-VkImageMemoryBarrier2KHR-srcQueueFamilyIndex-03939
  If srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define an image layout transition, and oldLayout or newLayout is VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR, 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-VkImageMemoryBarrier2KHR-oldLayout-02088
  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_FRAGMENT_SHADING_RATE_ATTACHMENT_OPTIMAL_KHR then image must have been created with VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR set

- VUID-VkImageMemoryBarrier2KHR-image-01671
  If image has a single-plane color format or is not disjoint, then the aspectMask member of subresourceRange must be VK_IMAGE_ASPECT_COLOR_BIT

- VUID-VkImageMemoryBarrier2KHR-image-01672
  If image has a multi-planar format and the image is disjoint, then the aspectMask member of subresourceRange must include either at least one of VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, and VK_IMAGE_ASPECT_PLANE_2_BIT; or must include VK_IMAGE_ASPECT_COLOR_BIT

- VUID-VkImageMemoryBarrier2KHR-image-01673
If `image` has a multi-planar format with only two planes, then the `aspectMask` member of `subresourceRange` must not include `VK_IMAGE_ASPECT_PLANE_2_BIT`.

- **VUID-VkImageMemoryBarrier2KHR-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-VkImageMemoryBarrier2KHR-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-VkImageMemoryBarrier2KHR-srcQueueFamilyIndex-04070**
  If `srcQueueFamilyIndex` is not equal to `dstQueueFamilyIndex`, at least one must not be a special queue family reserved for external memory ownership transfers, as described in Queue Family Ownership Transfer.

- **VUID-VkImageMemoryBarrier2KHR-image-04071**
  If `image` was created with a sharing mode of `VK_SHARING_MODE_CONCURRENT`, `srcQueueFamilyIndex` and `dstQueueFamilyIndex` are not equal, and one of `srcQueueFamilyIndex` and `dstQueueFamilyIndex` is one of the special queue family values reserved for external memory transfers, the other must be `VK_QUEUE_FAMILY_IGNORED`.

- **VUID-VkImageMemoryBarrier2KHR-image-04072**
  If `image` was created with a sharing mode of `VK_SHARING_MODE_EXCLUSIVE`, and `srcQueueFamilyIndex` and `dstQueueFamilyIndex` are not equal, `srcQueueFamilyIndex` and `dstQueueFamilyIndex` must both be valid queue families, or one of the special queue family values reserved for external memory transfers, as described in Queue Family Ownership Transfer.

- **VUID-VkImageMemoryBarrier2KHR-srcStageMask-03854**
  If either `srcStageMask` or `dstStageMask` includes `VK_PIPELINE_STAGE_2_HOST_BIT_KHR`, `srcQueueFamilyIndex` and `dstQueueFamilyIndex` must be equal.

- **VUID-VkImageMemoryBarrier2KHR-srcStageMask-03855**
  If `srcStageMask` includes `VK_PIPELINE_STAGE_2_HOST_BIT_KHR`, and `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, `oldLayout` must be one of `VK_IMAGE_LAYOUT_PREINITIALIZED`, `VK_IMAGE_LAYOUT_UNDEFINED`, or `VK_IMAGE_LAYOUT_GENERAL`.

---

**Valid Usage (Implicit)**

- **VUID-VkImageMemoryBarrier2KHR-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER_2_KHR`.

- **VUID-VkImageMemoryBarrier2KHR-pNext-pNext**
  `pNext` must be `NULL` or a pointer to a valid instance of `VkSampleLocationsInfoEXT`.

- **VUID-VkImageMemoryBarrier2KHR-sType-unique**
The `sType` value of each struct in the `pNext` chain must be unique

- VUID-VkImageMemoryBarrier2KHR-srcStageMask-parameter
  `srcStageMask` must be a valid combination of `VkPipelineStageFlagBits2KHR` values

- VUID-VkImageMemoryBarrier2KHR-srcAccessMask-parameter
  `srcAccessMask` must be a valid combination of `VkAccessFlagBits2KHR` values

- VUID-VkImageMemoryBarrier2KHR-dstStageMask-parameter
  `dstStageMask` must be a valid combination of `VkPipelineStageFlagBits2KHR` values

- VUID-VkImageMemoryBarrier2KHR-dstAccessMask-parameter
  `dstAccessMask` must be a valid combination of `VkAccessFlagBits2KHR` values

- VUID-VkImageMemoryBarrier2KHR-oldLayout-parameter
  `oldLayout` must be a valid `VkImageLayout` value

- VUID-VkImageMemoryBarrier2KHR-newLayout-parameter
  `newLayout` must be a valid `VkImageLayout` value

- VUID-VkImageMemoryBarrier2KHR-image-parameter
  `image` must be a valid `VkImage` handle

- VUID-VkImageMemoryBarrier2KHR-subresourceRange-parameter
  `subresourceRange` must be a valid `VkImageSubresourceRange` structure

The `VkImageMemoryBarrier` structure is defined as:

```c
// Provided by VK_VERSION_1.0
typedef struct VkImageMemoryBarrier {
    VkStructureType sType;
    const void* pNext;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    VkImageLayout oldLayout;
    VkImageLayout newLayout;
    uint32_t srcQueueFamilyIndex;
    uint32_t dstQueueFamilyIndex;
    VkImage image;
    VkImageSubresourceRange subresourceRange;
} VkImageMemoryBarrier;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcAccessMask` is a bitmask of `VkAccessFlagBits` specifying a source access mask.
- `dstAccessMask` is a bitmask of `VkAccessFlagBits` specifying a destination access mask.
- `oldLayout` is the old layout in an image layout transition.
- `newLayout` is the new layout in an image layout transition.
- `srcQueueFamilyIndex` is the source queue family for a queue family ownership transfer.
• *dstQueueFamilyIndex* is the destination queue family for a queue family ownership transfer.

• *image* is a handle to the image affected by this barrier.

• *subresourceRange* describes the image subresource range within *image* that is affected by this barrier.

The first access scope is limited to access to memory through the specified image subresource range, via access types in the source access mask specified by *srcAccessMask*. If *srcAccessMask* includes VK_ACCESS_HOST_WRITE_BIT, memory writes performed by that access type are also made visible, as that access type is not performed through a resource.

The second access scope is limited to access to memory through the specified image subresource range, via access types in the destination access mask specified by *dstAccessMask*. If *dstAccessMask* includes VK_ACCESS_HOST_WRITE_BIT or VK_ACCESS_HOST_READ_BIT, available memory writes are also made visible to accesses of those types, as those access types are not performed through a resource.

If *srcQueueFamilyIndex* is not equal to *dstQueueFamilyIndex*, and *srcQueueFamilyIndex* is equal to the current queue family, then the memory barrier defines a queue family release operation for the specified image subresource range, and the second access scope includes no access, as if *dstAccessMask* was 0.

If *dstQueueFamilyIndex* is not equal to *srcQueueFamilyIndex*, and *dstQueueFamilyIndex* is equal to the current queue family, then the memory barrier defines a queue family acquire operation for the specified image subresource range, and the first access scope includes no access, as if *srcAccessMask* was 0.

If the synchronization2 feature is not enabled or oldLayout is not equal to newLayout, oldLayout and newLayout define an image layout transition for the specified image subresource range.

Note

If the synchronization2 feature is enabled, when the old and new layout are equal, the layout values are ignored - data is preserved no matter what values are specified, or what layout the image is currently in.

If *image* has a multi-planar format and the image is disjoint, then including VK_IMAGE_ASPECT_COLOR_BIT in the aspectMask member of *subresourceRange* is equivalent to including VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, and (for three-plane formats only) VK_IMAGE_ASPECT_PLANE_2_BIT.

### Valid Usage

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

If subresourceRange.layerCount is not VK_REMAINING_ARRAY_LAYERS, subresourceRange.baseArrayLayer + subresourceRange.layerCount must be less than or equal to the arrayLayers specified in VkImageCreateInfo when image was created

If image is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

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

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

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

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

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

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

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 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-04065**
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL` then image must have been created with at least one of `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_SAMPLED_BIT`, or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`

- **VUID-VkImageMemoryBarrier-srcQueueFamilyIndex-04066**
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL` then image must have been created with at least one of `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_SAMPLED_BIT`, or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`

- **VUID-VkImageMemoryBarrier-srcQueueFamilyIndex-04067**
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL` then image must have been created with at least one of `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_SAMPLED_BIT`, or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`

- **VUID-VkImageMemoryBarrier-srcQueueFamilyIndex-04068**
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL` then image must have been created with `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT` set

- **VUID-VkImageMemoryBarrier-srcQueueFamilyIndex-04069**
  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_STENCIL_ATTACHMENT_OPTIMAL` then image must have been created with `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT` set

- **VUID-VkImageMemoryBarrier-srcQueueFamilyIndex-03938**
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL_KHR`, image must have been created with `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` or `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`

- **VUID-VkImageMemoryBarrier-srcQueueFamilyIndex-03939**
  If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer
or oldLayout and newLayout define an image layout transition, and oldLayout or newLayout is VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR, 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-oldLayout-02088
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_FRAGMENT_SHADING_RATE_ATTACHMENT_OPTIMAL_KHR then image must have been created with VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR set

• VUID-VkImageMemoryBarrier-image-01671
If image has a single-plane color format or is not disjoint, then the aspectMask member of subresourceRange must be VK_IMAGE_ASPECT_COLOR_BIT

• VUID-VkImageMemoryBarrier-image-01672
If image has a multi-planar format and the image is disjoint, then the aspectMask member of subresourceRange must include either at least one of VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, and VK_IMAGE_ASPECT_PLANE_2_BIT; or must include VK_IMAGE_ASPECT_COLOR_BIT

• VUID-VkImageMemoryBarrier-image-01673
If image has a multi-planar format with only two planes, then the aspectMask member of subresourceRange must not include VK_IMAGE_ASPECT_PLANE_2_BIT

• VUID-VkImageMemoryBarrier-image-03319
If image has a depth/stencil format with both depth and stencil and the separateDepthStencilLayouts feature is enabled, then the aspectMask member of subresourceRange must include either or both VK_IMAGE_ASPECT_DEPTH_BIT and VK_IMAGE_ASPECT_STENCIL_BIT

• VUID-VkImageMemoryBarrier-image-03320
If image has a depth/stencil format with both depth and stencil and the separateDepthStencilLayouts feature is not enabled, then the aspectMask member of subresourceRange must include both VK_IMAGE_ASPECT_DEPTH_BIT and VK_IMAGE_ASPECT_STENCIL_BIT

• VUID-VkImageMemoryBarrier-srcQueueFamilyIndex-04070
If srcQueueFamilyIndex is not equal to dstQueueFamilyIndex, at least one must not be a special queue family reserved for external memory ownership transfers, as described in Queue Family Ownership Transfer

• VUID-VkImageMemoryBarrier-image-04071
If image was created with a sharing mode of VK_SHARING_MODE_CONCURRENT, srcQueueFamilyIndex and dstQueueFamilyIndex are not equal, and one of srcQueueFamilyIndex and dstQueueFamilyIndex is one of the special queue family values reserved for external memory transfers, the other must be VK_QUEUE_FAMILY_IGNORED

• VUID-VkImageMemoryBarrier-image-04072
If image was created with a sharing mode of VK_SHARING_MODE_EXCLUSIVE, and srcQueueFamilyIndex and dstQueueFamilyIndex are not equal, srcQueueFamilyIndex and dstQueueFamilyIndex must both be valid queue families, or one of the special queue family values reserved for external memory transfers, as described in Queue Family Ownership
Transfer

• VUID-VkImageMemoryBarrier-synchronization2-03857
  If the synchronization2 feature is not enabled, and image was created with a sharing mode of VK_SHARING_MODE_CONCURRENT, at least one of srcQueueFamilyIndex and dstQueueFamilyIndex must be VK_QUEUE_FAMILY_IGNORED

Valid Usage (Implicit)

• VUID-VkImageMemoryBarrier-sType-sType
  sType must be VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER

• VUID-VkImageMemoryBarrier-pNext-pNext
  pNext must be NULL or a pointer to a valid instance of VkSampleLocationsInfoEXT

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

• 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.

#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.
The special queue family index `VK_QUEUE_FAMILY_FOREIGN_EXT` represents any queue external to the resource's current Vulkan instance, regardless of the queue's underlying physical device or driver version. This includes, for example, queues for fixed-function image processing devices, media codec devices, and display devices, as well as all queues that use the same underlying device group or physical device, and the same driver version as the resource's `VkDevice`.

If memory dependencies are correctly expressed between uses of such a resource between two queues in different families, but no ownership transfer is defined, the contents of that resource are undefined for any read accesses performed by the second queue family.

**Note**

If an application does not need the contents of a resource to remain valid when transferring from one queue family to another, then the ownership transfer **should** be skipped.

**Note**

Applications should expect transfers to/from `VK_QUEUE_FAMILY_FOREIGN_EXT` to be more expensive than transfers to/from `VK_QUEUE_FAMILY_EXTERNAL_KHR`.

A queue family ownership transfer consists of two distinct parts:

1. Release exclusive ownership from the source queue family
2. Acquire exclusive ownership for the destination queue family

An application **must** ensure that these operations occur in the correct order by defining an execution dependency between them, e.g. using a semaphore.

A **release operation** is used to release exclusive ownership of a range of a buffer or image subresource range. A release operation is defined by executing a buffer memory barrier (for a buffer range) or an image memory barrier (for an image subresource range) using 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>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>Any</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:

```
// 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 is defined by the host execution model, but includes execution of `vkQueueSubmit` on the host and anything that happened-before it.

The second synchronization scope includes all commands submitted in the same queue submission, and all commands that occur later in submission order.
The first access scope includes all host writes to mappable device memory that are available to the host memory domain.

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

7.10. Synchronization and Multiple Physical Devices

If a logical device includes more than one physical device, then fences, semaphores, and events all still have a single instance of the signaled state.

A fence becomes signaled when all physical devices complete the necessary queue operations.

Semaphore wait and signal operations all include a device index that is the sole physical device that performs the operation. These indices are provided in the VkDeviceGroupSubmitInfo 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).

7.11. Calibrated timestamps

In order to be able to correlate the time a particular operation took place at on timelines of different time domains (e.g. a device operation vs a host operation), Vulkan allows querying calibrated timestamps from multiple time domains.

To query calibrated timestamps from a set of time domains, call:

```c
// Provided by VK_EXT_calibrated_timestamps
VkResult vkGetCalibratedTimestampsEXT(
    VkDevice device,
    uint32_t timestampCount,
    const VkCalibratedTimestampInfoEXT* pTimestampInfos,
    uint64_t* pTimestamps,
    uint64_t* pMaxDeviation);
```

- `device` is the logical device used to perform the query.
- `timestampCount` is the number of timestamps to query.
- `pTimestampInfos` is a pointer to an array of `timestampCount` VkCalibratedTimestampInfoEXT structures, describing the time domains the calibrated timestamps should be captured from.
- `pTimestamps` is a pointer to an array of `timestampCount` 64-bit unsigned integer values in which the requested calibrated timestamp values are returned.
- `pMaxDeviation` is a pointer to a 64-bit unsigned integer value in which the strictly positive maximum deviation, in nanoseconds, of the calibrated timestamp values is returned.

**Note**
The maximum deviation may vary between calls to vkGetCalibratedTimestampsEXT.
even for the same set of time domains due to implementation and platform specific reasons. It is the application's responsibility to assess whether the returned maximum deviation makes the timestamp values suitable for any particular purpose and can choose to re-issue the timestamp calibration call pursuing a lower deviation value.

Calibrated timestamp values can be extrapolated to estimate future coinciding timestamp values, however, depending on the nature of the time domains and other properties of the platform extrapolating values over a sufficiently long period of time may no longer be accurate enough to fit any particular purpose, so applications are expected to re-calibrate the timestamps on a regular basis.

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

### Valid Usage (Implicit)

- VUID-vkGetCalibratedTimestampsEXT-device-parameter
  - device must be a valid VkDevice handle
- VUID-vkGetCalibratedTimestampsEXT-pTimestampInfos-parameter
  - pTimestampInfos must be a valid pointer to an array of timestampCount valid VkCalibratedTimestampInfoEXT structures
- VUID-vkGetCalibratedTimestampsEXT-pTimestamps-parameter
  - pTimestamps must be a valid pointer to an array of timestampCount uint64_t values
- VUID-vkGetCalibratedTimestampsEXT-pMaxDeviation-parameter
  - pMaxDeviation must be a valid pointer to a uint64_t value
- VUID-vkGetCalibratedTimestampsEXT-timestampCount-arraylength
  - timestampCount must be greater than 0

### Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkCalibratedTimestampInfoEXT structure is defined as:

```c
// Provided by VK_EXT_calibrated_timestamps
typedef struct VkCalibratedTimestampInfoEXT {
    VkStructureType sType;
    const void* pNext;
} VkCalibratedTimestampInfoEXT;
```
VkTimeDomainEXT  timeDomain;
} VkCalibratedTimestampInfoEXT;

• *sType* is the type of this structure.
• *pNext* is NULL or a pointer to a structure extending this structure.
• *timeDomain* is a VkTimeDomainEXT value specifying the time domain from which the calibrated timestamp value should be returned.

**Valid Usage**

• VUID-VkCalibratedTimestampInfoEXT-timeDomain-02354
  timeDomain  must be one of the VkTimeDomainEXT values returned by vkGetPhysicalDeviceCalibrateableTimeDomainsEXT

**Valid Usage (Implicit)**

• VUID-VkCalibratedTimestampInfoEXT-sType-sType
  sType  must be VK_STRUCTURE_TYPE_CALIBRATED_TIMESTAMP_INFO_EXT

• VUID-VkCalibratedTimestampInfoEXT-pNext-pNext
  pNext  must be NULL

• VUID-VkCalibratedTimestampInfoEXT-timeDomain-parameter
  timeDomain  must be a valid VkTimeDomainEXT value

The set of supported time domains consists of:

```c
// Provided by VK_EXT_calibrated_timestamps
typedef enum VkTimeDomainEXT {    VK_TIME_DOMAIN_DEVICE_EXT = 0,
    VK_TIME_DOMAIN_CLOCK_MONOTONIC_EXT = 1,
    VK_TIME_DOMAIN_CLOCK_MONOTONIC_RAW_EXT = 2,
    VK_TIME_DOMAIN_QUERY_PERFORMANCE_COUNTER_EXT = 3,
} VkTimeDomainEXT;
```

• **VK_TIME_DOMAIN_DEVICE_EXT** specifies the device time domain. Timestamp values in this time domain use the same units and are comparable with device timestamp values captured using vkCmdWriteTimestamp or vkCmdWriteTimestamp2KHR and are defined to be incrementing according to the *timestampPeriod* of the device.

• **VK_TIME_DOMAIN_CLOCK_MONOTONIC_EXT** specifies the CLOCK_MONOTONIC time domain available on POSIX platforms. Timestamp values in this time domain are in units of nanoseconds and are comparable with platform timestamp values captured using the POSIX clock_gettime API as computed by this example:
An implementation supporting `VK_EXT_calibrated_timestamps` will use the same time domain for all its `VkQueue` so that timestamp values reported for `VK_TIME_DOMAIN_DEVICE_EXT` can be matched to any timestamp captured through `vkCmdWriteTimestamp` or `vkCmdWriteTimestamp2KHR`.

```c
struct timespec tv;
clock_gettime(CLOCK_MONOTONIC, &tv);
return tv.tv_nsec + tv.tv_sec*1000000000ull;
```

- `VK_TIME_DOMAIN_CLOCK_MONOTONIC_RAW_EXT` specifies the `CLOCK_MONOTONIC_RAW` time domain available on POSIX platforms. Timestamp values in this time domain are in units of nanoseconds and are comparable with platform timestamp values captured using the POSIX `clock_gettime` API as computed by this example:

```c
struct timespec tv;
clock_gettime(CLOCK_MONOTONIC_RAW, &tv);
return tv.tv_nsec + tv.tv_sec*1000000000ull;
```

- `VK_TIME_DOMAIN_QUERY_PERFORMANCE_COUNTER_EXT` specifies the performance counter (QPC) time domain available on Windows. Timestamp values in this time domain are in the same units as those provided by the Windows `QueryPerformanceCounter` API and are comparable with platform timestamp values captured using that API as computed by this example:

```c
LARGE_INTEGER counter;
QueryPerformanceCounter(&counter);
return counter.QuadPart;
```
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:

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

An *attachment description* describes the properties of an attachment including its format, sample count, and how its contents are treated at the beginning and end of each render pass instance.

A *subpass* represents a phase of rendering that reads and writes a subset of the attachments in a render pass. Rendering commands are recorded into a particular subpass of a render pass instance.

A *subpass description* describes the subset of attachments that is involved in the execution of a subpass. Each subpass can read from some attachments as *input attachments*, write to some as *color attachments* or *depth/stencil attachments*, and perform *multisample resolve operations* to *resolve attachments*. A subpass description can also include a set of *preserve attachments*, which are attachments that are not read or written by the subpass but whose contents must be preserved throughout the subpass.

A subpass *uses an attachment* if the attachment is a color, depth/stencil, resolve, depth/stencil resolve, fragment shading rate, or input attachment for that subpass (as determined by the `pColorAttachments`, `pDepthStencilAttachment`, `pResolveAttachments`, `VkSubpassDescriptionDepthStencilResolve::pDepthStencilResolveAttachment`, `VkFragmentShadingRateAttachmentInfoKHR::pFragmentShadingRateAttachment->attachment`, 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. For multi-pixel fragments, the pixel read from an input attachment is selected from the pixels covered by that fragment in an implementation-dependent manner. However, this selection must be made consistently for any fragment with the same shading rate for the lifetime of the `VkDevice`.

---

**Note**

By describing a complete set of subpasses in advance, render passes provide the implementation an opportunity to optimize the storage and transfer of attachment data between subpasses.
In practice, this means that subpasses with a simple framebuffer-space dependency may be merged into a single tiled rendering pass, keeping the attachment data on-chip for the duration of a render pass instance. However, it is also quite common for a render pass to only contain a single subpass.

Subpass dependencies describe execution and memory dependencies between subpasses.

A subpass dependency chain is a sequence of subpass dependencies in a render pass, where the source subpass of each subpass dependency (after the first) equals the destination subpass of the previous dependency.

Execution of subpasses may overlap or execute out of order with regards to other subpasses, unless otherwise enforced by an execution dependency. Each subpass only respects submission order for commands recorded in the same subpass, and the vkCmdBeginRenderPass and vkCmdEndRenderPass commands that delimit the render pass - commands within other subpasses are not included. This affects most other implicit ordering guarantees.

A render pass describes the structure of subpasses and attachments independent of any specific image views for the attachments. The specific image views that will be used for the attachments, and their dimensions, are specified in VkFramebuffer objects. Framebuffers are created with respect to a specific render pass that the framebuffer is compatible with (see Render Pass Compatibility). Collectively, a render pass and a framebuffer define the complete render target state for one or more subpasses as well as the algorithmic dependencies between the subpasses.

The various pipeline stages of the drawing commands for a given subpass may execute concurrently and/or out of order, both within and across drawing commands, whilst still respecting pipeline order. However for a given (x,y,layer,sample) sample location, certain per-sample operations are performed in rasterization order.

VK_ATTACHMENT_UNUSED is a constant indicating that a render pass attachment is not used.

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

### 8.1. Render Pass Creation

To create a render pass, call:

```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.

- \texttt{pAllocator} controls host memory allocation as described in the Memory Allocation chapter.
- \texttt{pRenderPass} is a pointer to a \texttt{VkRenderPass} handle in which the resulting render pass object is returned.

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

### Valid Usage

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

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

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

### Valid Usage (Implicit)

- \textbf{VUID-vkCreateRenderPass-device-parameter}
  
  \texttt{device} must be a valid \texttt{VkDevice} handle.

- \textbf{VUID-vkCreateRenderPass-pCreateInfo-parameter}
  
  \texttt{pCreateInfo} must be a valid pointer to a valid \texttt{VkRenderPassCreateInfo} structure.

- \textbf{VUID-vkCreateRenderPass-pAllocator-null}
  
  \texttt{pAllocator} must be \texttt{NULL}.

- \textbf{VUID-vkCreateRenderPass-pRenderPass-parameter}
  
  \texttt{pRenderPass} must be a valid pointer to a \texttt{VkRenderPass} handle.

### Return Codes

**Success**

- \texttt{VK_SUCCESS}
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 the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **attachmentCount** is the number of attachments used by this render pass.
- **pAttachments** is a pointer to an array of `attachmentCount` `VkAttachmentDescription` structures describing the attachments used by the render pass.
- **subpassCount** is the number of subpasses to create.
- **pSubpasses** is a pointer to an array of `subpassCount` `VkSubpassDescription` structures describing each subpass.
- **dependencyCount** is the number of memory dependencies between pairs of subpasses.
- **pDependencies** is a pointer to an array of `dependencyCount` `VkSubpassDependency` structures describing dependencies between pairs of subpasses.

**Note**

Care should be taken to avoid a data race here; if any subpasses access attachments with overlapping memory locations, and one of those accesses is a write, a subpass dependency needs to be included between them.

**Valid Usage**

- **VUID-VkRenderPassCreateInfo-attachment-00834**
  If the `attachment` member of any element of `pInputAttachments`, `pColorAttachments`, `pResolveAttachments` or `pDepthStencilAttachment`, or any element of `pPreserveAttachments` in any element of `pSubpasses` is not `VK_ATTACHMENT_UNUSED`, then it **must** be less than
For any member of `pAttachments` with a `loadOp` equal to `VK_ATTACHMENT_LOAD_OP_CLEAR`, the first use of that attachment must not specify a layout equal to `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`.

For any member of `pAttachments` with a `stencilLoadOp` equal to `VK_ATTACHMENT_LOAD_OP_CLEAR`, the first use of that attachment must not specify a layout equal to `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`.

For any member of `pAttachments` with a `loadOp` equal to `VK_ATTACHMENT_LOAD_OP_CLEAR`, the first use of that attachment must not specify a layout equal to `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`.

For any member of `pAttachments` with a `stencilLoadOp` equal to `VK_ATTACHMENT_LOAD_OP_CLEAR`, the first use of that attachment must not specify a layout equal to `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`.

If the `pNext` chain includes a `VkRenderPassInputAttachmentAspectCreateInfo` structure, the subpass member of each element of its `pAspectReferences` member must be less than `subpassCount`.

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.

If the `pNext` chain includes a `VkRenderPassMultiviewCreateInfo` structure, and its `subpassCount` member is not zero, that member must be equal to the value of `subpassCount`.

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

If the `pNext` chain includes a `VkRenderPassMultiviewCreateInfo` structure, for each non-zero element of `pViewOffsets`, the `srcSubpass` and `dstSubpass` members of `pDependencies` at
the same index must not be equal

- 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-srcSubpass-02517
  The srcSubpass member of each element of pDependencies must be less than subpassCount

- VUID-VkRenderPassCreateInfo-dstSubpass-02518
  The dstSubpass member of each element of pDependencies 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

---

```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;
    const void* pNext;
    uint32_t subpassCount;
    const uint32_t* pViewMasks;
    uint32_t dependencyCount;
    const int32_t* pViewOffsets;
    uint32_t correlationMaskCount;
    const uint32_t* pCorrelationMasks;
} VkRenderPassMultiviewCreateInfo;
```

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

• **subpassCount** is zero or the number of subpasses in the render pass.

• **pViewMasks** is a pointer to an array of **subpassCount** view masks, where each mask is a bitfield of view indices describing which views rendering is broadcast to in each subpass, when multiview is enabled. If **subpassCount** is zero, each view mask is treated as zero.

• **dependencyCount** is zero or the number of dependencies in the render pass.

• **pViewOffsets** is a pointer to an array of **dependencyCount** view offsets, one for each dependency. If **dependencyCount** is zero, each dependency’s view offset is treated as zero. Each view offset controls which views in the source subpass the views in the destination subpass depend on.

• **correlationMaskCount** is zero or the number of correlation masks.

• **pCorrelationMasks** is a pointer to an array of **correlationMaskCount** view masks indicating sets of views that may be more efficient to render concurrently.

When a subpass uses a non-zero view mask, **multiview** functionality is considered to be enabled. Multiview is all-or-nothing for a render pass - that is, either all subpasses must have a non-zero view mask (though some subpasses may have only one view) or all must be zero. Multiview causes all drawing and clear commands in the subpass to behave as if they were broadcast to each view, where a view is represented by one layer of the framebuffer attachments. All draws and clears are broadcast to each **view index** whose bit is set in the view mask. The view index is provided in the **ViewIndex** shader input variable, and color, depth/stencil, and input attachments all read/write the layer of the framebuffer corresponding to the view index.

If the view mask is zero for all subpasses, multiview is considered to be disabled and all drawing commands execute normally, without this additional broadcasting.

Some implementations may not support multiview in conjunction with geometry shaders or tessellation shaders.

When multiview is enabled, the **VK_DEPENDENCY_VIEW_LOCAL_BIT** bit in a dependency can be used to express a view-local dependency, meaning that each view in the destination subpass depends on a single view in the source subpass. Unlike pipeline barriers, a subpass dependency can potentially have a different view mask in the source subpass and the destination subpass. If the dependency is view-local, then each view (dstView) in the destination subpass depends on the view dstView + **pViewOffsets**[dependency] in the source subpass. If there is not such a view in the source subpass, then this dependency does not affect that view in the destination subpass. If the dependency is not view-local, then all views in the destination subpass depend on all views in the source subpass, and the view offset is ignored. A non-zero view offset is not allowed in a self-dependency.

The elements of **pCorrelationMasks** are a set of masks of views indicating that views in the same mask may exhibit spatial coherency between the views, making it more efficient to render them concurrently. Correlation masks must not have a functional effect on the results of the multiview rendering.

When multiview is enabled, at the beginning of each subpass all non-render pass state is undefined. In particular, each time **vkCmdBeginRenderPass** or **vkCmdNextSubpass** is called the graphics pipeline must be bound, any relevant descriptor sets or vertex/index buffers must be bound, and any relevant dynamic state or push constants must be set before they are used.
Valid Usage

- VUID-VkRenderPassMultiviewCreateInfo-pCorrelationMasks-00841
  Each view index must not be set in more than one element of pCorrelationMasks

Valid Usage (Implicit)

- VUID-VkRenderPassMultiviewCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO

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

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

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

The VkAttachmentDescription structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkAttachmentDescription {
    VkAttachmentDescriptionFlags flags;
    VkFormat format;
    VkSampleCountFlagBits samples;
    VkAttachmentLoadOp loadOp;
    VkAttachmentStoreOp storeOp;
    VkAttachmentLoadOp stencilLoadOp;
    VkAttachmentStoreOp stencilStoreOp;
    VkImageLayout initialLayout;
    VkImageLayout finallayout;
} VkAttachmentDescription;
```

- flags is a bitmask of VkAttachmentDescriptionFlagBits specifying additional properties of the attachment.
- format is a VkFormat value specifying the format of the image view that will be used for the attachment.
- samples is a VkSampleCountFlagBits value specifying the number of samples of the image.
- loadOp is a VkAttachmentLoadOp value specifying how the contents of color and depth components of the attachment are treated at the beginning of the subpass where it is first used.
- storeOp is a VkAttachmentStoreOp value specifying how the contents of color and depth components of the attachment are treated at the end of the subpass where it is last used.
• stencilLoadOp is a VkAttachmentLoadOp value specifying how the contents of stencil components of the attachment are treated at the beginning of the subpass where it is first used.

• stencilStoreOp is a VkAttachmentStoreOp value specifying how the contents of stencil components of the attachment are treated at the end of the last subpass where it is used.

• initialLayout is the layout the attachment image subresource will be in when a render pass instance begins.

• finalLayout is the layout the attachment image subresource will be transitioned to when a render pass instance ends.

If the attachment uses a color format, then loadOp and storeOp are used, and stencilLoadOp and stencilStoreOp are ignored. If the format has depth and/or stencil components, loadOp and storeOp apply only to the depth data, while stencilLoadOp and stencilStoreOp define how the stencil data is handled. loadOp and stencilLoadOp define the load operations that execute as part of the first subpass that uses the attachment. storeOp and stencilStoreOp define the store operations that execute as part of the last subpass that uses the attachment.

The load operation for each sample in an attachment happens-before any recorded command which accesses the sample in the first subpass where the attachment is used. Load operations for attachments with a depth/stencil format execute in the VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT pipeline stage. Load operations for attachments with a color format execute in the VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT pipeline stage.

The store operation for each sample in an attachment happens-after any recorded command which accesses the sample in the last subpass where the attachment is used. Store operations for attachments with a depth/stencil format execute in the VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT pipeline stage. Store operations for attachments with a color format execute in the VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT pipeline stage.

If an attachment is not used by any subpass, then loadOp, storeOp, stencilStoreOp, and stencilLoadOp are ignored, and the attachment's memory contents will not be modified by execution of a render pass instance.

The load and store operations apply on the first and last use of each view in the render pass, respectively. If a view index of an attachment is not included in the view mask in any subpass that uses it, then the load and store operations are ignored, and the attachment's memory contents will not be modified by execution of a render pass instance.

During a render pass instance, input/color attachments with color formats that have a component size of 8, 16, or 32 bits must be represented in the attachment's format throughout the instance. Attachments with other floating- or fixed-point color formats, or with depth components may be represented in a format with a precision higher than the attachment format, but must be represented with the same range. When such a component is loaded via the loadOp, it will be converted into an implementation-dependent format used by the render pass. Such components must be converted from the render pass format, to the format of the attachment, before they are resolved or stored at the end of a render pass instance via storeOp. Conversions occur as described in Numeric Representation and Computation and Fixed-Point Data Conversions.

If flags includes VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT, then the attachment is treated as if it
shares physical memory with another attachment in the same render pass. This information limits
the ability of the implementation to reorder certain operations (like layout transitions and the
loadOp) such that it is not improperly reordered against other uses of the same physical memory via
different attachment. This is described in more detail below.

If a render pass uses multiple attachments that alias the same device memory, those attachments
must each include the VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT bit in their attachment description
flags. Attachments aliasing the same memory occurs in multiple ways:

- Multiple attachments being assigned the same image view as part of framebuffer creation.
- Attachments using distinct image views that correspond to the same image subresource of an
  image.
- Attachments using views of distinct image subresources which are bound to overlapping
  memory ranges.

Note
Render passes must include subpass dependencies (either directly or via a subpass
dependency chain) between any two subpasses that operate on the same
attachment or aliasing attachments and those subpass dependencies must include
execution and memory dependencies separating uses of the aliases, if at least one
of those subpasses writes to one of the aliases. These dependencies must not
include the VK_DEPENDENCY_BY_REGION_BIT if the aliases are views of distinct image
subresources which overlap in memory.

Multiple attachments that alias the same memory must not be used in a single subpass. A given
attachment index must not be used multiple times in a single subpass, with one exception: two
subpass attachments can use the same attachment index if at least one use is as an input
attachment and neither use is as a resolve or preserve attachment. In other words, the same view
can be used simultaneously as an input and color or depth/stencil attachment, but must not be
used as multiple color or depth/stencil attachments nor as resolve or preserve attachments. The
precise set of valid scenarios is described in more detail below.

If a set of attachments alias each other, then all except the first to be used in the render pass must
use an initialLayout of VK_IMAGE_LAYOUT_UNDEFINED, since the earlier uses of the other aliases make
their contents undefined. Once an alias has been used and a different alias has been used after it,
the first alias must not be used in any later subpasses. However, an application can assign the same
image view to multiple aliasing attachment indices, which allows that image view to be used
multiple times even if other aliases are used in between.

Note
Once an attachment needs the VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT bit, there
should be no additional cost of introducing additional aliases, and using these
additional aliases may allow more efficient clearing of the attachments on
multiple uses via VK_ATTACHMENT_LOAD_OP_CLEAR.
Valid Usage

- VUID-VkAttachmentDescription-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-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-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-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-format-03283
  If format is a depth/stencil format, finalLayout must not be
  VK_IMAGE_LAYOUT_COLOR_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-03288
  If format is a depth/stencil format which includes both depth and stencil aspects, 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-03289
  If format is a depth/stencil format which includes both depth and stencil aspects, 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-03290
  If format is a depth/stencil format which includes only the depth aspect, initialLayout must not be VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL

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

- VUID-VkAttachmentDescription-format-03292
  If format is a depth/stencil format which includes only the stencil aspect, initialLayout must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL

- VUID-VkAttachmentDescription-format-03293
  If format is a depth/stencil format which includes only the stencil aspect, finalLayout must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL

Valid Usage (Implicit)

- VUID-VkAttachmentDescription-flags-parameter
  flags must be a valid combination of VkAttachmentDescriptionFlagBits values

- VUID-VkAttachmentDescription-format-parameter
  format must be a valid VkFormat value

- VUID-VkAttachmentDescription-samples-parameter
  samples must be a valid VkSampleCountFlagBits value

- VUID-VkAttachmentDescription-loadOp-parameter
  loadOp must be a valid VkAttachmentLoadOp value

- VUID-VkAttachmentDescription-storeOp-parameter
  storeOp must be a valid VkAttachmentStoreOp value
• **VUID-VkAttachmentDescription-stencilLoadOp-parameter**
  
  `stencilLoadOp` must be a valid `VkAttachmentLoadOp` value

• **VUID-VkAttachmentDescription-stencilStoreOp-parameter**
  
  `stencilStoreOp` must be a valid `VkAttachmentStoreOp` value

• **VUID-VkAttachmentDescription-initialLayout-parameter**
  
  `initialLayout` must be a valid `VkImageLayout` value

• **VUID-VkAttachmentDescription-finalLayout-parameter**
  
  `finalLayout` must be a valid `VkImageLayout` value

Bits which **can** be set in `VkAttachmentDescription::flags` describing additional properties of the attachment are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkAttachmentDescriptionFlagBits {
    VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT = 0x00000001,
} VkAttachmentDescriptionFlagBits;
```

• **VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT** specifies that the attachment aliases the same device memory as other attachments.

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

`VkAttachmentDescriptionFlags` is a bitmask type for setting a mask of zero or more `VkAttachmentDescriptionFlagBits`.

Possible values of `VkAttachmentDescription::loadOp` and `stencilLoadOp`, specifying how the contents of the attachment are treated, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkAttachmentLoadOp {
    VK_ATTACHMENT_LOAD_OP_LOAD = 0,
    VK_ATTACHMENT_LOAD_OP_CLEAR = 1,
    VK_ATTACHMENT_LOAD_OP_DONT_CARE = 2,
} VkAttachmentLoadOp;
```

• **VK_ATTACHMENT_LOAD_OP_LOAD** specifies that the previous contents of the image within the render area will be preserved. For attachments with a depth/stencil format, this uses the access type `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT`. For attachments with a color format, this uses the access type `VK_ACCESS_COLOR_ATTACHMENT_READ_BIT`.

• **VK_ATTACHMENT_LOAD_OP_CLEAR** specifies that the contents within the render area will be cleared to a uniform value, which is specified when a render pass instance is begun. For attachments with a depth/stencil format, this uses the access type `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`. For attachments with a color format, this uses the access type `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`. For attachments with a depth/stencil format, this uses the access type `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`.
VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT.

• VK_ATTACHMENT_LOAD_OP_DONT_CARE specifies that the previous contents within the area need not be preserved; the contents of the attachment will be undefined inside the render area. For attachments with a depth/stencil format, this uses the access type VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT. For attachments with a color format, this uses the access type VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT.

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

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

The VkRenderPassInputAttachmentAspectCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkRenderPassInputAttachmentAspectCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t aspectReferenceCount;
    const VkInputAttachmentAspectReference* pAspectReferences;
} VkRenderPassInputAttachmentAspectCreateInfo;
```

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• aspectReferenceCount is the number of elements in the pAspectReferences array.
• pAspectReferences is a pointer to an array of aspectReferenceCount
**VkInputAttachmentAspectReference** structures containing a mask describing which aspect(s) can be accessed for a given input attachment within a given subpass.

To specify which aspects of an input attachment can be read, add a **VkRenderPassInputAttachmentAspectCreateInfo** structure to the **pNext** chain of the **VkRenderPassCreateInfo** structure:

An application can access any aspect of an input attachment that does not have a specified aspect mask in the **pAspectReferences** array. Otherwise, an application must not access aspect(s) of an input attachment other than those in its specified aspect mask.

### Valid Usage (Implicit)

- **VUID-VkRenderPassInputAttachmentAspectCreateInfo-sType-sType**
  
  *sType must be VK_STRUCTURE_TYPE_RENDER_PASS_INPUT_ATTACHMENT_ASPECT_CREATE_INFO*

- **VUID-VkRenderPassInputAttachmentAspectCreateInfo-pAspectReferences-parameter**
  
  *pAspectReferences must be a valid pointer to an array of **aspectReferenceCount** valid **VkInputAttachmentAspectReference** structures*

- **VUID-VkRenderPassInputAttachmentAspectCreateInfo-aspectReferenceCount-arraylength**
  
  *aspectReferenceCount must be greater than 0*

The **VkInputAttachmentAspectReference** structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkInputAttachmentAspectReference {
    uint32_t subpass;
    uint32_t inputAttachmentIndex;
    VkImageAspectFlags aspectMask;
} VkInputAttachmentAspectReference;
```

- **subpass** is an index into the **pSubpasses** array of the parent **VkRenderPassCreateInfo** structure.
- **inputAttachmentIndex** is an index into the **pInputAttachments** of the specified subpass.
- **aspectMask** is a mask of which aspect(s) can be accessed within the specified subpass.

This structure specifies an aspect mask for a specific input attachment of a specific subpass in the render pass.

**subpass** and **inputAttachmentIndex** index into the render pass as:

```c
pCreateInfo->pSubpasses[subpass].pInputAttachments[inputAttachmentIndex]
```

### Valid Usage

- **VUID-VkInputAttachmentAspectReference-aspectMask-01964**
aspectMask must not include VK_IMAGE_ASPECT_METADATA_BIT

- VUID-VkInputAttachmentAspectReference-aspectMask-02250
  aspectMask must not include VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT for any index i

Valid Usage (Implicit)

- VUID-VkInputAttachmentAspectReference-aspectMask-parameter
  aspectMask must be a valid combination of VkImageAspectFlagBits values

- VUID-VkInputAttachmentAspectReference-aspectMask-required bitmask
  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`, or if **Color Write Enable** has been disabled for the corresponding attachment index, then writes to the corresponding location by a fragment shader are discarded.

If **pResolveAttachments** is not NULL, each of its elements corresponds to a color attachment (the element in **pColorAttachments** at the same index), and a multisample resolve operation is defined for each attachment. At the end of each subpass, multisample resolve operations read the subpass's color attachments, and resolve the samples for each pixel within the render area to the same pixel location in the corresponding resolve attachments, unless the resolve attachment index is `VK_ATTACHMENT_UNUSED`.

Similarly, if **VkSubpassDescriptionDepthStencilResolve**::`pDepthStencilResolveAttachment` is not NULL and does not have the value `VK_ATTACHMENT_UNUSED`, it corresponds to the depth/stencil attachment in `pDepthStencilAttachment`, and multisample resolve operations for depth and stencil are defined by **VkSubpassDescriptionDepthStencilResolve**::`depthResolveMode` and **VkSubpassDescriptionDepthStencilResolve**::`stencilResolveMode`, respectively. At the end of each subpass, multisample resolve operations read the subpass's depth/stencil attachment, and resolve the samples for each pixel to the same pixel location in the corresponding resolve attachment. If **VkSubpassDescriptionDepthStencilResolve**::`depthResolveMode` is `VK_RESOLVE_MODE_NONE`, then the depth component of the resolve attachment is not written to and its contents are preserved. Similarly, if **VkSubpassDescriptionDepthStencilResolve**::`stencilResolveMode` is `VK_RESOLVE_MODE_NONE`, then the stencil component of the resolve attachment is not written to and its contents are preserved. **VkSubpassDescriptionDepthStencilResolve**::`depthResolveMode` is ignored if the **VkFormat** of the `pDepthStencilResolveAttachment` does not have a depth component. Similarly, **VkSubpassDescriptionDepthStencilResolve**::`stencilResolveMode` is ignored if the **VkFormat** of the `pDepthStencilResolveAttachment` does not have a stencil component.

If the image subresource range referenced by the depth/stencil attachment is created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT`, then the multisample resolve operation uses the sample locations state specified in the `sampleLocationsInfo` member of the element of the **VkRenderPassSampleLocationsBeginInfoEXT**::`pPostSubpassSampleLocations` for the subpass.

If **pDepthStencilAttachment** is NULL, or if its attachment index is `VK_ATTACHMENT_UNUSED`, it indicates that
no depth/stencil attachment will be used in the subpass.

The contents of an attachment within the render area become undefined at the start of a subpass $S$ if all of the following conditions are true:

- The attachment is used as a color, depth/stencil, or resolve attachment in any subpass in the render pass.
- There is a subpass $S_1$ that uses or preserves the attachment, and a subpass dependency from $S_1$ to $S$.
- The attachment is not used or preserved in subpass $S$.

Once the contents of an attachment become undefined in subpass $S$, they remain undefined for subpasses in subpass dependency chains starting with subpass $S$ until they are written again. However, they remain valid for subpasses in other subpass dependency chains starting with subpass $S_1$ if those subpasses use or preserve the attachment.

### Valid Usage

- **VUID-VkSubpassDescription-pipelineBindPoint-00844**
  
  pipelineBindPoint must be VK_PIPELINE_BIND_POINT_GRAPHICS

- **VUID-VkSubpassDescription-colorAttachmentCount-00845**
  
  colorAttachmentCount must be less than or equal to VkPhysicalDeviceLimits::maxColorAttachments

- **VUID-VkSubpassDescription-loadOp-00846**
  
  If the first use of an attachment in this render pass is as an input attachment, and the attachment is not also used as a color or depth/stencil attachment in the same subpass, then loadOp must not be VK_ATTACHMENT_LOAD_OP_CLEAR

- **VUID-VkSubpassDescription-pResolveAttachments-00847**
  
  If pResolveAttachments is not NULL, for each resolve attachment that is not VK_ATTACHMENT_UNUSED, the corresponding color attachment must not be VK_ATTACHMENT_UNUSED

- **VUID-VkSubpassDescription-pResolveAttachments-00848**
  
  If pResolveAttachments is not NULL, for each resolve attachment that is not VK_ATTACHMENT_UNUSED, the corresponding color attachment must not have a sample count of VK_SAMPLE_COUNT_1_BIT

- **VUID-VkSubpassDescription-pResolveAttachments-00849**
  
  If pResolveAttachments is not NULL, each resolve attachment that is not VK_ATTACHMENT_UNUSED must have a sample count of VK_SAMPLE_COUNT_1_BIT

- **VUID-VkSubpassDescription-pResolveAttachments-00850**
  
  If pResolveAttachments is not NULL, each resolve attachment that is not VK_ATTACHMENT_UNUSED must have the same VgFormat as its corresponding color attachment

- **VUID-VkSubpassDescription-pColorAttachments-01417**
  
  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 neither the VK_AMD_mixed_attachment_samples nor the VK_NV_framebuffer_mixed_samples extensions are enabled, and if pDepthStencilAttachment is not VK_ATTACHMENT_UNUSED and any attachments in pColorAttachments are not VK_ATTACHMENT_UNUSED, they must have the same sample count

• VUID-VkSubpassDescription-attachment-00853
  Each element of pPreserveAttachments must not be VK_ATTACHMENT_UNUSED

• VUID-VkSubpassDescription-pPreserveAttachments-00854
  Each element of pPreserveAttachments must not also be an element of any other member of the subpass description

• VUID-VkSubpassDescription-layout-02519
  If any attachment is used by more than one VkAttachmentReference member, then each use must use the same layout

• VUID-VkSubpassDescription-None-04437
  Each attachment must follow the image layout requirements specified for its attachment type

• 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-preserveAttachmentCount-05054
  preserveAttachmentCount must be less than or equal to maxSubpassPreserveAttachments
Valid Usage (Implicit)

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

- **VUID-VkSubpassDescription-pipelineBindPoint-parameter**
  
  `pipelineBindPoint must be a valid VkPipelineBindPoint value`

- **VUID-VkSubpassDescription-pInputAttachments-parameter**
  
  If `inputAttachmentCount` is not 0, `pInputAttachments must be a valid pointer to an array of inputAttachmentCount valid VkAttachmentReference structures`

- **VUID-VkSubpassDescription-pColorAttachments-parameter**
  
  If `colorAttachmentCount` is not 0, `pColorAttachments must be a valid pointer to an array of colorAttachmentCount valid VkAttachmentReference structures`

- **VUID-VkSubpassDescription-pResolveAttachments-parameter**
  
  If `colorAttachmentCount` is not 0, and `pResolveAttachments` is not NULL, `pResolveAttachments must be a valid pointer to an array of colorAttachmentCount valid VkAttachmentReference structures`

- **VUID-VkSubpassDescription-pDepthStencilAttachment-parameter**
  
  If `pDepthStencilAttachment` is not NULL, `pDepthStencilAttachment must be a valid pointer to a valid VkAttachmentReference structure`

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

Bits which **can be set** in `VkSubpassDescription::flags`, specifying usage of the subpass, are:

```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-00857
  If attachment is not VK_ATTACHMENT_UNUSED, layout must not be VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_PREINITIALIZED, VK_IMAGE_LAYOUT_PRESENT_SRC_KHR, VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, or VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL.

### Valid Usage (Implicit)

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

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

#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
• **srcStageMask** is a bitmask of `VkPipelineStageFlagBits` specifying the source stage mask.
• **dstStageMask** is a bitmask of `VkPipelineStageFlagBits` specifying the destination stage mask.
• **srcAccessMask** is a bitmask of `VkAccessFlagBits` specifying a source access mask.
• **dstAccessMask** is a bitmask of `VkAccessFlagBits` specifying a destination access mask.
• **dependencyFlags** is a bitmask of `VkDependencyFlagBits`.

If **srcSubpass** is equal to **dstSubpass** then the **VkSubpassDependency** describes a subpass self-dependency, and only constrains the pipeline barriers allowed within a subpass instance. Otherwise, when a render pass instance which includes a subpass dependency is submitted to a queue, it defines a memory dependency between the subpasses identified by **srcSubpass** and **dstSubpass**.

If **srcSubpass** is equal to **VK_SUBPASS_EXTERNAL**, the first synchronization scope includes commands that occur earlier in submission order than the `vkCmdBeginRenderPass` used to begin the render pass instance. Otherwise, the first set of commands includes all commands submitted as part of the subpass instance identified by **srcSubpass** and any load, store or multisample resolve operations on attachments used in **srcSubpass**. In either case, the first synchronization scope is limited to operations on the pipeline stages determined by the source stage mask specified by **srcStageMask**.

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

The first access scope is limited to 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 *geometry shaders* feature is not enabled, *srcStageMask* must not contain *VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT*

- **VUID-VkSubpassDependency-srcStageMask-04091**
  If the *tessellation shaders* feature is not enabled, *srcStageMask* must not contain *VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT* or *VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT*

- **VUID-VkSubpassDependency-srcStageMask-03937**
  If the *synchronization2* feature is not enabled, *srcStageMask* must not be 0

- **VUID-VkSubpassDependency-dstStageMask-04090**
  If the *geometry shaders* feature is not enabled, *dstStageMask* must not contain *VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT*

- **VUID-VkSubpassDependency-dstStageMask-04091**
  If the *tessellation shaders* feature is not enabled, *dstStageMask* must not contain *VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT* or *VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT*

- **VUID-VkSubpassDependency-dstStageMask-03937**
  If the *synchronization2* feature is not enabled, *dstStageMask* must not be 0

- **VUID-VkSubpassDependency-srcSubpass-00864**
  *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-VkSubpassDependency-srcSubpass-00865**
  *srcSubpass* and *dstSubpass* must not both be equal to *VK_SUBPASS_EXTERNAL*

- **VUID-VkSubpassDependency-srcSubpass-00867**
  If *srcSubpass* is equal to *dstSubpass* and not all of the stages in *srcStageMask* and *dstStageMask* are framebuffer-space stages, the logically latest pipeline stage in *srcStageMask* must be logically earlier than or equal to the logically earliest pipeline stage in *dstStageMask*

- **VUID-VkSubpassDependency-srcAccessMask-00868**
  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-VkSubpassDependency-dstAccessMask-00869**
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-VkSubpassDependency-srcSubpass-02243**
  If `srcSubpass` equals `dstSubpass`, and `srcStageMask` and `dstStageMask` both include a framebuffer-space stage, then `dependencyFlags` must include `VK_DEPENDENCY_BY_REGION_BIT`.

- **VUID-VkSubpassDependency-dstStageMask-02520**
  If `dependencyFlags` includes `VK_DEPENDENCY_VIEW_LOCAL_BIT`, `srcSubpass` must not be equal to `VK_SUBPASS_EXTERNAL`.

- **VUID-VkSubpassDependency-dstSubpass-02521**
  If `dependencyFlags` includes `VK_DEPENDENCY_VIEW_LOCAL_BIT`, `dstSubpass` must not be equal to `VK_SUBPASS_EXTERNAL`.

- **VUID-VkSubpassDependency-srcSubpass-00872**
  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

350
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.

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:

```
VkSubpassDependency implicitDependency = {
    .srcSubpass = VK_SUBPASS_EXTERNAL;
    .dstSubpass = firstSubpass; // First subpass attachment is used in
    .srcStageMask = VK_PIPELINE_STAGE_NONE_KHR;
    .dstStageMask = VK_PIPELINE_STAGE_ALL_COMMANDS_BIT;
    .srcAccessMask = 0;
    .dstAccessMask = VK_ACCESS_INPUT_ATTACHMENT_READ_BIT |
                     VK_ACCESS_COLOR_ATTACHMENT_READ_BIT |
                     VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
                     VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT |
                     VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
    .dependencyFlags = 0;
};
```

Similarly, if there is no subpass dependency from the last subpass that uses an attachment to
VK_SUBPASS_EXTERNAL, then an implicit subpass dependency exists from the last subpass it is used in
to VK_SUBPASS_EXTERNAL. The implicit subpass dependency only exists if there exists an automatic
layout transition into finalLayout. The subpass dependency operates as if defined with the
following parameters:

```
VkSubpassDependency implicitDependency = {
    .srcSubpass = lastSubpass; // Last subpass attachment is used in
    .dstSubpass = VK_SUBPASS_EXTERNAL;
    .srcStageMask = VK_PIPELINE_STAGE_ALL_COMMANDS_BIT;
    .dstStageMask = VK_PIPELINE_STAGE_NONE_KHR;
    .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 the attachment view is a 2D or 2D array view of a 3D image, even if the attachment view only refers to a subset of the slices of the selected mip level of the 3D image, automatic layout transitions apply to the entire subresource referenced which is the entire mip level in this case.

Automatic layout transitions away from the layout used in a subpass happen-after the availability operations for all dependencies with that subpass as the *srcSubpass*.

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

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

Automatic layout transitions into *finalLayout* happen-before the visibility operations for all dependencies with a *dstSubpass* equal to VK_SUBPASS_EXTERNAL, where *srcSubpass* uses the attachment that will be transitioned. For attachments created with VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT, automatic layout transitions into *finalLayout* happen-before the visibility operations for all dependencies with a *dstSubpass* equal to VK_SUBPASS_EXTERNAL, where *srcSubpass* uses any aliased attachment.

The image layout of the depth aspect of a depth/stencil attachment referring to an image created with VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT is dependent on the last sample locations used to render to the attachment, thus automatic layout transitions use the sample locations state specified in VkRenderPassSampleLocationsBeginInfoEXT.

Automatic layout transitions of an attachment referring to a depth/stencil image created with VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT use the sample locations the image subresource range referenced by the attachment was last rendered with. If the current render pass does not use the attachment as a depth/stencil attachment in any subpass that happens-before, the automatic layout transition uses the sample locations state specified in the sampleLocationsInfo member of the element of the VkRenderPassSampleLocationsBeginInfoEXT::*pAttachmentInitialSampleLocations* array for which the attachmentIndex member equals the attachment index of the attachment, if one is specified. Otherwise, the automatic layout transition uses the sample locations state specified in the sampleLocationsInfo member of the element of the VkRenderPassSampleLocationsBeginInfoEXT::*pPostSubpassSampleLocations* array for which the subpassIndex member equals the index of the subpass that last used the attachment as a depth/stencil attachment, if one is specified.

If no sample locations state has been specified for an automatic layout transition performed on an attachment referring to a depth/stencil image created with
The contents of the depth aspect of the depth/stencil attachment become undefined as if the layout of the attachment was transitioned from the \texttt{VK\_IMAGE\_LAYOUT\_UNDEFINED} layout.

If two subpasses use the same attachment, and both subpasses use the attachment in a read-only layout, no subpass dependency needs to be specified between those subpasses. If an implementation treats those layouts separately, it \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:

```c
// Used for input attachments
VkPipelineStageFlags inputAttachmentStages = VK\_PIPELINE\_STAGE\_FRAGMENT\_SHADER\_BIT;
VkAccessFlags inputAttachmentDstAccess = VK\_ACCESS\_INPUT\_ATTACHMENT\_READ\_BIT;

// Used for depth/stencil attachments
VkPipelineStageFlags depthStencilAttachmentStages =
    VK\_PIPELINE\_STAGE\_EARLY\_FRAGMENT\_TESTS\_BIT |
    VK\_PIPELINE\_STAGE\_LATE\_FRAGMENT\_TESTS\_BIT;
VkAccessFlags depthStencilAttachmentDstAccess =
    VK\_ACCESS\_DEPTH\_STENCIL\_ATTACHMENT\_READ\_BIT;

VkSubpassDependency implicitDependency = {
    .srcSubpass = firstSubpass;
    .dstSubpass = secondSubpass;
    .srcStageMask = inputAttachmentStages | depthStencilAttachmentStages;
    .dstStageMask = inputAttachmentStages | depthStencilAttachmentStages;
    .srcAccessMask = 0;
    .dstAccessMask = inputAttachmentDstAccess | depthStencilAttachmentDstAccess;
    .dependencyFlags = 0;
};
```

If a subpass uses the same attachment as both an input attachment and either a color attachment or a depth/stencil attachment, writes via the color or depth/stencil attachment are not automatically made visible to reads via the input attachment, causing a \textit{feedback loop}, except in any of the following conditions:

- If the color components or depth/stencil components read by the input attachment are mutually exclusive with the components written by the color or depth/stencil attachments, then there is no feedback loop. This requires the graphics pipelines used by the subpass to disable writes to color components that are read as inputs via the \texttt{colorWriteEnable} or \texttt{colorWriteMask}, and to disable writes to depth/stencil components that are read as inputs via \texttt{depthWriteEnable} or \texttt{stencilTestEnable}.

- If the attachment is used as an input attachment and depth/stencil attachment only, and the depth/stencil attachment is not written to.

Rendering within a subpass containing a feedback loop creates a \textit{data race}, except in the following cases:

- If a memory dependency is inserted between when the attachment is written and when it is
subsequently read by later fragments. **Pipeline barriers** expressing a subpass self-dependency are the only way to achieve this, and one **must** be inserted every time a fragment will read values at a particular sample (x, y, layer, sample) coordinate, if those values have been written since the most recent pipeline barrier; or since the start of the subpass, if there have been no pipeline barriers since the start of the subpass.

Attachments have requirements for a valid image layout depending on the usage

- An attachment used as an input attachment **must** be in the `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_GENERAL` layout.

- An attachment used only as a color attachment **must** be in the `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR` or `VK_IMAGE_LAYOUT_GENERAL` layout.

- An attachment used as both an input attachment and a color attachment **must** be in the `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR` or `VK_IMAGE_LAYOUT_GENERAL` layout.

- An attachment used only as a depth/stencil attachment **must** be in the `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_GENERAL` layout.

- An attachment used as an input attachment and depth/stencil attachment **must** be in the `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_GENERAL` layout.

An attachment **must** not be used as both a depth/stencil attachment and a color attachment.

A more extensible version of render pass creation is also defined below.

To create a render pass, call:

```c
// Provided by VK_VERSION_1_2
VkResult vkCreateRenderPass2(
    VkDevice device,
    const VkRenderPassCreateInfo2* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkRenderPass* pRenderPass);
```

- **device** is the logical device that creates the render pass.
• **pCreateInfo** is a pointer to a `VkRenderPassCreateInfo2` structure describing the parameters of the render pass.

• **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

• **pRenderPass** is a pointer to a `VkRenderPass` handle in which the resulting render pass object is returned.

This command is functionally identical to `vkCreateRenderPass`, but includes extensible sub-structures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

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 the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `attachmentCount` is the number of attachments used by this render pass.
- `pAttachments` is a pointer to an array of `attachmentCount` `VkAttachmentDescription2` structures describing the attachments used by the render pass.
- `subpassCount` is the number of subpasses to create.
- `pSubpasses` is a pointer to an array of `subpassCount` `VkSubpassDescription2` structures describing each subpass.
- `dependencyCount` is the number of dependencies between pairs of subpasses.
- `pDependencies` is a pointer to an array of `dependencyCount` `VkSubpassDependency2` structures describing dependencies between pairs of subpasses.
- `correlatedViewMaskCount` is the number of correlation masks.
- `pCorrelatedViewMasks` is a pointer to an array of view masks indicating sets of views that **may** be more efficient to render concurrently.

Parameters defined by this structure with the same name as those in `VkRenderPassCreateInfo` have
the identical effect to those parameters; the child structures are variants of those used in VkRenderPassCreateInfo which add sType and pNext parameters, allowing them to be extended.

If the VkSubpassDescription2::viewMask member of any element of pSubpasses is not zero, multiview functionality is considered to be enabled for this render pass.

correlatedViewMaskCount and pCorrelatedViewMasks have the same effect as VkRenderPassMultiviewCreateInfo::correlationMaskCount and VkRenderPassMultiviewCreateInfo::pCorrelationMasks, respectively.

Valid Usage

• VUID-VkRenderPassCreateInfo2-None-03049
  If any two subpasses operate on attachments with overlapping ranges of the same VkDeviceMemory object, and at least one subpass writes to that area of VkDeviceMemory, a subpass dependency must be included (either directly or via some intermediate subpasses) between them

• VUID-VkRenderPassCreateInfo2-attachment-03050
  If the attachment member of any element of pInputAttachments, pColorAttachments, pResolveAttachments or pDepthStencilAttachment, or the attachment indexed by any element of pPreserveAttachments in any given element of pSubpasses is bound to a range of a VkDeviceMemory object that overlaps with any other attachment in any subpass (including the same subpass), the VkAttachmentDescription2 structures describing them must include VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT in flags

• VUID-VkRenderPassCreateInfo2-attachment-03051
  If the attachment member of any element of pInputAttachments, pColorAttachments, pResolveAttachments or pDepthStencilAttachment, or any element of pPreserveAttachments in any given element of pSubpasses is not VK_ATTACHMENT_UNUSED, then it must be less than attachmentCount

• VUID-VkRenderPassCreateInfo2-pSubpasses-06473
  If the pSubpasses pNext chain includes a VkSubpassDescriptionDepthStencilResolve structure and the pDepthStencilResolveAttachment member is not NULL and does not have the value VK_ATTACHMENT_UNUSED, then attachment must be less than attachmentCount

• VUID-VkRenderPassCreateInfo2-pAttachments-02522
  For any member of pAttachments with a loadOp equal to VK_ATTACHMENT_LOAD_OP_CLEAR, the first use of that attachment must not specify a layout equal to VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL, or VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL

• VUID-VkRenderPassCreateInfo2-pAttachments-02523
  For any member of pAttachments with a stencilLoadOp equal to VK_ATTACHMENT_LOAD_OP_CLEAR, the first use of that attachment must not specify a layout equal to VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL, or VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL
For any element of `pDependencies`, if the `srcSubpass` is not `VK_SUBPASS_EXTERNAL`, all stage flags included in the `srcStageMask` member of that dependency must be a pipeline stage supported by the pipeline identified by the `pipelineBindPoint` member of the source subpass.

For any element of `pDependencies`, if the `dstSubpass` is not `VK_SUBPASS_EXTERNAL`, all stage flags included in the `dstStageMask` member of that dependency must be a pipeline stage supported by the pipeline identified by the `pipelineBindPoint` member of the destination subpass.

The set of bits included in any element of `pCorrelatedViewMasks` must not overlap with the set of bits included in any other element of `pCorrelatedViewMasks`.

If the `VkSubpassDescription2::viewMask` member of all elements of `pSubpasses` is 0, `correlatedViewMaskCount` must be 0.

The `VkSubpassDescription2::viewMask` member of all elements of `pSubpasses` must either all be 0, or all not be 0.

If the `VkSubpassDescription2::viewMask` member of all elements of `pSubpasses` is 0, the `dependencyFlags` member of any element of `pDependencies` must not include `VK_DEPENDENCY_VIEW_LOCAL_BIT`.

For any element of `pDependencies` where its `srcSubpass` member equals its `dstSubpass` member, if the `viewMask` member of the corresponding element of `pSubpasses` includes more than one bit, its `dependencyFlags` member must include `VK_DEPENDENCY_VIEW_LOCAL_BIT`.

If the `attachment` member of any element of the `pInputAttachments` member of any element of `pSubpasses` is not `VK_ATTACHMENT_UNUSED`, the `aspectMask` member of that element of `pInputAttachments` must only include aspects that are present in images of the format specified by the element of `pAttachments` specified by `attachment`.

The `srcSubpass` member of each element of `pDependencies` must be less than `subpassCount`.

The `dstSubpass` member of each element of `pDependencies` must be less than `subpassCount`.

If any element of `pAttachments` is used as a fragment shading rate attachment in any subpass, it must not be used as any other attachment in the render pass.

If any element of `pAttachments` is used as a fragment shading rate attachment in any subpass, it must have an image format whose potential format features contain
VK_FORMAT_FEATURE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR

- 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

Valid Usage (Implicit)

- VUID-VkRenderPassCreateInfo2-sType-sType
  sType must be VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO_2
- VUID-VkRenderPassCreateInfo2-pNext-pNext
  pNext must be NULL
- VUID-VkRenderPassCreateInfo2-flags-zerobitmask
  flags must be 0
- VUID-VkRenderPassCreateInfo2-pAttachments-parameter
  If attachmentCount is not 0, pAttachments must be a valid pointer to an array of attachmentCount valid VkAttachmentDescription2 structures
- VUID-VkRenderPassCreateInfo2-pSubpasses-parameter
  pSubpasses must be a valid pointer to an array of subpassCount valid VkSubpassDescription2 structures
- VUID-VkRenderPassCreateInfo2-pDependencies-parameter
  If dependencyCount is not 0, pDependencies must be a valid pointer to an array of dependencyCount valid VkSubpassDependency2 structures
- VUID-VkRenderPassCreateInfo2-pCorrelatedViewMasks-parameter
  If correlatedViewMaskCount is not 0, pCorrelatedViewMasks must be a valid pointer to an array of correlatedViewMaskCount uint32_t values
- VUID-VkRenderPassCreateInfo2-subpassCount-arraylength
  subpassCount must be greater than 0

The VkAttachmentDescription2 structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkAttachmentDescription2 {
    VkStructureType sType;
    const void* pNext;
    VkAttachmentDescriptionFlags flags;
    VkFormat format;
    VkSampleCountFlagBits samples;
    VkAttachmentLoadOp loadOp;
    VkAttachmentStoreOp storeOp;
} VkAttachmentDescription2;
```
VkAttachmentLoadOp stencilLoadOp;
VkAttachmentStoreOp stencilStoreOp;
VkImageLayout initialLayout;
VkImageLayout finalLayout;
}

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• flags is a bitmask of VkAttachmentDescriptionFlagBits specifying additional properties of the attachment.
• format is a VkFormat value specifying the format of the image that will be used for the attachment.
• samples is a VkSampleCountFlagBits value specifying the number of samples of the image.
• loadOp is a VkAttachmentLoadOp value specifying how the contents of color and depth components of the attachment are treated at the beginning of the subpass where it is first used.
• storeOp is a VkAttachmentStoreOp value specifying how the contents of color and depth components of the attachment are treated at the end of the subpass where it is last used.
• stencilLoadOp is a VkAttachmentLoadOp value specifying how the contents of stencil components of the attachment are treated at the beginning of the subpass where it is first used.
• stencilStoreOp is a VkAttachmentStoreOp value specifying how the contents of stencil components of the attachment are treated at the end of the last subpass where it is used.
• initialLayout is the layout the attachment image subresource will be in when a render pass instance begins.
• finalLayout is the layout the attachment image subresource will be transitioned to when a render pass instance ends.

Parameters defined by this structure with the same name as those in VkAttachmentDescription have the identical effect to those parameters.

If the separateDepthStencilLayouts feature is enabled, and format is a depth/stencil format, initialLayout and finalLayout can be set to a layout that only specifies the layout of the depth aspect.

If the pNext chain includes a VkAttachmentDescriptionStencilLayout structure, then the stencilInitialLayout and stencilFinalLayout members specify the initial and final layouts of the stencil aspect of a depth/stencil format, and initialLayout and finalLayout only apply to the depth aspect. For depth-only formats, the VkAttachmentDescriptionStencilLayout structure is ignored. For stencil-only formats, the initial and final layouts of the stencil aspect are taken from the VkAttachmentDescriptionStencilLayout structure if present, or initialLayout and finalLayout if not present.

If 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-finalLayout-03061
  finalLayout must not be VK_IMAGE_LAYOUT_UNDEFINED or VK_IMAGE_LAYOUT_PREINITIALIZED

- VUID-VkAttachmentDescription2-format-03294
  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,
  VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL,
  or VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL

- VUID-VkAttachmentDescription2-format-03295
  If format is a depth/stencil format, initialLayout must not be
  VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL

- VUID-VkAttachmentDescription2-format-03296
  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,
  VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL,
  or VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL

- VUID-VkAttachmentDescription2-format-03297
  If format is a depth/stencil format, finalLayout must not be
  VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL

- VUID-VkAttachmentDescription2-separateDepthStencilLayouts-03298
  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-VkAttachmentDescription2-separateDepthStencilLayouts-03299
  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-VkAttachmentDescription2-format-03300
  If format is a color format, initialLayout must not be
  VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL,
  or VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL

- VUID-VkAttachmentDescription2-format-03301
  If format is a color format, finalLayout must not be
  VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL,
  or VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL

- VUID-VkAttachmentDescription2-format-03302
  If format is a depth/stencil format which includes both depth and stencil aspects, and
If `initialLayout` is `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, the `pNext` chain must include a `VkAttachmentDescriptionStencilLayout` structure.

- **VUID-VkAttachmentDescription2-format-03303**
  If `format` is a depth/stencil format which includes both depth and stencil aspects, and `finalLayout` is `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, the `pNext` chain must include a `VkAttachmentDescriptionStencilLayout` structure.

- **VUID-VkAttachmentDescription2-format-03304**
  If `format` is a depth/stencil format which includes only the depth aspect, `initialLayout` must not be `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`.

- **VUID-VkAttachmentDescription2-format-03305**
  If `format` is a depth/stencil format which includes only the depth aspect, `finalLayout` must not be `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`.

- **VUID-VkAttachmentDescription2-format-03306**
  If `format` is a depth/stencil format which includes only the stencil aspect, `initialLayout` must not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`.

- **VUID-VkAttachmentDescription2-format-03307**
  If `format` is a depth/stencil format which includes only the stencil aspect, `finalLayout` must not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`.

**Valid Usage (Implicit)**

- **VUID-VkAttachmentDescription2-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_2`.

- **VUID-VkAttachmentDescription2-pNext-pNext**
  `pNext` must be `NULL` or a pointer to a valid instance of `VkAttachmentDescriptionStencilLayout`.

- **VUID-VkAttachmentDescription2-sType-unique**
  The `sType` value of each struct in the `pNext` chain must be unique.

- **VUID-VkAttachmentDescription2-flags-parameter**
  `flags` must be a valid combination of `VkAttachmentDescriptionFlagBits` values.

- **VUID-VkAttachmentDescription2-format-parameter**
  `format` must be a valid `VkFormat` value.

- **VUID-VkAttachmentDescription2-samples-parameter**
  `samples` must be a valid `VkSampleCountFlagBits` value.

- **VUID-VkAttachmentDescription2-loadOp-parameter**
  `loadOp` must be a valid `VkAttachmentLoadOp` value.

362
• 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 the type of this structure.

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

• `stencilInitialLayout` is the layout the stencil aspect of the attachment image subresource will be in when a render pass instance begins.

• `stencilFinalLayout` is the layout the stencil aspect of the attachment image subresource will be transitioned to when a render pass instance ends.

**Valid Usage**

• VUID-VkAttachmentDescriptionStencilLayout-stencilInitialLayout-03308  
  `stencilInitialLayout` **must** not be `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL` or

• 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`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_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-stencilInitialStateLayout-parameter
  stencilInitialStateLayout 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;
    const VkAttachmentReference2* colorAttachmentCount;
    const VkAttachmentReference2* pColorAttachments;
    const VkAttachmentReference2* pResolveAttachments;
    const VkAttachmentReference2* pDepthStencilAttachment;
    const uint32_t preserveAttachmentCount;
    const uint32_t* pPreserveAttachments;
} VkSubpassDescription2;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `flags` is a bitmask of VkSubpassDescriptionFlagBits specifying usage of the subpass.
- `pipelineBindPoint` is a VkPipelineBindPoint value specifying the pipeline type supported for this subpass.
- `viewMask` is a bitfield of view indices describing which views rendering is broadcast to in this subpass, when multiview is enabled.
- `inputAttachmentCount` is the number of input attachments.
- `pInputAttachments` is a pointer to an array of VkAttachmentReference2 structures defining the input attachments for this subpass and their layouts.
- `colorAttachmentCount` is the number of color attachments.
- `pColorAttachments` is a pointer to an array of `colorAttachmentCount` `VkAttachmentReference2` structures defining the color attachments for this subpass and their layouts.
- `pResolveAttachments` is `NULL` or a pointer to an array of `colorAttachmentCount` `VkAttachmentReference2` structures defining the resolve attachments for this subpass and their layouts.
- `pDepthStencilAttachment` is a pointer to a `VkAttachmentReference2` structure specifying the depth/stencil attachment for this subpass and its layout.
- `preserveAttachmentCount` is the number of preserved attachments.
- `pPreserveAttachments` is a pointer to an array of `preserveAttachmentCount` render pass attachment indices identifying attachments that are not used by this subpass, but whose contents must be preserved throughout the subpass.

Parameters defined by this structure with the same name as those in `VkSubpassDescription` have the identical effect to those parameters.

`viewMask` has the same effect for the described subpass as `VkRenderPassMultiviewCreateInfo`::`pViewMasks` has on each corresponding subpass.

If a `VkFragmentShadingRateAttachmentInfoKHR` structure is included in the `pNext` chain, `pFragmentShadingRateAttachment` is not `NULL`, and its `attachment` member is not `VK_ATTACHMENT_UNUSED`, the identified attachment defines a fragment shading rate attachment for that subpass.

### Valid Usage

- VUID-VkSubpassDescription2-pipelineBindPoint-03062
  - `pipelineBindPoint` must be `VK_PIPELINE_BIND_POINT_GRAPHICS`

- VUID-VkSubpassDescription2-colorAttachmentCount-03063
  - `colorAttachmentCount` must be less than or equal to `VkPhysicalDeviceLimits`::`maxColorAttachments`

- VUID-VkSubpassDescription2-loadOp-03064
  - 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-VkSubpassDescription2-pResolveAttachments-03065
  - 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-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-pResolveAttachments-03068
  Any given element of pResolveAttachments must have the same VkFormat as its corresponding color attachment

- VUID-VkSubpassDescription2-pColorAttachments-03069
  All attachments in pColorAttachments that are not VK_ATTACHMENT_UNUSED must have the same sample count

- 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

- VUID-VkSubpassDescription2-pResolveAttachments-02899
  All attachments in pResolveAttachments that are not VK_ATTACHMENT_UNUSED must have image formats whose potential format features contain

- 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-pDepthStencilAttachment-03071
  If neither the VK_AMD_mixed_attachment_samples nor the VK_NV_framebuffer_mixed_samples extensions are enabled, and if pDepthStencilAttachment is not VK_ATTACHMENT_UNUSED and any attachments in pColorAttachments are not VK_ATTACHMENT_UNUSED, they must have the same sample count

- VUID-VkSubpassDescription2-attachment-03073
  Each element of pPreserveAttachments must not be VK_ATTACHMENT_UNUSED

- VUID-VkSubpassDescription2-pPreserveAttachments-03074
  Any given element of pPreserveAttachments must not also be an element of any other member of the subpass description

- VUID-VkSubpassDescription2-layout-02528
  If any attachment is used by more than one VkAttachmentReference2 member, then each use must use the same layout

- VUID-VkSubpassDescription2-None-04439
  Attachments must follow the image layout requirements based on the type of attachment it is being used as

- 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-attachment-04563
  If the attachment member of any element of pInputAttachments is not VK_ATTACHMENT_UNUSED, then the aspectMask member must not include VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT for any index i

- 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

Valid Usage (Implicit)

- VUID-VkSubpassDescription2-sType-sType
  sType must be VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_2

- VUID-VkSubpassDescription2-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 VkFragmentShadingRateAttachmentInfoKHR or 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
If the `pNext` chain of `VkSubpassDescription2` includes a `VkSubpassDescriptionDepthStencilResolve` structure, then that structure describes multisample resolve operations for the depth/stencil attachment in a subpass.

The `VkSubpassDescriptionDepthStencilResolve` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkSubpassDescriptionDepthStencilResolve {
    VkStructureType sType;
    const void* pNext;
    VkResolveModeFlagBits depthResolveMode;
    VkResolveModeFlagBits stencilResolveMode;
    const VkAttachmentReference2* pDepthStencilResolveAttachment;
} VkSubpassDescriptionDepthStencilResolve;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `depthResolveMode` is a `VkResolveModeFlagBits` value describing the depth resolve mode.
- `stencilResolveMode` is a `VkResolveModeFlagBits` value describing the stencil resolve mode.
- `pDepthStencilResolveAttachment` is `NULL` or a pointer to a `VkAttachmentReference2` structure defining the depth/stencil resolve attachment for this subpass and its layout.

If `pDepthStencilResolveAttachment` is `NULL`, or if its attachment index is `VK_ATTACHMENT_UNUSED`, it indicates that no depth/stencil resolve attachment will be used in the subpass.

**Valid Usage**

- VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-03177
  If `pDepthStencilResolveAttachment` is not `NULL` and does not have the value `VK_ATTACHMENT_UNUSED`, `pDepthStencilAttachment` must not be `NULL` or have the value `VK_ATTACHMENT_UNUSED`.

- VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-03178
  If `pDepthStencilResolveAttachment` is not `NULL` and does not have the value `VK_ATTACHMENT_UNUSED`, `depthResolveMode` and `stencilResolveMode` must not both be `VK_RESOLVE_MODE_NONE`.

- VUID-VkSubpassDescriptionDepthStencilResolve-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`.  

If `pDepthStencilResolveAttachment` is not NULL and does not have the value `VK_ATTACHMENT_UNUSED`, `pDepthStencilResolveAttachment` must have a sample count of `VK_SAMPLE_COUNT_1_BIT`.

If `pDepthStencilResolveAttachment` is not NULL and does not have the value `VK_ATTACHMENT_UNUSED` then it must have an image format whose potential format features contain `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`.

If `pDepthStencilResolveAttachment` is not NULL and does not have the value `VK_ATTACHMENT_UNUSED` and `VkFormat` of `pDepthStencilResolveAttachment` has a depth component, then the `VkFormat` of `pDepthStencilAttachment` must have a depth component with the same number of bits and numerical type.

If `pDepthStencilResolveAttachment` is not NULL and does not have the value `VK_ATTACHMENT_UNUSED`, and `VkFormat` of `pDepthStencilResolveAttachment` has a stencil component, then the `VkFormat` of `pDepthStencilAttachment` must have a stencil component with the same number of bits and numerical type.

If `pDepthStencilResolveAttachment` is not NULL and does not have the value `VK_ATTACHMENT_UNUSED` and the `VkFormat` of `pDepthStencilResolveAttachment` has a depth component, then the value of `depthResolveMode` must be one of the bits set in `VkPhysicalDeviceDepthStencilResolveProperties::supportedDepthResolveModes` or `VK_RESOLVE_MODE_NONE`.

If `pDepthStencilResolveAttachment` is not NULL and does not have the value `VK_ATTACHMENT_UNUSED`, the `VkFormat` of `pDepthStencilResolveAttachment` has both depth and stencil components, `VkPhysicalDeviceDepthStencilResolveProperties::independentResolve` is `VK_FALSE`, and `VkPhysicalDeviceDepthStencilResolveProperties::independentResolveNone` is `VK_TRUE`, then the values of `depthResolveMode` and `stencilResolveMode` must be identical or one of them must be `VK_RESOLVE_MODE_NONE`. 
Valid Usage (Implicit)

- VUID-VkSubpassDescriptionDepthStencilResolve-sType-sType
  
  `sType` **must** be `VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_DEPTH_STENCIL_RESOLVE`

- VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-parameter
  
  If `pDepthStencilResolveAttachment` is not NULL, `pDepthStencilResolveAttachment` **must** be a valid pointer to a valid `VkAttachmentReference2` structure

Possible values of `VkSubpassDescriptionDepthStencilResolve::depthResolveMode` and `stencilResolveMode`, specifying the depth and stencil resolve modes, are:

```c
// Provided by VK_VERSION_1_2
typedef enum VkResolveModeFlagBits {
    VK_RESOLVE_MODE_NONE = 0,
    VK_RESOLVE_MODE_SAMPLE_ZERO_BIT = 0x00000001,
    VK_RESOLVE_MODE_AVERAGE_BIT = 0x00000002,
    VK_RESOLVE_MODE_MIN_BIT = 0x00000004,
    VK_RESOLVE_MODE_MAX_BIT = 0x00000008,
} VkResolveModeFlagBits;
```

- `VK_RESOLVE_MODE_NONE` indicates that no resolve operation is done.
- `VK_RESOLVE_MODE_SAMPLE_ZERO_BIT` indicates that result of the resolve operation is equal to the value of sample 0.
- `VK_RESOLVE_MODE_AVERAGE_BIT` indicates that result of the resolve operation is the average of the sample values.
- `VK_RESOLVE_MODE_MIN_BIT` indicates that result of the resolve operation is the minimum of the sample values.
- `VK_RESOLVE_MODE_MAX_BIT` indicates that result of the resolve operation is the maximum of the sample values.

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

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

The `VkFragmentShadingRateAttachmentInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_fragment_shading_rate
typedef struct VkFragmentShadingRateAttachmentInfoKHR {
    VkStructureType sType;
    const void* pNext;
    const VkAttachmentReference2* pFragmentShadingRateAttachment;
    VkExtent2D shadingRateAttachmentTexelSize;
};
```
\{ VkFragmentShadingRateAttachmentInfoKHR;

- \textit{sType} is the type of this structure.
- \textit{pNext} is NULL or a pointer to a structure extending this structure.
- \textit{pFragmentShadingRateAttachment} is NULL or a pointer to a \texttt{VkAttachmentReference2} structure defining the fragment shading rate attachment for this subpass.
- \textit{shadingRateAttachmentTexelSize} specifies the size of the portion of the framebuffer corresponding to each texel in \textit{pFragmentShadingRateAttachment}.

If no shading rate attachment is specified, or if this structure is not specified, the implementation behaves as if a valid shading rate attachment was specified with all texels specifying a single pixel per fragment.

**Valid Usage**

- VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-04524
  If \textit{pFragmentShadingRateAttachment} is not NULL and its \texttt{attachment} member is not \texttt{VK_ATTACHMENT_UNUSED}, its \texttt{layout} member \textbf{must} be equal to \texttt{VK_IMAGE_LAYOUT_GENERAL} or \texttt{VK_IMAGE_LAYOUT_FRAGMENT_SHADING_RATE_ATTACHMENT_OPTIMAL_KHR}

- VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-04525
  If \textit{pFragmentShadingRateAttachment} is not NULL and its \texttt{attachment} member is not \texttt{VK_ATTACHMENT_UNUSED}, \texttt{shadingRateAttachmentTexelSize.width} \textbf{must} be a power of two value

- VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-04526
  If \textit{pFragmentShadingRateAttachment} is not NULL and its \texttt{attachment} member is not \texttt{VK_ATTACHMENT_UNUSED}, \texttt{shadingRateAttachmentTexelSize.width} \textbf{must} be less than or equal to \texttt{maxFragmentShadingRateAttachmentTexelSize.width}

- VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-04527
  If \textit{pFragmentShadingRateAttachment} is not NULL and its \texttt{attachment} member is not \texttt{VK_ATTACHMENT_UNUSED}, \texttt{shadingRateAttachmentTexelSize.width} \textbf{must} be greater than or equal to \texttt{minFragmentShadingRateAttachmentTexelSize.width}

- VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-04528
  If \textit{pFragmentShadingRateAttachment} is not NULL and its \texttt{attachment} member is not \texttt{VK_ATTACHMENT_UNUSED}, \texttt{shadingRateAttachmentTexelSize.height} \textbf{must} be a power of two value

- VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-04529
  If \textit{pFragmentShadingRateAttachment} is not NULL and its \texttt{attachment} member is not \texttt{VK_ATTACHMENT_UNUSED}, \texttt{shadingRateAttachmentTexelSize.height} \textbf{must} be less than or equal
to \( \text{maxFragmentShadingRateAttachmentTexelSize.height} \)

- VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-04530
  If \( p\text{FragmentShadingRateAttachment} \) is not NULL and its attachment member is not \( VK\_\text{ATTACHMENT}\_\text{UNUSED} \), \( \text{shadingRateAttachmentTexelSize.height} \) must be greater than or equal to \( \text{minFragmentShadingRateAttachmentTexelSize.height} \)

- VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-04531
  If \( p\text{FragmentShadingRateAttachment} \) is not NULL and its attachment member is not \( VK\_\text{ATTACHMENT}\_\text{UNUSED} \), the quotient of \( \text{shadingRateAttachmentTexelSize.width} \) and \( \text{shadingRateAttachmentTexelSize.height} \) must be less than or equal to \( \text{maxFragmentShadingRateAttachmentTexelSizeAspectRatio} \)

- VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-04532
  If \( p\text{FragmentShadingRateAttachment} \) is not NULL and its attachment member is not \( VK\_\text{ATTACHMENT}\_\text{UNUSED} \), the quotient of \( \text{shadingRateAttachmentTexelSize.height} \) and \( \text{shadingRateAttachmentTexelSize.width} \) must be less than or equal to \( \text{maxFragmentShadingRateAttachmentTexelSizeAspectRatio} \)

Valid Usage (Implicit)

- VUID-VkFragmentShadingRateAttachmentInfoKHR-sType-sType
  \( s\text{Type} \) must be \( VK\_\text{STRUCTURE\_TYPE\_FRAGMENT\_SHADING\_RATE\_ATTACHMENT\_INFO\_KHR} \)

- VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-parameter
  If \( p\text{FragmentShadingRateAttachment} \) is not NULL, \( p\text{FragmentShadingRateAttachment} \) must be a valid pointer to a valid \( \text{VkAttachmentReference2} \) structure

The \( \text{VkAttachmentReference2} \) 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;
```

- \( s\text{Type} \) is the type of this structure.
- \( p\text{Next} \) 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 \( \text{VkRenderPassCreateInfo2::pAttachments} \), or \( VK\_\text{ATTACHMENT}\_\text{UNUSED} \) 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`.

- **VUID-VkAttachmentReference2-attachment-04754**
  If **attachment** is not `VK_ATTACHMENT_UNUSED`, and the format of the referenced attachment is a color format, **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`.

- **VUID-VkAttachmentReference2-attachment-04755**
  If **attachment** is not `VK_ATTACHMENT_UNUSED`, and the format of the referenced attachment is a depth/stencil format which includes both depth and stencil aspects, and **layout** is `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, the `pNext` chain must include a `VkAttachmentReferenceStencilLayout` structure.

- **VUID-VkAttachmentReference2-attachment-04756**
  If **attachment** is not `VK_ATTACHMENT_UNUSED`, and the format of the referenced attachment is a depth/stencil format which includes only the depth aspect, **layout** must not be `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`.

- **VUID-VkAttachmentReference2-attachment-04757**
  If **attachment** is not `VK_ATTACHMENT_UNUSED`, and the format of the referenced attachment is a
depth/stencil format which includes only the stencil aspect, layout must not be
VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_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 the type of 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
// Provided by VK_VERSION_1_2
typedef struct VkSubpassDependency2 {
    VkStructureType sType;
    const void* pNext;
    uint32_t srcSubpass;
    uint32_t dstSubpass;
    VkPipelineStageFlags srcStageMask;
    VkPipelineStageFlags dstStageMask;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    VkDependencyFlags dependencyFlags;
    int32_t viewOffset;
} VkSubpassDependency2;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **srcSubpass** is the subpass index of the first subpass in the dependency, or `VK_SUBPASS_EXTERNAL`.
- **dstSubpass** is the subpass index of the second subpass in the dependency, or `VK_SUBPASS_EXTERNAL`.
- **srcStageMask** is a bitmask of `VkPipelineStageFlagBits` specifying the source stage mask.
- **dstStageMask** is a bitmask of `VkPipelineStageFlagBits` specifying the destination stage mask.
- **srcAccessMask** is a bitmask of `VkAccessFlagBits` specifying a source access mask.
- **dstAccessMask** is a bitmask of `VkAccessFlagBits` specifying a destination access mask.
- **dependencyFlags** is a bitmask of `VkDependencyFlagBits`.
- **viewOffset** controls which views in the source subpass the views in the destination subpass depend on.

Parameters defined by this structure with the same name as those in `VkSubpassDependency` have the identical effect to those parameters.

*viewOffset* has the same effect for the described subpass dependency as `VkRenderPassMultiviewCreateInfo::pViewOffsets` has on each corresponding subpass dependency.

If a `VkMemoryBarrier2KHR` structure is included in the `pNext` chain, `srcStageMask`, `dstStageMask`,
The synchronization and access scopes instead are defined by the parameters of VkMemoryBarrier2KHR. 

### Valid Usage

- **VUID-VkSubpassDependency2-srcStageMask-04090**
  If the geometry shaders feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`.

- **VUID-VkSubpassDependency2-srcStageMask-04091**
  If the tessellation shaders feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`.

- **VUID-VkSubpassDependency2-srcStageMask-03937**
  If the synchronization feature is not enabled, `srcStageMask` must not be 0.

- **VUID-VkSubpassDependency2-dstStageMask-04090**
  If the geometry shaders feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`.

- **VUID-VkSubpassDependency2-dstStageMask-04091**
  If the tessellation shaders feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`.

- **VUID-VkSubpassDependency2-dstStageMask-03937**
  If the synchronization feature is not enabled, `dstStageMask` must not be 0.

- **VUID-VkSubpassDependency2-srcSubpass-03084**
  `srcSubpass` must be less than or equal to `dstSubpass`, unless one of them is `VK_SUBPASS_EXTERNAL`, to avoid cyclic dependencies and ensure a valid execution order.

- **VUID-VkSubpassDependency2-srcSubpass-03085**
  `srcSubpass` and `dstSubpass` must not both be equal to `VK_SUBPASS_EXTERNAL`.

- **VUID-VkSubpassDependency2-srcSubpass-03087**
  If `srcSubpass` is equal to `dstSubpass` and not all of the stages in `srcStageMask` and `dstStageMask` are framebuffer-space stages, the logically latest pipeline stage in `srcStageMask` must be logically earlier than or equal to the logically earliest pipeline stage in `dstStageMask`.

- **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`.
Valid Usage (Implicit)

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

- **VUID-VkSubpassDependency2-pNext-pNext**
  
pNext must be NULL or a pointer to a valid instance of VkMemoryBarrier2KHR

- **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
void vkDestroyRenderPass(
    VkDevice device,           // device
    VkRenderPass renderPass,   // renderPass
    const VkAllocationCallbacks* pAllocator);  // 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.
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 the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- flags is a bitmask of VkFramebufferCreateFlagBits
- renderPass is a render pass defining what render passes the framebuffer will be compatible with. See Render Pass Compatibility for details.
- attachmentCount is the number of attachments.
- pAttachments is a pointer to an array of VkImageView handles, each of which will be used as the
corresponding attachment in a render pass instance. If flags includes VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, this parameter is ignored.

- width, height and layers define the dimensions of the framebuffer. If the render pass uses multiview, then layers must be one and each attachment requires a number of layers that is greater than the maximum bit index set in the view mask in the subpasses in which it is used.

Other than the exceptions listed below, applications must ensure that all accesses to memory that backs image subresources used as attachments in a given render pass instance either happen-before the load operations for those attachments, or happen-after the store operations for those attachments.

The exceptions to the general rule are:

- For depth/stencil attachments, an aspect can be used separately as attachment and non-attachment if both accesses are read-only.

- For depth/stencil attachments, each aspect can be used separately as attachment and non-attachment as long as the non-attachment accesses are also via an image subresource in either the VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL layout or the VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL layout, and the attachment resource uses whichever of those two layouts the image accesses do not.

Use of non-attachment aspects in these cases is only well defined if the attachment is used in the subpass where the non-attachment access is being made, or the layout of the image subresource is constant throughout the entire render pass instance, including the initialLayout and finalLayout.

Note
These restrictions mean that the render pass has full knowledge of all uses of all of the attachments, so that the implementation is able to make correct decisions about when and how to perform layout transitions, when to overlap execution of subpasses, etc.

It is legal for a subpass to use no color or depth/stencil attachments, either because it has no attachment references or because all of them are VK_ATTACHMENT_UNUSED. This kind of subpass can use shader side effects such as image stores and atomics to produce an output. In this case, the subpass continues to use the width, height, and layers of the framebuffer to define the dimensions of the rendering area, and the rasterizationSamples from each pipeline’s VkPipelineMultisampleStateCreateInfo to define the number of samples used in rasterization; however, if VkPhysicalDeviceFeatures::variableMultisampleRate is VK_FALSE, then all pipelines to be bound with the subpass must have the same value for VkPipelineMultisampleStateCreateInfo::rasterizationSamples.

Valid Usage

- VUID-VkFramebufferCreateInfo-attachmentCount-00876
  If renderpass is not VK_NULL_HANDLE, attachmentCount must be equal to the attachment count specified in renderPass
- VUID-VkFramebufferCreateInfoflags-02778
If renderpass is not VK_NULL_HANDLE, 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 renderpass is not VK_NULL_HANDLE and 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 renderpass is not VK_NULL_HANDLE and 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 renderpass is not VK_NULL_HANDLE and 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 and 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 renderpass is not VK_NULL_HANDLE and 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 renderpass is not VK_NULL_HANDLE and 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 renderpass is not VK_NULL_HANDLE and 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::width greater than or equal to width

- VUID-VkFramebufferCreateInfo-flags-04534
  If renderpass is not VK_NULL_HANDLE and 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::height greater than or equal to height

- VUID-VkFramebufferCreateInfo-flags-04535
If renderpass is not VK_NULL_HANDLE and 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 is not VK_NULL_HANDLE and 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-flags-04537
  If renderpass is not VK_NULL_HANDLE and flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, and renderPass was specified with non-zero view masks, each element of pAttachments that is used as a fragment shading rate attachment by renderPass must have a layerCount that is either 1, or greater than the index of the most significant bit set in any of those view masks

- VUID-VkFramebufferCreateInfo-flags-04538
  If renderpass is not VK_NULL_HANDLE and flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, and renderPass was not specified with non-zero view masks, each element of pAttachments that is used as a fragment shading rate attachment must have a width at least as large as $\lceil \frac{\text{width}}{\text{texelWidth}} \rceil$, where texelWidth is the largest value of shadingRateAttachmentTexelSize.width in a VkFragmentShadingRateAttachmentInfoKHR which references that attachment.

- VUID-VkFramebufferCreateInfo-flags-04539
  If renderpass is not VK_NULL_HANDLE and flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, an element of pAttachments that is used as a fragment shading rate attachment must have a height at least as large as $\lceil \frac{\text{height}}{\text{texelHeight}} \rceil$, where texelHeight is the largest value of shadingRateAttachmentTexelSize.height in a VkFragmentShadingRateAttachmentInfoKHR which references that attachment.

- 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**

- **height** must be greater than 0

- **height** must be less than or equal to **maxFramebufferHeight**

- **layers** must be greater than 0

- **layers** must be less than or equal to **maxFramebufferLayers**

- If **renderpass** is not **VK_NULL_HANDLE** and **renderPass** was specified with non-zero view masks, **layers** must be 1

- If **flags** does not include **VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT**, each element of **pAttachments** that is a 2D or 2D array image view taken from a 3D image must not be a depth/stencil format

- If the **imageless framebuffer** feature is not enabled, **flags** must not include **VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT**

- If **flags** includes **VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT**, the **pNext** chain must include a **VkFramebufferAttachmentsCreateInfo** structure

- If **flags** includes **VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT**, the **attachmentImageInfoCount** member of a **VkFramebufferAttachmentsCreateInfo** structure in the **pNext** chain must be equal to either zero or **attachmentCount**

- If **renderpass** is not **VK_NULL_HANDLE** and **flags** includes **VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT**, the **width** member of any element of the **pAttachmentImageInfos** member of a **VkFramebufferAttachmentsCreateInfo** structure in the **pNext** chain that is used as an input, color, resolve or depth/stencil attachment in **renderPass** must be greater than or equal to **width**

- If **renderpass** is not **VK_NULL_HANDLE** and **flags** includes **VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT**, the **height** member of any element of the **pAttachmentImageInfos** member of a **VkFramebufferAttachmentsCreateInfo** structure in the **pNext** chain that is used as an input, color, resolve or depth/stencil attachment in **renderPass** must be greater than or equal to **height**

- If **renderpass** is not **VK_NULL_HANDLE** and **flags** includes **VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT**, the **width** member of any element of the **pAttachmentImageInfos** member of a **VkFramebufferAttachmentsCreateInfo** structure in the **pNext** chain that is used as a fragment shading rate attachment must be greater than...
or equal to \( \lceil \frac{\text{width}}{\text{texelWidth}} \rceil \), where \( \text{texelWidth} \) is the largest value of \( \text{shadingRateAttachmentTexelSize.width} \) in a \( \text{VkFragmentShadingRateAttachmentInfoKHR} \) which references that attachment

- **VUID-VkFramebufferCreateInfo-flags-04544**
  If \( \text{renderpass} \) is not \( \text{VK_NULL_HANDLE} \) and \( \text{flags} \) includes \( \text{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT} \), the \( \text{height} \) member of any element of the \( \text{pAttachmentImageInfos} \) member of a \( \text{VkFramebufferAttachmentsCreateInfo} \) structure in the \( \text{pNext} \) chain that is used as a fragment shading rate attachment must be greater than or equal to \( \lceil \frac{\text{height}}{\text{texelHeight}} \rceil \), where \( \text{texelHeight} \) is the largest value of \( \text{shadingRateAttachmentTexelSize.height} \) in a \( \text{VkFragmentShadingRateAttachmentInfoKHR} \) which references that attachment

- **VUID-VkFramebufferCreateInfo-flags-04545**
  If \( \text{renderpass} \) is not \( \text{VK_NULL_HANDLE} \) and \( \text{flags} \) includes \( \text{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT} \), the \( \text{layerCount} \) member of any element of the \( \text{pAttachmentImageInfos} \) member of a \( \text{VkFramebufferAttachmentsCreateInfo} \) structure in the \( \text{pNext} \) chain that is used as a fragment shading rate attachment must be either 1, or greater than or equal to \( \text{layers} \)

- **VUID-VkFramebufferCreateInfo-flags-04587**
  If \( \text{renderpass} \) is not \( \text{VK_NULL_HANDLE} \), \( \text{flags} \) includes \( \text{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT} \), and \( \text{renderPass} \) was specified with non-zero view masks, each element of \( \text{pAttachments} \) that is used as a fragment shading rate attachment by \( \text{renderPass} \) must have a \( \text{layerCount} \) that is either 1, or greater than the index of the most significant bit set in any of those view masks

- **VUID-VkFramebufferCreateInfo-renderPass-03198**
  If \( \text{renderpass} \) is not \( \text{VK_NULL_HANDLE} \), multiview is enabled for \( \text{renderPass} \), and \( \text{flags} \) includes \( \text{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT} \), the \( \text{layerCount} \) member of any element of the \( \text{pAttachmentImageInfos} \) member of a \( \text{VkFramebufferAttachmentsCreateInfo} \) structure included in the \( \text{pNext} \) chain used as an input, color, resolve, or depth/stencil attachment in \( \text{renderPass} \) must be greater than the maximum bit index set in the view mask in the subpasses in which it is used in \( \text{renderPass} \)

- **VUID-VkFramebufferCreateInfo-renderPass-04546**
  If \( \text{renderpass} \) is not \( \text{VK_NULL_HANDLE} \), multiview is not enabled for \( \text{renderPass} \), and \( \text{flags} \) includes \( \text{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT} \), the \( \text{layerCount} \) member of any element of the \( \text{pAttachmentImageInfos} \) member of a \( \text{VkFramebufferAttachmentsCreateInfo} \) structure included in the \( \text{pNext} \) chain used as an input, color, resolve, or depth/stencil attachment in \( \text{renderPass} \) must be greater than or equal to \( \text{layers} \)

- **VUID-VkFramebufferCreateInfo-flags-03201**
  If \( \text{renderpass} \) is not \( \text{VK_NULL_HANDLE} \) and \( \text{flags} \) includes \( \text{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT} \), the \( \text{usage} \) member of any element of the \( \text{pNext} \) chain that refers to an attachment used as a color attachment or resolve attachment by \( \text{renderPass} \) must include \( \text{VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT} \)

- **VUID-VkFramebufferCreateInfo-flags-03202**
  If \( \text{renderpass} \) is not \( \text{VK_NULL_HANDLE} \) and \( \text{flags} \) includes \( \text{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT} \), the \( \text{usage} \) member of any element of the
AttachmentImageInfos 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 renderpass is not VK_NULL_HANDLE and 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 renderpass is not VK_NULL_HANDLE and 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-flags-04548
  If renderpass is not VK_NULL_HANDLE and flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of pAttachments that is used as a fragment shading rate attachment by renderPass must have been created with a usage value including VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR

- VUID-VkFramebufferCreateInfo-flags-04549
  If renderpass is not VK_NULL_HANDLE and 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 fragment shading rate attachment by renderPass must include VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR

- 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:

```c
// Provided by VK_VERSION_1_2
typedef struct VkFramebufferAttachmentsCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t attachmentImageInfoCount;
    const VkFramebufferAttachmentImageInfo* pAttachmentImageInfos;
} VkFramebufferAttachmentsCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `attachmentImageInfoCount` is the number of attachments being described.
- `pAttachmentImageInfos` is a pointer to an array of `VkFramebufferAttachmentImageInfo` structures, each structure describing a number of parameters of the corresponding attachment in a render pass instance.

**Valid Usage (Implicit)**

- VUID-VkFramebufferAttachmentsCreateInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENTS_CREATE_INFO`

- VUID-VkFramebufferAttachmentsCreateInfo-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:

```c
// Provided by VK_VERSION_1_2
typedef struct VkFramebufferAttachmentImageInfo {
    VkStructureType sType;
    const void* pNext;
    VkImageCreateFlags flags;
    VkImageUsageFlags usage;
    uint32_t width;
} VkFramebufferAttachmentImageInfo;
```
uint32_t height;
uint32_t layerCount;
uint32_t viewFormatCount;
const VkFormat* pViewFormats;
}
VkFramebufferAttachmentImageInfo;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is a bitmask of VkImageCreateFlagBits, matching the value of VkImageCreateInfo::flags used to create an image that will be used with this framebuffer.
- **usage** is a bitmask of VkImageUsageFlagBits, matching the value of VkImageCreateInfo::usage used to create an image used with this framebuffer.
- **width** is the width of the image view used for rendering.
- **height** is the height of the image view used for rendering.
- **layerCount** is the number of array layers of the image view used for rendering.
- **viewFormatCount** is the number of entries in the pViewFormats array, matching the value of VkImageFormatListCreateInfo::viewFormatCount used to create an image used with this framebuffer.
- **pViewFormats** is a pointer to an array of VkFormat values specifying all of the formats which can be used when creating views of the image, matching the value of VkImageFormatListCreateInfo::pViewFormats used to create an image used with this framebuffer.

Images that can be used with the framebuffer when beginning a render pass, as specified by VkRenderPassAttachmentBeginInfo, must be created with parameters that are identical to those specified here.

### Valid Usage (Implicit)

- **VUID-VkFramebufferAttachmentImageInfo-sType-sType** `sType` must be VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENT_IMAGE_INFO
- **VUID-VkFramebufferAttachmentImageInfo-pNext-pNext** `pNext` must be NULL
- **VUID-VkFramebufferAttachmentImageInfo-flags-parameter** `flags` must be a valid combination of VkImageCreateFlagBits values
- **VUID-VkFramebufferAttachmentImageInfo-usage-parameter** `usage` must be a valid combination of VkImageUsageFlagBits values
- **VUID-VkFramebufferAttachmentImageInfo-usage-requiredbitmask** `usage` must not be 0
- **VUID-VkFramebufferAttachmentImageInfo-pViewFormats-parameter** If `viewFormatCount` is not 0, `pViewFormats` must be a valid pointer to an array of `viewFormatCount` valid VkFormat values
Bits which can be set in VkFramebufferCreateInfo::flags to specify options for framebuffers are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkFramebufferCreateFlagBits {
    // Provided by VK_VERSION_1_2
    VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT = 0x00000001,
} VkFramebufferCreateFlagBits;
```

- **VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT** specifies that image views are not specified, and only attachment compatibility information will be provided via a VkFramebufferAttachmentImageInfo structure.

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

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

To destroy a framebuffer, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyFramebuffer(
    VkDevice device,
    VkFramebuffer framebuffer,
    const VkAllocationCallbacks* pAllocator);
```

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

**Valid Usage**

- **VUID-vkDestroyFramebuffer-framebuffer-00892**
  All submitted commands that refer to framebuffer must have completed execution

**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 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 commandBuffer,  // commandBuffer is the command buffer in which to record the command.
    const VkRenderPassBeginInfo* pRenderPassBegin,  // pRenderPassBegin is a pointer to a VkRenderPassBeginInfo structure specifying the render pass to begin an instance of, and the framebuffer the instance uses.
    VkSubpassContents contents);  // 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_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_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`
structures or the layout member of the VkAttachmentReference structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL then the corresponding attachment image view of the framebuffer specified in the framebuffer member of pRenderPassBegin must have been created with a usage value including VK_IMAGE_USAGE_TRANSFER_DST_BIT

- VUID-vkCmdBeginRenderPass-initialLayout-00900
  If the initialLayout member of any of the VkAttachmentDescription structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is not VK_IMAGE_LAYOUT_UNDEFINED, then each such initialLayout must be equal to the current layout of the corresponding attachment image subresource of the framebuffer specified in the framebuffer member of pRenderPassBegin

- VUID-vkCmdBeginRenderPass-srcStageMask-06451
  The srcStageMask members of any element of the pDependencies member of VkRenderPassCreateInfo used to create renderPass must be supported by the capabilities of the queue family identified by the queueFamilyIndex member of the VkCommandPoolCreateInfo used to create the command pool which commandBuffer was allocated from

- VUID-vkCmdBeginRenderPass-dstStageMask-06452
  The dstStageMask members of any element of the pDependencies member of VkRenderPassCreateInfo used to create renderPass must be supported by the capabilities of the queue family identified by the queueFamilyIndex member of the VkCommandPoolCreateInfo used to create the command pool which commandBuffer was allocated from

- VUID-vkCmdBeginRenderPass-framebuffer-02532
  For any attachment in framebuffer that is used by renderPass and is bound to memory locations that are also bound to another attachment used by renderPass, and if at least one of those uses causes either attachment to be written to, both attachments must have had the VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT set

Valid Usage (Implicit)

- VUID-vkCmdBeginRenderPass-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdBeginRenderPass-pRenderPassBegin-parameter
  pRenderPassBegin must be a valid pointer to a valid VkRenderPassBeginInfo structure

- VUID-vkCmdBeginRenderPass-contents-parameter
  contents must be a valid VkSubpassContents value

- VUID-vkCmdBeginRenderPass-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdBeginRenderPass-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

- VUID-vkCmdBeginRenderPass-renderpass
This command **must** only be called outside of a render pass instance

- **VUID-vkCmdBeginRenderPass-bufferlevel**
  
  `commandBuffer must` be a primary `VkCommandBuffer`

### Host Synchronization

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
</tr>
</tbody>
</table>

Alternatively to begin a render pass, call:

```c
// Provided by VK_VERSION_1_2
void vkCmdBeginRenderPass2(
    VkCommandBuffer commandBuffer,
    const VkRenderPassBeginInfo* pRenderPassBegin,
    const VkSubpassBeginInfo* pSubpassBeginInfo);
```

- `commandBuffer` is the command buffer in which to record the command.
- `pRenderPassBegin` is a pointer to a `VkRenderPassBeginInfo` structure specifying the render pass to begin an instance of, and the framebuffer the instance uses.
- `pSubpassBeginInfo` is a pointer to a `VkSubpassBeginInfo` structure containing information about the subpass which is about to begin rendering.

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

### Valid Usage

- **VUID-vkCmdBeginRenderPass2-framebuffer-02779**
  
  Both the `framebuffer` and `renderPass` members of `pRenderPassBegin` **must** have been created on the same `VkDevice` that `commandBuffer` was allocated on

- **VUID-vkCmdBeginRenderPass2-initialLayout-03094**
  
  If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` then the corresponding attachment image view
of the framebuffer specified in the framebuffer member of pRenderPassBegin must have been created with a usage value including VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT.

• VUID-vkCmdBeginRenderPass2-initialLayout-03096
  If any of the initialLayout or finalLayout member of the VkAttachmentDescription structures or the layout member of the VkAttachmentReference structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is
  
  - VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL,
  - VK_IMAGE_LAYOUTDEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL,
  - VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL,
  - VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL
  
  then the corresponding attachment image view of the framebuffer specified in the framebuffer member of pRenderPassBegin must have been created with a usage value including VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT.

• VUID-vkCmdBeginRenderPass2-initialLayout-02844
  If any of the initialLayout or finalLayout member of the VkAttachmentDescription structures or the layout member of the VkAttachmentReference structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is
  
  - VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL,
  - VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL,
  - VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL,
  - 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 VkAttachmentDescriptionStencillaryout structures or the stencilLayout member of the VkAttachmentReferenceStencillaryout structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is
  
  - VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL,
  - 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`

- **VUID-vkCmdBeginRenderPass2-initialLayout-03100**
  If the `initialLayout` member of any of the `VkAttachmentDescription` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is not `VK_IMAGE_LAYOUT_UNDEFINED`, then each such `initialLayout` must be equal to the current layout of the corresponding attachment image subresource of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin`

- **VUID-vkCmdBeginRenderPass2-srcStageMask-06453**
  The `srcStageMask` members of any element of the `pDependencies` member of `VkRenderPassCreateInfo` used to create `renderPass` must be supported by the capabilities of the queue family identified by the `queueFamilyIndex` member of the `VkCommandPoolCreateInfo` used to create the command pool which `commandBuffer` was allocated from

- **VUID-vkCmdBeginRenderPass2-dstStageMask-06454**
  The `dstStageMask` members of any element of the `pDependencies` member of `VkRenderPassCreateInfo` used to create `renderPass` must be supported by the capabilities of the queue family identified by the `queueFamilyIndex` member of the `VkCommandPoolCreateInfo` used to create the command pool which `commandBuffer` was allocated from

- **VUID-vkCmdBeginRenderPass2-framebuffer-02533**
  For any attachment in `framebuffer` that is used by `renderPass` and is bound to memory locations that are also bound to another attachment used by `renderPass`, and if at least one of those uses causes either attachment to be written to, both attachments must have had the `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT` set

---

**Valid Usage (Implicit)**

- **VUID-vkCmdBeginRenderPass2-commandBuffer-parameter**
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdBeginRenderPass2-pRenderPassBegin-parameter**
  `pRenderPassBegin` must be a valid pointer to a valid `VkRenderPassBeginInfo` structure

- **VUID-vkCmdBeginRenderPass2-pSubpassBeginInfo-parameter**
  `pSubpassBeginInfo` must be a valid pointer to a valid `VkSubpassBeginInfo` structure

- **VUID-vkCmdBeginRenderPass2-commandBuffer-recording**
  `commandBuffer` must be in the recording state
The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations.

This command must only be called outside of a render pass instance.

`commandBuffer` must be a primary `VkCommandBuffer`.

### Host Synchronization

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
</tr>
</tbody>
</table>

The `VkRenderPassBeginInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkRenderPassBeginInfo {
    VkStructureType sType;
    const void* pNext;
    VkRenderPass renderPass;
    VkFramebuffer framebuffer;
    VkRect2D renderArea;
    uint32_t clearValueCount;
    const VkClearValue* pClearValues;
} VkRenderPassBeginInfo;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `renderPass` is the render pass to begin an instance of.
- `framebuffer` is the framebuffer containing the attachments that are used with the render pass.
- `renderArea` is the render area that is affected by the render pass instance, and is described in more detail below.
- `clearValueCount` is the number of elements in `pClearValues`.
- `pClearValues` is a pointer to an array of `clearValueCount` `VkClearValue` structures containing...
clear values for each attachment, if the attachment uses a `loadOp` value of `VK_ATTACHMENT_LOAD_OP_CLEAR` or if the attachment has a depth/stencil format and uses a `stencilLoadOp` value of `VK_ATTACHMENT_LOAD_OP_CLEAR`. The array is indexed by attachment number. Only elements corresponding to cleared attachments are used. Other elements of `pClearValues` are ignored.

`renderArea` is the render area that is affected by the render pass instance. The effects of attachment load, store and multisample resolve operations are restricted to the pixels whose x and y coordinates fall within the render area on all attachments. The render area extends to all layers of `framebuffer`. The application **must** ensure (using scissor if necessary) that all rendering is contained within the render area. The render area **must** be contained within the framebuffer dimensions.

When multiview is enabled, the resolve operation at the end of a subpass applies to all views in the view mask.

**Note**

There **may** be a performance cost for using a render area smaller than the framebuffer, unless it matches the render area granularity for the render pass.

---

**Valid Usage**

- **VUID-VkRenderPassBeginInfo-clearValueCount-00902**

  `clearValueCount` **must** be greater than the largest attachment index in `renderPass` specifying a `loadOp` (or `stencilLoadOp`, if the attachment has a depth/stencil format) of `VK_ATTACHMENT_LOAD_OP_CLEAR`.

- **VUID-VkRenderPassBeginInfo-clearValueCount-04962**

  If `clearValueCount` is not 0, `pClearValues` **must** be a valid pointer to an array of `clearValueCount` `VkClearValue` unions.

- **VUID-VkRenderPassBeginInfo-renderPass-00904**

  `renderPass` **must** be compatible with the `renderPass` member of the `VkFramebufferCreateInfo` structure specified when creating `framebuffer`.

- **VUID-VkRenderPassBeginInfo-pNext-02850**

  If the `pNext` chain does not contain `VkDeviceGroupRenderPassBeginInfo` or its `deviceRenderAreaCount` member is equal to 0, `renderArea.offset.x` **must** be greater than or equal to 0.

- **VUID-VkRenderPassBeginInfo-pNext-02851**

  If the `pNext` chain does not contain `VkDeviceGroupRenderPassBeginInfo` or its `deviceRenderAreaCount` member is equal to 0, `renderArea.offset.y` **must** be greater than or equal to 0.

- **VUID-VkRenderPassBeginInfo-pNext-02852**

  If the `pNext` chain does not contain `VkDeviceGroupRenderPassBeginInfo` or its `deviceRenderAreaCount` member is equal to 0, `renderArea.offset.x + renderArea.extent.width` **must** be less than or equal to `VkFramebufferCreateInfo::width` the `framebuffer` was created with.

- **VUID-VkRenderPassBeginInfo-pNext-02853**
If the pNext chain does not contain `VkDeviceGroupRenderPassBeginInfo` or its `deviceRenderAreaCount` member is equal to 0, `renderArea.offset.y + renderArea.extent.height` must be less than or equal to `VkFramebufferCreateInfo::height` the framebuffer was created with.

- VUID-VkRenderPassBeginInfo-pNext-02856
  If the pNext chain contains `VkDeviceGroupRenderPassBeginInfo`, `offset.x + extent.width` of each element of pDeviceRenderAreas must be less than or equal to `VkFramebufferCreateInfo::width` the framebuffer was created with.

- VUID-VkRenderPassBeginInfo-pNext-02857
  If the pNext chain contains `VkDeviceGroupRenderPassBeginInfo`, `offset.y + extent.height` of each element of pDeviceRenderAreas must be less than or equal to `VkFramebufferCreateInfo::height` the framebuffer was created with.

- VUID-VkRenderPassBeginInfo-framebuffer-03207
  If framebuffer was created with a `VkFramebufferCreateInfo::flags` value that did not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, and the pNext chain includes a `VkRenderPassAttachmentBeginInfo` structure, its attachmentCount must be zero.

- VUID-VkRenderPassBeginInfo-framebuffer-03208
  If framebuffer was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, the attachmentCount of a `VkRenderPassAttachmentBeginInfo` structure included in the pNext chain must be equal to the value of `VkFramebufferAttachmentsCreateInfo::attachmentImageInfoCount` used to create framebuffer.

- 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 usage 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-03217**
  If framebuffer was created with a VkFramebufferCreateInfo::flags value that included VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of the pAttachments member of a VkRenderPassAttachmentBeginInfo structure included in the pNext chain must be a VkImageView of an image created with a value of VkImageCreateInfo::samples equal to the corresponding value of VkAttachmentDescription::samples in renderPass
Valid Usage (Implicit)

- VUID-VkRenderPassBeginInfo-sType-sType
  \textit{sType} \textbf{must} be \texttt{VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO}

- VUID-VkRenderPassBeginInfo-pNext-pNext
  Each \texttt{pNext} member of any structure (including this one) in the \texttt{pNext} chain \textbf{must} be either \texttt{NULL} or a pointer to a valid instance of \texttt{VkDeviceGroupRenderPassBeginInfo}, \texttt{VkRenderPassAttachmentBeginInfo}, or \texttt{VkRenderPassSampleLocationsBeginInfoEXT}

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

- VUID-VkRenderPassBeginInfo-renderPass-parameter
  \textit{renderPass} \textbf{must} be a valid \texttt{VkRenderPass} handle

- VUID-VkRenderPassBeginInfo-framebuffer-parameter
  \textit{framebuffer} \textbf{must} be a valid \texttt{VkFramebuffer} handle

- VUID-VkRenderPassBeginInfo-commonparent
  Both of \textit{framebuffer}, and \textit{renderPass} \textbf{must} have been created, allocated, or retrieved from the same \texttt{VkDevice}

The image layout of the depth aspect of a depth/stencil attachment referring to an image created with \texttt{VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT} is dependent on the last sample locations used to render to the image subresource, thus preserving the contents of such depth/stencil attachments across subpass boundaries requires the application to specify these sample locations whenever a layout transition of the attachment \textbf{may} occur. This information \textbf{can} be provided by adding a \texttt{VkRenderPassSampleLocationsBeginInfoEXT} structure to the \texttt{pNext} chain of \texttt{VkRenderPassBeginInfo}.

The \texttt{VkRenderPassSampleLocationsBeginInfoEXT} structure is defined as:

```c
// Provided by VK_EXT_sample_locations
typedef struct VkRenderPassSampleLocationsBeginInfoEXT {
    VkStructureType sType;
    const void* pNext;
    uint32_t attachmentInitialSampleLocationsCount;
    const VkAttachmentSampleLocationsEXT* pAttachmentInitialSampleLocations;
    uint32_t postSubpassSampleLocationsCount;
    const VkSubpassSampleLocationsEXT* pPostSubpassSampleLocations;
} VkRenderPassSampleLocationsBeginInfoEXT;
```

- \textit{sType} is the type of this structure.
- \textit{pNext} is \texttt{NULL} or a pointer to a structure extending this structure.
- \texttt{attachmentInitialSampleLocationsCount} is the number of elements in the \texttt{pAttachmentInitialSampleLocations} array.
- \texttt{pAttachmentInitialSampleLocations} is a pointer to an array of...
The `VkAttachmentSampleLocationsEXT` structure is defined as:

```c
typedef struct VkAttachmentSampleLocationsEXT {
    uint32_t attachmentIndex;
    VkSampleLocationsInfoEXT sampleLocationsInfo;
} VkAttachmentSampleLocationsEXT;
```

- `attachmentIndex` is the index of the attachment for which the sample locations state is provided.
**sampleLocationsInfo** is the sample locations state to use for the layout transition of the given attachment from the initial layout of the attachment to the image layout specified for the attachment in the first subpass using it.

If the image referenced by the framebuffer attachment at index `attachmentIndex` was not created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` then the values specified in `sampleLocationsInfo` are ignored.

---

**Valid Usage**

- VUID-VkAttachmentSampleLocationsEXT-attachmentIndex-01531
  
  `attachmentIndex` **must** be less than the `attachmentCount` specified in `VkRenderPassCreateInfo` the render pass specified by `VkRenderPassBeginInfo::renderPass` was created with.

---

**Valid Usage (Implicit)**

- VUID-VkAttachmentSampleLocationsEXT-sampleLocationsInfo-parameter
  
  `sampleLocationsInfo` **must** be a valid `VkSampleLocationsInfoEXT` structure.

---

The `VkSubpassSampleLocationsEXT` structure is defined as:

```c
// Provided by VK_EXT_sample_locations
typedef struct VkSubpassSampleLocationsEXT {
    uint32_t subpassIndex;
    VkSampleLocationsInfoEXT sampleLocationsInfo;
} VkSubpassSampleLocationsEXT;
```

- `subpassIndex` is the index of the subpass for which the sample locations state is provided.
- `sampleLocationsInfo` is the sample locations state to use for the layout transition of the depth/stencil attachment away from the image layout the attachment is used with in the subpass specified in `subpassIndex`.

If the image referenced by the depth/stencil attachment used in the subpass identified by `subpassIndex` was not created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` or if the subpass does not use a depth/stencil attachment, and `VkPhysicalDeviceSampleLocationsPropertiesEXT::variableSampleLocations` is `VK_TRUE` then the values specified in `sampleLocationsInfo` are ignored.

---

**Valid Usage**

- VUID-VkSubpassSampleLocationsEXT-subpassIndex-01532
  
  `subpassIndex` **must** be less than the `subpassCount` specified in `VkRenderPassCreateInfo` the render pass specified by `VkRenderPassBeginInfo::renderPass` was created with.
Valid Usage (Implicit)

- VUID-VkSubpassSampleLocationsEXT-sampleLocationsInfo-parameter
  sampleLocationsInfo must be a valid VkSampleLocationsInfoEXT structure

The VkSubpassBeginInfo structure is defined as:

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

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `contents` is a VkSubpassContents value specifying how the commands in the next subpass will be provided.

Valid Usage (Implicit)

- VUID-VkSubpassBeginInfo-sType-sType
  `sType` must be VK_STRUCTURE_TYPE_SUBPASS_BEGIN_INFO

- VUID-VkSubpassBeginInfo-pNext-pNext
  `pNext` must be NULL

- VUID-VkSubpassBeginInfo-contents-parameter
  `contents` must be a valid VkSubpassContents value

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

```c
// Provided by VK_VERSION_1_0
typedef enum VkSubpassContents {
    VK_SUBPASS_CONTENTS_INLINE = 0,
    VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS = 1,
} VkSubpassContents;
```

- `VK_SUBPASS_CONTENTS_INLINE` specifies that the contents of the subpass will be recorded inline in the primary command buffer, and secondary command buffers must not be executed within the subpass.
- `VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS` specifies that the contents are recorded in secondary command buffers that will be called from the primary command buffer, and `vkCmdExecuteCommands` is the only valid command on the command buffer until
vkCmdNextSubpass or vkCmdEndRenderPass.

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

The VkDeviceGroupRenderPassBeginInfo structure is defined as:

```
// Provided by VK_VERSION_1_1
typedef struct VkDeviceGroupRenderPassBeginInfo {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           deviceMask;
    uint32_t           deviceRenderAreaCount;
    const VkRect2D*     pDeviceRenderAreas;
} VkDeviceGroupRenderPassBeginInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **deviceMask** is the device mask for the render pass instance.
- **deviceRenderAreaCount** is the number of elements in the pDeviceRenderAreas array.
- **pDeviceRenderAreas** is a pointer to an array of VkRect2D structures defining the render area for each physical device.

The deviceMask serves several purposes. It is an upper bound on the set of physical devices that can be used during the render pass instance, and the initial device mask when the render pass instance begins. In addition, commands transitioning to the next subpass in a render pass instance and commands ending the render pass instance, and, accordingly render pass attachment load, store, and resolve operations and subpass dependencies corresponding to the render pass instance, are executed on the physical devices included in the device mask provided here.

If deviceRenderAreaCount is not zero, then the elements of pDeviceRenderAreas override the value of VkRenderPassBeginInfo::renderArea, and provide a render area specific to each physical device. These render areas serve the same purpose as VkRenderPassBeginInfo::renderArea, including controlling the region of attachments that are cleared by VK_ATTACHMENT_LOAD_OP_CLEAR and that are resolved into resolve attachments.

If this structure is not present, the render pass instance’s device mask is the value of VkDeviceGroupCommandBufferBeginInfo::deviceMask. If this structure is not present or if deviceRenderAreaCount is zero, VkRenderPassBeginInfo::renderArea is used for all physical devices.

### Valid Usage

- **VUID-VkDeviceGroupRenderPassBeginInfo-deviceMask-00905**
  
  deviceMask must be a valid device mask value

- **VUID-VkDeviceGroupRenderPassBeginInfo-deviceMask-00906**

  deviceMask must not be zero
• VUID-VkDeviceGroupRenderPassBeginInfo-deviceMask-00907
deviceMask must be a subset of the command buffer’s initial device mask

• VUID-VkDeviceGroupRenderPassBeginInfo-deviceRenderAreaCount-00908
deviceRenderAreaCount must either be zero or equal to the number of physical devices in
the logical device

• VUID-VkDeviceGroupRenderPassBeginInfo-offset-06166
The offset.x member of any element of pDeviceRenderAreas must be greater than or equal
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

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:

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

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• attachmentCount is the number of attachments.
• pAttachments is a pointer to an array of VkImageView handles, each of which will be used as the
corresponding attachment in the render pass instance.
Valid Usage

- 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
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.

Moving to the next subpass automatically performs any multisample resolve operations in the subpass being ended. End-of-subpass multisample resolves are treated as color attachment writes for the purposes of synchronization. This applies to resolve operations for both color and depth/stencil attachments. That is, they are considered to execute in the `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` pipeline stage and their writes are synchronized with `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`. Synchronization between rendering within a subpass and any resolve operations at the end of the subpass occurs automatically, without need for explicit dependencies or pipeline barriers. However, if the resolve attachment is also used in a different subpass, an explicit dependency is needed.

After transitioning to the next subpass, the application can record the commands for that subpass.

### Valid Usage

- VUID-vkCmdNextSubpass-None-00909
The current subpass index **must** be less than the number of subpasses in the render pass minus one

Valid Usage (Implicit)

- **VUID-vkCmdNextSubpass-commandBuffer-parameter**
  - `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- **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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
</tr>
</tbody>
</table>

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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</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

### 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
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 previous 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

### 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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
</tr>
</tbody>
</table>

The `VkSubpassEndInfo` structure is defined as:

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

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

Valid Usage (Implicit)

- VUID-VkSubpassEndInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_SUBPASS_END_INFO`
- VUID-VkSubpassEndInfo-pNext-pNext
  `pNext` must be `NULL`
Chapter 9. Shaders

A shader specifies programmable operations that execute for each vertex, control point, tessellated vertex, primitive, fragment, or workgroup in the corresponding stage(s) of the graphics and compute pipelines.

Graphics pipelines include vertex shader execution as a result of primitive assembly, followed, if enabled, by tessellation control and evaluation shaders operating on patches, geometry shaders, if enabled, operating on primitives, and fragment shaders, if present, operating on fragments generated by Rasterization. In this specification, vertex, tessellation control, tessellation evaluation and geometry shaders are collectively referred to as pre-rasterization shader stages and occur in the logical pipeline before rasterization. The fragment shader occurs logically after rasterization.

Only the compute shader stage is included in a compute pipeline. Compute shaders operate on compute invocations in a workgroup.

Shaders can read from input variables, and read from and write to output variables. Input and output variables can be used to transfer data between shader stages, or to allow the shader to interact with values that exist in the execution environment. Similarly, the execution environment provides constants describing capabilities.

Shader variables are associated with execution environment-provided inputs and outputs using built-in decorations in the shader. The available decorations for each stage are documented in the following subsections.

9.1. Shader Modules

Shader modules contain shader code and one or more entry points. Shaders are selected from a shader module by specifying an entry point as part of pipeline creation. The stages of a pipeline can use shaders that come from different modules. The shader code defining a shader module must be in the SPIR-V format, as described by the Vulkan Environment for SPIR-V appendix.

Shader modules are represented by VkShaderModule handles:

```c
// 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. 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. 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.
9.4. Shader Inputs and Outputs

Data is passed into and out of shaders using variables with input or output storage class, respectively. User-defined inputs and outputs are connected between stages by matching their Location decorations. Additionally, data can be provided by or communicated to special functions provided by the execution environment using BuiltIn decorations.

In many cases, the same BuiltIn decoration can be used in multiple shader stages with similar meaning. The specific behavior of variables decorated as BuiltIn is documented in the following sections.

9.5. Vertex Shaders

Each vertex shader invocation operates on one vertex and its associated vertex attribute data, and outputs one vertex and associated data. Graphics pipelines must include a vertex shader, and the vertex shader stage is always the first shader stage in the graphics pipeline.

9.5.1. Vertex Shader Execution

A vertex shader must be executed at least once for each vertex specified by a drawing command. If the subpass includes multiple views in its view mask, the shader may be invoked separately for each view. During execution, the shader is presented with the index of the vertex and instance for which it has been invoked. Input variables declared in the vertex shader are filled by the implementation with the values of vertex attributes associated with the invocation being executed.

If the same vertex is specified multiple times in a drawing command (e.g. by including the same index value multiple times in an index buffer) the implementation may reuse the results of vertex shading if it can statically determine that the vertex shader invocations will produce identical results.

Note

It is implementation-dependent when and if results of vertex shading are reused, and thus how many times the vertex shader will be executed. This is true also if the vertex shader contains stores or atomic operations (see vertexPipelineStoresAndAtomics).

9.6. Tessellation Control Shaders

The tessellation control shader is used to read an input patch provided by the application and to produce an output patch. Each tessellation control shader invocation operates on an input patch (after all control points in the patch are processed by a vertex shader) and its associated data, and outputs a single control point of the output patch and its associated data, and can also output additional per-patch data. The input patch is sized according to the patchControlPoints member of VkPipelineTessellationStateCreateInfo, as part of input assembly.

The input patch can also be dynamically sized with patchControlPoints parameter of vkCmdSetPatchControlPointsEXT.
To dynamically set the number of control points per patch, call:

```c
// Provided by VK_EXT_extended_dynamic_state2
void vkCmdSetPatchControlPointsEXT(
    VkCommandBuffer commandBuffer,  
    uint32_t patchControlPoints);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `patchControlPoints` specifies the number of control points per patch.

This command sets the number of control points per patch for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineTessellationStateCreateInfo::patchControlPoints` value used to create the currently active pipeline.

**Valid Usage**

- VUID-vkCmdSetPatchControlPointsEXT-None-04873
  The `extendedDynamicState2PatchControlPoints` feature must be enabled

- VUID-vkCmdSetPatchControlPointsEXT-patchControlPoints-04874
  `patchControlPoints` must be greater than zero and less than or equal to `VkPhysicalDeviceLimits::maxTessellationPatchSize`

**Valid Usage (Implicit)**

- VUID-vkCmdSetPatchControlPointsEXT-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetPatchControlPointsEXT-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdSetPatchControlPointsEXT-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
The size of the output patch is controlled by the `OpExecutionMode OutputVertices` specified in the tessellation control or tessellation evaluation shaders, which must be specified in at least one of the shaders. The size of the input and output patches must each be greater than zero and less than or equal to `VkPhysicalDeviceLimits::maxTessellationPatchSize`.

### 9.6.1. Tessellation Control Shader Execution

A tessellation control shader is invoked at least once for each output vertex in a patch. If the subpass includes multiple views in its view mask, the shader may be invoked separately for each view.

Inputs to the tessellation control shader are generated by the vertex shader. Each invocation of the tessellation control shader can read the attributes of any incoming vertices and their associated data. The invocations corresponding to a given patch execute logically in parallel, with undefined relative execution order. However, the `OpControlBarrier` instruction can be used to provide limited control of the execution order by synchronizing invocations within a patch, effectively dividing tessellation control shader execution into a set of phases. Tessellation control shaders will read undefined values if one invocation reads a per-vertex or per-patch output written by another invocation at any point during the same phase, or if two invocations attempt to write different values to the same per-patch output in a single phase.

### 9.7. Tessellation Evaluation Shaders

The Tessellation Evaluation Shader operates on an input patch of control points and their associated data, and a single input barycentric coordinate indicating the invocation’s relative position within the subdivided patch, and outputs a single vertex and its associated data.

#### 9.7.1. Tessellation Evaluation Shader Execution

A tessellation evaluation shader is invoked at least once for each unique vertex generated by the tessellator. If the subpass includes multiple views in its view mask, the shader may be invoked separately for each view.

### 9.8. Geometry Shaders

The geometry shader operates on a group of vertices and their associated data assembled from a single input primitive, and emits zero or more output primitives and the group of vertices and their associated data required for each output primitive.
9.8.1. Geometry Shader Execution

A geometry shader is invoked at least once for each primitive produced by the tessellation stages, or at least once for each primitive generated by primitive assembly when tessellation is not in use. A shader can request that the geometry shader runs multiple instances. A geometry shader is invoked at least once for each instance. If the subpass includes multiple views in its view mask, the shader may be invoked separately for each view.

9.9. Fragment Shaders

Fragment shaders are invoked as a fragment operation in a graphics pipeline. Each fragment shader invocation operates on a single fragment and its associated data. With few exceptions, fragment shaders do not have access to any data associated with other fragments and are considered to execute in isolation of fragment shader invocations associated with other fragments.

9.10. Compute Shaders

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

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

9.11. Interpolation Decorations

Interpolation decorations control the behavior of attribute interpolation in the fragment shader stage. Interpolation decorations can be applied to Input storage class variables in the fragment shader stage’s interface, and control the interpolation behavior of those variables.

Inputs that could be interpolated can be decorated by at most one of the following decorations:

- **Flat**: no interpolation
- **NoPerspective**: linear interpolation (for lines and polygons)

Fragment input variables decorated with neither Flat nor NoPerspective use perspective-correct interpolation (for lines and polygons).

The presence of and type of interpolation is controlled by the above interpolation decorations as well as the auxiliary decorations Centroid and Sample.

A variable decorated with Flat will not be interpolated. Instead, it will have the same value for
every fragment within a triangle. This value will come from a single provoking vertex. A variable decorated with Flat can also be decorated with Centroid or Sample, which will mean the same thing as decorating it only as Flat.

For fragment shader input variables decorated with neither Centroid nor Sample, the assigned variable may be interpolated anywhere within the fragment and a single value may be assigned to each sample within the fragment.

If a fragment shader input is decorated with Centroid, a single value may be assigned to that variable for all samples in the fragment, but that value must be interpolated to a location that lies in both the fragment and in the primitive being rendered, including any of the fragment’s samples covered by the primitive. Because the location at which the variable is interpolated may be different in neighboring fragments, and derivatives may be computed by computing differences between neighboring fragments, derivatives of centroid-sampled inputs may be less accurate than those for non-centroid interpolated variables. The PostDepthCoverage execution mode does not affect the determination of the centroid location.

If a fragment shader input is decorated with Sample, a separate value must be assigned to that variable for each covered sample in the fragment, and that value must be sampled at the location of the individual sample. When rasterizationSamples is VK_SAMPLE_COUNT_1_BIT, the fragment center must be used for Centroid, Sample, and undecorated attribute interpolation.

Fragment shader inputs that are signed or unsigned integers, integer vectors, or any double-precision floating-point type must be decorated with Flat.

9.12. Static Use

A SPIR-V module declares a global object in memory using the OpVariable instruction, which results in a pointer \( x \) to that object. A specific entry point in a SPIR-V module is said to statically use that object if that entry point’s call tree contains a function containing a memory instruction or image instruction with \( x \) as an id operand. See the “Memory Instructions” and “Image Instructions” subsections of section 3 “Binary Form” of the SPIR-V specification for the complete list of SPIR-V memory instructions.

Static use is not used to control the behavior of variables with Input and Output storage. The effects of those variables are applied based only on whether they are present in a shader entry point’s interface.

9.13. Scope

A scope describes a set of shader invocations, where each such set is a scope instance. Each invocation belongs to one or more scope instances, but belongs to no more than one scope instance for each scope.

The operations available between invocations in a given scope instance vary, with smaller scopes generally able to perform more operations, and with greater efficiency.

All invocations executed in a Vulkan instance fall into a single cross device scope instance. Whilst the CrossDevice scope is defined in SPIR-V, it is disallowed in Vulkan. API synchronization commands can be used to communicate between devices.

9.13.2. Device

All invocations executed on a single device form a device scope instance.

If the vulkanMemoryModel and vulkanMemoryModelDeviceScope features are enabled, this scope is represented in SPIR-V by the Device Scope, which can be used as a Memory Scope for barrier and atomic operations.

If both the shaderDeviceClock and vulkanMemoryModelDeviceScope features are enabled, using the Device Scope with the OpReadClockKHR instruction will read from a clock that is consistent across invocations in the same device scope instance.

There is no method to synchronize the execution of these invocations within SPIR-V, and this can only be done with API synchronization primitives.

Invocations executing on different devices in a device group operate in separate device scope instances.

9.13.3. Queue Family

Invocations executed by queues in a given queue family form a queue family scope instance.

This scope is identified in SPIR-V as the QueueFamily Scope if the vulkanMemoryModel feature is enabled, or if not, the Device Scope, which can be used as a Memory Scope for barrier and atomic operations.

If the shaderDeviceClock feature is enabled, but the vulkanMemoryModelDeviceScope feature is not enabled, using the Device Scope with the OpReadClockKHR instruction will read from a clock that is consistent across invocations in the same queue family scope instance.

There is no method to synchronize the execution of these invocations within SPIR-V, and this can only be done with API synchronization primitives.

Each invocation in a queue family scope instance must be in the same device scope instance.

9.13.4. Command

Any shader invocations executed as the result of a single command such as vkCmdDispatch or vkCmdDraw form a command scope instance. For indirect drawing commands with drawCount greater than one, invocations from separate draws are in separate command scope instances.

There is no specific Scope for communication across invocations in a command scope instance. As this has a clear boundary at the API level, coordination here can be performed in the API, rather than in SPIR-V.
Each invocation in a command scope instance must be in the same queue-family scope instance.

For shaders without defined workgroups, this set of invocations forms an invocation group as defined in the SPIR-V specification.

### 9.13.5. Primitive

Any fragment shader invocations executed as the result of rasterization of a single primitive form a primitive scope instance.

There is no specific Scope for communication across invocations in a primitive scope instance.

Any generated helper invocations are included in this scope instance.

Each invocation in a primitive scope instance must be in the same command scope instance.

Any input variables decorated with Flat are uniform within a primitive scope instance.

### 9.13.6. Workgroup

A local workgroup is a set of invocations that can synchronize and share data with each other using memory in the Workgroup storage class.

The Workgroup Scope can be used as both an Execution Scope and Memory Scope for barrier and atomic operations.

Each invocation in a local workgroup must be in the same command scope instance.

Only compute shaders have defined workgroups - other shader types cannot use workgroup functionality. For shaders that have defined workgroups, this set of invocations forms an invocation group as defined in the SPIR-V specification.

### 9.13.7. Subgroup

A subgroup (see the subsection “Control Flow” of section 2 of the SPIR-V 1.3 Revision 1 specification) is a set of invocations that can synchronize and share data with each other efficiently.

The Subgroup Scope can be used as both an Execution Scope and Memory Scope for barrier and atomic operations. Other subgroup features allow the use of group operations with subgroup scope.

If the shaderSubgroupClock feature is enabled, using the Subgroup Scope with the OpReadClockKHR instruction will read from a clock that is consistent across invocations in the same subgroup.

For shaders that have defined workgroups, each invocation in a subgroup must be in the same local workgroup.

In other shader stages, each invocation in a subgroup must be in the same device scope instance.

Only shader stages that support subgroup operations have defined subgroups.
9.13.8. Quad

A quad scope instance is formed of four shader invocations.

In a fragment shader, each invocation in a quad scope instance is formed of invocations in neighboring framebuffer locations \((x_i, y_i)\), where:

- \(i\) is the index of the invocation within the scope instance.
- \(w\) and \(h\) are the number of pixels the fragment covers in the x and y axes.
- \(w\) and \(h\) are identical for all participating invocations.
- \((x_0) = (x_1 - w) = (x_2) = (x_3 - w)\)
- \((y_0) = (y_1) = (y_2 - h) = (y_3 - h)\)
- Each invocation has the same layer and sample indices.

In all shaders, each invocation in a quad scope instance is formed of invocations in adjacent subgroup invocation indices \((s_i)\), where:

- \(i\) is the index of the invocation within the quad scope instance.
- \((s_0) = (s_1 - 1) = (s_2 - 2) = (s_3 - 3)\)
- \(s_0\) is an integer multiple of 4.

Each invocation in a quad scope instance must be in the same subgroup.

In a fragment shader, each invocation in a quad scope instance must be in the same primitive scope instance.

Fragment and compute shaders have defined quad scope instances. If the \texttt{quadOperationsInAllStages} limit is supported, any shader stages that support subgroup operations also have defined quad scope instances.

9.13.9. Fragment Interlock

A fragment interlock scope instance is formed of fragment shader invocations based on their framebuffer locations \((x,y,layer,sample)\), executed by commands inside a single subpass.

The specific set of invocations included varies based on the execution mode as follows:

- If the \texttt{SampleInterlockOrderedEXT} or \texttt{SampleInterlockUnorderedEXT} execution modes are used, only invocations with identical framebuffer locations \((x,y,layer,sample)\) are included.
- If the \texttt{PixelInterlockOrderedEXT} or \texttt{PixelInterlockUnorderedEXT} execution modes are used, fragments with different sample ids are also included.
- If the \texttt{ShadingRateInterlockOrderedEXT} or \texttt{ShadingRateInterlockUnorderedEXT} execution modes are used, fragments from neighbouring framebuffer locations are also included, as determined by the shading rate.

Only fragment shaders with one of the above execution modes have defined fragment interlock scope instances.
There is no specific Scope value for communication across invocations in a fragment interlock scope instance. However, this is implicitly used as a memory scope by OpBeginInvocationInterlockEXT and OpEndInvocationInterlockEXT.

Each invocation in a fragment interlock scope instance must be in the same queue family scope instance.

### 9.13.10. 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 fragment shaders that have a defined fragment interlock scope must be in a fragment interlock scope instance.

Invocations in shaders that have defined workgroups must be in a local workgroup.

Invocations in shaders that have a defined subgroup scope must be in a subgroup.

Invocations in shaders that have a defined quad scope must be in a quad scope instance.

All invocations in all stages must be in a command scope instance.


Group operations are executed by multiple invocations within a scope instance; with each invocation involved in calculating the result. This provides a mechanism for efficient 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.


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.

#### 9.14.2. Vote Group Operations

The vote group operations allow invocations within a group to compare values across a group. The types of votes enabled are:

- Do all active group invocations agree that an expression is true?
• Do any active group invocations evaluate an expression to true?
• Do all active group invocations have the same value of an expression?

**Note**
These operations are useful in combination with control flow in that they allow for developers to check whether conditions match across the group and choose potentially faster code-paths in these cases.

### 9.14.3. Arithmetic Group Operations

The arithmetic group operations allow invocations to perform scans and reductions across a group. The operators supported are add, mul, min, max, and, or, xor.

For reductions, every invocation in a group will obtain the cumulative result of these operators applied to all values in the group. For exclusive scans, each invocation in a group will obtain the cumulative result of these operators applied to all values in invocations with a lower index in the group. Inclusive scans are identical to exclusive scans, except the cumulative result includes the operator applied to the value in the current invocation.

The order in which these operators are applied is implementation-dependent.


The ballot group operations allow invocations to perform more complex votes across the group. The ballot functionality allows all invocations within a group to provide a boolean value and get as a result what each invocation provided as their boolean value. The broadcast functionality allows values to be broadcast from an invocation to all other invocations within the group.

### 9.14.5. Shuffle Group Operations

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


The shuffle relative group operations allow invocations to read values from other invocations within the group relative to the current invocation in the group. The relative operations supported allow data to be shifted up and down through the invocations within a group.


The clustered group operations allow invocations to perform an operation among partitions of a group, such that the operation is only performed within the group invocations within a partition. The partitions for clustered group operations are consecutive power-of-two size groups of invocations and the cluster size must be known at pipeline creation time. The operations supported are add, mul, min, max, and, or, xor.
9.15. Quad Group Operations

Quad group operations (OpGroupNonUniformQuad*) are a specialized type of group operations that only operate on quad scope instances. Whilst these instructions do include a Scope parameter, this scope is always overridden; only the quad scope instance is included in its execution scope.

Fragment shaders that statically execute quad group operations must launch sufficient invocations to ensure their correct operation; additional helper invocations are launched for framebuffer locations not covered by rasterized fragments if necessary.

The index used to select participating invocations is i, as described for a quad scope instance, defined as the quad index in the SPIR-V specification.

For OpGroupNonUniformQuadBroadcast this value is equal to Index. For OpGroupNonUniformQuadSwap, it is equal to the implicit Index used by each participating invocation.

9.16. Derivative Operations

Derivative operations calculate the partial derivative for an expression P as a function of an invocation’s x and y coordinates.

Derivative operations operate on a set of invocations known as a derivative group as defined in the SPIR-V specification. A derivative group is equivalent to the primitive scope instance for a fragment shader invocation.

Derivatives are calculated assuming that P is piecewise linear and continuous within the derivative group. All dynamic instances of explicit derivative instructions (OpDPdx*, OpDPdy*, and OpFwidth*) must be executed in control flow that is uniform within a derivative group. For other derivative operations, results are undefined if a dynamic instance is executed in control flow that is not uniform within the derivative group.

Fragment shaders that statically execute derivative operations must launch sufficient invocations to ensure their correct operation; additional helper invocations are launched for framebuffer locations not covered by rasterized fragments if necessary.

Derivative operations calculate their results as the difference between the result of P across invocations in the quad. For fine derivative operations (OpDPdxFine and OpDPdyFine), the values of DPdx(Pi) are calculated as

\[
DPdx(P_0) = DPdx(P_1) = P_1 - P_0
\]

\[
DPdx(P_2) = DPdx(P_3) = P_3 - P_2
\]

and the values of DPdy(Pi) are calculated as

\[
DPdy(P_0) = DPdy(P_2) = P_2 - P_0
\]
DPdy(P_i) = DPdy(P_{m-1}) = P_{m-1} - P_1

where i is the index of each invocation as described in Quad.

Coarse derivative operations (OpDPdxCoarse and OpDPdyCoarse), calculate their results in roughly the same manner, but may only calculate two values instead of four (one for each of DPdx and DPdy), reusing the same result no matter the originating invocation. If an implementation does this, it should use the fine derivative calculations described for P_0.

**Note**

Derivative values are calculated between fragments rather than pixels. If the fragment shader invocations involved in the calculation cover multiple pixels, these operations cover a wider area, resulting in larger derivative values. This in turn will result in a coarser level of detail being selected for image sampling operations using derivatives.

Applications may want to account for this when using multi-pixel fragments; if pixel derivatives are desired, applications should use explicit derivative operations and divide the results by the size of the fragment in each dimension as follows:

\[
\begin{align*}
DPdx(P_n)' &= DPdx(P_n) / w \\
DPdy(P_n)' &= DPdy(P_n) / h
\end{align*}
\]

where w and h are the size of the fragments in the quad, and DPdx(P_n)' and DPdy(P_n)' are the pixel derivatives.

The results for OpDPdx and 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 OpFwidthFine, OpFwidthCoarse, or OpFwidth is equivalent to executing the corresponding OpDPdx* and OpDPdy* instructions, taking the absolute value of the results, and summing them.

Executing an OpImage*Sample*ImplicitLod instruction is equivalent to executing OpDPdx(Coordinate) and OpDPdy(Coordinate), and passing the results as the Grad operands dx and dy.

**Note**

It is expected that using the ImplicitLod variants of sampling functions will be substantially more efficient than using the ExplicitLod variants with explicitly generated derivatives.

### 9.17. Helper Invocations

When performing derivative or quad group operations in a fragment shader, additional
invocations may be spawned in order to ensure correct results. These additional invocations are known as *helper invocations* and can be identified by a non-zero value in the `HelperInvocation` built-in. Stores and atomics performed by helper invocations must not have any effect on memory, and values returned by atomic instructions in helper invocations are undefined.

For *group operations* other than *derivative* and *quad group* operations, helper invocations may be treated as inactive even if they would be considered otherwise active.

Helper invocations may become permanently inactive if all invocations in a quad scope instance become helper invocations.
Chapter 10. Pipelines

The following figure shows a block diagram of the Vulkan pipelines. Some Vulkan commands specify geometric objects to be drawn or computational work to be performed, while others specify state controlling how objects are handled by the various pipeline stages, or control data transfer between memory organized as images and buffers. Commands are effectively sent through a processing pipeline, either a graphics pipeline, or a compute pipeline.

The first stage of the graphics pipeline (Input Assembler) assembles vertices to form geometric primitives such as points, lines, and triangles, based on a requested primitive topology. In the next stage (Vertex Shader) vertices can be transformed, computing positions and attributes for each vertex. If tessellation and/or geometry shaders are supported, they can then generate multiple primitives from a single input primitive, possibly changing the primitive topology or generating additional attribute data in the process.

The final resulting primitives are clipped to a clip volume in preparation for the next stage, Rasterization. The rasterizer produces a series of fragments associated with a region of the framebuffer, from a two-dimensional description of a point, line segment, or triangle. These fragments are processed by fragment operations to determine whether generated values will be written to the framebuffer. Fragment shading determines the values to be written to the framebuffer attachments. Framebuffer operations then read and write the color and depth/stencil attachments of the framebuffer for a given subpass of a render pass instance. The attachments can be used as input attachments in the fragment shader in a later subpass of the same render pass.

The compute pipeline is a separate pipeline from the graphics pipeline, which operates on one-, two-, or three-dimensional workgroups which can read from and write to buffer and image memory.

This ordering is meant only as a tool for describing Vulkan, not as a strict rule of how Vulkan is implemented, and we present it only as a means to organize the various operations of the pipelines. Actual ordering guarantees between pipeline stages are explained in detail in the synchronization chapter.
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-pipelineCache-05022  
  pipelineCache must not be VK_NULL_HANDLE

- 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 the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• flags is a bitmask of VkPipelineCreateFlagBits specifying how the pipeline will be generated.
• stage is a VkPipelineShaderStageCreateInfo structure describing the compute shader.
• layout is the description of binding locations used by both the pipeline and descriptor sets used with the pipeline.
• **basePipelineHandle** is a pipeline to derive from. 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-stage-00701
  
  The **stage** member of **stage** must be **VK_SHADER_STAGE_COMPUTE_BIT**

- VUID-VkComputePipelineCreateInfo-stage-00702
  
  The shader code for the entry point identified by **stage** and the rest of the state identified by this structure must adhere to the pipeline linking rules described in the Shader Interfaces chapter

- VUID-VkComputePipelineCreateInfo-layout-00703
  
  **layout** must be consistent with the layout of the compute shader specified in **stage**

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

---

### 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 the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is a bitmask of `VkPipelineShaderStageCreateFlagBits` specifying how the pipeline shader stage will be generated.
- `stage` is a `VkShaderStageFlagBits` value specifying a single pipeline stage.
- `module` is a `VkShaderModule` object containing the shader 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 and the `module`, `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 geometry shaders feature is not enabled, `stage` must not be `VK_SHADER_STAGE_GEOMETRY_BIT`.

- **VUID-VkPipelineShaderStageCreateInfo-stage-00705**
  If the tessellation shaders feature is not enabled, `stage` must not be `VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT` or `VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT`.

- **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`. 
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`.

If the identified entry point includes any variable in its interface that is declared with the `ClipDistance BuiltIn` decoration, that variable **must** not have an array size greater than `VkPhysicalDeviceLimits::maxClipDistances`.

If the identified entry point includes any variable in its interface that is declared with the `CullDistance BuiltIn` decoration, that variable **must** not have an array size greater than `VkPhysicalDeviceLimits::maxCullDistances`.

If the identified entry point includes any variables in its interface that are declared with the `ClipDistance` or `CullDistance` `BuiltIn` decoration, those variables **must** not have array sizes which sum to more than `VkPhysicalDeviceLimits::maxCombinedClipAndCullDistances`.

If the identified entry point includes any variable in its interface that is declared with the `SampleMask BuiltIn` decoration, that variable **must** not have an array size greater than `VkPhysicalDeviceLimits::maxSampleMaskWords`.

If `stage` is `VK_SHADER_STAGE_VERTEX_BIT`, the identified entry point **must** not include any input variable in its interface that is decorated with `CullDistance`.

If `stage` is `VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT` or `VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT`, and the identified entry point has an `OpExecutionMode` instruction specifying a patch size with `OutputVertices`, the patch size **must** be greater than $0$ and less than or equal to `VkPhysicalDeviceLimits::maxTessellationPatchSize`.

If `stage` is `VK_SHADER_STAGE_GEOMETRY_BIT`, the identified entry point **must** have an `OpExecutionMode` instruction specifying a maximum output vertex count that is greater than $0$ and less than or equal to `VkPhysicalDeviceLimits::maxGeometryOutputVertices`.

If `stage` is `VK_SHADER_STAGE_GEOMETRY_BIT`, the identified entry point **must** have an `OpExecutionMode` instruction specifying an invocation count that is greater than $0$ and less than or equal to `VkPhysicalDeviceLimits::maxGeometryShaderInvocations`.

If `stage` is a pre-rasterization shader stage, and the identified entry point writes to `Layer` for any primitive, it **must** write the same value to `Layer` for all vertices of a given primitive.

If `stage` is a pre-rasterization shader stage, and the identified entry point writes to `ViewportIndex` for any primitive, it **must** write the same value to `ViewportIndex` for all vertices of a given primitive.
• VUID-VkPipelineShaderStageCreateInfo-stage-00718
  If stage is VK_SHADER_STAGE_FRAGMENT_BIT, the identified entry point must not include any output variables in its interface decorated with CullDistance

• VUID-VkPipelineShaderStageCreateInfo-stage-00719
  If stage is VK_SHADER_STAGE_FRAGMENT_BIT, and the identified entry point writes to FragDepth in any execution path, it must write to FragDepth in all execution paths

• VUID-VkPipelineShaderStageCreateInfo-stage-01511
  If stage is VK_SHADER_STAGE_FRAGMENT_BIT, and the identified entry point writes to FragStencilRefEXT in any execution path, it must write to FragStencilRefEXT in all execution paths

• VUID-VkPipelineShaderStageCreateInfo-flags-02784
  If flags has the VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT flag set, the subgroupSizeControl feature must be enabled

• VUID-VkPipelineShaderStageCreateInfo-flags-02785
  If flags has the VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT flag set, the computeFullSubgroups feature must be enabled

• VUID-VkPipelineShaderStageCreateInfo-pNext-02754
  If a VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT structure is included in the pNext chain, flags must not have the VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT flag set

• VUID-VkPipelineShaderStageCreateInfo-pNext-02755
  If a VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT structure is included in the pNext chain, the subgroupSizeControl feature must be enabled, and stage must be a valid bit specified in requiredSubgroupSizeStages

• VUID-VkPipelineShaderStageCreateInfo-pNext-02756
  If a VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT structure is included in the pNext chain and stage is VK_SHADER_STAGE_COMPUTE_BIT, the local workgroup size of the shader must be less than or equal to the product of VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT::requiredSubgroupSize and maxComputeWorkgroupSubgroups

• VUID-VkPipelineShaderStageCreateInfo-pNext-02757
  If a VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT structure is included in the pNext chain, and flags has the VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT flag set, the local workgroup size in the X dimension of the pipeline must be a multiple of VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT::requiredSubgroupSize

• VUID-VkPipelineShaderStageCreateInfo-flags-02758
  If flags has both the VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT and VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT flags set, the local workgroup size in the X dimension of the pipeline must be a multiple of maxSubgroupSize

• VUID-VkPipelineShaderStageCreateInfo-flags-02759
  If flags has the VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT flag set and flags does not have the
VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT flag set and no VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT structure is included in the pNext chain, the local workgroup size in the X dimension of the pipeline must be a multiple of subgroupSize

- VUID-VkPipelineShaderStageCreateInfo-module-04145
  The SPIR-V code that was used to create module must be valid as described by the Khronos SPIR-V Specification after applying the specializations provided in pSpecializationInfo, if any, and then converting all specialization constants into fixed constants

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

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

- VUID-VkPipelineShaderStageCreateInfo-flags-parameter
  flags must be a valid combination of VkPipelineShaderStageCreateFlagBits values

- 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:

```c
// Provided by VK_VERSION_1_0
typedef enum VkPipelineShaderStageCreateFlagBits {
  // Provided by VK_EXT_subgroup_size_control
```
VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT = 0x00000001,
    // Provided by VK_EXT_subgroup_size_control
    VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT = 0x00000002,
} VkPipelineShaderStageCreateFlagBits;

- **VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT** specifies that the SubgroupSize may vary in the shader stage.
- **VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT** specifies that the subgroup sizes must be launched with all invocations active in the compute stage.

**Note**

If **VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT** and **VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT** are specified and minSubgroupSize does not equal maxSubgroupSize and no required subgroup size is specified, then the only way to guarantee that the 'X' dimension of the local workgroup size is a multiple of SubgroupSize is to make it a multiple of maxSubgroupSize. Under these conditions, you are guaranteed full subgroups but not any particular subgroup size.

Commands and structures which need to specify one or more shader stages do so using a bitmask whose bits correspond to stages. Bits which can be set to specify shader stages are:

```
// Provided by VK_VERSION_1_0
typedef enum VkShaderStageFlagBits {
    VK_SHADER_STAGE_VERTEX_BIT = 0x00000001,
    VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT = 0x00000002,
    VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT = 0x00000004,
    VK_SHADER_STAGE_GEOMETRY_BIT = 0x00000008,
    VK_SHADER_STAGE_FRAGMENT_BIT = 0x00000010,
    VK_SHADER_STAGE_COMPUTE_BIT = 0x00000020,
    VK_SHADER_STAGE_ALL_GRAPHICS = 0x0000001F,
    VK_SHADER_STAGE_ALL = 0x7FFFFFFF,
} VkShaderStageFlagBits;
```

- **VK_SHADER_STAGE_VERTEX_BIT** specifies the vertex stage.
- **VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT** specifies the tessellation control stage.
- **VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT** specifies the tessellation evaluation stage.
- **VK_SHADER_STAGE_GEOMETRY_BIT** specifies the geometry stage.
- **VK_SHADER_STAGE_FRAGMENT_BIT** specifies the fragment stage.
- **VK_SHADER_STAGE_COMPUTE_BIT** specifies the compute stage.
- **VK_SHADER_STAGE_ALL_GRAPHICS** is a combination of bits used as shorthand to specify all graphics stages defined above (excluding the compute stage).
- **VK_SHADER_STAGE_ALL** is a combination of bits used as shorthand to specify all shader stages supported by the device, including all additional stages which are introduced by extensions.
Note

**VK_SHADER_STAGE_ALL_GRAPHICS** only includes the original five graphics stages included in Vulkan 1.0, and not any stages added by extensions. Thus, it may not have the desired effect in all cases.

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

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

The **VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT** structure is defined as:

```c
// Provided by VK_EXT_subgroup_size_control
typedef struct VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT {
    VkStructureType sType;
    void* pNext;
    uint32_t requiredSubgroupSize;
} VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **requiredSubgroupSize** is an unsigned integer value specifying the required subgroup size for the newly created pipeline shader stage.

**Valid Usage**

- VUID-VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT-requiredSubgroupSize-02760
  
  **requiredSubgroupSize** must be a power-of-two integer.

- VUID-VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT-requiredSubgroupSize-02761

  **requiredSubgroupSize** must be greater or equal to **minSubgroupSize**

- VUID-VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT-requiredSubgroupSize-02762

  **requiredSubgroupSize** must be less than or equal to **maxSubgroupSize**

If a **VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT** structure is included in the **pNext** chain of **VkPipelineShaderStageCreateInfo**, it specifies that the pipeline shader stage being compiled has a required subgroup size.

**Valid Usage (Implicit)**

- VUID-VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT-sType-sType
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-pipelineCache-05031
pipelineCache must not be VK_NULL_HANDLE

- 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
  If pipelineCache is not VK_NULL_HANDLE, pipelineCache must be a valid VkPipelineCache handle

- VUID-vkCreateGraphicsPipelines-pCreateInfos-parameter
  pCreateInfos must be a valid pointer to an array of createInfoCount valid VkGraphicsPipelineCreateInfo structures

- VUID-vkCreateGraphicsPipelines-pAllocator-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
  If pipelineCache is a valid handle, it must have been created, allocated, or retrieved from device

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_NO_PIPELINE_MATCH
- VK_ERROR_OUT_OF_POOL_MEMORY

The VkGraphicsPipelineCreateInfo structure is defined as:

// Provided by VK_VERSION_1_0
typedef struct VkGraphicsPipelineCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineCreateFlags flags;
    uint32_t stageCount;
    const VkPipelineShaderStageCreateInfo* pStages;
    const VkPipelineVertexInputStateCreateInfo* pVertexInputState;
    const VkPipelineInputAssemblyStateCreateInfo* pInputAssemblyState;
    const VkPipelineTessellationStateCreateInfo* pTessellationState;
    const VkPipelineViewportStateCreateInfo* pViewportState;
    const VkPipelineRasterizationStateCreateInfo* pRasterizationState;
    const VkPipelineMultisampleStateCreateInfo* pMultisampleState;
    const VkPipelineDepthStencilStateCreateInfo* pDepthStencilState;
    const VkPipelineColorBlendStateCreateInfo* pColorBlendState;
    const VkPipelineDynamicStateCreateInfo* pDynamicState;
    VkPipelineLayout layout;
    VkRenderPass renderPass;
    uint32_t subpass;
    VkPipeline basePipelineHandle;
    int32_t basePipelineIndex;
} VkGraphicsPipelineCreateInfo;

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• flags is a bitmask of VkPipelineCreateFlagBits specifying how the pipeline will be generated.
• stageCount is the number of entries in the pStages array.
• pStages is a pointer to an array of stageCount VkPipelineShaderStageCreateInfo structures describing the set of the shader stages to be included in the graphics pipeline.
• pVertexInputState is a pointer to a VkPipelineVertexInputStateCreateInfo structure. It is ignored if the pipeline is created with the VK_DYNAMIC_STATE_VERTEX_INPUT_EXT dynamic state set.
• pInputAssemblyState is a pointer to a VkPipelineInputAssemblyStateCreateInfo structure which determines input assembly behavior, as described in Drawing Commands.
• pTessellationState is a pointer to a VkPipelineTessellationStateCreateInfo structure, and is ignored if the pipeline does not include a tessellation control shader stage and tessellation evaluation shader stage.
• pViewportState is a pointer to a VkPipelineViewportStateCreateInfo structure, and is ignored if the pipeline has rasterization disabled.
• pRasterizationState is a pointer to a VkPipelineRasterizationStateCreateInfo structure.
• pMultisampleState is a pointer to a VkPipelineMultisampleStateCreateInfo structure, and is ignored if the pipeline has rasterization disabled.
• pDepthStencilState is a pointer to a VkPipelineDepthStencilStateCreateInfo structure, and is ignored if the pipeline has rasterization disabled or if no depth/stencil attachment is used.
• pColorBlendState is a pointer to a VkPipelineColorBlendStateCreateInfo structure, and is ignored if the pipeline has rasterization disabled or if no color attachments are used.
• **pDynamicState** is a pointer to a `VkPipelineDynamicStateCreateInfo` structure, and is used to indicate which properties of the pipeline state object are dynamic and can be changed independently of the pipeline state. This can be **NULL**, which means no state in the pipeline is considered dynamic.

• **layout** is the description of binding locations used by both the pipeline and descriptor sets used with the pipeline.

• **renderPass** is a handle to a render pass object describing the environment in which the pipeline will be used. The pipeline **must** only be used with 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 is defined by:

- `VkPipelineVertexInputStateCreateInfo`
- `VkPipelineInputAssemblyStateCreateInfo`

Pre-rasterization shader state is defined by:

- `VkPipelineShaderStageCreateInfo` entries for:
  - Vertex shaders
  - Tessellation control shaders
  - Tessellation evaluation shaders
  - Geometry shaders
- **Within the **`VkPipelineLayout`, all bindings that affect the specified shader stages
- `VkPipelineViewportStateCreateInfo`
- `VkPipelineRasterizationStateCreateInfo`
- `VkPipelineTessellationStateCreateInfo` if tessellation stages are included.
- `VkRenderPass` and **subpass** parameter
- `VkPipelineDiscardRectangleStateCreateInfoEXT`
- `VkPipelineFragmentShadingRateStateCreateInfoKHR`

Fragment shader state is defined by:

- A `VkPipelineShaderStageCreateInfo` entry for the fragment shader
• Within the `VkPipelineLayout`, all bindings that affect the fragment shader
• `VkPipelineMultisampleStateCreateInfo`
• `VkPipelineDepthStencilStateCreateInfo`
• `VkRenderPass` and `subpass` parameter
• `VkPipelineFragmentShadingRateStateCreateInfoKHR`

Fragment output state is defined by:

• `VkPipelineColorBlendStateCreateInfo`
• The `alphaToCoverageEnable` and `alphaToOneEnable` members of `VkPipelineMultisampleStateCreateInfo`
• `VkRenderPass` and `subpass` parameter

A complete graphics pipeline always includes pre-rasterization shader state, with other subsets included depending on that state. If the pre-rasterization shader state includes a vertex shader, then vertex input state is included in a complete graphics pipeline. If the value of `VkPipelineRasterizationStateCreateInfo::rasterizerDiscardEnable` in the pre-rasterization shader state is `VK_FALSE` or the `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT` dynamic state is enabled, fragment shader state and fragment output interface state is included in a complete graphics pipeline.

Pipelines must be created with a complete set of pipeline state.

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-00533
  `basePipelineHandle` must be `VK_NULL_HANDLE`

• VUID-VkGraphicsPipelineCreateInfo-basePipelineIndex-00534
  `basePipelineIndex` must be zero

• VUID-VkGraphicsPipelineCreateInfo-stage-00726
  The `stage` member of each element of `pStages` must be unique

• VUID-VkGraphicsPipelineCreateInfo-stage-00727
  If the pipeline is being created with pre-rasterization shader state the `stage` member of one element of `pStages` must be `VK_SHADER_STAGE_VERTEX_BIT`

• VUID-VkGraphicsPipelineCreateInfo-stage-00728
  The `stage` member of each element of `pStages` must not be `VK_SHADER_STAGE_COMPUTE_BIT`

• VUID-VkGraphicsPipelineCreateInfo-pStages-00729
  If the pipeline is being created with pre-rasterization shader state and `pStages` includes a tessellation control shader stage, it must include a tessellation evaluation shader stage

• VUID-VkGraphicsPipelineCreateInfo-pStages-00730
If the pipeline is being created with pre-rasterization shader state and the `pStages` includes a tessellation evaluation shader stage, it must include a tessellation control shader stage.

- **VUID-VkGraphicsPipelineCreateInfo-pStages-00731**
  If the pipeline is being created with pre-rasterization shader state and the `pStages` includes a tessellation control shader stage and a tessellation evaluation shader stage, the `pTessellationState` must be a valid pointer to a valid `VkPipelineTessellationStateCreateInfo` structure.

- **VUID-VkGraphicsPipelineCreateInfo-pStages-00732**
  If the pipeline is being created with pre-rasterization shader state and the `pStages` includes tessellation shader stages, the shader code of at least one stage must contain an `OpExecutionMode` instruction specifying the type of subdivision in the pipeline.

- **VUID-VkGraphicsPipelineCreateInfo-pStages-00733**
  If the pipeline is being created with pre-rasterization shader state and the `pStages` includes tessellation shader stages, and the shader code of both stages contain an `OpExecutionMode` instruction specifying the type of subdivision in the pipeline, they must both specify the same subdivision mode.

- **VUID-VkGraphicsPipelineCreateInfo-pStages-00734**
  If the pipeline is being created with pre-rasterization shader state and the `pStages` includes tessellation shader stages, the shader code of at least one stage must contain an `OpExecutionMode` instruction specifying the output patch size in the pipeline.

- **VUID-VkGraphicsPipelineCreateInfo-pStages-00735**
  If the pipeline is being created with pre-rasterization shader state and the `pStages` includes tessellation shader stages, and the shader code of both contain an `OpExecutionMode` instruction specifying the output patch size in the pipeline, they must both specify the same patch size.

- **VUID-VkGraphicsPipelineCreateInfo-pStages-00736**
  If the pipeline is being created with pre-rasterization shader state and the `pStages` includes tessellation shader stages, the `topology` member of `pInputAssembly` must be `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`.

- **VUID-VkGraphicsPipelineCreateInfo-pStages-00737**
  If the pipeline is being created with pre-rasterization shader state and the `topology` member of `pInputAssembly` is `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`, the `pStages` must include tessellation shader stages.

- **VUID-VkGraphicsPipelineCreateInfo-pStages-00738**
  If the pipeline is being created with pre-rasterization shader state and the `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`.

- **VUID-VkGraphicsPipelineCreateInfo-pStages-00739**
  If the pipeline is being created with pre-rasterization shader state and the `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.

- **VUID-VkGraphicsPipelineCreateInfo-pStages-00740**
  If the pipeline is being created with pre-rasterization shader state and the `pStages` includes a geometry shader stage, and also includes tessellation shader stages, its shader code must contain an `OpExecutionMode` instruction specifying an input primitive type that is compatible with the primitive topology that is output by the tessellation stages.
If the pipeline is being created with pre-rasterization shader state and fragment shader state, it includes both a fragment shader and a geometry shader, and the fragment shader code reads from an input variable that is decorated with PrimitiveId, then the geometry shader code must write to a matching output variable, decorated with PrimitiveId, in all execution paths.

- VUID-VkGraphicsPipelineCreateInfo-renderPass-06038
  If renderPass is not VK_NULL_HANDLE and the pipeline is being created with fragment shader state the fragment shader must not read from any input attachment that is defined as VK_ATTACHMENT_UNUSED in subpass.

- VUID-VkGraphicsPipelineCreateInfo-pStages-00742
  If the pipeline is being created with pre-rasterization shader state and multiple pre-rasterization shader stages are included in pStages, the shader code for the entry points identified by those pStages and the rest of the state identified by this structure must adhere to the pipeline linking rules described in the Shader Interfaces chapter.

- VUID-VkGraphicsPipelineCreateInfo-None-04889
  If the pipeline is being created with pre-rasterization shader state and fragment shader state, the fragment shader and last pre-rasterization shader stage and any relevant state must adhere to the pipeline linking rules described in the Shader Interfaces chapter.

- VUID-VkGraphicsPipelineCreateInfo-renderPass-06039
  If renderPass is not VK_NULL_HANDLE, the pipeline is being created with fragment shader state, and subpass uses a depth/stencil attachment in renderPass with a read-only layout for the depth aspect in the VkAttachmentReference defined by subpass, the depthWriteEnable member of pDepthStencilState must be VK_FALSE.

- VUID-VkGraphicsPipelineCreateInfo-renderPass-06040
  If renderPass is not VK_NULL_HANDLE, the pipeline is being created with fragment shader state, and subpass uses a depth/stencil attachment in renderPass with a read-only layout for the stencil aspect in the VkAttachmentReference defined by subpass, the failOp, passOp and depthFailOp members of each of the front and back members of pDepthStencilState must be VK_STENCIL_OP_KEEP.

- VUID-VkGraphicsPipelineCreateInfo-renderPass-06041
  If renderPass is not VK_NULL_HANDLE, and the pipeline is being created with fragment output interface state, then for each color attachment in the subpass, if the potential format features of the format of the corresponding attachment description do not contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT, then the blendEnable member of the corresponding element of the pAttachments member of pColorBlendState must be VK_FALSE.

- VUID-VkGraphicsPipelineCreateInfo-renderPass-06042
  If renderPass is not VK_NULL_HANDLE, and the pipeline is being created with fragment output interface state, and the subpass uses color attachments, the attachmentCount member of pColorBlendState must be equal to the colorAttachmentCount used to create subpass.

- VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-04130
  If the pipeline is being created with pre-rasterization shader state, and no element of the pDynamicStates member of pDynamicState is VK_DYNAMIC_STATE_VIEWPORT or VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT, the pViewports member of pViewportState must be a valid pointer to an array of pViewportState->viewportCount valid VkViewport structures.
If the pipeline is being created with pre-rasterization shader state, and no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_SCISSOR` or `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT`, the `pScissors` member of `pViewportState` must be a valid pointer to an array of `pViewportState->scissorCount` `VkRect2D` structures.

If the pipeline is being created with pre-rasterization shader state, and the wide lines feature is not enabled, and no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_LINE_WIDTH`, the `lineWidth` member of `pRasterizationState` must be `1.0`.

If the pipeline is being created with pre-rasterization shader state, and the rasterizerDiscardEnable member of `pRasterizationState` is `VK_FALSE`, `pViewportState` must be a valid pointer to a valid `VkPipelineViewportStateCreateInfo` structure.

If the pipeline is being created with fragment shader state, `pMultisampleState` must be a valid pointer to a valid `VkPipelineMultisampleStateCreateInfo` structure.

If `renderPass` is not `VK_NULL_HANDLE`, the pipeline is being created with fragment shader state, and `subpass` uses a depth/stencil attachment, `pDepthStencilState` must be a valid pointer to a valid `VkPipelineDepthStencilStateCreateInfo` structure.

If `renderPass` is not `VK_NULL_HANDLE`, the pipeline is being created with fragment output interface state, and `subpass` uses color attachments, `pColorBlendState` must be a valid pointer to a valid `VkPipelineColorBlendStateCreateInfo` structure.

If `renderPass` is not `VK_NULL_HANDLE` and the pipeline is being created with fragment output interface state, `pColorBlendState->attachmentCount` must be greater than the index of all color attachments that are not `VK_ATTACHMENT_UNUSED` for the `subpass` index in `renderPass`.

If the pipeline is being created with pre-rasterization shader state, the depth bias clamping feature is not enabled, no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_DEPTH_BIAS`, and the `depthBiasEnable` member of `pRasterizationState` is `VK_TRUE`, the `depthBiasClamp` member of `pRasterizationState` must be `0.0`.

If the pipeline is being created with fragment shader state, and the `VK_EXT_depth_range_unrestricted` extension is not enabled and no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_DEPTH_BOUNDS`, and the
depthBoundsTestEnable member of pDepthStencilState is VK_TRUE, the minDepthBounds and maxDepthBounds members of pDepthStencilState must be between 0.0 and 1.0, inclusive.

- VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-01521
  If the pipeline is being created with fragment shader state, and no element of the pDynamicStates member of pDynamicState is VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT, and the sampleLocationsEnable member of a VkPipelineSampleLocationsStateCreateInfoEXT structure included in the pNext chain of pMultisampleState is VK_TRUE, sampleLocationsInfo.sampleLocationGridSize.width must evenly divide VkMultisamplePropertiesEXT::sampleLocationGridSize.width as returned by vkGetPhysicalDeviceMultisamplePropertiesEXT with a samples parameter equaling rasterizationSamples.

- VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-01522
  If the pipeline is being created with fragment shader state, and no element of the pDynamicStates member of pDynamicState is VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT, and the sampleLocationsEnable member of a VkPipelineSampleLocationsStateCreateInfoEXT structure included in the pNext chain of pMultisampleState is VK_TRUE, sampleLocationsInfo.sampleLocationGridSize.height must evenly divide VkMultisamplePropertiesEXT::sampleLocationGridSize.height as returned by vkGetPhysicalDeviceMultisamplePropertiesEXT with a samples parameter equaling rasterizationSamples.

- VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-01523
  If the pipeline is being created with fragment shader state, and no element of the pDynamicStates member of pDynamicState is VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT, and the sampleLocationsEnable member of a VkPipelineSampleLocationsStateCreateInfoEXT structure included in the pNext chain of pMultisampleState is VK_TRUE, sampleLocationsInfo.sampleLocationsPerPixel must equal rasterizationSamples.

- VUID-VkGraphicsPipelineCreateInfo-sampleLocationsEnable-01524
  If the pipeline is being created with fragment shader state, and the sampleLocationsEnable member of a VkPipelineSampleLocationsStateCreateInfoEXT structure included in the pNext chain of pMultisampleState is VK_TRUE, the fragment shader code must not statically use the extended instruction InterpolateAtSample.

- VUID-VkGraphicsPipelineCreateInfo-layout-00756
  layout must be consistent with all shaders specified in pStages.

- VUID-VkGraphicsPipelineCreateInfo-subpass-00757
  If the pipeline is being created with fragment shader state, and neither the VK_AMD_mixed_attachment_samples nor the VK_NV_framebuffer_mixed_samples extensions are enabled, and if subpass uses color and/or depth/stencil attachments, then the rasterizationSamples member of pMultisampleState must be the same as the sample count for those subpass attachments.

- VUID-VkGraphicsPipelineCreateInfo-subpass-00758
  If the pipeline is being created with fragment shader state and subpass does not use any color and/or depth/stencil attachments, then the rasterizationSamples member of pMultisampleState must follow the rules for a zero-attachment subpass.

- VUID-VkGraphicsPipelineCreateInfo-renderPass-06046
  If renderPass is a valid renderPass, subpass must be a valid subpass within renderPass.
If \texttt{renderPass} is a valid renderPass, the pipeline is being created with \textit{pre-rasterization shader state}, and the \texttt{renderPass} has multiview enabled and \texttt{subpass} has more than one bit set in the view mask and \textit{multiviewTessellationShader} is not enabled, then \texttt{pStages} \textbf{must} not include tessellation shaders.

If \texttt{renderPass} is a valid renderPass, the pipeline is being created with \textit{pre-rasterization shader state}, and the \texttt{renderPass} has multiview enabled and \texttt{subpass} has more than one bit set in the view mask and \textit{multiviewGeometryShader} is not enabled, then \texttt{pStages} \textbf{must} not include a geometry shader.

If \texttt{renderPass} is a valid renderPass, the pipeline is being created with \textit{pre-rasterization shader state}, and the \texttt{renderPass} has multiview enabled and \texttt{subpass} has more than one bit set in the view mask, shaders in the pipeline \textbf{must} not write to the \textit{Layer} built-in output.

If \texttt{renderPass} is a valid renderPass and the pipeline is being created with \textit{pre-rasterization shader state}, and the \texttt{renderPass} has multiview enabled, then all shaders \textbf{must} not include variables decorated with the \textit{Layer} built-in decoration in their interfaces.

flags \textbf{must} not contain the \texttt{VK_PIPELINE_CREATE_DISPATCH_BASE} flag.

If the pipeline is being created with \textit{fragment shader state} and an input attachment was referenced by an \textit{aspectMask} at \texttt{renderPass} creation time, the fragment shader \textbf{must} only read from the aspects that were specified for that input attachment.

The number of resources in \texttt{layout} accessible to each shader stage that is used by the pipeline \textbf{must} be less than or equal to \texttt{VkPhysicalDeviceLimits::maxPerStageResources}.

If the pipeline is being created with \textit{pre-rasterization shader state}, and no element of the \texttt{pDynamicStates} member of \texttt{pDynamicState} is \texttt{VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT}, and if \texttt{pNext} chain includes a \texttt{VkPipelineDiscardRectangleStateCreateInfoEXT} structure, and if its \texttt{discardRectangleCount} member is not 0, then its \texttt{pDiscardRectangles} member \textbf{must} be a valid pointer to an array of \texttt{discardRectangleCount} \texttt{VkRect2D} structures.

If the pipeline is being created with \textit{vertex input state}, \texttt{pInputAssemblyState} \textbf{must} be a valid pointer to a valid \texttt{VkPipelineInputAssemblyStateCreateInfo} structure.

If the pipeline is being created with \textit{pre-rasterization shader state} and at least one of \textit{fragment output interface state} or \textit{fragment shader state}, the \texttt{lineRasterizationMode} member of a \texttt{VkPipelineRasterizationLineStateCreateInfoEXT} structure included in the \texttt{pNext} chain of \texttt{pRasterizationState} is \texttt{VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT} or \texttt{VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT}, then the \texttt{alphaToCoverageEnable}, \texttt{alphaToOneEnable}, and \texttt{sampleShadingEnable} members of \texttt{pMultisampleState} \textbf{must} all be \texttt{VK_FALSE}.
If the pipeline is being created with pre-rasterization shader state, the stippledLineEnable member of VkPipelineRasterizationLineStateCreateInfoEXT is VK_TRUE, and no element of the pDynamicStates member of pDynamicState is VK_DYNAMIC_STATE_LINE_STIPPLE_EXT, then the lineStippleFactor member of VkPipelineRasterizationLineStateCreateInfoEXT must be in the range [1, 256].

If the extendedDynamicState feature is not enabled, there must be no element of the pDynamicStates member of pDynamicState set to VK_DYNAMIC_STATE_CULL_MODE_EXT, VK_DYNAMIC_STATE_FRONT_FACE_EXT, VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT, VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT, VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT, VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT, VK_DYNAMIC_STATE_DEPTH_TEST_ENABLE_EXT, VK_DYNAMIC_STATE_DEPTH_WRITE_ENABLE_EXT, VK_DYNAMIC_STATE_DEPTH_COMPARE_OP_EXT, VK_DYNAMIC_STATE_DEPTH_BOUNDS_TEST_ENABLE_EXT, VK_DYNAMIC_STATE_STENCIL_TEST_ENABLE_EXT, or VK_DYNAMIC_STATE_STENCIL_OP_EXT.

If the pipeline is being created with pre-rasterization shader state, and VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT is included in the pDynamicStates array then viewportCount must be zero.

If the pipeline is being created with pre-rasterization shader state, and VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT is included in the pDynamicStates array then scissorCount must be zero.

If the pipeline is being created with pre-rasterization shader state, and VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT is included in the pDynamicStates array then VK_DYNAMIC_STATE_VIEWPORT must not be present.

If the pipeline is being created with pre-rasterization shader state, and VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT is included in the pDynamicStates array then VK_DYNAMIC_STATE_SCISSOR must not be present.

If the extendedDynamicState2 feature is not enabled, there must be no element of the pDynamicStates member of pDynamicState set to VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT, VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT, or VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT.

If the extendedDynamicState2LogicOp feature is not enabled, there must be no element of the pDynamicStates member of pDynamicState set to VK_DYNAMIC_STATE_LOGIC_OP_EXT.

If the extendedDynamicState2PatchControlPoints feature is not enabled, there must be no element of the pDynamicStates member of pDynamicState set to VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT.
If the pipeline is being created with pre-rasterization shader state or fragment shader state and `VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR` is not included in `pDynamicState->pDynamicStates`, `VkPipelineFragmentShadingRateStateCreateInfoKHR::fragmentSize.width` must be greater than or equal to 1

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicState-04495**
  If the pipeline is being created with pre-rasterization shader state or fragment shader state and `VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR` is not included in `pDynamicState->pDynamicStates`, `VkPipelineFragmentShadingRateStateCreateInfoKHR::fragmentSize.height` must be greater than or equal to 1

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicState-04496**
  If the pipeline is being created with pre-rasterization shader state or fragment shader state and `VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR` is not included in `pDynamicState->pDynamicStates`, `VkPipelineFragmentShadingRateStateCreateInfoKHR::fragmentSize.width` must be a power-of-two value

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicState-04497**
  If the pipeline is being created with pre-rasterization shader state or fragment shader state and `VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR` is not included in `pDynamicState->pDynamicStates`, `VkPipelineFragmentShadingRateStateCreateInfoKHR::fragmentSize.height` must be a power-of-two value

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicState-04498**
  If the pipeline is being created with pre-rasterization shader state or fragment shader state and `VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR` is not included in `pDynamicState->pDynamicStates`, and the `pipelineFragmentShadingRate` feature is not enabled, `VkPipelineFragmentShadingRateStateCreateInfoKHR::fragmentSize.width` and `VkPipelineFragmentShadingRateStateCreateInfoKHR::fragmentSize.height` must both be equal to 1

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicState-04500**
  If the pipeline is being created with pre-rasterization shader state or fragment shader state and `VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR` is not included in `pDynamicState->pDynamicStates`, and the `pipelineFragmentShadingRate` feature is not enabled, `VkPipelineFragmentShadingRateStateCreateInfoKHR::combinerOps[0]` must be `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR`

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicState-04502**
  If the pipeline is being created with pre-rasterization shader state or fragment shader state and `VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR` is not included in `pDynamicState->pDynamicStates`, and the `primitiveFragmentShadingRate` feature is not enabled, `VkPipelineFragmentShadingRateStateCreateInfoKHR::combinerOps[0]` must be `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR`
DynamicStates, and the attachmentFragmentShadingRate feature is not enabled, VkPipelineFragmentShadingRateStateCreateInfoKHR::combinerOps[1] must be VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR

- VUID-VkGraphicsPipelineCreateInfo-primitiveFragmentShadingRateWithMultipleViewports-04503
  If the pipeline is being created with pre-rasterization shader state and the primitiveFragmentShadingRateWithMultipleViewports limit is not supported, VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT is not included in pDynamicState->pDynamicStates, and VkPipelineViewportStateCreateInfo::viewportCount is greater than 1, entry points specified in pStages must not write to the PrimitiveShadingRateKHR built-in

- VUID-VkGraphicsPipelineCreateInfo-primitiveFragmentShadingRateWithMultipleViewports-04504
  If the pipeline is being created with pre-rasterization shader state and the primitiveFragmentShadingRateWithMultipleViewports limit is not supported, and entry points specified in pStages write to the ViewportIndex built-in, they must not also write to the PrimitiveShadingRateKHR built-in

- VUID-VkGraphicsPipelineCreateInfo-fragmentShadingRateNonTrivialCombinerOps-04506
  If the pipeline is being created with pre-rasterization shader state or fragment shader state, the fragmentShadingRateNonTrivialCombinerOps limit is not supported, and VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR is not included in pDynamicState->pDynamicStates, elements of VkPipelineFragmentShadingRateStateCreateInfoKHR::combinerOps must be VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR or VK_FRAGMENT_SHADING_RATE_COMBINER_OP_REPLACE_KHR

- VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-04807
  If the pipeline is being created with pre-rasterization shader state and the vertexInputDynamicState feature is not enabled, there must be no element of the pDynamicStates member of pDynamicState set to VK_DYNAMIC_STATE_VERTEX_INPUT_EXT

- VUID-VkGraphicsPipelineCreateInfo-None-04893
  The pipeline must be created with a complete set of state

- VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-04800
  If the colorWriteEnable feature is not enabled, there must be no element of the pDynamicStates member of pDynamicState set to VK_DYNAMIC_STATE_COLOR_WRITE_ENABLE_EXT

- VUID-VkGraphicsPipelineCreateInfo-renderPass-06051
  renderPass must not be VK_NULL_HANDLE

**Valid Usage (Implicit)**

- VUID-VkGraphicsPipelineCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO

- VUID-VkGraphicsPipelineCreateInfo-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 VkPipelineDiscardRectangleStateCreateInfoEXT, VkPipelineFragmentShadingRateStateCreateInfoKHR, or VkPipelineOfflineCreateInfo
The `sType` value of each struct in the `pNext` chain must be unique.

Flags must be a valid combination of `VkPipelineCreateFlagBits` values.

If `stageCount` is not 0, `pStages` must be a valid pointer to an array of `stageCount` valid `VkPipelineShaderStageCreateInfo` structures.

`pRasterizationState` must be a valid pointer to a valid `VkPipelineRasterizationStateCreateInfo` structure.

If `pDynamicState` is not `NULL`, `pDynamicState` must be a valid pointer to a valid `VkPipelineDynamicStateCreateInfo` structure.

`layout` must be a valid `VkPipelineLayout` handle.

If `renderPass` is not `VK_NULL_HANDLE`, `renderPass` must be a valid `VkRenderPass` handle.

Each of `basePipelineHandle`, `layout`, and `renderPass` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`.

Possible values of the `flags` member of `VkGraphicsPipelineCreateInfo`, and `VkComputePipelineCreateInfo`, specifying how a pipeline is created, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkPipelineCreateFlagBits {
    VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT = 0x00000001,
    VK_PIPELINE_CREATE_VIEW_INDEX_FROM_DEVICE_INDEX_BIT = 0x00000008,
    VK_PIPELINE_CREATE_DISPATCH_BASE_BIT = 0x00000010,
    VK_PIPELINE_CREATE_DISPATCH_BASE = VK_PIPELINE_CREATE_DISPATCH_BASE_BIT,
} VkPipelineCreateFlagBits;
```

- `VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT` specifies that the created pipeline will not be optimized. Using this flag may reduce the time taken to create the pipeline.
- `VK_PIPELINE_CREATE_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.
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 the type of 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:

```c
// Provided by VK_VERSION_1_0
typedef enum VkDynamicState {
    VK_DYNAMIC_STATE_VIEWPORT = 0,
    VK_DYNAMIC_STATE_SCISSOR = 1,
    VK_DYNAMIC_STATE_LINE_WIDTH = 2,
    VK_DYNAMIC_STATE_DEPTH_BIAS = 3,
    VK_DYNAMIC_STATE_BLEND_CONSTANTS = 4,
    VK_DYNAMIC_STATE_DEPTH_BOUNDS = 5,
    VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK = 6,
    VK_DYNAMIC_STATE_STENCIL_WRITE_MASK = 7,
    VK_DYNAMIC_STATE_STENCIL_REFERENCE = 8,
    // Provided by VK_EXT_discard_rectangles
    VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT = 100099000,
    // Provided by VK_EXT_sample_locations
    VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT = 100143000,
    // Provided by VK_KHR_fragment_shading_rate
    VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR = 1000226000,
    // Provided by VK_EXT_line_rasterization
    VK_DYNAMIC_STATE_LINE_STIPPLE_EXT = 1000259000,
    // Provided by VK_EXT_extended_dynamic_state
    VK_DYNAMIC_STATE_CULL_MODE_EXT = 1000267000,
    // Provided by VK_EXT_extended_dynamic_state
    VK_DYNAMIC_STATE_FRONT_FACE_EXT = 1000267001,
    // Provided by VK_EXT_extended_dynamic_state
    VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT = 1000267002,
    // Provided by VK_EXT_extended_dynamic_state
    VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT = 1000267003,
    // Provided by VK_EXT_extended_dynamic_state
    VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT = 1000267004,
    // Provided by VK_EXT_extended_dynamic_state
    VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT = 1000267005,
    // Provided by VK_EXT_extended_dynamic_state
    VK_DYNAMIC_STATE_DEPTH_TEST_ENABLE_EXT = 1000267006,
    // Provided by VK_EXT_extended_dynamic_state
    VK_DYNAMIC_STATE_DEPTH_WRITE_ENABLE_EXT = 1000267007,
    // Provided by VK_EXT_extended_dynamic_state
    VK_DYNAMIC_STATE_DEPTH_COMPARE_OP_EXT = 1000267008,
    // Provided by VK_EXT_extended_dynamic_state
    VK_DYNAMIC_STATE_DEPTH_BOUNDS_TEST_ENABLE_EXT = 1000267009,
    // Provided by VK_EXT_extended_dynamic_state
    VK_DYNAMIC_STATE_STENCIL_TEST_ENABLE_EXT = 1000267010,
};
```
• **VK_DYNAMIC_STATE_VIEWPORT** specifies that the `pViewports` state in
  `VkPipelineViewportStateCreateInfo` will be ignored and **must** be set dynamically with
  `vkCmdSetViewport` before any drawing commands. The number of viewports used by a pipeline
  is still specified by the `viewportCount` member of `VkPipelineViewportStateCreateInfo`.

• **VK_DYNAMIC_STATE_SCISSOR** specifies that the `pScissors` state in
  `VkPipelineViewportStateCreateInfo` will be ignored and **must** be set dynamically with
  `vkCmdSetScissor` before any drawing commands. The number of scissor rectangles used by a
  pipeline is still specified by the `scissorCount` member of `VkPipelineViewportStateCreateInfo`.

• **VK_DYNAMIC_STATE_LINE_WIDTH** specifies that the `lineWidth` state in
  `VkPipelineRasterizationStateCreateInfo` will be ignored and **must** be set dynamically with
  `vkCmdSetLineWidth` before any drawing commands that generate line primitives for the
  rasterizer.

• **VK_DYNAMIC_STATE_DEPTH_BIAS** specifies that the `depthBiasConstantFactor`, `depthBiasClamp` and
  `depthBiasSlopeFactor` states in `VkPipelineRasterizationStateCreateInfo` will be ignored and **must**
  be set dynamically with `vkCmdSetDepthBias` before any draws are performed with
  `depthBiasEnable` in `VkPipelineRasterizationStateCreateInfo` set to `VK_TRUE`.

• **VK_DYNAMIC_STATE_BLEND_CONSTANTS** specifies that the `blendConstants` state in
  `VkPipelineColorBlendStateCreateInfo` will be ignored and **must** be set dynamically with
  `vkCmdSetBlendConstants` before any draws are performed with a pipeline state with
  `VkPipelineColorBlendAttachmentState` member `blendEnable` set to `VK_TRUE` and any of the blend
  functions using a constant blend color.

• **VK_DYNAMIC_STATE_DEPTH_BOUNDS** specifies that the `minDepthBounds` and `maxDepthBounds` states of
  `VkPipelineDepthStencilStateCreateInfo` will be ignored and **must** be set dynamically with
  `vkCmdSetDepthBounds` before any draws are performed with a pipeline state with
  `VkPipelineDepthStencilStateCreateInfo` member `depthBoundsTestEnable` set to `VK_TRUE`.

• **VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK** specifies that the `compareMask` state in
  `VkPipelineDepthStencilStateCreateInfo` for both `front` and `back` will be ignored and **must** be set
dynamically with `vkCmdSetStencilCompareMask` before any draws are performed with a
pipeline state with VkPipelineDepthStencilStateCreateInfo member stencilTestEnable set to VK_TRUE

- **VK_DYNAMIC_STATE_STENCIL_WRITE_MASK** specifies that the writeMask state in VkPipelineDepthStencilStateCreateInfo for both front and back will be ignored and **must** be set dynamically with vkCmdSetStencilWriteMask before any draws are performed with a pipeline state with VkPipelineDepthStencilStateCreateInfo member stencilTestEnable set to VK_TRUE.

- **VK_DYNAMIC_STATE_STENCIL_REFERENCE** specifies that the reference state in VkPipelineDepthStencilStateCreateInfo for both front and back will be ignored and **must** be set dynamically with vkCmdSetStencilReference before any draws are performed with a pipeline state with VkPipelineDepthStencilStateCreateInfo member stencilTestEnable set to VK_TRUE.

- **VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT** specifies that the pDiscardRectangles state in VkPipelineDiscardRectangleStateCreateInfoEXT will be ignored and **must** be set dynamically with vkCmdSetDiscardRectangleEXT before any draw or clear commands. The VkDiscardRectangleModeEXT and the number of active discard rectangles is still specified by the discardRectangleMode and discardRectangleCount members of VkPipelineDiscardRectangleStateCreateInfoEXT.

- **VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT** specifies that the sampleLocationsInfo state in VkPipelineSampleLocationsStateCreateInfoEXT will be ignored and **must** be set dynamically with vkCmdSetSampleLocationsEXT before any draw or clear commands. Enabling custom sample locations is still indicated by the sampleLocationsEnable member of VkPipelineSampleLocationsStateCreateInfoEXT.

- **VK_DYNAMIC_STATE_LINE_STIPPLE_EXT** specifies that the lineStippleFactor and lineStipplePattern state in VkPipelineRasterizationLineStateCreateInfoEXT will be ignored and **must** be set dynamically with vkCmdSetLineStippleEXT before any draws are performed with a pipeline state with VkPipelineRasterizationLineStateCreateInfoEXT member stippledLineEnable set to VK_TRUE.

- **VK_DYNAMIC_STATE_CULL_MODE_EXT** specifies that the cullMode state in VkPipelineRasterizationStateCreateInfo will be ignored and **must** be set dynamically with vkCmdSetCullModeEXT before any drawing commands.

- **VK_DYNAMIC_STATE_FRONT_FACE_EXT** specifies that the frontFace state in VkPipelineRasterizationStateCreateInfo will be ignored and **must** be set dynamically with vkCmdSetFrontFaceEXT before any drawing commands.

- **VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT** specifies that the topology state in VkPipelineInputAssemblyStateCreateInfo only specifies the topology class, and the specific topology order and adjacency **must** be set dynamically with vkCmdSetPrimitiveTopologyEXT before any drawing commands.

- **VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT** specifies that the viewportCount and pViewports state in VkPipelineViewportStateCreateInfo will be ignored and **must** be set dynamically with vkCmdSetViewportWithCountEXT before any draw call.

- **VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT** specifies that the scissorCount and pScissors state in VkPipelineViewportStateCreateInfo will be ignored and **must** be set dynamically with vkCmdSetScissorWithCountEXT before any draw call.

- **VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT** specifies that the stride state in
**VkVertexInputBindingDescription** will be ignored and **must** be set dynamically with **vkCmdBindVertexBuffers2EXT** before any draw call.

- **VK_DYNAMIC_STATE_DEPTH_TEST_ENABLE_EXT** specifies that the **depthTestEnable** state in **VkPipelineDepthStencilStateCreateInfo** will be ignored and **must** be set dynamically with **vkCmdSetDepthTestEnableEXT** before any draw call.

- **VK_DYNAMIC_STATE_DEPTH_WRITE_ENABLE_EXT** specifies that the **depthWriteEnable** state in **VkPipelineDepthStencilStateCreateInfo** will be ignored and **must** be set dynamically with **vkCmdSetDepthWriteEnableEXT** before any draw call.

- **VK_DYNAMIC_STATE_DEPTHCOMPARE_OP_EXT** specifies that the **depthCompareOp** state in **VkPipelineDepthStencilStateCreateInfo** will be ignored and **must** be set dynamically with **vkCmdSetDepthCompareOpEXT** before any draw call.

- **VK_DYNAMIC_STATE_DEPTH_BOUNDS_TEST_ENABLE_EXT** specifies that the **depthBoundsTestEnable** state in **VkPipelineDepthStencilStateCreateInfo** will be ignored and **must** be set dynamically with **vkCmdSetDepthBoundsTestEnableEXT** before any draw call.

- **VK_DYNAMIC_STATE_STENCIL_TEST_ENABLE_EXT** specifies that the **stencilTestEnable** state in **VkPipelineDepthStencilStateCreateInfo** will be ignored and **must** be set dynamically with **vkCmdSetStencilTestEnableEXT** before any draw call.

- **VK_DYNAMIC_STATE_STENCIL_OP_EXT** specifies that the **failOp**, **passOp**, **depthFailOp**, and **compareOp** states in **VkPipelineDepthStencilStateCreateInfo** for both **front** and **back** will be ignored and **must** be set dynamically with **vkCmdSetStencilOpEXT** before any draws are performed with a pipeline state with **VkPipelineDepthStencilStateCreateInfo** member **stencilTestEnable** set to **VK_TRUE**.

- **VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT** specifies that the **patchControlPoints** state in **VkPipelineTessellationStateCreateInfo** will be ignored and **must** be set dynamically with **vkCmdSetPatchControlPointsEXT** before any drawing commands.

- **VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT** specifies that the **rasterizerDiscardEnable** state in **VkPipelineRasterizationStateCreateInfo** will be ignored and **must** be set dynamically with **vkCmdSetRasterizerDiscardEnableEXT** before any drawing commands.

- **VK_DYNAMIC_STATE_DEPTHBIAS_ENABLE_EXT** specifies that the **depthBiasEnable** state in **VkPipelineRasterizationStateCreateInfo** will be ignored and **must** be set dynamically with **vkCmdSetDepthBiasEnableEXT** before any drawing commands.

- **VK_DYNAMIC_STATE_LOGIC_OP_EXT** specifies that the **logicOp** state in **VkPipelineColorBlendStateCreateInfo** will be ignored and **must** be set dynamically with **vkCmdSetLogicOpEXT** before any drawing commands.

- **VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT** specifies that the **primitiveRestartEnable** state in **VkPipelineInputAssemblyStateCreateInfo** will be ignored and **must** be set dynamically with **vkCmdSetPrimitiveRestartEnableEXT** before any drawing commands.

- **VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR** specifies that state in **VkPipelineFragmentShadingRateStateCreateInfoKHR** will be ignored and **must** be set dynamically with **vkCmdSetFragmentShadingRateKHR** before any drawing commands.

- **VK_DYNAMIC_STATE_VERTEX_INPUT_EXT** specifies that the **pVertexInputState** state will be ignored and **must** be set dynamically with **vkCmdSetVertexInputEXT** before any drawing commands.
• **VK_DYNAMIC_STATE_COLOR_WRITE_ENABLE_EXT** specifies that the `pColorWriteEnables` state in `VkPipelineColorWriteCreateInfoEXT` will be ignored and must be set dynamically with `vkCmdSetColorWriteEnableEXT` before any draw call.

### 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 unmodified. This can be useful for depth-only rendering.

Presence of a shader stage in a pipeline is indicated by including a valid `VkPipelineShaderStageCreateInfo` with `module` and `pName` selecting an entry point from a shader module, where that entry point is valid for the stage specified by `stage`.

Presence of some of the fixed-function stages in the pipeline is implicitly derived from enabled shaders and provided state. For example, the fixed-function tessellator is always present when the pipeline has valid Tessellation Control and Tessellation Evaluation shaders.

*For example:*

- Depth/stencil-only rendering in a subpass with no color attachments
  - Active Pipeline Shader Stages
    - Vertex Shader
  - Required: Fixed-Function Pipeline Stages
    - `VkPipelineVertexInputStateCreateInfo`
    - `VkPipelineInputAssemblyStateCreateInfo`
    - `VkPipelineViewportStateCreateInfo`
    - `VkPipelineRasterizationStateCreateInfo`
    - `VkPipelineMultisampleStateCreateInfo`
    - `VkPipelineDepthStencilStateCreateInfo`

- Color-only rendering in a subpass with no depth/stencil attachment
  - Active Pipeline Shader Stages
    - Vertex Shader
    - Fragment Shader
  - Required: Fixed-Function Pipeline Stages
    - `VkPipelineVertexInputStateCreateInfo`
    - `VkPipelineInputAssemblyStateCreateInfo`
    - `VkPipelineViewportStateCreateInfo`
    - `VkPipelineRasterizationStateCreateInfo`
• VkPipelineMultisampleStateCreateInfo
• VkPipelineColorBlendStateCreateInfo

• Rendering pipeline with tessellation and geometry shaders
  ◦ Active Pipeline Shader Stages
    • Vertex Shader
    • Tessellation Control Shader
    • Tessellation Evaluation Shader
    • Geometry Shader
    • Fragment Shader
  ◦ Required: Fixed-Function Pipeline Stages
    • VkPipelineVertexInputStateCreateInfo
    • VkPipelineInputAssemblyStateCreateInfo
    • VkPipelineTessellationStateCreateInfo
    • VkPipelineViewportStateCreateInfo
    • VkPipelineRasterizationStateCreateInfo
    • VkPipelineMultisampleStateCreateInfo
    • VkPipelineDepthStencilStateCreateInfo
    • VkPipelineColorBlendStateCreateInfo

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.

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 object is returned.

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

- **VUID-vkCreatePipelineCache-pPipelineCache-parameter**
  - `pPipelineCache` must be a valid pointer to a `VkPipelineCache` handle

Return Codes

**Success**
- **VK_SUCCESS**

**Failure**
- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**
- **VK_ERROR_INVALID_PIPELINE_CACHE_DATA**

The `VkPipelineCacheCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineCacheCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineCacheCreateFlags flags;
    size_t initialDataSize;
    const void* pInitialData;
} VkPipelineCacheCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **flags** is a bitmask of `VkPipelineCacheCreateFlagBits` specifying the behavior of the pipeline cache.
- **initialDataSize** is the number of bytes in `pInitialData`.
- **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-05045
  The contents of this structure and data pointed to by pInitialData must be the same as specified in one of the VkDeviceObjectReservationCreateInfo::pPipelineCacheCreateInfos structures when the device was created

- 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

// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineCacheCreateFlags;

VkPipelineCacheCreateFlags is a bitmask type for setting a mask of zero or more VkPipelineCacheCreateFlagBits.

Possible values of the flags member of VkPipelineCacheCreateInfo, specifying the behavior of the pipeline cache, are:

// Provided by VKSC_VERSION_1_0
typedef enum VkPipelineCacheCreateFlagBits {
  // Provided by VKSC_VERSION_1_0
  VK_PIPELINE_CACHE_CREATE_USE_APPLICATION_STORAGE_BIT = 0x00000004,
  VK_PIPELINE_CACHE_CREATE_RESERVED_1_BIT_EXT = 0x00000002,
  // Provided by VKSC_VERSION_1_0
}

463
• `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.

Version one of the pipeline cache header is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineCacheHeaderVersionOne {
    uint32_t headerSize;
    VkPipelineCacheHeaderVersion headerVersion;
    uint32_t vendorID;
    uint32_t deviceID;
    uint8_t pipelineCacheUUID[VK_UUID_SIZE];
} VkPipelineCacheHeaderVersionOne;
```

• `headerSize` is the length in bytes of the pipeline cache header.

• `headerVersion` is a `VkPipelineCacheHeaderVersion` enum value specifying the version of the header. A consumer of the pipeline cache should use the cache version to interpret the remainder of the cache header.

• `vendorID` is the `VkPhysicalDeviceProperties::vendorID` of the implementation.

• `deviceID` is the `VkPhysicalDeviceProperties::deviceID` of the implementation.

• `pipelineCacheUUID` is the `VkPhysicalDeviceProperties::pipelineCacheUUID` of the implementation.

Unlike most structures declared by the Vulkan API, all fields of this structure are written with the least significant byte first, regardless of host byte-order.

The C language specification does not define the packing of structure members. This layout assumes tight structure member packing, with members laid out in the order listed in the structure, and the intended size of the structure is 32 bytes. If a compiler produces code that diverges from that pattern, applications must employ another method to set values at the correct offsets.
Valid Usage

- VUID-VkPipelineCacheHeaderVersionOne-headerSize-05075
  headerSize must be 56

- VUID-VkPipelineCacheHeaderVersionOne-headerVersion-05076
  headerVersion must be VK_PIPELINE_CACHE_HEADER_VERSION_SAFETY_CRITICAL_ONE

Valid Usage (Implicit)

- VUID-VkPipelineCacheHeaderVersionOne-headerVersion-parameter
  headerVersion must be a valid VkPipelineCacheHeaderVersion value

Possible values of the headerVersion value of the pipeline cache header are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkPipelineCacheHeaderVersion {
    VK_PIPELINE_CACHE_HEADER_VERSION_ONE = 1,
    // 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
• `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.

Note
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-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;
};
```
• 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-pipelineCache-parent
  If *pipelineCache* is a valid handle, it must have been created, allocated, or retrieved from *device*

---

**Host Synchronization**

- Host access to *pipelineCache* must be externally synchronized

---

### 10.7. 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](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_NO_PIPELINE_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 the type of this structure.
- **pNext** is NULL or a pointer to an extension-specific structure.
- **pipelineIdentifier** is an array of VK_UUID_SIZE uint8_t values representing an identifier for the pipeline.
• **matchControl** is an enum of type `VkPipelineMatchControl` that describes 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_NO_PIPELINE_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_OUT_OF_POOL_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` are:

```c
// Provided by VKSC_VERSION_1_0
typedef enum VkPipelineMatchControl {
    VK_PIPELINE_MATCH_CONTROL_APPLICATION_UUID_EXACT_MATCH = 0,
} VkPipelineMatchControl;
```
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:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSpecializationMapEntry {
    uint32_t constantID;
    uint32_t offset;
    size_t size;
} VkSpecializationMapEntry;
```

- `constantID` is the ID of the specialization constant in SPIR-V.
- `offset` is the byte offset of the specialization constant value within the supplied data buffer.
- `size` is the byte size of the specialization constant value within the supplied data buffer.

If a `constantID` value is not a specialization constant ID used in the shader, that map entry does not affect the behavior of the pipeline.

Valid Usage

- VUID-VkSpecializationMapEntry-constantID-00776
  For a `constantID` specialization constant declared in a shader, `size` must match the byte size of the `constantID`. If the specialization constant is of type `boolean`, `size` must be the byte size of `VkBool32`.

In human readable SPIR-V:

```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 %wgs.size 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
%wgs.size = 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[] = {
    { 13, // constantID
        0 * sizeof(uint32_t), // offset
        sizeof(uint32_t) // size
    },
    { 42, // constantID
        1 * sizeof(uint32_t), // offset
        sizeof(uint32_t) // size
    },
    { 3, // constantID
        2 * sizeof(uint32_t), // offset
        sizeof(uint32_t) // size
    }
};

const uint32_t data[] = { 16, 8, 4 }; // our workgroup size is 16x8x4

const VkSpecializationInfo info = {
    3, // mapEntryCount
    entries, // pMapEntries
    3 * sizeof(uint32_t), // dataSize
    data, // pData
};
```

Then when calling `vkCreateComputePipelines`, and passing the `VkSpecializationInfo` we defined as the `pSpecializationInfo` parameter of `VkPipelineShaderStageCreateInfo`, we will create a compute pipeline with the runtime specified local workgroup size.

Another example would be that an application has a SPIR-V module that has some platform-dependent constants they wish to use.

In human readable SPIR-V:

```
OpDecorate %1 SpecId 0 ; decorate our signed 32-bit integer constant
OpDecorate %2 SpecId 12 ; decorate our 32-bit floating-point constant
%32 = OpTypeInt 32 1 ; declare a signed 32-bit type
%float = OpTypeFloat 32 ; declare a 32-bit floating-point type
%1 = OpSpecConstant %32 -1 ; some signed 32-bit integer 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:

```c
struct SpecializationData {
    int32_t data0;
    float data1;
};

const VkSpecializationMapEntry entries[] = {
    { 0, // constantID
       offsetof(SpecializationData, data0), // offset
       sizeof(SpecializationData::data0) // size
    },
    { 12, // constantID
       offsetof(SpecializationData, data1), // offset
       sizeof(SpecializationData::data1) // size
    }
};

SpecializationData data;
data.data0 = -42; // set the data for the 32-bit integer
data.data1 = 42.0f; // set the data for the 32-bit floating-point

const VkSpecializationInfo info = {
    2, // mapEntryCount
    entries, // pMapEntries
    sizeof(data), // dataSize
    &data, // pData
};
```

It is legal for a SPIR-V module with specializations to be compiled into a pipeline where no specialization 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
void vkCmdBindPipeline(
    VkCommandBuffer commandBuffer,
    VkPipelineBindPoint pipelineBindPoint,
    VkPipeline pipeline);
```

- `commandBuffer` is the command buffer that the pipeline will be bound to.
- `pipelineBindPoint` is a `VkPipelineBindPoint` value specifying to which bind point the pipeline is bound. Binding one does not disturb the others.
- `pipeline` is the pipeline to be bound.

Once bound, a pipeline binding affects subsequent commands that interact with the given pipeline type in the command buffer until a different pipeline of the same type is bound to the bind point. Commands that do not interact with the given pipeline type must not be affected by the pipeline state.

- The pipeline bound to `VK_PIPELINE_BIND_POINT_COMPUTE` controls the behavior of all dispatching commands.
- The pipeline bound to `VK_PIPELINE_BIND_POINT_GRAPHICS` controls the behavior of all drawing commands.

## 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-variableSampleLocations-01525**
  If `VkPhysicalDeviceSampleLocationsPropertiesEXT::variableSampleLocations` is `VK_FALSE`, and pipeline is 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

- **VUID-vkCmdBindPipeline-variableSampleLocations-01525**
  If `VkPhysicalDeviceSampleLocationsPropertiesEXT::variableSampleLocations` is `VK_FALSE`, and pipeline is a graphics pipeline created with a `VkPipelineSampleLocationsStateCreateInfoEXT` structure having its `sampleLocationsEnable`
member set to `VK_TRUE` but without `VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT` enabled then the current render pass instance must have been begun by specifying a `VkRenderPassSampleLocationsBeginInfoEXT` structure whose `pPostSubpassSampleLocations` member contains an element with a `subpassIndex` matching the current subpass index and the `sampleLocationsInfo` member of that element must match the `sampleLocationsInfo` specified in `VkPipelineSampleLocationsStateCreateInfoEXT` when the pipeline was created.

- **VUID-vkCmdBindPipeline-commandBuffer-04809**
  If `commandBuffer` is a secondary command buffer with `VkCommandBufferInheritanceViewportScissorInfoNV::viewportScissor2D` enabled and `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_GRAPHICS` and `pipeline` was created with `VkPipelineDiscardRectangleStateCreateInfoEXT` structure and its `discardRectangleCount` member is not 0, then the pipeline must have been created with `VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT` enabled.

### 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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</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 another pipeline bind.

When a pipeline object is bound, the following applies to each state parameter:

- If the state is not specified as dynamic in the new pipeline object, then that command buffer state is overwritten by the state in the new pipeline object. 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 since the command buffer recording was begun, or the last bound pipeline object with that state specified as static, whichever was the latter.

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_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` | `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` | `VK_MEMORY_PROPERTY_HOST_CACHED_BIT`
- `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` | `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` | `VK_MEMORY_PROPERTY_HOST_CACHED_BIT` | `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`
- `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` | `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` | `VK_MEMORY_PROPERTY_HOST_COHERENT_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)

---

*i Note*
There is no ordering requirement between X and Y elements for the case their propertyFlags members are not in a subset relation. That potentially allows more than one possible way to order the same set of memory types. Notice that the list of all allowed memory property flag combinations is written in a valid order. But if instead VK_MEMORY_PROPERTYDEVICE_LOCAL_BITHAS_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTYHOST_COHERENT_BIT, the list would still be in a valid order.

This ordering requirement enables applications to use a simple search loop to select the desired memory type along the lines of:

```c
// Find a memory in `memoryTypeBitsRequirement` that includes all of
// `requiredProperties`
int32_t findProperties(const VkPhysicalDeviceMemoryProperties* pMemoryProperties,
    uint32_t memoryTypeBitsRequirement,
    VkMemoryPropertyFlags requiredProperties) {
    const uint32_t memoryCount = pMemoryProperties->memoryTypeCount;
    for (uint32_t memoryIndex = 0; memoryIndex < memoryCount; ++memoryIndex) {
        const uint32_t memoryTypeBits = (1 << memoryIndex);
        const bool isRequiredMemoryType = memoryTypeBitsRequirement & memoryTypeBits;
        const VkMemoryPropertyFlags properties = pMemoryProperties->memoryTypes[memoryIndex].propertyFlags;
        const bool hasRequiredProperties = (properties & requiredProperties) == requiredProperties;
        if (isRequiredMemoryType && hasRequiredProperties) {
            return static_cast<int32_t>(memoryIndex);
        }
    }
    // failed to find memory type
    return -1;
}
```

// Try to find an optimal memory type, or if it does not exist try fallback memory type
// `device` is the VkDevice
// `image` is the VkImage that requires memory to be bound
// `memoryProperties` properties as returned by vkGetPhysicalDeviceMemoryProperties
// `requiredProperties` are the property flags that must be present
// `optimalProperties` are the property flags that are preferred by the application
VkMemoryRequirements memoryRequirements;
vkGetImageMemoryRequirements(device, image, &memoryRequirements);
int32_t memoryType = findProperties(&memoryProperties, memoryRequirements.memoryTypeBits,
    optimalProperties);
if (memoryType == -1) // not found; try fallback properties
    memoryType = findProperties(&memoryProperties, memoryRequirements.memoryTypeBits,
```
**VK_MAX_MEMORY_TYPES** is the length of an array of *VkMemoryType* structures describing memory types, as returned in *VkPhysicalDeviceMemoryProperties::*memoryTypes.

```
#define VK_MAX_MEMORY_TYPES 32U
```

**VK_MAX_MEMORY_HEAPS** is the length of an array of *VkMemoryHeap* structures describing memory heaps, as returned in *VkPhysicalDeviceMemoryProperties::*memoryHeaps.

```
#define VK_MAX_MEMORY_HEAPS 16U
```

To query memory properties, call:

```
// 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:

```
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceMemoryProperties2 {
  VkStructureType sType;
  void* pNext;
  VkPhysicalDeviceMemoryProperties memoryProperties;
} VkPhysicalDeviceMemoryProperties2;
```
• \texttt{sType} is the type of this structure.
• \texttt{pNext} is \texttt{NULL} or a pointer to a structure extending this structure.
• \texttt{memoryProperties} is a \texttt{VkPhysicalDeviceMemoryProperties} structure which is populated with the same values as in \texttt{vkGetPhysicalDeviceMemoryProperties}.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceMemoryProperties2-sType-sType
  \texttt{sType} must be \texttt{VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_MEMORY\_PROPERTIES\_2}

- VUID-VkPhysicalDeviceMemoryProperties2-pNext-pNext
  \texttt{pNext} must be \texttt{NULL} or a pointer to a valid instance of \texttt{VkPhysicalDeviceMemoryBudgetPropertiesEXT}

- VUID-VkPhysicalDeviceMemoryProperties2-sType-unique
  The \texttt{sType} value of each struct in the \texttt{pNext} chain must be unique

The \texttt{VkMemoryHeap} structure is defined as:

```c
// Provided by VK\_VERSION\_1\_0
typedef struct VkMemoryHeap {
    VkDeviceSize size;
    VkMemoryHeapFlags flags;
} VkMemoryHeap;
```

• \texttt{size} is the total memory size in bytes in the heap.
• \texttt{flags} is a bitmask of \texttt{VkMemoryHeapFlagBits} specifying attribute flags for the heap.

Bits which may be set in \texttt{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;
```

- \texttt{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.

- \texttt{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 heap, 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`.

- **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`.

If the `VkPhysicalDeviceMemoryBudgetPropertiesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceMemoryProperties2`, it is filled with the current memory budgets and usages.

The `VkPhysicalDeviceMemoryBudgetPropertiesEXT` structure is defined as:

```c
// Provided by VK_EXT_memory_budget
typedef struct VkPhysicalDeviceMemoryBudgetPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    VkDeviceSize heapBudget[VK_MAX_MEMORY_HEAPS];
    VkDeviceSize heapUsage[VK_MAX_MEMORY_HEAPS];
} VkPhysicalDeviceMemoryBudgetPropertiesEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `heapBudget` is an array of `VK_MAX_MEMORY_HEAPS` `VkDeviceSize` values in which memory budgets are returned, with one element for each memory heap. A heap's budget is a rough estimate of how much memory the process can allocate from that heap before allocations may fail or cause performance degradation. The budget includes any currently allocated device memory.
• **heapUsage** is an array of VK_MAX_MEMORY_HEAPS VkDeviceSize values in which memory usages are returned, with one element for each memory heap. A heap’s usage is an estimate of how much memory the process is currently using in that heap.

The values returned in this structure are not invariant. The heapBudget and heapUsage values must be zero for array elements greater than or equal to VkPhysicalDeviceMemoryProperties::memoryHeapCount. The heapBudget value must be non-zero for array elements less than VkPhysicalDeviceMemoryProperties::memoryHeapCount. The heapBudget value must be less than or equal to VkMemoryHeap::size for each heap.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceMemoryBudgetPropertiesEXT-sType-sType
  - sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_BUDGET_PROPERTIES_EXT

### 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,
    const VkMemoryAllocateInfo* pAllocateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkDeviceMemory* pMemory);
```

- **device** is the logical device that owns the memory.
- **pAllocateInfo** is a pointer to a VkMemoryAllocateInfo structure describing parameters of the allocation. A successfully returned allocation must use the requested parameters—no substitution is permitted by the implementation.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
- **pMemory** is a pointer to a VkDeviceMemory handle in which information about the allocated memory is returned.

Allocations returned by vkAllocateMemory are guaranteed to meet any alignment requirement of the implementation. For example, if an implementation requires 128 byte alignment for images and 64 byte alignment for buffers, the device memory returned through this mechanism would be 128-
byte aligned. This ensures that applications can correctly suballocate objects of different types (with potentially different alignment requirements) in the same memory object.

When memory is allocated, its contents are undefined with the following constraint:

- The contents of unprotected memory must not be a function of the contents of data protected memory objects, even if those memory objects were previously freed.

Note
The contents of memory allocated by one application should not be a function of data from protected memory objects of another application, even if those memory objects were previously freed.

The maximum number of valid memory allocations that can exist simultaneously within a VkDevice may be restricted by implementation- or platform-dependent limits. The maxMemoryAllocationCount feature describes the number of allocations that can exist simultaneously before encountering these internal limits.

Note
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 **vkGetPhysicalDeviceMemoryProperties** for the **VkPhysicalDevice** that `device` was created from

- **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**
- **VK_ERROR_INVALID_EXTERNAL_HANDLE**

The **VkMemoryAllocateInfo** structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkMemoryAllocateInfo {
    VkStructureType sType;
    const void* pNext;
    VkDeviceSize allocationSize;
} VkMemoryAllocateInfo;
```
uint32_t memoryTypeIndex;
} VkMemoryAllocateInfo;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **allocationSize** is the size of the allocation in bytes.
- **memoryTypeIndex** is an index identifying a memory type from the memoryTypes array of the VkPhysicalDeviceMemoryProperties structure.

The internal data of an allocated device memory object **must** include a reference to implementation-specific resources, referred to as the memory object's **payload**. Applications **can** also import and export that internal data to and from device memory objects to share data between Vulkan instances and other compatible APIs. A VkMemoryAllocateInfo structure defines a memory import operation if its **pNext** chain includes one of the following structures:

- VkImportMemoryFdInfoKHR with a non-zero handleType value
- VkImportMemoryHostPointerInfoEXT with a non-zero handleType value
- VkImportMemorySciBufInfoNV with a non-zero handleType value

If the parameters define an import operation and the external handle type is VK_EXTERNAL_MEMORY_HANDLE_TYPE_SCI_BUF_BIT_NV, allocationSize is ignored. The implementation must query the size of this allocation from the NvSciBufAttrList associated with the external NvSciBufObj.

Whether device memory objects constructed via a memory import operation hold a reference to their payload depends on the properties of the handle type used to perform the import, as defined below for each valid handle type. Importing memory must not modify the content of the memory. Implementations must ensure that importing memory does not enable the importing Vulkan instance to access any memory or resources in other Vulkan instances other than that corresponding to the memory object imported. Implementations must also ensure accessing imported memory which has not been initialized does not allow the importing Vulkan instance to obtain data from the exporting Vulkan instance or vice-versa.

**Note**

How exported and imported memory is isolated is left to the implementation, but applications should be aware that such isolation may prevent implementations from placing multiple exportable memory objects in the same physical or virtual page. Hence, applications should avoid creating many small external memory objects whenever possible.

Importing memory must not increase overall heap usage within a system. However, it must affect the following per-process values:

- VkPhysicalDeviceMaintenance3Properties::maxMemoryAllocationCount
- VkPhysicalDeviceMemoryBudgetPropertiesEXT::heapUsage
When performing a memory import operation, it is the responsibility of the application to ensure the external handles and their associated payloads meet all valid usage requirements. However, implementations must perform sufficient validation of external handles and payloads to ensure that the operation results in a valid memory object which will not cause program termination, device loss, queue stalls, or corruption of other resources when used as allowed according to its allocation parameters. If the external handle provided does not meet these requirements, the implementation must fail the memory import operation with the error code VK_ERROR_INVALID_EXTERNAL_HANDLE.

Valid Usage

- VUID-VkMemoryAllocateInfo-allocationSize-00638
  allocationSize must be greater than 0

- VUID-VkMemoryAllocateInfo-allocationSize-01742
  If the parameters define an import operation, the external handle specified was created by the Vulkan API, and the external handle type is VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT, then the values of allocationSize and memoryTypeIndex must match those specified when the payload being imported was created

- VUID-VkMemoryAllocateInfo-memoryTypeIndex-00648
  If the parameters define an import operation and the external handle is a POSIX file descriptor created outside of the Vulkan API, the value of memoryTypeIndex must be one of those returned by vkGetMemoryFdPropertiesKHR

- VUID-VkMemoryAllocateInfo-memoryTypeIndex-01872
  If the protected memory feature is not enabled, the VkMemoryAllocateInfo::memoryTypeIndex must not indicate a memory type that reports VK_MEMORY_PROPERTY_PROTECTED_BIT

- VUID-VkMemoryAllocateInfo-memoryTypeIndex-01744
  If the parameters define an import operation and the external handle is a host pointer, the value of memoryTypeIndex must be one of those returned by vkGetMemoryHostPointerPropertiesEXT

- VUID-VkMemoryAllocateInfo-allocationSize-01745
  If the parameters define an import operation and the external handle is a host pointer, allocationSize must be an integer multiple of VkPhysicalDeviceExternalMemoryHostPropertiesEXT::minImportedHostPointerAlignment

- 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-pNext-03332
  If the pNext chain includes a VkImportMemoryHostPointerInfoEXT structure, VkMemoryOpaqueCaptureAddressAllocateInfo::opaqueCaptureAddress must be zero

- VUID-VkMemoryAllocateInfo-opaqueCaptureAddress-03333
  If the parameters define an import operation, VkMemoryOpaqueCaptureAddressAllocateInfo::opaqueCaptureAddress must be zero

- VUID-VkMemoryAllocateInfo-pNext-05097
  If the pNext chain includes a VkExportMemorySciBufInfoNV structure, VkPhysicalDeviceExternalMemorySciBufFeaturesNV::sciBufExport must be enabled

- VUID-VkMemoryAllocateInfo-pNext-05098
  If the pNext chain includes a VkImportMemorySciBufInfoNV structure, VkPhysicalDeviceExternalMemorySciBufFeaturesNV::sciBufImport must be enabled

- VUID-VkMemoryAllocateInfo-memoryTypeIndex-05099
  If the parameters define an import operation and the external handle is a NvSciBufObj, the value of memoryTypeIndex must be one of those returned by vkGetPhysicalDeviceExternalMemorySciBufPropertiesNV

### 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, VkExportMemorySciBufInfoNV, VkImportMemoryFdInfoKHR, VkImportMemoryHostPointerInfoEXT, VkImportMemorySciBufInfoNV, 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;
```
• sType is the type of 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-01433
  If image is not VK_NULL_HANDLE, VkMemoryAllocateInfo::allocationSize must equal the VkMemoryRequirements::size of the image

• VUID-VkMemoryDedicatedAllocateInfo-image-01434
  If image is not VK_NULL_HANDLE, image must have been created without VK_IMAGE_CREATE_SPARSE_BINDING_BIT set in VkImageCreateInfo::flags

• VUID-VkMemoryDedicatedAllocateInfo-buffer-01435
  If buffer is not VK_NULL_HANDLE, VkMemoryAllocateInfo::allocationSize must equal the VkMemoryRequirements::size of the buffer

• VUID-VkMemoryDedicatedAllocateInfo-buffer-01436
  If buffer is not VK_NULL_HANDLE, buffer must have been created without VK_BUFFER_CREATE_SPARSE_BINDING_BIT set in VkBufferCreateInfo::flags

• VUID-VkMemoryDedicatedAllocateInfo-image-01878
  If image is not VK_NULL_HANDLE and VkMemoryAllocateInfo defines a memory import operation with handle type VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT, the memory being imported must also be a dedicated image allocation and image must be identical to the image associated with the imported memory

• VUID-VkMemoryDedicatedAllocateInfo-buffer-01879
  If buffer is not VK_NULL_HANDLE and VkMemoryAllocateInfo defines a memory import operation with handle type VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT, the memory being imported must also be a dedicated buffer allocation and buffer must be identical to the buffer associated with the imported memory

**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
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 the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- handleTypes is a bitmask of VkExternalMemoryHandleTypeFlagBits specifying one or more memory handle types the application can export from the resulting allocation. The application can request multiple handle types for the same allocation.

**Valid Usage**

- VUID-VkExportMemoryAllocateInfo-handleTypes-00656
  The bits in handleTypes must be supported and compatible, as reported by VkExternalImageFormatProperties or VkExternalBufferProperties

**Valid Usage (Implicit)**

- VUID-VkExportMemoryAllocateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO

- VUID-VkExportMemoryAllocateInfo-handleTypes-parameter
  handleTypes must be a valid combination of VkExternalMemoryHandleTypeFlagBits values

### 11.2.4. File Descriptor External Memory

To import memory from a POSIX file descriptor handle, add a VkImportMemoryFdInfoKHR structure to the pNext chain of the VkMemoryAllocateInfo structure. The VkImportMemoryFdInfoKHR structure is defined as:
typedef struct VkImportMemoryFdInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkExternalMemoryHandleTypeFlagBits handleType;
    int fd;
} VkImportMemoryFdInfoKHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **handleType** is a VkExternalMemoryHandleTypeFlagBits value specifying the handle type of **fd**.
- **fd** is the external handle to import.

Importing memory from a file descriptor transfers ownership of the file descriptor from the application to the Vulkan implementation. The application **must** not perform any operations on the file descriptor after a successful import. The imported memory object holds a reference to its payload.

Applications **can** import the same payload into multiple instances of Vulkan, into the same instance from which it was exported, and multiple times into a given Vulkan instance. In all cases, each import operation **must** create a distinct VkDeviceMemory object.

### Valid Usage

- **VUID-VkImportMemoryFdInfoKHR-handleType-00667**
  If **handleType** is not 0, it **must** be supported for import, as reported by VkExternalImageFormatProperties or VkExternalBufferProperties

- **VUID-VkImportMemoryFdInfoKHR-fd-00668**
  The memory from which **fd** was exported **must** have been created on the same underlying physical device as **device**

- **VUID-VkImportMemoryFdInfoKHR-handleType-00669**
  If **handleType** is not 0, it **must** be VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT or VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT

- **VUID-VkImportMemoryFdInfoKHR-handleType-00670**
  If **handleType** is not 0, **fd** **must** be a valid handle of the type specified by **handleType**

- **VUID-VkImportMemoryFdInfoKHR-fd-01746**
  The memory represented by **fd** **must** have been created from a physical device and driver that is compatible with **device** and **handleType**, as described in External memory handle types compatibility

- **VUID-VkImportMemoryFdInfoKHR-fd-01520**
  **fd** **must** obey any requirements listed for **handleType** in external memory handle types compatibility
Valid Usage (Implicit)

- **VUID-VkImportMemoryFdInfoKHR-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_IMPORT_MEMORY_FD_INFO_KHR`

- **VUID-VkImportMemoryFdInfoKHR-handleType-parameter**
  
  If `handleType` is not 0, `handleType` **must** be a valid `VkExternalMemoryHandleTypeFlagBits` value

To export a POSIX file descriptor referencing the payload of a Vulkan device memory object, call:

```c
// Provided by VK_KHR_external_memory_fd
VkResult vkGetMemoryFdKHR(VkDevice device, const VkMemoryGetFdInfoKHR* pGetFdInfo, int* pFd);
```

- **device** is the logical device that created the device memory being exported.

- **pGetFdInfo** is a pointer to a `VkMemoryGetFdInfoKHR` structure containing parameters of the export operation.

- **pFd** will return a file descriptor referencing the payload of the device memory object.

Each call to `vkGetMemoryFdKHR` **must** create a new file descriptor holding a reference to the memory object’s payload and transfer ownership of the file descriptor to the application. To avoid leaking resources, the application **must** release ownership of the file descriptor using the `close` system call when it is no longer needed, or by importing a Vulkan memory object from it. Where supported by the operating system, the implementation **must** set the file descriptor to be closed automatically when an `execve` system call is made.

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

Valid Usage (Implicit)

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

- **VUID-vkGetMemoryFdKHR-pGetFdInfo-parameter**
  
  `pGetFdInfo` **must** be a valid pointer to a valid `VkMemoryGetFdInfoKHR` structure

- **VUID-vkGetMemoryFdKHR-pFd-parameter**
  
  `pFd` **must** be a valid pointer to an `int` value
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_TOO_MANY_OBJECTS
• VK_ERROR_OUT_OF_HOST_MEMORY

The `VkMemoryGetFdInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_external_memory_fd
typedef struct VkMemoryGetFdInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkDeviceMemory memory;
    VkExternalMemoryHandleTypeFlagBits handleType;
} VkMemoryGetFdInfoKHR;
```

• `sType` is the type of this structure.
• `pNext` is NULL or a pointer to a structure extending this structure.
• `memory` is the memory object from which the handle will be exported.
• `handleType` is a `VkExternalMemoryHandleTypeFlagBits` value specifying the type of handle requested.

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

>Note
The size of the exported file may be larger than the size requested by `VkMemoryAllocateInfo::allocationSize`. If `handleType` is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT`, then the application can query the file's actual size with `lseek`.

Valid Usage

• VUID-VkMemoryGetFdInfoKHR-handleType-00671
`handleType` must have been included in `VkExportMemoryAllocateInfo::handleTypes` when `memory` was created

• VUID-VkMemoryGetFdInfoKHR-handleType-00672
`handleType` must be `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT` or `VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT`
POSIX file descriptor memory handles compatible with Vulkan may also be created by non-Vulkan APIs using methods beyond the scope of this specification. To determine the correct parameters to use when importing such handles, call:

```c
// Provided by VK_KHR_external_memory_fd
VkResult vkGetMemoryFdPropertiesKHR(
    VkDevice device,
    VkExternalMemoryHandleTypeFlagBits handleType,
    int fd,
    VkMemoryFdPropertiesKHR* pMemoryFdProperties);
```

- `device` is the logical device that will be importing `fd`.
- `handleType` is a `VkExternalMemoryHandleTypeFlagBits` value specifying the type of the handle `fd`.
- `fd` is the handle which will be imported.
- `pMemoryFdProperties` is a pointer to a `VkMemoryFdPropertiesKHR` structure in which the properties of the handle `fd` are returned.

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

### Valid Usage

- VUID-vkGetMemoryFdPropertiesKHR-fd-00673
  - `fd` must be an external memory handle created outside of the Vulkan API
- VUID-vkGetMemoryFdPropertiesKHR-handleType-00674
  - `handleType` must not be `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT`
device must be a valid VkDevice handle

- VUID-vkGetMemoryFdPropertiesKHR-handleType-parameter
  handleType must be a valid VkExternalMemoryHandleTypeFlagBits value

- VUID-vkGetMemoryFdPropertiesKHR-pMemoryFdProperties-parameter
  pMemoryFdProperties must be a valid pointer to a VkMemoryFdPropertiesKHR structure

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_INVALID_EXTERNAL_HANDLE

The VkMemoryFdPropertiesKHR structure returned is defined as:

```c
// Provided by VK_KHR_external_memory_fd
typedef struct VkMemoryFdPropertiesKHR {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           memoryTypeBits;
} VkMemoryFdPropertiesKHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- memoryTypeBits is a bitmask containing one bit set for every memory type which the specified file descriptor can be imported as.

Valid Usage (Implicit)

- VUID-VkMemoryFdPropertiesKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_MEMORY_FD_PROPERTIES_KHR

- VUID-VkMemoryFdPropertiesKHR-pNext-pNext
  pNext must be NULL

11.2.5. Host External Memory

To import memory from a host pointer, add a VkImportMemoryHostPointerInfoEXT structure to the pNext chain of the VkMemoryAllocateInfo structure. The VkImportMemoryHostPointerInfoEXT structure is defined as:
typedef struct VkImportMemoryHostPointerInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkExternalMemoryHandleTypeFlagBits handleType;
    void* pHostPointer;
} VkImportMemoryHostPointerInfoEXT;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **handleType** is a VkExternalMemoryHandleTypeFlagBits value specifying the handle type.
- **pHostPointer** is the host pointer to import from.

Importing memory from a host pointer shares ownership of the memory between the host and the Vulkan implementation. The application can continue to access the memory through the host pointer but it is the application’s responsibility to synchronize device and non-device access to the payload as defined in Host Access to Device Memory Objects.

Applications can import the same payload into multiple instances of Vulkan and multiple times into a given Vulkan instance. However, implementations may fail to import the same payload multiple times into a given physical device due to platform constraints.

Importing memory from a particular host pointer may not be possible due to additional platform-specific restrictions beyond the scope of this specification in which case the implementation must fail the memory import operation with the error code VK_ERROR_INVALID_EXTERNAL_HANDLE_KHR.

Whether device memory objects imported from a host pointer hold a reference to their payload is undefined. As such, the application must ensure that the imported memory range remains valid and accessible for the lifetime of the imported memory object.

### Valid Usage

- **VUID-VkImportMemoryHostPointerInfoEXT-handleType-01747**
  If handleType is not 0, it must be supported for import, as reported in VkExternalMemoryProperties

- **VUID-VkImportMemoryHostPointerInfoEXT-handleType-01748**
  If handleType is not 0, it must be VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT or VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT

- **VUID-VkImportMemoryHostPointerInfoEXT-pHostPointer-01749**
  pHostPointer must be a pointer aligned to an integer multiple of VkPhysicalDeviceExternalMemoryHostPropertiesEXT::minImportedHostPointerAlignment

- **VUID-VkImportMemoryHostPointerInfoEXT-handleType-01750**
  If handleType is VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT, pHostPointer must be a pointer to allocationSize number of bytes of host memory, where allocationSize is the member of the VkMemoryAllocateInfo structure this structure is
chained to

• VUID-VkImportMemoryHostPointerInfoEXT-handleType-01751
  If `handleType` is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT`, `pHostPointer` must be a pointer to `allocationSize` number of bytes of host mapped foreign memory, where `allocationSize` is the member of the `VkMemoryAllocateInfo` structure this structure is chained to

---

**Valid Usage (Implicit)**

- VUID-VkImportMemoryHostPointerInfoEXT-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_IMPORT_MEMORY_HOST_POINTER_INFO_EXT`
- VUID-VkImportMemoryHostPointerInfoEXT-handleType-parameter
  `handleType` must be a valid `VkExternalMemoryHandleTypeFlagBits` value

To determine the correct parameters to use when importing host pointers, call:

```c
// Provided by VK_EXT_external_memory_host
VkResult vkGetMemoryHostPointerPropertiesEXT(
  VkDevice device,
  VkExternalMemoryHandleTypeFlagBits handleType,
  const void* pHostPointer,
  VkMemoryHostPointerPropertiesEXT* pMemoryHostPointerProperties);
```

- `device` is the logical device that will be importing `pHostPointer`.
- `handleType` is a `VkExternalMemoryHandleTypeFlagBits` value specifying the type of the handle `pHostPointer`.
- `pHostPointer` is the host pointer to import from.
- `pMemoryHostPointerProperties` is a pointer to a `VkMemoryHostPointerPropertiesEXT` structure in which the host pointer properties are returned.

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

---

**Valid Usage**

- VUID-vkGetMemoryHostPointerPropertiesEXT-handleType-01752
  `handleType` must be `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT` or `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT`
- VUID-vkGetMemoryHostPointerPropertiesEXT-pHostPointer-01753
  `pHostPointer` must be a pointer aligned to an integer multiple of `VkPhysicalDeviceExternalMemoryHostPropertiesEXT::minImportedHostPointerAlignment`
- VUID-vkGetMemoryHostPointerPropertiesEXT-handleType-01754
  If `handleType` is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT`, `pHostPointer`
must be a pointer to host memory

- VUID-vkGetMemoryHostPointerPropertiesEXT-handleType-01755
  If handleType is VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_mapped_FOREIGN_MEMORY_BIT_EXT, pHostPointer must be a pointer to host mapped foreign memory

Valid Usage (Implicit)

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

- VUID-vkGetMemoryHostPointerPropertiesEXT-handleType-parameter
dhandleType must be a valid VkExternalMemoryHandleTypeFlagBits value

- VUID-vkGetMemoryHostPointerPropertiesEXT-pMemoryHostPointerProperties-parameter
  pMemoryHostPointerProperties must be a valid pointer to a VkMemoryHostPointerPropertiesEXT structure

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_INVALID_EXTERNAL_HANDLE

The VkMemoryHostPointerPropertiesEXT structure is defined as:

```c
// Provided by VK_EXT_external_memory_host
typedef struct VkMemoryHostPointerPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    uint32_t memoryTypeBits;
} VkMemoryHostPointerPropertiesEXT;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `memoryTypeBits` is a bitmask containing one bit set for every memory type which the specified host pointer can be imported as.

The value returned by `memoryTypeBits` must only include bits that identify memory types which are host visible.
11.2.6. NvSciBuf External Memory

To export a NvSciBufObj from memory, add a VkExportMemorySciBufInfoNV structure to the pNext chain of the VkMemoryAllocateInfo structure. The VkExportMemorySciBufInfoNV structure is defined as:

```c
// Provided by VK_NV_external_memory_sci_buf
typedef struct VkExportMemorySciBufInfoNV {
    VkStructureType sType;
    const void* pNext;
    NvSciBufAttrList pAttributes;
} VkExportMemorySciBufInfoNV;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **pAttributes** is an opaque NvSciBufAttrList describing the attributes of the NvSciBuf object that will be exported.

If VkExportMemoryAllocateInfo is not present in the same pNext chain, this structure is ignored.

If VkExportMemoryAllocateInfo is present in the pNext chain of VkMemoryAllocateInfo with the handleType mask containing the VK_EXTERNAL_MEMORY_HANDLE_TYPE_SCI_BUF_BIT_NV bit, but either VkExportMemorySciBufInfoNV is not present in the pNext chain, or if it is present but pAttributes is set to NULL, vkAllocateMemory will return VK_ERROR_INITIALIZATION_FAILED.

The pAttributes parameter must be a reconciled NvSciBufAttrList. NvSciBufAttrList consists of both public and private attributes. It is the application's responsibility to set the public attributes. To set the private attributes, the application must use the vkGetPhysicalDeviceSciBufAttributesNV command. The NvSciBufAttrList is then reconciled using the NvSciBuf APIs.
Valid Usage (Implicit)

- VUID-VkExportMemorySciBufInfoNV-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_EXPORT_MEMORY_SCI_BUF_INFO_NV

To fill the private attributes of an unreconciled NvSciBufAttrList, call:

```cpp
// Provided by VK_NV_external_memory_sci_buf
VkResult vkGetPhysicalDeviceSciBufAttributesNV(
    VkPhysicalDevice physicalDevice,
    NvSciBufAttrList pAttributes);
```

- `physicalDevice` is the handle to the physical device that will be used to determine the attributes.
- `pAttributes` is an opaque `NvSciBufAttrList` in which the implementation will set the requested attributes.

On success, `pAttributes` will contain an unreconciled `NvSciBufAttrList` whose private attributes are filled in by the implementation. If the private attributes of `physicalDevice` could not be obtained, `VK_ERROR_INITIALIZATION_FAILED` is returned.

Valid Usage

- VUID-vkGetPhysicalDeviceSciBufAttributesNV-pAttributes-05101
  
  `pAttributes` must be a valid `NvSciBufAttrList` and must not be NULL

Valid Usage (Implicit)

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

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_INITIALIZATION_FAILED

To import memory from a `NvSciBufObj`, add a `VkImportMemorySciBufInfoNV` structure to the `pNext` chain of the `VkMemoryAllocateInfo` structure.

The `VkImportMemorySciBufInfoNV` structure is defined as:
typedef struct VkImportMemorySciBufInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkExternalMemoryHandleTypeFlagBits handleType;
    NvSciBufObj handle;
} VkImportMemorySciBufInfoNV;

- \textit{sType} is the type of this structure.
- \textit{pNext} is NULL or a pointer to a structure extending this structure.
- \textit{handleType} specifies the type of handle or name.
- \textit{handle} is the external handle to import.

Importing memory from a \texttt{NvSciBufObj} does not transfer ownership of the \texttt{NvSciBufObj} from the application to the Vulkan implementation. Vulkan will increment the reference count of the underlying memory of the imported \texttt{NvSciBufObj}. The application \textbf{must} release its ownership using \texttt{NvSciBuf APIs} when that ownership is no longer needed.

Applications \textbf{can} import the same payload into multiple instances of Vulkan, into the same instance from which it was exported, and multiple times into a given Vulkan instance. In all cases, each import operation \textbf{must} create a distinct \texttt{VkDeviceMemory} object.

After successfully importing the \texttt{NvSciBufObj} to \texttt{VkDeviceMemory}, the application \textbf{can} use it as a normal \texttt{VkDeviceMemory} object. It is the application's responsibility to synchronize the different \texttt{NvSciBufObj} accesses.

### Valid Usage

- VUID-VkImportMemorySciBufInfoNV-handleType-05102
  \texttt{handleType} \textbf{must} be \texttt{VK_EXTERNAL_MEMORY_HANDLE_TYPE_ENUMSCI_BUF_BIT_NV}

### Valid Usage (Implicit)

- VUID-VkImportMemorySciBufInfoNV-sType-sType
  \texttt{sType} \textbf{must} be \texttt{VK_STRUCTURE_TYPE_IMPORT_MEMORY_SCI_BUF_INFO_NV}

- VUID-VkImportMemorySciBufInfoNV-handleType-parameter
  \texttt{handleType} \textbf{must} be a valid \texttt{VkExternalMemoryHandleTypeFlagBits} value

To export a \texttt{NvSciBufObj} representing the payload of a Vulkan device memory object, call:

```c
// Provided by VK_NV_external_memory_sci_buf
VkResult vkGetMemorySciBufNV(
    VkDevice device,
    const VkMemoryGetSciBufInfoNV* pGetSciBufInfo,
)`
• **device** is the logical device that created the device memory being exported.

• **pGetSciBufInfo** is a pointer to a `VkMemoryGetSciBufInfoNV` structure containing parameters of the export operation.

• **pHandle** will return the `NvSciBufObj` representing the payload of the device memory object.

A call to `vkGetMemorySciBufNV` will not transfer the ownership of the `NvSciBufObj` handle to the application. The application will hold a reference to the `NvSciBufObj`, but it does not add a reference count to the `NvSciBufObj`, so the application must not release it.

### Valid Usage (Implicit)

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

- **VUID-vkGetMemorySciBufNV-pGetSciBufInfo-parameter**
  - `pGetSciBufInfo` must be a valid pointer to a valid `VkMemoryGetSciBufInfoNV` structure

- **VUID-vkGetMemorySciBufNV-pHandle-parameter**
  - `pHandle` must be a valid pointer to a `NvSciBufObj` value

### Return Codes

**Success**

- **VK_SUCCESS**

**Failure**

- **VK_ERROR_INITIALIZATION_FAILED**

The `VkMemoryGetSciBufInfoNV` structure is defined as:

```c
// Provided by VK_NV_external_memory_sci_buf
typedef struct VkMemoryGetSciBufInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkDeviceMemory memory;
    VkExternalMemoryHandleTypeFlagBits handleType;
} VkMemoryGetSciBufInfoNV;
```

- **sType** is the type of this structure.

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

- **memory** is the memory object from which the handle will be exported.

- **handleType** is the type of handle requested.
A `NvSciBufObj` handle compatible with Vulkan can also be created by non-Vulkan APIs using methods beyond the scope of this specification. To determine the correct parameters to use when importing such handles, call:

```c
// Provided by VK_NV_external_memory_sci_buf
VkResult vkGetPhysicalDeviceExternalMemorySciBufPropertiesNV(
    VkPhysicalDevice physicalDevice,    // physicalDevice is the handle to the physical device whose properties will be queried.
    VkExternalMemoryHandleTypeFlagBits handleType,    // handleType is the type of the handle handle.
    NvSciBufObj handle,    // handle is the NvSciBufObj handle which will be imported.
)
```

This command will return properties of `handle`, it contains the memory type bitmask that can be used to determine the `VkMemoryAllocateInfo`::`memoryTypeIndex` when calling `vkAllocateMemory`.

### Valid Usage

- **VUID-vkGetPhysicalDeviceExternalMemorySciBufPropertiesNV-handleType-05104**
  
  `handleType must be VK_EXTERNAL_MEMORY_HANDLE_TYPE_SCI_BUF_BIT_NV`

- **VUID-vkGetPhysicalDeviceExternalMemorySciBufPropertiesNV-sciBufImport-05105**
  
  `VkPhysicalDeviceExternalMemorySciBufFeaturesNV::sciBufImport must be enabled`
Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceExternalMemorySciBufPropertiesNV-physicalDevice-parameter
  `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceExternalMemorySciBufPropertiesNV-handleType-parameter
  `handleType` must be a valid `VkExternalMemoryHandleTypeFlagBits` value
- VUID-vkGetPhysicalDeviceExternalMemorySciBufPropertiesNV-
  pMemorySciBufProperties-parameter
  `pMemorySciBufProperties` must be a valid pointer to a `VkMemorySciBufPropertiesNV` structure

Return Codes

Success
- `VK_SUCCESS`

Failure
- `VK_ERROR_INITIALIZATION_FAILED`
- `VK_ERROR_INVALID_EXTERNAL_HANDLE`

The `VkMemorySciBufPropertiesNV` structure is defined as:

```
// Provided by VK_NV_external_memory_sci_buf
typedef struct VkMemorySciBufPropertiesNV {
    VkStructureType sType;
    const void* pNext;
    uint32_t memoryTypeBits;
} VkMemorySciBufPropertiesNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `memoryTypeBits` is a bitmask containing one bit set for every memory type for which the specified `NvSciBufObj` handle can be imported.

Valid Usage (Implicit)

- VUID-VkMemorySciBufPropertiesNV-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_MEMORY_SCI_BUF_PROPERTIES_NV`
- VUID-VkMemorySciBufPropertiesNV-pNext-pNext
  `pNext` must be `NULL`
### 11.2.7. Device Group Memory Allocations

If the `pNext` chain of `VkMemoryAllocateInfo` includes a `VkMemoryAllocateFlagsInfo` structure, then that structure includes flags and a device mask controlling how many instances of the memory will be allocated.

The `VkMemoryAllocateFlagsInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkMemoryAllocateFlagsInfo {
    VkStructureType sType;
    const void* pNext;
    VkMemoryAllocateFlags flags;
    uint32_t deviceMask;
} VkMemoryAllocateFlagsInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is a bitmask of `VkMemoryAllocateFlagBits` controlling the allocation.
- `deviceMask` is a mask of physical devices in the logical device, indicating that memory must be allocated on each device in the mask, if `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT` is set in `flags`.

If `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT` is not set, the number of instances allocated depends on whether `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` is set in the memory heap. If `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` is set, then memory is allocated for every physical device in the logical device (as if `deviceMask` has bits set for all device indices). If `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` is not set, then a single instance of memory is allocated (as if `deviceMask` is set to one).

On some implementations, allocations from a multi-instance heap may consume memory on all physical devices even if the `deviceMask` excludes some devices. If `VkPhysicalDeviceGroupProperties::subsetAllocation` is `VK_TRUE`, then memory is only consumed for the devices in the device mask.

**Note**

In practice, most allocations on a multi-instance heap will be allocated across all physical devices. Unicast allocation support is an optional optimization for a minority of allocations.

### Valid Usage

- **VUID-VkMemoryAllocateFlagsInfo-deviceMask-00675**
  If `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT` is set, `deviceMask` must be a valid device mask

- **VUID-VkMemoryAllocateFlagsInfo-deviceMask-00676**
  If `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT` is set, `deviceMask` must not be zero
Valid Usage (Implicit)

- VUID-VkMemoryAllocateFlagsInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_FLAGS_INFO

- VUID-VkMemoryAllocateFlagsInfo-flags-parameter
  flags must be a valid combination of VkMemoryAllocateFlagBits values

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,
} 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_2
typedef VkFlags VkMemoryAllocateFlags;
```

VkMemoryAllocateFlags is a bitmask type for setting a mask of zero or more VkMemoryAllocateFlagBits.

11.2.8. 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:

```c
// Provided by VK_VERSION_1_2
typedef struct VkMemoryOpaqueCaptureAddressAllocateInfo {
```
VkStructureType sType;
const void* pNext;
uint64_t opaqueCaptureAddress;
} VkMemoryOpaqueCaptureAddressAllocateInfo;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **opaqueCaptureAddress** is the opaque capture address requested for the memory allocation.

If **opaqueCaptureAddress** is zero, no specific address is requested.

If **opaqueCaptureAddress** is not zero, it **should** be an address retrieved from `vkGetDeviceMemoryOpaqueCaptureAddress` on an identically created memory allocation on the same implementation.

**Note**

In most cases, it is expected that a non-zero **opaqueAddress** is an address retrieved from `vkGetDeviceMemoryOpaqueCaptureAddress` on an identically created memory allocation. If this is not the case, it is likely that `VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS` errors will occur.

This is, however, not a strict requirement because trace capture/replay tools may need to adjust memory allocation parameters for imported memory.

If this structure is not present, it is as if **opaqueCaptureAddress** is zero.

**Valid Usage (Implicit)**

- VUID-VkMemoryOpaqueCaptureAddressAllocateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_MEMORY_OPAQUE_CAPTURE_ADDRESS_ALLOCATE_INFO

**11.2.9. 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.

**11.2.10. 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:
// Provided by VK_VERSION_1_0

VkResult vkMapMemory(
    VkDevice device,
    VkDeviceMemory memory,
    VkDeviceSize offset,
    VkDeviceSize size,
    VkMemoryMapFlags flags,
    void** ppData);

• device is the logical device that owns the memory.
• memory is the VkDeviceMemory object to be mapped.
• offset is a zero-based byte offset from the beginning of the memory object.
• size is the size of the memory range to map, or VK_WHOLE_SIZE to map from offset to the end of the allocation.
• flags is reserved for future use.
• ppData is a pointer to a void * variable in which is returned a host-accessible pointer to the beginning of the mapped range. This pointer minus offset must be aligned to at least VkPhysicalDeviceLimits::minMemoryMapAlignment.

After a successful call to vkMapMemory the memory object memory is considered to be currently host mapped.

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-zerobitsetmask
  
  `flags` must be 0

- VUID-vkMapMemory-ppData-parameter
  
  `ppData` must be a valid pointer to a pointer value

- VUID-vkMapMemory-memory-parent
  
  `memory` must have been created, allocated, or retrieved from `device`

### Host Synchronization

- Host access to `memory` must be externally synchronized
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY
• VK_ERROR_MEMORY_MAP_FAILED

// Provided by VK_VERSION_1_0
typedef VkFlags VkMemoryMapFlags;

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

Two commands are provided to enable applications to work with non-coherent memory allocations: vkFlushMappedMemoryRanges and vkInvalidateMappedMemoryRanges.

**Note**
If the memory object was created with the VK_MEMORY_PROPERTY_HOST_COHERENT_BIT set, vkFlushMappedMemoryRanges and vkInvalidateMappedMemoryRanges are unnecessary and may have a performance cost. However, availability and visibility operations still need to be managed on the device. See the description of host access types for more information.

**Note**
While memory objects imported from a handle type of VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT or VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT are inherently mapped to host address space, they are not considered to be host mapped device memory unless they are explicitly host mapped using vkMapMemory. That means flushing or invalidating host caches with respect to host accesses performed on such memory through the original host pointer specified at import time is the responsibility of the application and must be performed with appropriate synchronization primitives provided by the platform which are outside the scope of Vulkan. vkFlushMappedMemoryRanges and vkInvalidateMappedMemoryRanges, however, can still be used on such memory objects to synchronize host accesses performed through the host pointer of the host mapped device memory range returned by vkMapMemory.

To flush ranges of non-coherent memory from the host caches, call:

// Provided by VK_VERSION_1_0
VkResult vkFlushMappedMemoryRanges(
VkDevice

uint32_t

const VkMappedMemoryRange*

device,

memoryRangeCount,

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:

// 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 the type of this structure.
• `pNext` is NULL or a pointer to a structure extending this structure.
• `memory` is the memory object to which this range belongs.
• `offset` is the zero-based byte offset from the beginning of the memory object.
• `size` is either the size of range, or VK_WHOLE_SIZE to affect the range from `offset` to the end of the current mapping of the allocation.

Valid Usage

• VUID-VkMappedMemoryRange-memory-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
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 `size` is not equal to `VK_WHOLE_SIZE`, `size` must either be a multiple of `VkPhysicalDeviceLimits::nonCoherentAtomSize`, or `offset` plus `size` must equal the size of `memory`.

Valid Usage (Implicit)

- VUID-VkMappedMemoryRange-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE`

- VUID-VkMappedMemoryRange-pNext-pNext
  `pNext` must be `NULL`

- VUID-VkMappedMemoryRange-memory-parameter
  `memory` must be a valid `VkDeviceMemory` handle

To unmapping a memory object once host access to it is no longer needed by the application, call:

```c
// Provided by VK_VERSION_1_0
void vkUnmapMemory(
    VkDevice device, 
    VkDeviceMemory memory);
```

- `device` is the logical device that owns the memory.
- `memory` is the memory object to be unmapped.

Valid Usage

- VUID-vkUnmapMemory-memory-00689
  `memory` must be currently host mapped

Valid Usage (Implicit)

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

- VUID-vkUnmapMemory-memory-parameter
  `memory` must be a valid `VkDeviceMemory` handle

- VUID-vkUnmapMemory-memory-parent
  `memory` must have been created, allocated, or retrieved from `device`
Host Synchronization

• Host access to memory must be externally synchronized

11.2.11. 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-memory-00690
  memory must have been created with a memory type that reports VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT
Valid Usage (Implicit)

- VUID-vkGetDeviceMemoryCommitment-device-parameter
  device must be a valid VkDevice handle
- VUID-vkGetDeviceMemoryCommitment-memory-parameter
  memory must be a valid VkDeviceMemory handle
- VUID-vkGetDeviceMemoryCommitment-pCommittedMemoryInBytes-parameter
  pCommittedMemoryInBytes must be a valid pointer to a VkDeviceSize value
- VUID-vkGetDeviceMemoryCommitment-memory-parent
  memory must have been created, allocated, or retrieved from device

11.2.12. 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
Protected Memory Access Rules

If `VkPhysicalDeviceProtectedMemoryProperties::protectedNoFault` is `VK_FALSE`, applications must not perform any of the following operations:

- Write to unprotected memory within protected queue operations.
- Access protected memory within protected queue operations other than in framebuffer-space pipeline stages, the compute shader stage, or the transfer stage.
- Perform a query within protected queue operations.

If `VkPhysicalDeviceProtectedMemoryProperties::protectedNoFault` is `VK_TRUE`, these operations are valid, but reads will return undefined values, and writes will either be dropped or store undefined values.

Additionally, indirect operations must not be performed within protected queue operations.

Whether these operations are valid or not, or if any other invalid usage is performed, the implementation must guarantee that:

- Protected device memory must never be visible to the host.
- Values written to unprotected device memory must not be a function of values from protected memory.

11.2.13. 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);
```

- `device` is the logical device that owns the memory.
- `heapIndex` is the index of the memory heap from which the memory is allocated.
- `localDeviceIndex` is the device index of the physical device that performs the memory access.
- `remoteDeviceIndex` is the device index of the physical device that the memory is allocated for.
• pPeerMemoryFeatures is a pointer to a VkPeerMemoryFeatureFlags bitmask indicating which types of memory accesses are supported for the combination of heap, local, and remote devices.

### Valid Usage

- **VUID-vkGetDeviceGroupPeerMemoryFeatures-heapIndex-00691**
  
  heapIndex **must** be less than memoryHeapCount

- **VUID-vkGetDeviceGroupPeerMemoryFeatures-localDeviceIndex-00692**
  
  localDeviceIndex **must** be a valid device index

- **VUID-vkGetDeviceGroupPeerMemoryFeatures-remoteDeviceIndex-00693**
  
  remoteDeviceIndex **must** be a valid device index

- **VUID-vkGetDeviceGroupPeerMemoryFeatures-localDeviceIndex-00694**
  
  localDeviceIndex **must** not equal remoteDeviceIndex

### Valid Usage (Implicit)

- **VUID-vkGetDeviceGroupPeerMemoryFeatures-device-parameter**
  
  device **must** be a valid VkDevice handle

- **VUID-vkGetDeviceGroupPeerMemoryFeatures-pPeerMemoryFeatures-parameter**
  
  pPeerMemoryFeatures **must** be a valid pointer to a VkPeerMemoryFeatureFlags value

Bits which **may** be set in the value returned for **vkGetDeviceGroupPeerMemoryFeatures::pPeerMemoryFeatures**, indicating the 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**.

### 11.2.14. 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
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 the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `memory` specifies the memory whose address is being queried.

### Valid Usage
- VUID-VkDeviceMemoryOpaqueCaptureAddressInfo-memory-03336
  - `memory` **must** have been allocated with `VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT`

### Valid Usage (Implicit)
- VUID-VkDeviceMemoryOpaqueCaptureAddressInfo-sType-sType
  - `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_MEMORY_OPAQUE_CAPTURE_ADDRESS_INFO`
- VUID-VkDeviceMemoryOpaqueCaptureAddressInfo-pNext-pNext
  - `pNext` **must** be `NULL`
- VUID-VkDeviceMemoryOpaqueCaptureAddressInfo-memory-parameter
  - `memory` **must** be a valid `VkDeviceMemory` handle
Chapter 12. Resource Creation

Vulkan supports two primary resource types: buffers and images. Resources are views of memory with associated formatting and dimensionality. Buffers are essentially unformatted arrays of bytes whereas images contain format information, can be multidimensional and may have associated metadata.

12.1. Buffers

Buffers represent linear arrays of data which are used for various purposes by binding them to a graphics or compute pipeline via descriptor sets or via certain commands, or by directly specifying them as parameters to certain commands.

Buffers are represented by VkBuffer handles:

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

To create buffers, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateBuffer(
    VkDevice device,
    const VkBufferCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkBuffer* pBuffer);
```

- `device` is the logical device that creates the buffer object.
- `pCreateInfo` is a pointer to a VkBufferCreateInfo structure containing parameters affecting creation of the buffer.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pBuffer` is a pointer to a VkBuffer handle in which the resulting buffer object is returned.

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
  
  **pAllocate 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 the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **flags** is a bitmask of **VkBufferCreateFlagBits** specifying additional parameters of the buffer.
- **size** is the size in bytes of the buffer to be created.
- **usage** is a bitmask of **VkBufferUsageFlagBits** specifying allowed usages of the buffer.
- **sharingMode** is a **VkSharingMode** value specifying the sharing mode of the buffer when it will be accessed by multiple queue families.
- **queueFamilyIndexCount** is the number of entries in the **pQueueFamilyIndices** array.
• `pQueueFamilyIndices` is a pointer to an array of queue families that will access this buffer. It is ignored if `sharingMode` is not `VK_SHARING_MODE_CONCURRENT`.

### Valid Usage

- **VUID-VkBufferCreateInfo-size-00912**
  `size` must be greater than 0

- **VUID-VkBufferCreateInfo-sharingMode-00913**
  If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, `pQueueFamilyIndices` must be a valid pointer to an array of `queueFamilyIndexCount uint32_t` values

- **VUID-VkBufferCreateInfo-sharingMode-00914**
  If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, `queueFamilyIndexCount` must be greater than 1

- **VUID-VkBufferCreateInfo-sharingMode-01419**
  If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, each element of `pQueueFamilyIndices` must be unique and must be less than `pQueueFamilyPropertyCount` returned by either `vkGetPhysicalDeviceQueueFamilyProperties` or `vkGetPhysicalDeviceQueueFamilyProperties2` for the `physicalDevice` that was used to create device

- **VUID-VkBufferCreateInfo-flags-05061**
  `flags` must not contain `VK_BUFFER_CREATE_SPARSE_BINDING_BIT`, `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT`, or `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 protected memory 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-06549**
  If `flags` includes `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT`, the `bufferDeviceAddressCaptureReplay` feature must be enabled

### Valid Usage (Implicit)

- **VUID-VkBufferCreateInfo-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO`
Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkBufferOpaqueCaptureAddressCreateInfo` or `VkExternalMemoryBufferCreateInfo`.

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

Flags `must` be a valid combination of `VkBufferCreateFlagBits` values.

Usage `must` be a valid combination of `VkBufferUsageFlagBits` values.

Usage `must` not be 0.

Sharing mode `must` be a valid `VkSharingMode` value.

Bits which can be set in `VkBufferCreateInfo::usage`, specifying usage behavior of a buffer, are:

```c
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.

// 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:

// 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 the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• handleTypes is zero, or a bitmask of VkExternalMemoryHandleTypeFlagBits specifying one or more external memory handle types.

Valid Usage (Implicit)

• VUID-VkExternalMemoryBufferCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO

• VUID-VkExternalMemoryBufferCreateInfo-handleTypes-parameter
  handleTypes must be a valid combination of VkExternalMemoryHandleTypeFlagBits values

To request a specific device address for a buffer, add a VkBufferOpaqueCaptureAddressCreateInfo structure to the pNext chain of the VkBufferCreateInfo structure. The VkBufferOpaqueCaptureAddressCreateInfo structure is defined as:
typedef struct VkBufferOpaqueCaptureAddressCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint64_t opaqueCaptureAddress;
} VkBufferOpaqueCaptureAddressCreateInfo;

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- opaqueCaptureAddress is the opaque capture address requested for the buffer.

If opaqueCaptureAddress is zero, no specific address is requested.

If opaqueCaptureAddress is not zero, then it should be an address retrieved from vkGetBufferOpaqueCaptureAddress for an identically created buffer on the same implementation.

If this structure is not present, it is as if opaqueCaptureAddress is zero.

Apps should avoid creating buffers with app-provided addresses and implementation-provided addresses in the same process, to reduce the likelihood of VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS errors.

Note
The expected usage for this is that a trace capture/replay tool will add the VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT flag to all buffers that use VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT, and during capture will save the queried opaque device addresses in the trace. During replay, the buffers will be created specifying the original address so any address values stored in the trace data will remain valid.

Implementations are expected to separate such buffers in the GPU address space so normal allocations will avoid using these addresses. Apps/tools should avoid mixing app-provided and implementation-provided addresses for buffers created with VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT, to avoid address space allocation conflicts.

Valid Usage (Implicit)
- VUID-VkBufferOpaqueCaptureAddressCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_BUFFER_OPAQUE_CAPTURE_ADDRESS_CREATE_INFO

To destroy a buffer, call:

// 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.
To create a buffer view, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateBufferView(
    VkDevice device,  // device is the logical device that creates the buffer view.
    const VkBufferViewCreateInfo* pCreateInfo,  // pCreateInfo is a pointer to a VkBufferViewCreateInfo structure containing parameters to be used to create the buffer view.
    const VkAllocationCallbacks* pAllocator,  // pAllocator controls host memory allocation as described in the Memory Allocation chapter.
    VkBufferView* pView);  // 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 the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `buffer` is a `VkBuffer` on which the view will be created.
- `format` is a `VkFormat` describing the format of the data elements in the buffer.
- `offset` is an offset in bytes from the base address of the buffer. Accesses to the buffer view from shaders use addressing that is relative to this starting offset.
- `range` is a size in bytes of the buffer view. If `range` is equal to `VK_WHOLE_SIZE`, the range from `offset` to the end of the buffer is used. If `VK_WHOLE_SIZE` is used and the remaining size of the buffer is not a multiple of the texel block size of `format`, the nearest smaller multiple is used.

Valid Usage

- VUID-VkBufferViewCreateInfo-offset-00925
  `offset` must be less than the size of `buffer`

- VUID-VkBufferViewCreateInfo-range-00928
  If `range` is not equal to `VK_WHOLE_SIZE`, `range` must be greater than 0

- VUID-VkBufferViewCreateInfo-range-00929
  If `range` is not equal to `VK_WHOLE_SIZE`, `range` must be an integer multiple of the texel block size of `format`
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`.

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`.

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`.

`buffer` **must** have been created with a usage value containing at least one of `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT` or `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT`.

If `buffer` was created with usage containing `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT`, `format` **must** be supported for uniform texel buffers, as specified by the `VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT` flag in `VkFormatProperties::bufferFeatures` returned by `vkGetPhysicalDeviceFormatProperties`.

If `buffer` was created with usage containing `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT`, `format` **must** be supported for storage texel buffers, as specified by the `VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT` flag in `VkFormatProperties::bufferFeatures` returned by `vkGetPhysicalDeviceFormatProperties`.

If `buffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object.

If the `texelBufferAlignment` feature is not enabled, `offset` **must** be a multiple of `VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment`.

If the `texelBufferAlignment` feature is enabled and if `buffer` was created with usage containing `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT`, `offset` **must** be a multiple of the lesser of `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT::storageTexelBufferOffsetAlignmentBytes` or, if `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT::storageTexelBufferOffsetSingleTexelAlignment` is `VK_TRUE`, the size of a texel of the requested `format`. If the size of a texel is a multiple of three bytes, then the size of a single component of `format` is used instead.

If the `texelBufferAlignment` feature is enabled and if `buffer` was created with usage containing `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT`, `offset` **must** be a multiple of the lesser of `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT::uniformTexelBufferOffsetAlignment`.
::uniformTexelBufferOffsetAlignmentBytes or, if
VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT::uniformTexelBufferOffsetSingleTex
elAlignment is VK_TRUE, the size of a texel of the requested format. If the size of a texel is a
multiple of three bytes, then the size of a single component of format is used instead

---

**Valid Usage (Implicit)**

- VUID-VkBufferViewCreateInfo-sType-sType
  sType **must** be VK_STRUCTURE_TYPE_BUFFER_VIEW_CREATE_INFO
- VUID-VkBufferViewCreateInfo-pNext-pNext
  pNext **must** be NULL
- VUID-VkBufferViewCreateInfo-flags-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

---

// Provided by VK_VERSION_1_0
typedef VkFlags VkBufferViewCreateFlags;

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

To destroy a buffer view, call:

// Provided by VK_VERSION_1_0
void vkDestroyBufferView(
  VkDevice device,
  VkBufferView bufferView,
  const VkAllocationCallbacks* pAllocator);

- **device** is the logical device that destroys the buffer view.
- **bufferView** is the buffer view to destroy.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

---

**Valid Usage**

- VUID-vkDestroyBufferView-bufferView-00936
  All submitted commands that refer to bufferView **must** have completed execution


**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-bufferView-parent
  
  If `bufferView` is a valid handle, it must have been created, allocated, or retrieved from `device`

**Host Synchronization**

- Host access to `bufferView` must be externally synchronized

### 12.3. Images

Images represent multidimensional - up to 3 - arrays of data which can be used for various purposes (e.g. attachments, textures), by binding them to a graphics or compute pipeline via descriptor sets, or by directly specifying them as parameters to certain commands.

Images are represented by `VkImage` handles:

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

To create images, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateImage(  
    VkDevice device,  
    const VkImageCreateInfo* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkImage* pImage);  
```

- `device` is the logical device that creates the image.
- `pCreateInfo` is a pointer to a `VkImageCreateInfo` structure containing parameters to be used to create the image.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pImage` is a pointer to a `VkImage` handle in which the resulting image object is returned.
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
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;
} VkImageCreateInfo;
```
VkImageUsageFlags usage;
VkSharingMode sharingMode;
uint32_t queueFamilyIndexCount;
const uint32_t* pQueueFamilyIndices;
VkImageLayout initialLayout;
} VkImageCreateInfo;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is a bitmask of VkImageCreateFlagBits describing additional parameters of the image.
- **imageType** is a VkImageType value specifying the basic dimensionality of the image. Layers in array textures do not count as a dimension for the purposes of the image type.
- **format** is a VkFormat describing the format and type of the texel blocks that will be contained in the image.
- **extent** is a VkExtent3D describing the number of data elements in each dimension of the base level.
- **mipLevels** describes the number of levels of detail available for minified sampling of the image.
- **arrayLayers** is the number of layers in the image.
- **samples** is a VkSampleCountFlagBits value specifying the number of samples per texel.
- **tiling** is a VkImageTiling value specifying the tiling arrangement of the texel blocks in memory.
- **usage** is a bitmask of VkImageUsageFlagBits describing the intended usage of the image.
- **sharingMode** is a VkSharingMode value specifying the sharing mode of the image when it will be accessed by multiple queue families.
- **queueFamilyIndexCount** is the number of entries in the pQueueFamilyIndices array.
- **pQueueFamilyIndices** is a pointer to an array of queue families that will access this image. It is ignored if sharingMode is not VK_SHARING_MODE_CONCURRENT.
- **initialLayout** is a VkImageLayout value specifying the initial VkImageLayout of all image subresources of the image. See Image Layouts.

Images created with **tiling** equal to VK_IMAGE_TILING_LINEAR have further restrictions on their limits and capabilities compared to images created with **tiling** equal to VK_IMAGE_TILING_OPTIMAL. Creation of images with tiling VK_IMAGE_TILING_LINEAR may not be supported unless other parameters meet all of the constraints:

- **imageType** is VK_IMAGE_TYPE_2D
- **format** is not a depth/stencil format
- **mipLevels** is 1
- **arrayLayers** is 1
- **samples** is VK_SAMPLE_COUNT_1_BIT
- **usage** only includes VK_IMAGE_USAGE_TRANSFER_SRC_BIT and/or VK_IMAGE_USAGE_TRANSFER_DST_BIT
Images created with one of the formats that require a sampler $Y' CB R$ conversion, have further restrictions on their limits and capabilities compared to images created with other formats. Creation of images with a format requiring $Y' CB R$ conversion may not be supported unless other parameters meet all of the constraints:

- `imageType` is `VK_IMAGE_TYPE_2D`
- `mipLevels` is 1
- `arrayLayers` is 1
- `samples` is `VK_SAMPLE_COUNT_1_BIT`

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

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

If the size of the resultant image would exceed `maxResourceSize`, then `vkCreateImage` must fail and return `VK_ERROR_OUT_OF_DEVICE_MEMORY`. This failure may occur even when all image creation parameters satisfy their valid usage requirements.

**Note**

For images created without `VK_IMAGE_CREATE_EXTENDED_USAGE_BIT` a usage bit is valid if it is supported for the format the image is created with.

For images created with `VK_IMAGE_CREATE_EXTENDED_USAGE_BIT` a usage bit is valid if it is supported for at least one of the formats a `VkImageView` created from the image can have (see Image Views for more detail).

---

### Image Creation Limits

Valid values for some image creation parameters are limited by a numerical upper bound or by inclusion in a bitset. For example, `VkImageCreateInfo::arrayLayers` is limited by `imageCreateMaxArrayLayers`, defined below; and `VkImageCreateInfo::samples` is limited by `imageCreateSampleCounts`, also defined below.

Several limiting values are defined below, as well as assisting values from which the limiting values are derived. The limiting values are referenced by the relevant valid usage statements of `VkImageCreateInfo`.

- Let `uint64_t imageCreateDrmFormatModifiers[]` be the set of Linux DRM format modifiers that the resultant image may have.

  - If tiling is not `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then `imageCreateDrmFormatModifiers` is empty.
  - If `VkImageCreateInfo::pNext` contains `VkImageDrmFormatModifierExplicitCreateInfoEXT`, then `imageCreateDrmFormatModifiers` contains exactly one modifier, `VkImageDrmFormatModifierExplicitCreateInfoEXT::drmFormatModifier`.

---

541
If `VkImageCreateInfo::pNext` contains `VkImageDrmFormatModifierListCreateInfoEXT`, then `imageCreateDrmFormatModifiers` contains the entire array `VkImageDrmFormatModifierListCreateInfoEXT::pDrmFormatModifiers`.

- Let `VkBool32 imageCreateMaybeLinear` indicate if the resultant image may be linear.
  - If tiling is `VK_IMAGE_TILING_LINEAR`, then `imageCreateMaybeLinear` is `VK_TRUE`.
  - If tiling is `VK_IMAGE_TILING_OPTIMAL`, then `imageCreateMaybeLinear` is `VK_FALSE`.
  - If tiling is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then `imageCreateMaybeLinear` is `VK_TRUE` if and only if `imageCreateDrmFormatModifiers` contains `DRM_FORMAT_MOD_LINEAR`.

- 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`.
  - If tiling is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then the value of `imageCreateFormatFeatures` is found by calling `vkGetPhysicalDeviceFormatProperties2` with ` VkImageFormatProperties::format ` equal to `VkImageCreateInfo::format` and with `VkDrmFormatModifierPropertiesListEXT` chained into `VkImageFormatProperties2`; by collecting all members of the returned array `VkDrmFormatModifierPropertiesListEXT::pDrmFormatModifierProperties` whose `drmFormatModifier` belongs to `imageCreateDrmFormatModifiers`; and by taking the bitwise intersection, over the collected array members, of `drmFormatModifierTilingFeatures`. (The resultant `imageCreateFormatFeatures` may be empty).

- Let `VkImageFormatProperties2 imageCreateImageFormatPropertiesList[]` be the list of structures obtained by calling `vkGetPhysicalDeviceImageFormatProperties2`, possibly multiple times, as follows:
  - The parameters `VkPhysicalDeviceImageFormatInfo2::format`, `imageType`, `tiling`, `usage`, and `flags` must be equal to those in `VkImageCreateInfo`.
  - If `VkImageCreateInfo::pNext` contains a `VkExternalMemoryImageCreateInfo` structure whose `handleTypes` is not 0, then `VkPhysicalDeviceImageFormatInfo2::pNext` must contain a `VkPhysicalDeviceExternalImageFormatInfo` structure whose `handleType` is not 0; and `vkGetPhysicalDeviceImageFormatProperties2` must be called for each handle type in `VkExternalMemoryImageCreateInfo::handleTypes`, successively setting `VkPhysicalDeviceExternalImageFormatInfo::handleType` on each call.
  - If `VkImageCreateInfo::pNext` contains no `VkExternalMemoryImageCreateInfo` structure, or contains a structure whose `handleTypes` is 0, then `VkPhysicalDeviceImageFormatInfo2::pNext` must either contain no `VkPhysicalDeviceExternalImageFormatInfo` structure, or contain a structure whose `handleType` is 0.
• If tiling is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then `VkPhysicalDeviceImageFormatInfo2::pNext` must contain a `VkPhysicalDeviceImageDrmFormatModifierInfoEXT` structure where `sharingMode` is equal to `VkImageCreateInfo::sharingMode`; and, if `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, then `queueFamilyIndexCount` and `pQueueFamilyIndices` must be equal to those in `VkImageCreateInfo`; and, if `flags` contains `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT`, then the `VkImageFormatListCreateInfo` structure included in the `pNext` chain of `VkPhysicalDeviceImageFormatInfo2` must be equivalent to the one included in the `pNext` chain of `VkImageCreateInfo`; and `vkGetPhysicalDeviceImageFormatProperties2` must be called for each modifier in `imageCreateDrmFormatModifiers`, successively setting `VkPhysicalDeviceImageDrmFormatModifierInfoEXT::drmFormatModifier` on each call.

• If tiling is not `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then `VkPhysicalDeviceImageFormatInfo2::pNext` must contain no `VkPhysicalDeviceImageDrmFormatModifierInfoEXT` structure.

• If any call to `vkGetPhysicalDeviceImageFormatProperties2` returns an error, then `imageCreateImageFormatPropertiesList` is defined to be the empty list.

  • Let `uint32_t imageCreateMaxMipLevels` be the minimum value of `VkImageFormatProperties::maxMipLevels` in `imageCreateImageFormatPropertiesList`. The value is undefined if `imageCreateImageFormatPropertiesList` is empty.

  • Let `uint32_t imageCreateMaxArrayLayers` be the minimum value of `VkImageFormatProperties::maxArrayLayers` in `imageCreateImageFormatPropertiesList`. The value is undefined if `imageCreateImageFormatPropertiesList` is empty.

  • Let `VkExtent3D imageCreateMaxExtent` be the component-wise minimum over all `VkImageFormatProperties::maxExtent` values in `imageCreateImageFormatPropertiesList`. The value is undefined if `imageCreateImageFormatPropertiesList` is empty.

  • Let `VkSampleCountFlags imageCreateSampleCounts` be the intersection of each `VkImageFormatProperties::sampleCounts` in `imageCreateImageFormatPropertiesList`. The value is undefined if `imageCreateImageFormatPropertiesList` is empty.

Valid Usage

• VUID-VkImageCreateInfo-imageCreateMaxMipLevels-02251
Each of the following values (as described in Image Creation Limits) must not be undefined: `imageCreateMaxMipLevels`, `imageCreateMaxArrayLayers`, `imageCreateMaxExtent`, and `imageCreateSampleCounts`

• VUID-VkImageCreateInfo-sharingMode-00941
If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, `pQueueFamilyIndices` must be a valid pointer to an array of `queueFamilyIndexCount uint32_t` values

• VUID-VkImageCreateInfo-sharingMode-00942
If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, `queueFamilyIndexCount` must be greater than 1

• VUID-VkImageCreateInfo-sharingMode-01420
If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, each element of `pQueueFamilyIndices` must be unique and must be less than `pQueueFamilyPropertyCount` returned by either `vkGetPhysicalDeviceQueueFamilyProperties` or `vkGetPhysicalDeviceQueueFamilyProperties2` for the `physicalDevice` that was used to create device

- VUID-VkImageCreateInfo-format-00943
  `format` must not be `VK_FORMAT_UNDEFINED`

- VUID-VkImageCreateInfo-extent-00944
  `extent.width` must be greater than 0

- VUID-VkImageCreateInfo-extent-00945
  `extent.height` must be greater than 0

- VUID-VkImageCreateInfo-extent-00946
  `extent.depth` must be greater than 0

- VUID-VkImageCreateInfo-mipLevels-00947
  `mipLevels` must be greater than 0

- VUID-VkImageCreateInfo-arrayLayers-00948
  `arrayLayers` must be greater than 0

- VUID-VkImageCreateInfo-flags-00949
  If `flags` contains `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`, `imageType` must be `VK_IMAGE_TYPE_2D`

- VUID-VkImageCreateInfo-flags-00950
  If `flags` contains `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT`, `imageType` must be `VK_IMAGE_TYPE_3D`

- VUID-VkImageCreateInfo-extent-02252
  `extent.width` must be less than or equal to `imageCreateMaxExtent.width` (as defined in Image Creation Limits)

- VUID-VkImageCreateInfo-extent-02253
  `extent.height` must be less than or equal to `imageCreateMaxExtent.height` (as defined in Image Creation Limits)

- VUID-VkImageCreateInfo-extent-02254
  `extent.depth` must be less than or equal to `imageCreateMaxExtent.depth` (as defined in Image Creation Limits)

- VUID-VkImageCreateInfo-imageType-00954
  If `imageType` is `VK_IMAGE_TYPE_2D` and `flags` contains `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`, `extent.width` and `extent.height` must be equal and `arrayLayers` must be greater than or equal to 6

- VUID-VkImageCreateInfo-imageType-00956
  If `imageType` is `VK_IMAGE_TYPE_1D`, both `extent.height` and `extent.depth` must be 1

- VUID-VkImageCreateInfo-imageType-00957
  If `imageType` is `VK_IMAGE_TYPE_2D`, `extent.depth` must be 1

- VUID-VkImageCreateInfo-mipLevels-00958
  `mipLevels` must be less than or equal to the number of levels in the complete mipmap
chain based on \(\text{extent.width, extent.height, and extent.depth}\)

- **VUID-VkImageCreateInfo-mipLevels-02255**
  \(\text{mipLevels must}\) be less than or equal to \(\text{imageCreateMaxMipLevels}\) (as defined in \text{Image Creation Limits})

- **VUID-VkImageCreateInfo-arrayLayers-02256**
  \(\text{arrayLayers must}\) be less than or equal to \(\text{imageCreateMaxArrayLayers}\) (as defined in \text{Image Creation Limits})

- **VUID-VkImageCreateInfo-imageType-00961**
  If \(\text{imageType}\) is \(\text{VK_IMAGE_TYPE_3D}\), \(\text{arrayLayers must}\) be 1

- **VUID-VkImageCreateInfo-samples-02257**
  If \(\text{samples}\) is not \(\text{VK_SAMPLE_COUNT_1_BIT}\), then \(\text{imageType must}\) be \(\text{VK_IMAGE_TYPE_2D}\), flags \(\text{must}\) not contain \(\text{VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT}\), \(\text{mipLevels must}\) be equal to 1, and \(\text{imageCreateMaybeLinear}\) (as defined in \text{Image Creation Limits}) \(\text{must}\) be \(\text{VK_FALSE}\)

- **VUID-VkImageCreateInfo-usage-00963**
  If \(\text{usage}\) includes \(\text{VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT}\), then bits other than \(\text{VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, and VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT}\) \(\text{must}\) not be set

- **VUID-VkImageCreateInfo-usage-00964**
  If \(\text{usage}\) includes \(\text{VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT, or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT}\), \(\text{extent.width must}\) be less than or equal to \(\text{VkPhysicalDeviceLimits::maxFramebufferWidth}\)

- **VUID-VkImageCreateInfo-usage-00965**
  If \(\text{usage}\) includes \(\text{VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT, or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT}\), \(\text{extent.height must}\) be less than or equal to \(\text{VkPhysicalDeviceLimits::maxFramebufferHeight}\)

- **VUID-VkImageCreateInfo-usage-00966**
  If \(\text{usage}\) includes \(\text{VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT}\), \(\text{usage must}\) also contain at least one of \(\text{VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT}\)

- **VUID-VkImageCreateInfo-samples-02258**
  \(\text{samples must}\) be a bit value that is set in \(\text{imageCreateSampleCounts}\) (as defined in \text{Image Creation Limits})

- **VUID-VkImageCreateInfo-usage-00968**
  If the \text{multisampled storage images} feature is not enabled, and \(\text{usage}\) contains \(\text{VK_IMAGE_USAGE_STORAGE_BIT}\), \(\text{samples must}\) be \(\text{VK_SAMPLE_COUNT_1_BIT}\)

- **VUID-VkImageCreateInfo-flags-05062**
  \(\text{flags must}\) not contain \(\text{VK_IMAGE_CREATE_SPARSE_BINDING_BIT, VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT, VK_IMAGE_CREATE_SPARSE_ALIASED_BIT, or VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT}\)

- **VUID-VkImageCreateInfo-flags-01890**
  If the \text{protected memory feature} is not enabled, \(\text{flags must}\) not contain
If the `pNext` chain includes a `VkExternalMemoryImageCreateInfo` structure, its `handleTypes` member must only contain bits that are also in `VkExternalImageFormatProperties::externalMemoryProperties.compatibleHandleTypes`, as returned by `vkGetPhysicalDeviceImageFormatProperties2` with `format`, `imageType`, `tiling`, `usage`, and `flags` equal to those in this structure, and with a `VkPhysicalDeviceExternalImageFormatInfo` structure included in the `pNext` chain, with a `handleType` equal to any one of the handle types specified in `VkExternalMemoryImageCreateInfo::handleTypes`.

If `flags` contains `VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT`, then `format` must be a compressed image format.

If `flags` contains `VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT`, then `flags` must also contain `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT`.

`initialLayout` must be `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`.

If the `pNext` chain includes a `VkExternalMemoryImageCreateInfo` or `VkExternalMemoryImageCreateInfoNV` structure whose `handleTypes` member is not 0, `initialLayout` must be `VK_IMAGE_LAYOUT_UNDEFINED`.

If the image `format` is one of the formats that require a sampler Y’C’bC’r conversion, `mipLevels` must be 1.

If the image `format` is one of the formats that require a sampler Y’C’bC’r conversion, `samples` must be `VK_SAMPLE_COUNT_1_BIT`.

If the image `format` is one of the formats that require a sampler Y’C’bC’r conversion, `imageType` must be `VK_IMAGE_TYPE_2D`.

If the image `format` is one of the formats that require a sampler Y’C’bC’r conversion, and the `ycbcrImageArrays` feature is not enabled, `arrayLayers` must be 1.

If `format` is a `multi-planar` format, and if `imageCreateFormatFeatures` (as defined in Image Creation Limits) does not contain `VK_FORMAT_FEATURE_DISJOINT_BIT`, then `flags` must not contain `VK_IMAGE_CREATE_DISJOINT_BIT`.

If `format` is not a `multi-planar` format, and `flags` does not include `VK_IMAGE_CREATE_ALIAS_BIT`, `flags` must not contain `VK_IMAGE_CREATE_DISJOINT_BIT`.

If `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-tiling-02261
  If tiling is VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT, then the pNext chain must include exactly one of VkImageDrmFormatModifierListCreateInfoEXT or VkImageDrmFormatModifierExplicitCreateInfoEXT structures

• VUID-VkImageCreateInfo-pNext-02262
  If the pNext chain includes a VkImageDrmFormatModifierListCreateInfoEXT or VkImageDrmFormatModifierExplicitCreateInfoEXT structure, then tiling must be VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT

• VUID-VkImageCreateInfo-tiling-02353
  If tiling is VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT and flags contains VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT, then the pNext chain must include a VkImageFormatListCreateInfo structure with non-zero viewFormatCount

• VUID-VkImageCreateInfo-flags-01533
  If flags contains VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT format must be a depth or depth/stencil format

• 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_INPUT_ATTACHMENT_BIT, extent.width must be less than or equal to VkPhysicalDeviceLimits::maxFramebufferWidth

• VUID-VkImageCreateInfo-format-02537
If `format` is a depth-stencil format and the `pNext` chain includes a `VkImageStencilUsageCreateInfo` structure with its `stencilUsage` member including `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`, `extent.height` must be less than or equal to `VkPhysicalDeviceLimits::maxFramebufferHeight`.

- **VUID-VkImageCreateInfo-format-02538**
  If the multisampled storage images feature is not enabled, `format` is a depth-stencil format and the `pNext` chain includes a `VkImageStencilUsageCreateInfo` structure with its `stencilUsage` including `VK_IMAGE_USAGE_STORAGE_BIT`, `samples` must be `VK_SAMPLE_COUNT_1_BIT`.

- **VUID-VkImageCreateInfo-imageType-02082**
  If `usage` includes `VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR`, `imageType` must be `VK_IMAGE_TYPE_2D`.

- **VUID-VkImageCreateInfo-samples-02083**
  If `usage` includes `VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR`, `samples` must be `VK_SAMPLE_COUNT_1_BIT`.

- **VUID-VkImageCreateInfo-pNext-04737**
  If a `VkImageFormatListCreateInfo` structure was included in the `pNext` chain and `VkImageFormatListCreateInfo::viewFormatCount` is not zero, then all of the formats in `VkImageFormatListCreateInfo::pViewFormats` must be compatible with the `format` as described in the compatibility table.

- **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`, `VkImageDrmFormatModifierExplicitCreateInfoEXT`, `VkImageDrmFormatModifierListCreateInfoEXT`, `VkImageFormatListCreateInfo`, `VkImageStencilUsageCreateInfo`, or `VkImageSwapchainCreateInfoKHR`.

- **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;
    const void* pNext;
    VkImageUsageFlags stencilUsage;
} VkImageStencilUsageCreateInfo;
```

• sType is the type of this structure.

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

• stencilUsage is a bitmask of VkImageUsageFlagBits describing the intended usage of the stencil aspect of the image.

If the pNext chain of VkImageCreateInfo includes a VkImageStencilUsageCreateInfo structure, then that structure includes the usage flags specific to the stencil aspect of the image for an image with a depth-stencil format.

This structure specifies image usages which only apply to the stencil aspect of a depth/stencil format image. When this structure is included in the pNext chain of VkImageCreateInfo, the stencil aspect of the image must only be used as specified by stencilUsage. When this structure is not included in the pNext chain of VkImageCreateInfo, the stencil aspect of an image must only be used as specified by VkImageCreateInfo::usage. Use of other aspects of an image are unaffected by this structure.

This structure can also be included in the pNext chain of VkPhysicalDeviceImageFormatInfo2 to query additional capabilities specific to image creation parameter combinations including a separate set of usage flags for the stencil aspect of the image using vkGetPhysicalDeviceImageFormatProperties2. When this structure is not included in the pNext chain of VkPhysicalDeviceImageFormatInfo2 then the implicit value of stencilUsage matches that of VkPhysicalDeviceImageFormatInfo2::usage.
Valid Usage

- VUID-VkImageStencilUsageCreateInfo-stencilUsage-02539
  If stencilUsage includes VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT, it must not include bits other than VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT

Valid Usage (Implicit)

- VUID-VkImageStencilUsageCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_IMAGE_STENCIL_USAGE_CREATE_INFO

- VUID-VkImageStencilUsageCreateInfo-stencilUsage-parameter
  stencilUsage must be a valid combination of VkImageUsageFlagBits values

- VUID-VkImageStencilUsageCreateInfo-stencilUsage-requiredbitmask
  stencilUsage must not be 0

To define a set of external memory handle types that may be used as backing store for an image, add a VkExternalMemoryImageCreateInfo structure to the pNext chain of the VkImageCreateInfo structure. The VkExternalMemoryImageCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkExternalMemoryImageCreateInfo {
    VkStructureType              sType;
    const void*                  pNext;
    VkExternalMemoryHandleTypeFlags handleTypes;
} VkExternalMemoryImageCreateInfo;
```

**Note**

A VkExternalMemoryImageCreateInfo structure with a non-zero handleTypes field must be included in the creation parameters for an image that will be bound to memory that is either exported or imported.

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **handleTypes** is zero, or a bitmask of VkExternalMemoryHandleTypeFlagBits specifying one or more external memory handle types.

Valid Usage (Implicit)

- VUID-VkExternalMemoryImageCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO

- VUID-VkExternalMemoryImageCreateInfo-handleTypes-parameter
**handleTypes** must be a valid combination of `VkExternalMemoryHandleTypeFlagBits` values.

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

The `VkImageSwapchainCreateInfoKHR` structure is defined as:

```c
// Provided by VK_VERSION_1_1 with VK_KHR_swapchain
typedef struct VkImageSwapchainCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkSwapchainKHR swapchain;
} VkImageSwapchainCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `swapchain` is `VK_NULL_HANDLE` or a handle of a swapchain that the image will be bound to.

**Valid Usage**

- VUID-VkImageSwapchainCreateInfoKHR-swapchain-00995
  If `swapchain` is not `VK_NULL_HANDLE`, the fields of `VkImageCreateInfo` must match the implied image creation parameters of the swapchain.

**Valid Usage (Implicit)**

- VUID-VkImageSwapchainCreateInfoKHR-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_IMAGE_SWAPCHAIN_CREATE_INFO_KHR`.

- VUID-VkImageSwapchainCreateInfoKHR-swapchain-parameter
  If `swapchain` is not `VK_NULL_HANDLE`, `swapchain` must be a valid `VkSwapchainKHR` handle.

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;
} VkImageFormatListCreateInfo;
```
const VkFormat* pViewFormats;
} VkImageFormatListCreateInfo;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **viewFormatCount** is the number of entries in the **pViewFormats** array.
- **pViewFormats** is a pointer to an array of **VkFormat** values specifying all formats which can be used when creating views of this image.

If **viewFormatCount** is zero, **pViewFormats** is ignored and the image is created as if the **VkImageFormatListCreateInfo** structure were not included in the **pNext** chain of **VkImageCreateInfo**.

### Valid Usage (Implicit)

- VUID-VkImageFormatListCreateInfo-sType-sType
  
  **sType** must be **VK_STRUCTURE_TYPE_IMAGE_FORMAT_LIST_CREATE_INFO**

- VUID-VkImageFormatListCreateInfo-pViewFormats-parameter
  
  If **viewFormatCount** is not 0, **pViewFormats** must be a valid pointer to an array of **viewFormatCount** valid **VkFormat** values.

If the **pNext** chain of **VkImageCreateInfo** includes a **VkImageDrmFormatModifierListCreateInfoEXT** structure, then the image will be created with one of the Linux DRM format modifiers listed in the structure. The choice of modifier is implementation-dependent.

The **VkImageDrmFormatModifierListCreateInfoEXT** structure is defined as:

```c
// Provided by VK_EXT_image_drm_format_modifier
typedef struct VkImageDrmFormatModifierListCreateInfoEXT {
  VkStructureType sType;
  const void* pNext;
  uint32_t drmFormatModifierCount;
  const uint64_t* pDrmFormatModifiers;
} VkImageDrmFormatModifierListCreateInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **drmFormatModifierCount** is the length of the **pDrmFormatModifiers** array.
- **pDrmFormatModifiers** is a pointer to an array of Linux DRM format modifiers.

### Valid Usage

- VUID-VkImageDrmFormatModifierListCreateInfoEXT-pDrmFormatModifiers-02263
  
  Each modifier in **pDrmFormatModifiers** must be compatible with the parameters in
VkImageCreateInfo and its pNext chain, as determined by querying VkPhysicalDeviceImageFormatInfo2 extended with VkPhysicalDeviceImageDrmFormatModifierInfoEXT

### Valid Usage (Implicit)

- **VUID-VkImageDrmFormatModifierListCreateInfoEXT-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_LIST_CREATE_INFO_EXT`

- **VUID-VkImageDrmFormatModifierListCreateInfoEXT-pDrmFormatModifiers-parameter**
  - `pDrmFormatModifiers` must be a valid pointer to an array of `drmFormatModifierCount` uint64_t values

- **VUID-VkImageDrmFormatModifierListCreateInfoEXT-drmFormatModifierCount-arraylength**
  - `drmFormatModifierCount` must be greater than 0

If the pNext chain of VkImageCreateInfo includes a VkImageDrmFormatModifierExplicitCreateInfoEXT structure, then the image will be created with the Linux DRM format modifier and memory layout defined by the structure.

The VkImageDrmFormatModifierExplicitCreateInfoEXT structure is defined as:

```c
// Provided by VK_EXT_image_drm_format_modifier
typedef struct VkImageDrmFormatModifierExplicitCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    uint64_t drmFormatModifier;
    uint32_t drmFormatModifierPlaneCount;
    const VkSubresourceLayout* pPlaneLayouts;
} VkImageDrmFormatModifierExplicitCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `drmFormatModifier` is the Linux DRM format modifier with which the image will be created.
- `drmFormatModifierPlaneCount` is the number of memory planes in the image (as reported by VkDrmFormatModifierPropertiesEXT) as well as the length of the `pPlaneLayouts` array.
- `pPlaneLayouts` is a pointer to an array of VkSubresourceLayout structures describing the image’s memory planes.

The \( i \)th member of `pPlaneLayouts` describes the layout of the image’s \( i \)th memory plane (that is, VK_IMAGE_ASPECT_MEMORY_PLANE_\( i \)_BIT_EXT). In each element of `pPlaneLayouts`, the implementation must ignore size. The implementation calculates the size of each plane, which the application can query with vkGetImageSubresourceLayout.

When creating an image with VkImageDrmFormatModifierExplicitCreateInfoEXT, it is the
application’s responsibility to satisfy all valid usage requirements. However, the implementation must validate that the provided pPlaneLayouts, when combined with the provided drmFormatModifier and other creation parameters in VkImageCreateInfo and its pNext chain, produce a valid image. (This validation is necessarily implementation-dependent and outside the scope of Vulkan, and therefore not described by valid usage requirements). If this validation fails, then vkCreateImage returns VK_ERROR_INVALID_DRM_FORMAT_MODIFIER_PLANE_LAYOUT_EXT.

Valid Usage

- VUID-VkImageDrmFormatModifierExplicitCreateInfoEXT-drmFormatModifier-02264
  drmFormatModifier must be compatible with the parameters in VkImageCreateInfo and its pNext chain, as determined by querying VkPhysicalDeviceImageDrmFormatModifierInfoEXT

- VUID-VkImageDrmFormatModifierExplicitCreateInfoEXT-drmFormatModifierPlaneCount-02265
  drmFormatModifierPlaneCount must be equal to the VkDrmFormatModifierPropertiesEXT::drmFormatModifierPlaneCount associated with VkImageCreateInfo::format and drmFormatModifier, as found by querying VkDrmFormatModifierPropertiesListEXT

- VUID-VkImageDrmFormatModifierExplicitCreateInfoEXT-size-02267
  For each element of pPlaneLayouts, size must be 0

- VUID-VkImageDrmFormatModifierExplicitCreateInfoEXT-arrayPitch-02268
  For each element of pPlaneLayouts, arrayPitch must be 0 if VkImageCreateInfo::arrayLayers is 1

- VUID-VkImageDrmFormatModifierExplicitCreateInfoEXT-depthPitch-02269
  For each element of pPlaneLayouts, depthPitch must be 0 if VkImageCreateInfo::extent.depth is 1

Valid Usage (Implicit)

- VUID-VkImageDrmFormatModifierExplicitCreateInfoEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_EXPLICIT_CREATE_INFO_EXT

- VUID-VkImageDrmFormatModifierExplicitCreateInfoEXT-pPlaneLayouts-parameter
  If drmFormatModifierPlaneCount is not 0, pPlaneLayouts must be a valid pointer to an array of drmFormatModifierPlaneCount VkSubresourceLayout structures

Bits which can be set in VkImageViewUsageCreateInfo::usage, or VkImageStencilUsageCreateInfo::stencilUsage, or VkImageCreateInfo::usage, specifying intended usage of an image, 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,
// Provided by VK_KHR_fragment_shading_rate
VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR = 0x00000100,
}

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.
- VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR specifies that the image can be used to create a VkImageView suitable for use as a fragment shading rate attachment.

typedef VkFlags VkImageUsageFlags;
• **VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT**
• **VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT**
• **VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT**
• **VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR**

Bits which **can** be set in `VkImageCreateInfo::flags`, specifying additional parameters of an image, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkImageCreateFlagBits {
    VK_IMAGE_CREATE_SPARSE_BINDING_BIT = 0x00000001,
    VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT = 0x00000002,
    VK_IMAGE_CREATE_SPARSE_ALIASED_BIT = 0x00000004,
    VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT = 0x00000008,
    VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT = 0x00000010,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_ALIAS_BIT = 0x00000400,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT = 0x00000040,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT = 0x00000020,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT = 0x00000080,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_EXTENDED_USAGE_BIT = 0x00000100,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_PROTECTED_BIT = 0x00000800,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_DISJOINT_BIT = 0x00000200,
    // Provided by VK_EXT_sample_locations
    VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT = 0x00001000,
} VkImageCreateFlagBits;
```

• **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.

VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT specifies that an image with a depth or depth/stencil format can be used with custom sample locations when used as a depth/stencil attachment.

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:
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,
    // Provided by VK_EXT_image_drm_format_modifier
    VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT = 1000158000,
} 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).
- **VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT** indicates that the image’s tiling is defined by a Linux DRM format modifier. The modifier is specified at image creation with `VkImageDrmFormatModifierListCreateInfoEXT` or `VkImageDrmFormatModifierExplicitCreateInfoEXT`, and can be queried with `vkGetImageDrmFormatModifierPropertiesEXT`.

To query the memory layout of an image subresource, call:

```c
// Provided by VK_VERSION_1_0
void vkGetImageSubresourceLayout(
    VkDevice device,                  // device, 
    VkImage image,                   // image, 
    const VkImageSubresource* pSubresource,    
    VkSubresourceLayout* pLayout);  // 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.

If the image is linear, then the returned layout is valid for host access.

If the image's tiling is `VK_IMAGE_TILING_LINEAR` and its format is a multi-planar format, then `vkGetImageSubresourceLayout` describes one format plane of the image. If the image's tiling is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then `vkGetImageSubresourceLayout` describes one memory plane of the image. If the image's tiling is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT` and the image is non-linear, then the returned layout has an implementation-dependent meaning; the vendor of the image's DRM format modifier may provide documentation that explains how to interpret the returned layout.

`vkGetImageSubresourceLayout` is invariant for the lifetime of a single image.

### Valid Usage

- **VUID-vkGetImageSubresourceLayout-image-02270**
  - `image` must have been created with `tiling` equal to `VK_IMAGE_TILING_LINEAR` or `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`

- **VUID-vkGetImageSubresourceLayout-aspectMask-00997**
  - The `aspectMask` member of `pSubresource` must only have a single bit set

- **VUID-vkGetImageSubresourceLayout-mipLevel-01716**
  - The `mipLevel` member of `pSubresource` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `image` was created

- **VUID-vkGetImageSubresourceLayout-arrayLayer-01717**
  - The `arrayLayer` member of `pSubresource` must be less than the `arrayLayers` specified in `VkImageCreateInfo` when `image` was created

- **VUID-vkGetImageSubresourceLayout-format-04461**
  - If `format` is a color format, the `aspectMask` member of `pSubresource` must be `VK_IMAGE_ASPECT_COLOR_BIT`

- **VUID-vkGetImageSubresourceLayout-format-04462**
  - If `format` has a depth component, the `aspectMask` member of `pSubresource` must contain `VK_IMAGE_ASPECT_DEPTH_BIT`

- **VUID-vkGetImageSubresourceLayout-format-04463**
  - If `format` has a stencil component, the `aspectMask` member of `pSubresource` must contain `VK_IMAGE_ASPECT_STENCIL_BIT`

- **VUID-vkGetImageSubresourceLayout-format-04464**
  - If `format` does not contain a stencil or depth component, the `aspectMask` member of `pSubresource` must not contain `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT`

- **VUID-vkGetImageSubresourceLayout-format-01581**
  - If the tiling of the `image` is `VK_IMAGE_TILING_LINEAR` and its `format` is a multi-planar format with two planes, the `aspectMask` member of `pSubresource` must be `VK_IMAGE_ASPECT_PLANE_0_BIT` or `VK_IMAGE_ASPECT_PLANE_1_BIT`

- **VUID-vkGetImageSubresourceLayout-format-01582**
  - If the tiling of the `image` is `VK_IMAGE_TILING_LINEAR` and its `format` is a multi-planar format
with three planes, the `aspectMask` member of `pSubresource` must be
`VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT` or `VK_IMAGE_ASPECT_PLANE_2_BIT`

- VUID-vkGetImageSubresourceLayout-tiling-02271
  If the tiling of the image is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then the `aspectMask`
  member of `pSubresource` must be `VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT` and the index `i`
  must be less than the `VkDrmFormatModifierPropertiesEXT::drmFormatModifierPlaneCount`
  associated with the image’s `format` and `VkImageDrmFormatModifierPropertiesEXT::drmFormatModifier`

**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 \texttt{VkSubresourceLayout} structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSubresourceLayout {
    VkDeviceSize offset;
    VkDeviceSize size;
    VkDeviceSize rowPitch;
    VkDeviceSize arrayPitch;
    VkDeviceSize depthPitch;
} VkSubresourceLayout;
```

- \texttt{offset} is the byte offset from the start of the image or the plane where the image subresource begins.
- \texttt{size} is the size in bytes of the image subresource. \texttt{size} includes any extra memory that is required based on \texttt{rowPitch}.
- \texttt{rowPitch} describes the number of bytes between each row of texels in an image.
- \texttt{arrayPitch} describes the number of bytes between each array layer of an image.
- \texttt{depthPitch} describes the number of bytes between each slice of 3D image.

If the image is \texttt{linear}, then \texttt{rowPitch}, \texttt{arrayPitch} and \texttt{depthPitch} describe the layout of the image subresource in linear memory. For uncompressed formats, \texttt{rowPitch} is the number of bytes between texels with the same \texttt{x} coordinate in adjacent rows (\texttt{y} coordinates differ by one). \texttt{arrayPitch} is the number of bytes between texels with the same \texttt{x} and \texttt{y} coordinate in adjacent array layers of the image (array layer values differ by one). \texttt{depthPitch} is the number of bytes between texels with the same \texttt{x} and \texttt{y} coordinate in adjacent slices of a 3D image (\texttt{z} coordinates differ by one). Expressed as an addressing formula, the starting byte of a texel in the image subresource has address:

```c
// (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 \texttt{rowPitch} is the number of bytes between compressed texel blocks in adjacent rows. \texttt{arrayPitch} is the number of bytes between compressed texel blocks in adjacent array layers. \texttt{depthPitch} is the number of bytes between compressed texel blocks in adjacent slices of a 3D image.

```c
// (x,y,z,layer) are in compressed texel block coordinates
address(x,y,z,layer) = layer*arrayPitch + z*depthPitch + y*rowPitch + x *compressedTexelBlockSize + offset;
```

The value of \texttt{arrayPitch} is undefined for images that were not created as arrays. \texttt{depthPitch} is defined only for 3D images.

If the image has a \texttt{single-plane} color format and its tiling is \texttt{VK_IMAGE_TILING_LINEAR}, then the
aspectMask member of VkImageSubresource must be VK_IMAGE_ASPECT_COLOR_BIT.

If the image has a depth/stencil format and its tiling is VK_IMAGE_TILING_LINEAR, then aspectMask must be either VK_IMAGE_ASPECT_DEPTH_BIT or VK_IMAGE_ASPECT_STENCIL_BIT. On implementations that store depth and stencil aspects separately, querying each of these image subresource layouts will return a different offset and size representing the region of memory used for that aspect. On implementations that store depth and stencil aspects interleaved, the same offset and size are returned and represent the interleaved memory allocation.

If the image has a multi-planar format and its tiling is VK_IMAGE_TILING_LINEAR, then the aspectMask member of VkImageSubresource must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or (for 3-plane formats only) VK_IMAGE_ASPECT_PLANE_2_BIT. Querying each of these image subresource layouts will return a different offset and size representing the region of memory used for that plane. If the image is disjoint, then the offset is relative to the base address of the plane. If the image is non-disjoint, then the offset is relative to the base address of the image.

If the image’s tiling is VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT, then the aspectMask member of VkImageSubresource must be one of VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT, where the maximum allowed plane index \( i \) is defined by the VkDrmFormatModifierPropertiesEXT::drmFormatModifierPlaneCount associated with the image’s VkImageCreateInfo::format and modifier. The memory range used by the subresource is described by offset and size. If the image is disjoint, then the offset is relative to the base address of the memory plane. If the image is non-disjoint, then the offset is relative to the base address of the image. If the image is non-linear, then rowPitch, arrayPitch, and depthPitch have an implementation-dependent meaning.

If an image was created with VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT, then the image has a Linux DRM format modifier. To query the modifier, call:

```cpp
// Provided by VK_EXT_image_drm_format_modifier
VkResult vkGetImageDrmFormatModifierPropertiesEXT(
    VkDevice device, 
    VkImage image, 
    VkImageDrmFormatModifierPropertiesEXT* pProperties);
```

- **device** is the logical device that owns the image.
- **image** is the queried image.
- **pProperties** is a pointer to a VkImageDrmFormatModifierPropertiesEXT structure in which properties of the image’s DRM format modifier are returned.

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

**Valid Usage**

- VUID-vkGetImageDrmFormatModifierPropertiesEXT-image-02272
  image must have been created with tiling equal to VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT
Valid Usage (Implicit)

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

- VUID-vkGetImageDrmFormatModifierPropertiesEXT-image-parameter
image must be a valid VkImage handle

- VUID-vkGetImageDrmFormatModifierPropertiesEXT-pProperties-parameter
pProperties must be a valid pointer to a VkImageDrmFormatModifierPropertiesEXT structure

- VUID-vkGetImageDrmFormatModifierPropertiesEXT-image-parent
image must have been created, allocated, or retrieved from device

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY

The VkImageDrmFormatModifierPropertiesEXT structure is defined as:

```c
// Provided by VK_EXT_image_drm_format_modifier
typedef struct VkImageDrmFormatModifierPropertiesEXT {
    VkStructureType         sType;
    void*                   pNext;
    uint64_t                drmFormatModifier;
} VkImageDrmFormatModifierPropertiesEXT;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- drmFormatModifier returns the image's Linux DRM format modifier.

If the image was created with VkImageDrmFormatModifierListCreateInfoEXT, then the returned drmFormatModifier must belong to the list of modifiers provided at time of image creation in VkImageDrmFormatModifierListCreateInfoEXT::pDrmFormatModifiers. If the image was created with VkImageDrmFormatModifierExplicitCreateInfoEXT, then the returned drmFormatModifier must be the modifier provided at time of image creation in VkImageDrmFormatModifierExplicitCreateInfoEXT::drmFormatModifier.

Valid Usage (Implicit)

- VUID-VkImageDrmFormatModifierPropertiesEXT-sType-sType
To destroy an image, call:

```c
// Provided by VK_VERSION_1_0
define vkDestroyImage(
    VkDevice device, 
    VkImage image, 
    const VkAllocationCallbacks* pAllocator);
```

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

### Valid Usage

- **VUID-vkDestroyImage-image-01000**
  All submitted commands that refer to image, either directly or via a VkImageView, must have completed execution

- **VUID-vkDestroyImage-image-04882**
  image must not have been acquired from vkGetSwapchainImagesKHR

### 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.
- If the image was created with VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT, then:
  - The image’s DRM format modifier is the value of VkImageDrmFormatModifierListCreateInfoEXT::drmFormatModifier found by calling vkGetImageDrmFormatModifierPropertiesEXT.
  - Let VkDrmFormatModifierPropertiesListEXT::pDrmFormatModifierProperties be the array found by calling vkGetPhysicalDeviceFormatProperties2 on the same format as VkImageCreateInfo::format.
  - Let VkDrmFormatModifierPropertiesEXT prop be an array element whose drmFormatModifier member is the value of the image’s DRM format modifier.
  - Then the image set of format features is the value of taking the bitwise intersection over the collected prop::drmFormatModifierTilingFeatures.

12.3.2. Image Miplevel Sizing

A complete mipmap chain is the full set of mipmap levels, from the largest mipmap provided, down to the minimum mipmap size.

Conventional Images

For conventional images, the dimensions of each successive mipmap level, n+1, are:

\[
\begin{align*}
width_{n+1} &= \max(\lceil width_n / 2 \rceil, 1) \\
height_{n+1} &= \max(\lceil height_n / 2 \rceil, 1) \\
depth_{n+1} &= \max(\lceil depth_n / 2 \rceil, 1)
\end{align*}
\]

where \( width_n \), \( height_n \), and \( depth_n \) are the dimensions of the next larger mipmap level, n.

The minimum mipmap size is:

- 1 for one-dimensional images,
- 1x1 for two-dimensional images, and
• 1x1x1 for three-dimensional images.

The number of levels in a complete mipmap chain is:

$$\lceil \log_2(\max(\text{width}_0, \text{height}_0, \text{depth}_0)) \rceil + 1$$

where width$_0$, height$_0$, and depth$_0$ are the dimensions of the largest (most detailed) mipmap 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 vkCmdPipelineBarrier or a vkCmdWaitEvents command buffer command (see Image Memory Barriers), or as part of a subpass dependency within a render pass (see VkSubpassDependency).

Image layout is per-image subresource. Separate image subresources of the same image can be in different layouts at the same time, with the exception that depth and stencil aspects of a given image subresource can only be in different layouts if the separateDepthStencilLayouts feature is enabled.

#### Note

Each layout may offer optimal performance for a specific usage of image memory. For example, an image with a layout of VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL may provide optimal performance for use as a color attachment, but be unsupported for use in transfer commands. Applications can transition an image subresource from one layout to another in order to achieve optimal performance when the image subresource is used for multiple kinds of operations. After initialization, applications need not use any layout other than the general layout, though this may produce suboptimal performance on some implementations.

Upon creation, all image subresources of an image are initially in the same layout, where that layout is selected by the VkImageCreateInfo::initialLayout member. The initialLayout must be either VK_IMAGE_LAYOUT_UNDEFINED or VK_IMAGE_LAYOUT_PREINITIALIZED. If it is VK_IMAGE_LAYOUT_PREINITIALIZED, then the image data can be preinitialized by the host while using this layout, and the transition away from this layout will preserve that data. If it is VK_IMAGE_LAYOUT_UNDEFINED, then the contents of the data are considered to be undefined, and the transition away from this layout is not guaranteed to preserve that data. For either of these initial layouts, any image subresources must be transitioned to another layout before they are accessed by the device.

Host access to image memory is only well-defined for linear images and for image subresources of those images which are currently in either the VK_IMAGE_LAYOUT_PREINITIALIZED or
VK_IMAGE_LAYOUT_GENERAL layout. Calling vkGetImageSubresourceLayout for a linear image returns a subresource layout mapping that is valid for either of those image layouts.

The set of image layouts consists of:

```c
// Provided by VK_VERSION_1_0
typedef enum VkImageLayout {
    VK_IMAGE_LAYOUT_UNDEFINED = 0,
    VK_IMAGE_LAYOUT_GENERAL = 1,
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL = 2,
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL = 3,
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL = 4,
    VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL = 5,
    VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL = 6,
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL = 7,
    VK_IMAGE_LAYOUT_PREINITIALIZED = 8,
// Provided by VK_VERSION_1_1
    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,
// Provided by VK_KHR_swapchain
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR = 1000001002,
// Provided by VK_KHR_shared_presentable_image
    VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR = 1000111000,
// Provided by VK_KHR_fragment_shading_rate
    VK_IMAGE_LAYOUT_FRAGMENT_SHADING_RATE_ATTACHMENT_OPTIMAL_KHR = 1000164003,
// Provided by VK_KHR_synchronization2
    VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR = 1000314000,
// Provided by VK_KHR_synchronization2
    VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL_KHR = 1000314001,
} 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_ATTACHMENT_OPTIMAL_KHR** specifies a layout that must only be used with attachment accesses in the graphics pipeline.
- **VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR** specifies a layout allowing read only access as an attachment, or in shaders as a sampled image, combined image/sampler, or input attachment.
- **VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL_KHR** specifies a layout that must only be used with attachment accesses in the graphics pipeline.
- **VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR** specifies a layout allowing read only access as an attachment, or in shaders as a sampled image, combined image/sampler, or input attachment.
- **VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL** specifies a layout for the depth aspect of a depth/stencil format image allowing read and write access as a depth/stencil attachment.
- **VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL** specifies a layout for the depth and stencil aspects of a depth/stencil format image 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_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL**.
- **VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL** specifies a layout for the depth and stencil aspects of a depth/stencil format image allowing read-only 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_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL**.
- **VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL** specifies a layout allowing read-only access in a shader.

---

568
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.

- **VK_IMAGE_LAYOUT_PRESENT_SRC_KHR** must only be used for presenting a presentable image for display. A swapchain’s image must be transitioned to this layout before calling vkQueuePresentKHR, and must be transitioned away from this layout after calling vkAcquireNextImageKHR.

- **VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR** is valid only for shared presentable images, and must be used for any usage the image supports.

- **VK_IMAGE_LAYOUT_FRAGMENT_SHADING_RATE_ATTACHMENT_OPTIMAL_KHR** must only be used as a fragment shading rate attachment or This layout is valid only for image subresources of images created with the VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR 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...
The new layout used in a transition must not be VK_IMAGE_LAYOUT_UNDEFINED or VK_IMAGE_LAYOUT_PREINITIALIZED.

The image layout of each image subresource of a depth/stencil image created with VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT is dependent on the last sample locations used to render to the image subresource as a depth/stencil attachment, thus applications must provide the same sample locations that were last used to render to the given image subresource whenever a layout transition of the image subresource happens, otherwise the contents of the depth aspect of the image subresource become undefined.

In addition, depth reads from a depth/stencil attachment referring to an image subresource range of a depth/stencil image created with VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT using different sample locations than what have been last used to perform depth writes to the image subresources of the same image subresource range return undefined values.

Similarly, depth writes to a depth/stencil attachment referring to an image subresource range of a depth/stencil image created with VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT using different sample locations than what have been last used to perform depth writes to the image subresources of the same image subresource range make the contents of the depth aspect of those image subresources undefined.

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:
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-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 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 the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- flags is a bitmask of VkImageViewCreateFlagBits describing additional parameters of the image view.
- image is a VkImage on which the view will be created.
- viewType is a VkImageViewType value specifying the type of the image view.
- format is a VkFormat describing the format and type used to interpret texel blocks in the image.
- components is a VkComponentMapping 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.

**Note**
Values intended to be used with one view format may not be exactly preserved when written or read through a different format. For example, an integer value that happens to have the bit pattern of a floating point denorm or NaN may be flushed or canonicalized when written or read through a view with a floating point format. Similarly, a value written through a signed normalized format that has a bit pattern exactly equal to \(-2^{b}\) may be changed to \(-2^{b} + 1\) as described in Conversion from Normalized Fixed-Point to Floating-Point.

If `image` was created with the `VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT` flag, format must be compatible with the image's format as described above, or must be an uncompressed format in which case it must be size-compatible with the image's format, as defined for copying data between images. In this case, the resulting image view's texel dimensions equal the dimensions of the selected mip level divided by the compressed texel block size and rounded up.

The `VkComponentMapping components` member describes a remapping from components of the image to components of the vector returned by shader image instructions. This remapping must be the identity swizzle for storage image descriptors, input attachment descriptors, framebuffer attachments, and any `VkImageView` used with a combined image sampler that enables sampler Y'CbCr conversion.
If the image view is to be used with a sampler which supports \( Y'C_bC_r \) conversion, an identically defined object of type \( \text{VkSamplerYcbcrConversion} \) to that used to create the sampler must be passed to \( \text{vkCreateImageView} \) in a \( \text{VkSamplerYcbcrConversionInfo} \) included in the \( p\text{Next} \) chain of \( \text{VkImageViewCreateInfo} \). Conversely, if a \( \text{VkSamplerYcbcrConversion} \) object is passed to \( \text{vkCreateImageView} \), an identically defined \( \text{VkSamplerYcbcrConversion} \) object must be used when sampling the image.

If the image has a multi-planar format and \( \text{subresourceRange.aspectMask} \) is \( \text{VK_IMAGE_ASPECT_COLOR_BIT} \), then the format must be identical to the image format, and the sampler to be used with the image view must enable \( Y'C_bC_r \) conversion.

If image was created with the \( \text{VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT} \) and the image has a multi-planar format, and if \( \text{subresourceRange.aspectMask} \) is \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT}, \text{VK_IMAGE_ASPECT_PLANE_1_BIT}, \text{or VK_IMAGE_ASPECT_PLANE_2_BIT} \), format must be compatible with the corresponding plane of the image, and the sampler to be used with the image view must not enable \( Y'C_bC_r \) conversion. The width and height of the single-plane image view must be derived from the multi-planar image’s dimensions in the manner listed for plane compatibility for the plane.

Any view of an image plane will have the same mapping between texel coordinates and memory locations as used by the components of the color aspect, subject to the formulae relating texel coordinates to lower-resolution planes as described in Chroma Reconstruction. That is, if an R or B plane has a reduced resolution relative to the G plane of the multi-planar image, the image view operates using the \((u_{\text{plane}}, v_{\text{plane}})\) unnormalized coordinates of the reduced-resolution plane, and these coordinates access the same memory locations as the \((u_{\text{color}}, v_{\text{color}})\) unnormalized coordinates of the color aspect for which chroma reconstruction operations operate on the same \((u_{\text{plane}}, v_{\text{plane}})\) or \((i_{\text{plane}}, j_{\text{plane}})\) coordinates.

Table 12. Image type and image view type compatibility requirements

<table>
<thead>
<tr>
<th>Image View Type</th>
<th>Compatible Image Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_IMAGE_VIEW_TYPE_1D</td>
<td>VK_IMAGE_TYPE_1D</td>
</tr>
<tr>
<td>VK_IMAGE_VIEW_TYPE_1D_ARRAY</td>
<td>VK_IMAGE_TYPE_1D</td>
</tr>
<tr>
<td>VK_IMAGE_VIEW_TYPE_2D</td>
<td>VK_IMAGE_TYPE_2D, VK_IMAGE_TYPE_3D</td>
</tr>
<tr>
<td>VK_IMAGE_VIEW_TYPE_2D_ARRAY</td>
<td>VK_IMAGE_TYPE_2D, VK_IMAGE_TYPE_3D</td>
</tr>
<tr>
<td>VK_IMAGE_VIEW_TYPE_CUBE</td>
<td>VK_IMAGE_TYPE_2D</td>
</tr>
<tr>
<td>VK_IMAGE_VIEW_TYPE_CUBE_ARRAY</td>
<td>VK_IMAGE_TYPE_2D</td>
</tr>
<tr>
<td>VK_IMAGE_VIEW_TYPE_3D</td>
<td>VK_IMAGE_TYPE_3D</td>
</tr>
</tbody>
</table>

Valid Usage

- **VUID-VkImageViewCreateInfo-image-01003**
  If image was not created with \( \text{VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT} \) then \( \text{viewType} \) must not be \( \text{VK_IMAGE_VIEW_TYPE_CUBE} \) or \( \text{VK_IMAGE_VIEW_TYPE_CUBE_ARRAY} \)

- **VUID-VkImageViewCreateInfo-viewType-01004**
  If the image cube map arrays feature is not enabled, \( \text{viewType} \) must not be
• VUID-VkImageViewCreateInfo-image-01005
If image was created with VK_IMAGE_TYPE_3D but without VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT set then viewType must not be VK_IMAGE_VIEW_TYPE_2D or VK_IMAGE_VIEW_TYPE_2D_ARRAY

• 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-02652
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 Miplevel Sizing.

- VUID-VkImageViewCreateInfo-subresourceRange-02725
  If subresourceRange.layerCount is not VK_REMAINING_ARRAY_LAYERS, image is a 3D image created with **VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT** set, and viewType is **VK_IMAGE_VIEW_TYPE_2D** or **VK_IMAGE_VIEW_TYPE_2D_ARRAY**, subresourceRange.layerCount must be non-zero and subresourceRange.baseArrayLayer + subresourceRange.layerCount must be less than or equal to the depth computed from baseMipLevel and extent.depth specified in **VkImageCreateInfo** when image was created, according to the formula defined in Image Miplevel Sizing.

- VUID-VkImageViewCreateInfo-image-01761
  If image was created with the **VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT** flag, but without the **VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT** flag, and if the format of the image is not a multi-planar format, format must be compatible with the format used to create image, as defined in Format Compatibility Classes.

- VUID-VkImageViewCreateInfo-image-01583
  If image was created with the **VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT** flag, format must be compatible with, or must be an uncompressed format that is size-compatible with, the format used to create image.

- VUID-VkImageViewCreateInfo-image-01584
  If image was created with the **VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT** flag, the levelCount and layerCount members of subresourceRange must both be 1.

- VUID-VkImageViewCreateInfo-image-04739
  If image was created with the **VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT** flag and format is a non-compressed format, viewType must not be **VK_IMAGE_VIEW_TYPE_3D**.

- VUID-VkImageViewCreateInfo-pNext-01585
  If a **VkImageFormatListCreateInfo** structure was included in the pNext chain of the **VkImageCreateInfo** structure used when creating image and ** VkImageFormatListCreateInfo::viewFormatCount** is not zero then format must be one of the formats in **VkImageFormatListCreateInfo::pViewFormats**.

- VUID-VkImageViewCreateInfo-image-01586
  If image was created with the **VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT** flag, if the format of the
image is a multi-planar format, and if subresourceRange.aspectMask is one of VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or VK_IMAGE_ASPECT_PLANE_2_BIT, then format must be compatible with the VkFormat for the plane of the image format indicated by subresourceRange.aspectMask, as defined in Compatible formats of planes of multi-planar formats

• VUID-VkImageViewCreateInfo-image-01762
  If image was not created with the VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT flag, or if the format of the image is a multi-planar format and if subresourceRange.aspectMask is VK_IMAGE_ASPECT_COLOR_BIT, format must be identical to the format used to create image

• VUID-VkImageViewCreateInfo-format-06415
  If the image format is one of the formats that require a sampler Y’CnCnR conversion, then the pNext chain must include a VkSamplerYcbcrConversion structure with a conversion value other than VK_NULL_HANDLE

• VUID-VkImageViewCreateInfo-format-04714
  If format has a _422 or _420 suffix then image must have been created with a width that is a multiple of 2

• VUID-VkImageViewCreateInfo-format-04715
  If format has a _420 suffix then image must have been created with a height that is a multiple of 2

• VUID-VkImageViewCreateInfo-pNext-01970
  If the pNext chain includes a VkSamplerYcbcrConversion structure with a conversion value other than VK_NULL_HANDLE, all members of components must have the identity swizzle

• 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-image-02086
  If image was created with usage containing VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR, viewType must be VK_IMAGE_VIEW_TYPE_2D or VK_IMAGE_VIEW_TYPE_2D_ARRAY

• VUID-VkImageViewCreateInfo-usage-04550
  If the attachmentFragmentShadingRate feature is enabled, and the usage for the image view includes VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR, then the image view’s format features must contain VK_FORMAT_FEATURE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR

• VUID-VkImageViewCreateInfo-usage-04551
  If the attachmentFragmentShadingRate feature is enabled, the usage for the image view includes VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR, and layeredShadingRateAttachments is VK_FALSE, subresourceRange.layerCount must be 1

• 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**
  `subresourceRange.levelCount` must be less than or equal to `VkDeviceObjectReservationCreateInfo::maxImageViewMipLevels`.

- **VUID-VkImageViewCreateInfo-subresourceRange-05065**
  `subresourceRange.layerCount` must be less than or equal to `VkDeviceObjectReservationCreateInfo::maxImageViewArrayLayers`.

---

578
If `subresourceRange.layerCount` is greater than 1, `subresourceRange.levelCount` 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 `VkImageViewASTCDecodeModeEXT`, `VkImageViewUsageCreateInfo`, or `VkSamplerYcbcrConversionInfo`.

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

- **VUID-VkImageViewCreateInfo-flags-zerobitmask**
  - `flags` must be 0.

- **VUID-VkImageViewCreateInfo-image-parameter**
  - `image` must be a valid `VkImage` handle.

- **VUID-VkImageViewCreateInfo-viewType-parameter**
  - `viewType` must be a valid `VkImageViewType` value.

- **VUID-VkImageViewCreateInfo-format-parameter**
  - `format` must be a valid `VkFormat` value.

- **VUID-VkImageViewCreateInfo-components-parameter**
  - `components` must be a valid `VkComponentMapping` structure.

- **VUID-VkImageViewCreateInfo-subresourceRange-parameter**
  - `subresourceRange` must be a valid `VkImageSubresourceRange` structure.

Bits which can be set in `VkImageViewCreateInfo::flags`, specifying additional parameters of an image view, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkImageViewCreateFlagBits {
} VkImageViewCreateFlagBits;
```

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

`VkImageViewCreateFlags` is a bitmask type for setting a mask of zero or more `VkImageViewCreateFlagBits`.

The set of usages for the created image view can be restricted compared to the parent image's usage.
flags by adding a `VkImageViewUsageCreateInfo` structure to the `pNext` chain of `VkImageViewCreateInfo`.

The `VkImageViewUsageCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkImageViewUsageCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkImageUsageFlags usage;
} VkImageViewUsageCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `usage` is a bitmask of `VkImageUsageFlagBits` specifying allowed usages of the image view.

When this structure is chained to `VkImageViewCreateInfo` the `usage` field overrides the implicit `usage` parameter inherited from image creation time and its value is used instead for the purposes of determining the valid usage conditions of `VkImageViewCreateInfo`.

**Valid Usage (Implicit)**

- `VUID-VkImageViewUsageCreateInfo-sType-sType` `sType` must be `VK_STRUCTURE_TYPE_IMAGE_VIEW_USAGE_CREATE_INFO`
- `VUID-VkImageViewUsageCreateInfo-usage-parameter` `usage` must be a valid combination of `VkImageUsageFlagBits` values
- `VUID-VkImageViewUsageCreateInfo-usage-requiredmask` `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 mipmap 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 mipmap 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’C₆C₇R 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’C₆C₇R 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`.

- VUID-VkImageSubresourceRange-aspectMask-02278
  `aspectMask` must not include `VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT` for any index `i`.

### Valid Usage (Implicit)

- VUID-VkImageSubresourceRange-aspectMask-parameter
  `aspectMask` must be a valid combination of `VkImageAspectFlagBits` values.

- VUID-VkImageSubresourceRange-aspectMask-requiredbitsetmask
  `aspectMask` must not be 0.

Bits which can be set in an aspect mask to specify aspects of an image for purposes such as identifying a subresource, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkImageAspectFlagBits {
    VK_IMAGE_ASPECT_COLOR_BIT = 0x00000001,
    VK_IMAGE_ASPECT_DEPTH_BIT = 0x00000002,
    VK_IMAGE_ASPECT_STENCIL_BIT = 0x00000004,
    VK_IMAGE_ASPECT_METADATA_BIT = 0x00000008,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_ASPECT_PLANE_0_BIT = 0x00000010,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_ASPECT_PLANE_1_BIT = 0x00000020,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_ASPECT_PLANE_2_BIT = 0x00000040,
    // Provided by VK_EXT_image_drm_format_modifier
    VK_IMAGE_ASPECT_MEMORY_PLANE_0_BIT_EXT = 0x00000080,
    // Provided by VK_EXT_image_drm_format_modifier
    VK_IMAGE_ASPECT_MEMORY_PLANE_1_BIT_EXT = 0x00000100,
    // Provided by VK_EXT_image_drm_format_modifier
    VK_IMAGE_ASPECT_MEMORY_PLANE_2_BIT_EXT = 0x00000200,
    // Provided by VK_EXT_image_drm_format_modifier
    VK_IMAGE_ASPECT_MEMORY_PLANE_3_BIT_EXT = 0x00000400,
} 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.

• **VK_IMAGE_ASPECT_MEMORY_PLANE_0_BIT_EXT** specifies memory plane 0.

• **VK_IMAGE_ASPECT_MEMORY_PLANE_1_BIT_EXT** specifies memory plane 1.

• **VK_IMAGE_ASPECT_MEMORY_PLANE_2_BIT_EXT** specifies memory plane 2.

• **VK_IMAGE_ASPECT_MEMORY_PLANE_3_BIT_EXT** specifies memory plane 3.

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

**VkImageAspectFlags** is a bitmask type for setting a mask of zero or more **VkImageAspectFlagBits**.

The **VkComponentMapping** structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkComponentMapping {
    VkComponentSwizzle r;
    VkComponentSwizzle g;
    VkComponentSwizzle b;
    VkComponentSwizzle a;
} VkComponentMapping;
```

• **r** is a **VkComponentSwizzle** specifying the component value placed in the R component of the output vector.

• **g** is a **VkComponentSwizzle** specifying the component value placed in the G component of the output vector.

• **b** is a **VkComponentSwizzle** specifying the component value placed in the B component of the output vector.

• **a** is a **VkComponentSwizzle** specifying the component value placed in the A component of the output vector.

**Valid Usage (Implicit)**

• **VUID-VkComponentMapping-r-parameter** **r must** be a valid **VkComponentSwizzle** value

• **VUID-VkComponentMapping-g-parameter** **g must** be a valid **VkComponentSwizzle** value

• **VUID-VkComponentMapping-b-parameter** **b must** be a valid **VkComponentSwizzle** value
Possible values of the members of \texttt{VkComponentMapping}, specifying the component values placed in each component of the output vector, are:

\begin{verbatim}
#include <vk.xml> // Provided by VK_VERSION_1_0
type 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;
\end{verbatim}

- \texttt{VK_COMPONENT_SWIZZLE_IDENTITY} specifies that the component is set to the identity swizzle.
- \texttt{VK_COMPONENT_SWIZZLE_ZERO} specifies that the component is set to zero.
- \texttt{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 \texttt{VkFormat}.
- \texttt{VK_COMPONENT_SWIZZLE_R} specifies that the component is set to the value of the R component of the image.
- \texttt{VK_COMPONENT_SWIZZLE_G} specifies that the component is set to the value of the G component of the image.
- \texttt{VK_COMPONENT_SWIZZLE_B} specifies that the component is set to the value of the B component of the image.
- \texttt{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:

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Component & Identity Mapping \\
\hline
\texttt{components.r} & \texttt{VK_COMPONENT_SWIZZLE_R} \\
\texttt{components.g} & \texttt{VK_COMPONENT_SWIZZLE_G} \\
\texttt{components.b} & \texttt{VK_COMPONENT_SWIZZLE_B} \\
\texttt{components.a} & \texttt{VK_COMPONENT_SWIZZLE_A} \\
\hline
\end{tabular}
\caption{Component Mappings Equivalent To \texttt{VK_COMPONENT_SWIZZLE_IDENTITY}}
\end{table}

If the $\text{pNext}$ chain includes a \texttt{VkImageViewASTCDecodeModeEXT} structure, then that structure includes a parameter specifying the decode mode for image views using ASTC compressed formats.
The `VkImageViewASTCDecodeModeEXT` structure is defined as:

```c
// Provided by VK_EXT_astc_decode_mode
typedef struct VkImageViewASTCDecodeModeEXT {
    VkStructureType sType;
    const void* pNext;
    VkFormat decodeMode;
} VkImageViewASTCDecodeModeEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `decodeMode` is the intermediate format used to decode ASTC compressed formats.

### Valid Usage

- **VUID-VkImageViewASTCDecodeModeEXT-decodeMode-02230**
  - `decodeMode` must be one of `VK_FORMAT_R16G16B16A16_SFLOAT`, `VK_FORMAT_R8G8B8A8_UNORM`, or `VK_FORMAT_E5B9G9R9_UFLOAT_PACK32`.

- **VUID-VkImageViewASTCDecodeModeEXT-decodeMode-02231**
  - If the `decodeModeSharedExponent` feature is not enabled, `decodeMode` must not be `VK_FORMAT_E5B9G9R9_UFLOAT_PACK32`.

- **VUID-VkImageViewASTCDecodeModeEXT-decodeMode-02232**
  - If `decodeMode` is `VK_FORMAT_R8G8B8A8_UNORM` the image view must not include blocks using any of the ASTC HDR modes.

- **VUID-VkImageViewASTCDecodeModeEXT-format-04084**
  - `format` of the image view must be one of the [ASTC Compressed Image Formats](https://www.khronos.org/registry/vulkan/specs/1.0/xhtml/vkspec.html#astc-compressed).

If `format` uses sRGB encoding then the `decodeMode` has no effect.

### Valid Usage (Implicit)

- **VUID-VkImageViewASTCDecodeModeEXT-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_IMAGE_VIEW_ASTC_DECODE_MODE_EXT`.

- **VUID-VkImageViewASTCDecodeModeEXT-decodeMode-parameter-decodeMode**
  - `decodeMode` must be a valid `VkFormat` value.

To destroy an image view, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyImageView(
    VkDevice device,
    VkImageView imageView,
);```

585
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`.

- If `VkImageViewCreateInfo::image` was created with `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then:
The image's DRM format modifier is the value of `VkImageDrmFormatModifierListCreateInfoEXT::drmFormatModifier` found by calling `vkGetImageDrmFormatModifierPropertiesEXT`.

Let `VkDrmFormatModifierPropertiesListEXT::pDrmFormatModifierProperties` be the array found by calling `vkGetPhysicalDeviceFormatProperties2` on the same format as `VkImageViewCreateInfo::format`.

Let `VkDrmFormatModifierPropertiesEXT prop` be an array element whose `drmFormatModifier` member is the value of the image's DRM format modifier.

Then the image view's set of format features is the value of taking the bitwise intersection, over the collected `prop::drmFormatModifierTilingFeatures`.

### 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:
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:

---

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 *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
define 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 the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `buffer` is the buffer to query.
Valid Usage (Implicit)

- VUID-VkBufferMemoryRequirementsInfo2-sType-sType
  
sType must be VK_STRUCTURE_TYPE_BUFFER_MEMORY_REQUIREMENTS_INFO_2

- VUID-VkBufferMemoryRequirementsInfo2-pNext-pNext
  
pNext must be NULL

- VUID-VkBufferMemoryRequirementsInfo2-buffer-parameter
  
buffer must be a valid VkBuffer handle

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

image is the image to query.

Valid Usage

• VUID-VkImageMemoryRequirementsInfo2-image-01589
  If image was created with a multi-planar format and the VK_IMAGE_CREATE_DISJOINT_BIT flag, there must be a VkImagePlaneMemoryRequirementsInfo included in the pNext chain of the VkImageMemoryRequirementsInfo2 structure

• VUID-VkImageMemoryRequirementsInfo2-image-02279
  If image was created with VK_IMAGE_CREATE_DISJOINT_BIT and with VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT, then there must be a VkImagePlaneMemoryRequirementsInfo 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-02280
  If image was created with a single-plane format and with any tiling other than VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT, then there must not be a VkImagePlaneMemoryRequirementsInfo included in the pNext chain of the VkImageMemoryRequirementsInfo2 structure

Valid Usage (Implicit)

• VUID-VkImageMemoryRequirementsInfo2-sType-sType
  sType must be VK_STRUCTURE_TYPE_IMAGE_MEMORY_REQUIREMENTS_INFO_2

• VUID-VkImageMemoryRequirementsInfo2-pNext-pNext
  pNext must be NULL or a pointer to a valid instance of VkImagePlaneMemoryRequirementsInfo

• VUID-VkImageMemoryRequirementsInfo2-sType-unique
  The sType value of each struct in the pNext chain must be unique

• VUID-VkImageMemoryRequirementsInfo2-image-parameter
  image must be a valid VkImage handle

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:

// Provided by VK_VERSION_1_1
typedef struct VkImagePlaneMemoryRequirementsInfo {
    VkStructureType sType;
    const void* pNext;
    VkImageAspectFlagBits planeAspect;
} VkImagePlaneMemoryRequirementsInfo;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **planeAspect** is a VkImageAspectFlagBits value specifying the aspect corresponding to the image plane to query.

### Valid Usage

- **VUID-VkImagePlaneMemoryRequirementsInfo-planeAspect-02281**
  If the image's tiling is VK_IMAGE_TILING_LINEAR or VK_IMAGE_TILING_OPTIMAL, then **planeAspect** must be a single valid format plane for the image (that is, for a two-plane image **planeAspect** must be VK_IMAGE_ASPECT_PLANE_0_BIT or VK_IMAGE_ASPECT_PLANE_1_BIT, and for a three-plane image **planeAspect** must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT or VK_IMAGE_ASPECT_PLANE_2_BIT)

- **VUID-VkImagePlaneMemoryRequirementsInfo-planeAspect-02282**
  If the image's tiling is VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT, then **planeAspect** must be a single valid memory plane for the image (that is, **aspectMask** must specify a plane index that is less than the VkDrmFormatModifierPropertiesEXT::drmFormatModifierPlaneCount associated with the image's format and VkImageDrmFormatModifierPropertiesEXT::drmFormatModifier)

### Valid Usage (Implicit)

- **VUID-VkImagePlaneMemoryRequirementsInfo-sType-sType**
  **sType** must be VK_STRUCTURE_TYPE_IMAGE_PLANE_MEMORY_REQUIREMENTS_INFO

- **VUID-VkImagePlaneMemoryRequirementsInfo-planeAspect-parameter**
  **planeAspect** must be a valid VkImageAspectFlagBits value

The **VkMemoryRequirements2** structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkMemoryRequirements2 {
    VkStructureType sType;
    void* pNext;
    VkMemoryRequirements memoryRequirements;
} VkMemoryRequirements2;
```

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

• `memoryRequirements` is a `VkMemoryRequirements` structure describing the memory requirements of the resource.

### Valid Usage (Implicit)

- VUID-VkMemoryRequirements2-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_MEMORY_REQUIREMENTS_2`.

- VUID-VkMemoryRequirements2-pNext-pNext
  
  `pNext` must be `NULL` or a pointer to a valid instance of `VkMemoryDedicatedRequirements`.

- VUID-VkMemoryRequirements2-sType-unique
  
  The `sType` value of each struct in the `pNext` chain must be unique.

The `VkMemoryDedicatedRequirements` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkMemoryDedicatedRequirements {
    VkStructureType sType;
    void* pNext;
    VkBool32 prefersDedicatedAllocation;
    VkBool32 requiresDedicatedAllocation;
} VkMemoryDedicatedRequirements;
```

- `sType` is the type of this structure.

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

- `prefersDedicatedAllocation` specifies that the implementation would prefer a dedicated allocation for this resource. The application is still free to suballocate the resource but it may get better performance if a dedicated allocation is used.

- `requiresDedicatedAllocation` specifies that a dedicated allocation is required for this resource.

To determine the dedicated allocation requirements of a buffer or image resource, add a `VkMemoryDedicatedRequirements` structure to the `pNext` chain of the `VkMemoryRequirements2` structure passed as the `pMemoryRequirements` parameter of `vkGetBufferMemoryRequirements2` or `vkGetImageMemoryRequirements2`, respectively.

Constraints on the values returned for buffer resources are:

- `requiresDedicatedAllocation` may be `VK_TRUE` if the `pNext` chain of `VkBufferCreateInfo` for the call to `vkCreateBuffer` used to create the buffer being queried included a `VkExternalMemoryBufferCreateInfo` structure, and any of the handle types specified in `VkExternalMemoryBufferCreateInfo::handleTypes` requires dedicated allocation, as reported by `vkGetPhysicalDeviceExternalBufferProperties::externalMemoryProperties.externalMemoryFeatures`. Otherwise, `requiresDedicatedAllocation` will be `VK_FALSE`. 
• When the implementation sets `requiresDedicatedAllocation` to `VK_TRUE`, it **must** also set `prefersDedicatedAllocation` to `VK_TRUE`.

• If `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` was set in `VkBufferCreateInfo::flags` when `buffer` was created, then both `prefersDedicatedAllocation` and `requiresDedicatedAllocation` will be `VK_FALSE`.

Constraints on the values returned for image resources are:

• `requiresDedicatedAllocation` **may** be `VK_TRUE` if the `pNext` chain of `VkImageCreateInfo` for the call to `vkCreateImage` used to create the image being queried included a `VkExternalMemoryImageCreateInfo` structure, and any of the handle types specified in `VkExternalMemoryImageCreateInfo::handleTypes` requires dedicated allocation, as reported by `vkGetPhysicalDeviceImageFormatProperties2` in `VkExternalImageFormatProperties::externalMemoryProperties.externalMemoryFeatures`. Otherwise, `requiresDedicatedAllocation` will 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,          // Provided by VK_VERSION_1_0
    VkBuffer buffer,         // Provided by VK_VERSION_1_0
    VkDeviceMemory memory,   // Provided by VK_VERSION_1_0
    VkDeviceSize memoryOffset);
```

• `device` is the logical device that owns the buffer and memory.

• `buffer` is the buffer to be attached to memory.

• `memory` is a `VkDeviceMemory` object describing the device memory to attach.

• `memoryOffset` is the start offset of the region of `memory` which is to be bound to the buffer. The number of bytes returned in the `VkMemoryRequirements::size` member in `memory`, starting from `memoryOffset` bytes, will be bound to the specified buffer.

`vkBindBufferMemory` is equivalent to passing the same parameters through `VkBindBufferMemoryInfo` to `vkBindBufferMemory2`.

If the `memory` was obtained by a memory import operation with `VkExternalMemoryBufferCreateInfo::handleTypes` assigned to `VK_EXTERNAL_MEMORY_HANDLE_TYPE_SCI_BUF_BIT_NV`, the properties of `buffer` and the `memoryOffset` must be compatible with the attributes used to create `NvSciBufObj`, otherwise the implementation will
return VK_ERROR_VALIDATION_FAILED.

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

Valid Usage

- VUID-vkBindBufferMemory-buffer-01029
  buffer must not already be backed by a memory object

- VUID-vkBindBufferMemory-buffer-01030
  buffer must not have been created with any sparse memory binding flags

- VUID-vkBindBufferMemory-memoryOffset-01031
  memoryOffset must be less than the size of memory

- VUID-vkBindBufferMemory-memory-01035
  memory must have been allocated using one of the memory types allowed in the 
  memoryTypeBits member of the VkMemoryRequirements structure returned from a call to 
vkGetBufferMemoryRequirements with buffer

- VUID-vkBindBufferMemory-memoryOffset-01036
  memoryOffset must be an integer multiple of the alignment member of the 
  VkMemoryRequirements structure returned from a call to vkGetBufferMemoryRequirements with buffer

- VUID-vkBindBufferMemory-size-01037
  The size member of the VkMemoryRequirements structure returned from a call to 
vkGetBufferMemoryRequirements with buffer must be less than or equal to the size of memory 
minus memoryOffset

- VUID-vkBindBufferMemory-buffer-01444
  If buffer requires a dedicated allocation (as reported by 
vkGetBufferMemoryRequirements2 in VkMemoryDedicatedRequirements ::requiresDedicatedAllocation for buffer), memory must have been allocated with 
VkMemoryDedicatedAllocateInfo::buffer equal to buffer

- VUID-vkBindBufferMemory-memory-01508
  If the VkMemoryAllocateInfo provided when memory was allocated included a 
VkMemoryDedicatedAllocateInfo structure in its pNext chain, and 
VkMemoryDedicatedAllocateInfo::buffer was not VK_NULL_HANDLE, then buffer must 
equal VkMemoryDedicatedAllocateInfo::buffer, and memoryOffset must be zero

- VUID-vkBindBufferMemory-None-01898
  If buffer was created with the VK_BUFFER_CREATE_PROTECTED_BIT bit set, the buffer must be 
bound to a memory object allocated with a memory type that reports 
VK_MEMORY_PROPERTY_PROTECTED_BIT

- VUID-vkBindBufferMemory-None-01899
  If buffer was created with the VK_BUFFER_CREATE_PROTECTED_BIT bit not set, the buffer must 
not be bound to a memory object allocated with a memory type that reports 
VK_MEMORY_PROPERTY_PROTECTED_BIT

- VUID-vkBindBufferMemory-memory-02726
If the value of `VkExportMemoryAllocateInfo::handleTypes` used to allocate `memory` is not 0, it must include at least one of the handles set in `VkExternalMemoryBufferCreateInfo::handleTypes` when `buffer` was created.

- **VUID-vkBindBufferMemory-memory-02727**
  If `memory` was allocated by a memory import operation, the external handle type of the imported memory must also have been set in `VkExternalMemoryBufferCreateInfo::handleTypes` when `buffer` was created.

- **VUID-vkBindBufferMemory-bufferDeviceAddress-03339**
  If the `VkPhysicalDeviceBufferDeviceAddressFeatures::bufferDeviceAddress` feature is enabled and `buffer` was created with the `VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT` bit set, `memory` must have been allocated with the `VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT` bit set.

### Valid Usage (Implicit)

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

- **VUID-vkBindBufferMemory-buffer-parameter**
  `buffer` must be a valid `VkBuffer` handle.

- **VUID-vkBindBufferMemory-memory-parameter**
  `memory` must be a valid `VkDeviceMemory` handle.

- **VUID-vkBindBufferMemory-buffer-parent**
  `buffer` must have been created, allocated, or retrieved from `device`.

- **VUID-vkBindBufferMemory-memory-parent**
  `memory` must have been created, allocated, or retrieved from `device`.

### Host Synchronization

- Host access to `buffer` must be externally synchronized.

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

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

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;
} VkBindBufferMemoryInfo;
```
const void* pNext;
VkBuffer buffer;
VkDeviceMemory memory;
VkDeviceSize memoryOffset;

VkBindBufferMemoryInfo;

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• buffer is the buffer to be attached to memory.
• memory is a VkDeviceMemory object describing the device memory to attach.
• memoryOffset is the start offset of the region of memory which is to be bound to the buffer. The number of bytes returned in the VkMemoryRequirements::size member in memory, starting from memoryOffset bytes, will be bound to the specified buffer.

Valid Usage

• VUID-VkBindBufferMemoryInfo-buffer-01029
  buffer must not already be backed by a memory object

• VUID-VkBindBufferMemoryInfo-buffer-01030
  buffer must not have been created with any sparse memory binding flags

• VUID-VkBindBufferMemoryInfo-memoryOffset-01031
  memoryOffset must be less than the size of memory

• VUID-VkBindBufferMemoryInfo-memory-01035
  memory must have been allocated using one of the memory types allowed in the memoryTypeBits member of the VkMemoryRequirements structure returned from a call to vkGetBufferMemoryRequirements with buffer

• VUID-VkBindBufferMemoryInfo-memoryOffset-01036
  memoryOffset must be an integer multiple of the alignment member of the VkMemoryRequirements structure returned from a call to vkGetBufferMemoryRequirements with buffer

• VUID-VkBindBufferMemoryInfo-size-01037
  The size member of the VkMemoryRequirements structure returned from a call to vkGetBufferMemoryRequirements with buffer must be less than or equal to the size of memory minus memoryOffset

• VUID-VkBindBufferMemoryInfo-buffer-01444
  If 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

• 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-02727
If memory was allocated by a memory import operation, the external handle type of the imported memory must also have been set in VkExternalMemoryBufferCreateInfo::handleTypes when buffer was created

• VUID-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-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
The 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
The `VkBindBufferMemoryDeviceGroupInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkBindBufferMemoryDeviceGroupInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t deviceIndexCount;
    const uint32_t* pDeviceIndices;
} VkBindBufferMemoryDeviceGroupInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **deviceIndexCount** is the number of elements in **pDeviceIndices**.
- **pDeviceIndices** is a pointer to an array of device indices.

If the **pNext** chain of `VkBindBufferMemoryInfo` includes a `VkBindBufferMemoryDeviceGroupInfo` structure, then that structure determines how memory is bound to buffers across multiple devices in a device group.

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

If **deviceIndexCount** is zero and **memory** comes from a memory heap with the `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` bit set, then it is as if `pDeviceIndices` contains consecutive indices from zero to the number of physical devices in the logical device, minus one. In other words, by default each physical device attaches to its own instance of **memory**.

If **deviceIndexCount** is zero and **memory** comes from a memory heap without the `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` bit set, then it is as if `pDeviceIndices` contains an array of zeros. In other words, by default each physical device attaches to instance zero.

### Valid Usage

- **VUID-VkBindBufferMemoryDeviceGroupInfo-deviceIndexCount-01606**
  
  `deviceIndexCount` **must** either be zero or equal to the number of physical devices in the logical device

- **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`
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,       // device is the logical device that owns the image and memory.
    VkImage image,        // image is the image.
    VkDeviceMemory memory,  // memory is the 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 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 the `memory` is allocated by a memory import operation with `VkExternalMemoryBufferCreateInfo`::`handleTypes` assigned to `VK_EXTERNAL_MEMORY_HANDLE_TYPE_SCI_BUF_BIT_NV`, the properties of `image` and the `memoryOffset` must be compatible with the attributes used to create `NvSciBufObj`, otherwise the implementation will return `VK_ERROR_VALIDATION_FAILED`.

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

### Valid Usage

- **VUID-vkBindImageMemory-image-01044**
  `image` must not already be backed by a memory object

- **VUID-vkBindImageMemory-image-01045**
  `image` must not have been created with any sparse memory binding flags

- **VUID-vkBindImageMemory-memoryOffset-01046**
  `memoryOffset` must be less than the size of `memory`

- **VUID-vkBindImageMemory-image-01445**
  If `image` requires a dedicated allocation (as reported by `vkGetImageMemoryRequirements2` in `VkMemoryDedicatedRequirements`::`requiresDedicatedAllocation` for `image`), `memory` must have been created with `VkMemoryDedicatedAllocateInfo::image` equal to `image`
If the `VkMemoryAllocateInfo` provided when memory was allocated included a `VkMemoryDedicatedAllocateInfo` structure in its `pNext` chain, and `VkMemoryDedicatedAllocateInfo::image` was not `VK_NULL_HANDLE`, then `image` must equal `VkMemoryDedicatedAllocateInfo::image` and `memoryOffset` must be zero.

If image was created with the `VK_IMAGE_CREATE_PROTECTED_BIT` bit set, the image must be bound to a memory object allocated with a memory type that reports `VK_MEMORYPROPERTY_PROTECTED_BIT`.

If image was created with the `VK_IMAGE_CREATE_PROTECTED_BIT` bit not set, the image must not be bound to a memory object created with a memory type that reports `VK_MEMORYPROPERTY_PROTECTED_BIT`.

If the value of `VkExportMemoryAllocateInfo::handleTypes` used to allocate memory is not 0, it must include at least one of the handles set in `VkExternalMemoryImageCreateInfo::handleTypes` when image was created.

If memory was created by a memory import operation, the external handle type of the imported memory must also have been set in `VkExternalMemoryImageCreateInfo::handleTypes` when image was created.

image must not have been created with the `VK_IMAGE_CREATE_DISJOINT_BIT` set.

memory must have been allocated using one of the memory types allowed in the `memoryTypeBits` member of the `VkMemoryRequirements` structure returned from a call to `vkGetImageMemoryRequirements` with image.

memoryOffset must be an integer multiple of the `alignment` member of the `VkMemoryRequirements` structure returned from a call to `vkGetImageMemoryRequirements` with image.

The difference of the size of memory and memoryOffset must be greater than or equal to the size member of the `VkMemoryRequirements` structure returned from a call to `vkGetImageMemoryRequirements` with the same image.

Valid Usage (Implicit)

- device must be a valid `VkDevice` handle
- image must be a valid `VkImage` handle
- memory must be a valid `VkMemory` handle
memory must be a valid VkDeviceMemory handle

- VUID-vkBindImageMemory-image-parent
  
  image must have been created, allocated, or retrieved from device

- VUID-vkBindImageMemory-memory-parent
  
  memory must have been created, allocated, or retrieved from device

**Host Synchronization**

- Host access to image must be externally synchronized

**Return Codes**

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

To attach memory to image objects for one or more images at a time, call:

```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.

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

- VUID-vkBindImageMemory2-pBindInfos-04006
  pBindInfos must not refer to the same image subresource more than once

### Valid Usage (Implicit)

- VUID-vkBindImageMemory2-device-parameter
device must be a valid VkDevice handle

- VUID-vkBindImageMemory2-pBindInfos-parameter
  pBindInfos must be a valid pointer to an array of bindInfoCount valid
  VkBindImageMemoryInfo structures

- VUID-vkBindImageMemory2-bindInfoCount-arraylength
  bindInfoCount must be greater than 0

### Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

VkBindImageMemoryInfo contains members corresponding to the parameters of
vkBindImageMemory.

The VkBindImageMemoryInfo structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkBindImageMemoryInfo {
    VkStructureType sType;
    const void* pNext;
    VkImage image;
    VkDeviceMemory memory;
    VkDeviceSize memoryOffset;
} VkBindImageMemoryInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **image** is the image to be attached to memory.
- **memory** is a VkDeviceMemory object describing the device memory to attach.
- **memoryOffset** is the start offset of the region of memory which is to be bound to the image. The
number of bytes returned in the `VkMemoryRequirements::size` member in `memory`, starting from `memoryOffset` bytes, will be bound to the specified image.

**Valid Usage**

- **VUID-VkBindImageMemoryInfo-image-01044**
  `image` must not already be backed by a memory object

- **VUID-VkBindImageMemoryInfo-image-01045**
  `image` must not have been created with any sparse memory binding flags

- **VUID-VkBindImageMemoryInfo-memoryOffset-01046**
  `memoryOffset` must be less than the size of `memory`

- **VUID-VkBindImageMemoryInfo-image-01445**
  If `image` requires a dedicated allocation (as reported by `vkGetImageMemoryRequirements2` in `VkMemoryDedicatedRequirements`::`requiresDedicatedAllocation` for `image`), `memory` must have been created with `VkMemoryDedicatedAllocateInfo`::`image` equal to `image`

- **VUID-VkBindImageMemoryInfo-memory-01509**
  If the `VkMemoryAllocateInfo` provided when `memory` was allocated included a `VkMemoryDedicatedAllocateInfo` structure in its `pNext` chain, and `VkMemoryDedicatedAllocateInfo`::`image` was not `VK_NULL_HANDLE`, then `image` must equal `VkMemoryDedicatedAllocateInfo`::`image` and `memoryOffset` must be zero

- **VUID-VkBindImageMemoryInfo-None-01901**
  If image was created with the `VK_IMAGE_CREATE_PROTECTED_BIT` bit set, the image must be bound to a memory object allocated with a memory type that reports `VK_MEMORY_PROPERTY_PROTECTED_BIT`

- **VUID-VkBindImageMemoryInfo-None-01902**
  If image was created with the `VK_IMAGE_CREATE_PROTECTED_BIT` bit not set, the image must not be bound to a memory object created with a memory type that reports `VK_MEMORY_PROPERTY_PROTECTED_BIT`

- **VUID-VkBindImageMemoryInfo-memory-02728**
  If the value of `VkExportMemoryAllocateInfo`::`handleTypes` used to allocate `memory` is not 0, it must include at least one of the handles set in `VkExternalMemoryImageCreateInfo`::`handleTypes` when `image` was created

- **VUID-VkBindImageMemoryInfo-memory-02729**
  If `memory` was created by a memory import operation, the external handle type of the imported memory must also have been set in `VkExternalMemoryImageCreateInfo`::`handleTypes` when `image` was created

- **VUID-VkBindImageMemoryInfo-pNext-01615**
  If the `pNext` chain does not include a `VkBindImagePlaneMemoryInfo` structure, `memory` must have been allocated using one of the memory types allowed in the `memoryTypeBits` member of the `VkMemoryRequirements` structure returned from a call to `vkGetImageMemoryRequirements2` with `image`

- **VUID-VkBindImageMemoryInfo-pNext-01616**
If the pNext chain does not include a VkBindImagePlaneMemoryInfo structure, memoryOffset must be an integer multiple of the alignment member of the VkMemoryRequirements structure returned from a call to vkGetImageMemoryRequirements2 with image

- VUID-VkBindImageMemoryInfo-pNext-01617
  If the pNext chain does not include a VkBindImagePlaneMemoryInfo structure, the difference of the size of memory and memoryOffset must be greater than or equal to the size member of the VkMemoryRequirements structure returned from a call to vkGetImageMemoryRequirements2 with the same image

- VUID-VkBindImageMemoryInfo-pNext-01618
  If the pNext chain includes a VkBindImagePlaneMemoryInfo structure, image must have been created with the VK_IMAGE_CREATE_DISJOINT_BIT bit set

- VUID-VkBindImageMemoryInfo-pNext-01619
  If the pNext chain includes a VkBindImagePlaneMemoryInfo structure, memory must have been allocated using one of the memory types allowed in the memoryTypeBits member of the VkMemoryRequirements structure returned from a call to vkGetImageMemoryRequirements2 with image and where VkBindImagePlaneMemoryInfo::planeAspect corresponds to the VkImagePlaneMemoryRequirementsInfo::planeAspect in the VkImageMemoryRequirementsInfo2 structure's pNext chain

- VUID-VkBindImageMemoryInfo-pNext-01620
  If the pNext chain includes a VkBindImagePlaneMemoryInfo structure, memoryOffset must be an integer multiple of the alignment member of the VkMemoryRequirements structure returned from a call to vkGetImageMemoryRequirements2 with image and where VkBindImagePlaneMemoryInfo::planeAspect corresponds to the VkImagePlaneMemoryRequirementsInfo::planeAspect in the VkImageMemoryRequirementsInfo2 structure's pNext chain

- VUID-VkBindImageMemoryInfo-pNext-01621
  If the pNext chain includes a VkBindImagePlaneMemoryInfo structure, the difference of the size of memory and memoryOffset must be greater than or equal to the size member of the VkMemoryRequirements structure returned from a call to vkGetImageMemoryRequirements2 with the same image and where VkBindImagePlaneMemoryInfo::planeAspect corresponds to the VkImagePlaneMemoryRequirementsInfo::planeAspect in the VkImageMemoryRequirementsInfo2 structure's pNext chain

- VUID-VkBindImageMemoryInfo-pNext-01626
  If the pNext chain includes a VkBindImageMemoryDeviceGroupInfo structure, all instances of memory specified by VkBindImageMemoryDeviceGroupInfo::pDeviceIndices must have been allocated

- VUID-VkBindImageMemoryInfo-image-01630
  If image was created with a valid swapchain handle in VkImageSwapchainCreateInfoKHR::swapchain, then the pNext chain must include a VkBindImageMemorySwapchainInfoKHR structure containing the same swapchain handle

- VUID-VkBindImageMemoryInfo-pNext-01631
If the `pNext` chain includes a `VkBindImageMemorySwapchainInfoKHR` structure, memory must be `VK_NULL_HANDLE`

- VUID-VkBindImageMemoryInfo-pNext-01632
  If the `pNext` chain does not include a `VkBindImageMemorySwapchainInfoKHR` structure, memory must be a valid `VkDeviceMemory` handle

### 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`, `VkBindImageMemorySwapchainInfoKHR`, or `VkBindImagePlaneMemoryInfo`
- VUID-VkBindImageMemoryInfo-sType-unique The `sType` value of each struct in the `pNext` chain must be unique
- VUID-VkBindImageMemoryInfo-image-parameter `image` must be a valid `VkImage` handle
- VUID-VkBindImageMemoryInfo-commonparent Both of `image`, and `memory` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`

The `VkBindImageMemoryDeviceGroupInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkBindImageMemoryDeviceGroupInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t deviceIndexCount;
    const uint32_t* pDeviceIndices;
    uint32_t splitInstanceBindRegionCount;
    const VkRect2D* pSplitInstanceBindRegions;
} VkBindImageMemoryDeviceGroupInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `deviceIndexCount` is the number of elements in `pDeviceIndices`.
- `pDeviceIndices` is a pointer to an array of device indices.
- `splitInstanceBindRegionCount` is the number of elements in `pSplitInstanceBindRegions`.
- `pSplitInstanceBindRegions` is a pointer to an array of `VkRect2D` structures describing which regions of the image are attached to each instance of memory.
If the `pNext` chain of `VkBindImageMemoryInfo` includes a `VkBindImageMemoryDeviceGroupInfo` structure, then that structure determines how memory is bound to images across multiple devices in a device group.

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

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

If the `pNext` chain of `VkBindImageMemoryInfo` includes a `VkBindImageMemorySwapchainInfoKHR` structure, then that structure includes a swapchain handle and image index indicating that the image will be bound to memory from that swapchain.
The `VkBindImageMemorySwapchainInfoKHR` structure is defined as:

```c
// Provided by VK_VERSION_1_1 with VK_KHR_swapchain
typedef struct VkBindImageMemorySwapchainInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkSwapchainKHR swapchain;
    uint32_t imageIndex;
} VkBindImageMemorySwapchainInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `swapchain` is `VK_NULL_HANDLE` or a swapchain handle.
- `imageIndex` is an image index within `swapchain`.

If `swapchain` is not `NULL`, the `swapchain` and `imageIndex` are used to determine the memory that the image is bound to, instead of `memory` and `memoryOffset`.

Memory can be bound to a swapchain and use the `pDeviceIndices` or `pSplitInstanceBindRegions` members of `VkBindImageMemoryDeviceGroupInfo`.

### Valid Usage

- VUID-VkBindImageMemorySwapchainInfoKHR-imageIndex-01644
  `imageIndex` must be less than the number of images in `swapchain`

### Valid Usage (Implicit)

- VUID-VkBindImageMemorySwapchainInfoKHR-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_SWAPCHAIN_INFO_KHR`

- VUID-VkBindImageMemorySwapchainInfoKHR-swapchain-parameter
  `swapchain` must be a valid `VkSwapchainKHR` handle

### Host Synchronization

- Host access to `swapchain` must be externally synchronized

In order to bind planes of a disjoint image, add a `VkBindImagePlaneMemoryInfo` structure to the `pNext` chain of `VkBindImageMemoryInfo`.

The `VkBindImagePlaneMemoryInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
```
typedef struct VkBindImagePlaneMemoryInfo {
    VkStructureType sType;
    const void* pNext;
    VkImageAspectFlagBits planeAspect;
} VkBindImagePlaneMemoryInfo;

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- planeAspect is a VkImageAspectFlagBits value specifying the aspect of the disjoint image plane to bind.

### Valid Usage

- VUID-VkBindImagePlaneMemoryInfo-planeAspect-02283
  If the image’s tiling is VK_IMAGE_TILING_LINEAR or VK_IMAGE_TILING_OPTIMAL, then planeAspect must be a single valid format plane for the image (that is, for a two-plane image planeAspect must be VK_IMAGE_ASPECT_PLANE_0_BIT or VK_IMAGE_ASPECT_PLANE_1_BIT, and for a three-plane image planeAspect must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT or VK_IMAGE_ASPECT_PLANE_2_BIT)

- VUID-VkBindImagePlaneMemoryInfo-planeAspect-02284
  If the image’s tiling is VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT, then planeAspect must be a single valid memory plane for the image (that is, aspectMask must specify a plane index that is less than the VkDrmFormatModifierPropertiesEXT::drmFormatModifierPlaneCount associated with the image’s format and VkImageDrmFormatModifierPropertiesEXT::drmFormatModifier)

### Valid Usage (Implicit)

- VUID-VkBindImagePlaneMemoryInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_BIND_IMAGE_PLANE_MEMORY_INFO

- VUID-VkBindImagePlaneMemoryInfo-planeAspect-parameter
  planeAspect must be a valid VkImageAspectFlagBits value

### Buffer-Image Granularity

The implementation-dependent limit bufferImageGranularity specifies a page-like granularity at which linear and non-linear resources must be placed in adjacent memory locations to avoid aliasing. Two resources which do not satisfy this granularity requirement are said to alias. bufferImageGranularity is specified in bytes, and must be a power of two. Implementations which do not impose a granularity restriction may report a bufferImageGranularity value of one.

#### Note

Despite its name, bufferImageGranularity is really a granularity between “linear” and “non-linear” resources.
Given resourceA at the lower memory offset and resourceB at the higher memory offset in the same VkDeviceMemory object, where one resource is linear and the other is non-linear (as defined in the Glossary), and the following:

\[
\begin{align*}
\text{resourceA.end} &= \text{resourceA.memoryOffset} + \text{resourceA.size} - 1 \\
\text{resourceA.endPage} &= \text{resourceA.end} \& \sim (\text{bufferImageGranularity} - 1) \\
\text{resourceB.start} &= \text{resourceB.memoryOffset} \\
\text{resourceB.startPage} &= \text{resourceB.start} \& \sim (\text{bufferImageGranularity} - 1)
\end{align*}
\]

The following property must hold:

\[
\text{resourceA.endPage} < \text{resourceB.startPage}
\]

That is, the end of the first resource (A) and the beginning of the second resource (B) must be on separate “pages” of size bufferImageGranularity. bufferImageGranularity may be different than the physical page size of the memory heap. This restriction is only needed when a linear resource and a non-linear resource are adjacent in memory and will be used simultaneously. The memory ranges of adjacent resources can be closer than bufferImageGranularity, provided they meet the alignment requirement for the objects in question.

Sparse block size in bytes and sparse image and buffer memory alignments must all be multiples of the bufferImageGranularity. Therefore, memory bound to sparse resources naturally satisfies the bufferImageGranularity.

### 12.7. Resource Sharing Mode

Buffer and image objects are created with a sharing mode controlling how they can be accessed from queues. The supported sharing modes are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkSharingMode {
    VK_SHARING_MODE_EXCLUSIVE = 0,
    VK_SHARING_MODE_CONCURRENT = 1,
} VkSharingMode;
```

- **VK_SHARING_MODE_EXCLUSIVE** specifies that access to any range or image subresource of the object will be exclusive to a single queue family at a time.

- **VK_SHARING_MODE_CONCURRENT** specifies that concurrent access to any range or image subresource of the object from multiple queue families is supported.

Note

VK_SHARING_MODE_CONCURRENT may result in lower performance access to the buffer or image than VK_SHARING_MODE_EXCLUSIVE.

Ranges of buffers and image subresources of image objects created using VK_SHARING_MODE_EXCLUSIVE...
must only be accessed by queues in the queue family that has ownership of the resource. Upon creation, such resources are not owned by any queue family; ownership is implicitly acquired upon first use within a queue. Once a resource using VK_SHARING_MODE_EXCLUSIVE is owned by some queue family, the application must perform a queue family ownership transfer to make the memory contents of a range or image subresource accessible to a different queue family.

Note
Images still require a layout transition from VK_IMAGE_LAYOUT_UNDEFINED or VK_IMAGE_LAYOUT_PREINITIALIZED before being used on the first queue.

A queue family can take ownership of an image subresource or buffer range of a resource created with VK_SHARING_MODE_EXCLUSIVE, without an ownership transfer, in the same way as for a resource that was just created; however, taking ownership in this way has the effect that the contents of the image subresource or buffer range are undefined.

Ranges of buffers and image subresources of image objects created using VK_SHARING_MODE_CONCURRENT must only be accessed by queues from the queue families specified through the queueFamilyIndexCount and pQueueFamilyIndices members of the corresponding create info structures.

12.7.1. External Resource Sharing

Resources should only be accessed in the Vulkan instance that has exclusive ownership of their underlying memory. Only one Vulkan instance has exclusive ownership of a resource's underlying memory at a given time, regardless of whether the resource was created using VK_SHARING_MODE_EXCLUSIVE or VK_SHARING_MODE_CONCURRENT. Applications can transfer ownership of a resource's underlying memory only if the memory has been imported from or exported to another instance or external API using external memory handles. The semantics for transferring ownership outside of the instance are similar to those used for transferring ownership of VK_SHARING_MODE_EXCLUSIVE resources between queues, and is also accomplished using VkBufferMemoryBarrier or VkImageMemoryBarrier operations. To make the contents of the underlying memory accessible in the destination instance or API, applications must

1. Release exclusive ownership from the source instance or API.
2. Ensure the release operation has completed using semaphores or fences.
3. Acquire exclusive ownership in the destination instance or API

Unlike queue ownership transfers, the destination instance or API is not specified explicitly when releasing ownership, nor is the source instance or API specified when acquiring ownership. Instead, the image or memory barrier's dstQueueFamilyIndex or srcQueueFamilyIndex parameters are set to the reserved queue family index VK_QUEUE_FAMILY_EXTERNAL or VK_QUEUE_FAMILY_FOREIGN_EXT to represent the external destination or source respectively.

Binding a resource to a memory object shared between multiple Vulkan instances or other APIs does not change the ownership of the underlying memory. The first entity to access the resource implicitly acquires ownership. 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.

• If either image’s tiling is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then both images *must* be linear.

• All other creation parameters *must* be identical

Aliases created by binding the same memory to resources in multiple Vulkan instances or external APIs using external memory handle export and import mechanisms interpret the contents of the memory in consistent ways, and data written to one alias *can* be read by the other alias.

Otherwise, the aliases interpret the contents of the memory differently, and writes via one alias make the contents of memory partially or completely undefined to the other alias. If the first alias is a host-accessible subresource, then the bytes affected are those written by the memory operations according to its addressing scheme. If the first alias is not host-accessible, then the bytes affected are those overlapped by the image subresources that were written. If the second alias is a host-accessible subresource, the affected bytes become undefined. If the second alias is not host-accessible, all sparse image blocks (for sparse partially-resident images) or all image subresources
(for non-sparse image and fully resident sparse images) that overlap the affected bytes become undefined.

If any image subresources are made undefined due to writes to an alias, then each of those image subresources **must** have its layout transitioned from `VK_IMAGE_LAYOUT_UNDEFINED` to a valid layout before it is used, or from `VK_IMAGE_LAYOUT_PREINITIALIZED` if the memory has been written by the host. If any sparse blocks of a sparse image have been made undefined, then only the image subresources containing them **must** be transitioned.

Use of an overlapping range by two aliases **must** be separated by a memory dependency using the appropriate access types if at least one of those uses performs writes, whether the aliases interpret memory consistently or not. If buffer or image memory barriers are used, the scope of the barrier **must** contain the entire range and/or set of image subresources that overlap.

If two aliasing image views are used in the same framebuffer, then the render pass **must** declare the attachments using the `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT`, and follow the other rules listed in that section.

**Note**

Memory recycled via an application suballocator (i.e. without freeing and reallocating the memory objects) is not substantially different from memory aliasing. However, a suballocator usually waits on a fence before recycling a region of memory, and signaling a fence involves sufficient implicit dependencies to satisfy all the above requirements.
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
• VUID-vkCreateSampler-pCreateInfo-parameter
  \texttt{pCreateInfo} must be a valid pointer to a valid \texttt{VkSamplerCreateInfo} structure

• VUID-vkCreateSampler-pAllocator-null
  \texttt{pAllocator} must be \texttt{NULL}

• VUID-vkCreateSampler-pSampler-parameter
  \texttt{pSampler} must be a valid pointer to a \texttt{VkSampler} handle

\section*{Return Codes}

\textbf{Success}
  \begin{itemize}
    \item \texttt{VK_SUCCESS}
  \end{itemize}

\textbf{Failure}
  \begin{itemize}
    \item \texttt{VK_ERROR_OUT_OF_HOST_MEMORY}
    \item \texttt{VK_ERROR_OUT_OF_DEVICE_MEMORY}
  \end{itemize}

The \texttt{VkSamplerCreateInfo} structure is defined as:

\begin{verbatim}
// 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;
\end{verbatim}

\begin{itemize}
  \item \texttt{sType} is the type of this structure.
  \item \texttt{pNext} is \texttt{NULL} or a pointer to a structure extending this structure.
  \item \texttt{flags} is a bitmask of \texttt{VkSamplerCreateFlagBits} describing additional parameters of the sampler.
  \item \texttt{magFilter} is a \texttt{VkFilter} value specifying the magnification filter to apply to lookups.
\end{itemize}
• **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 (level-of-detail) calculation and bias provided by image sampling functions in SPIR-V, as described in the Level-of-Detail Operation section.

• **anisotropyEnable** is **VK_TRUE** to enable anisotropic filtering, as described in the Texel Anisotropic Filtering section, or **VK_FALSE** otherwise.

• **maxAnisotropy** is the anisotropy value clamp used by the sampler when **anisotropyEnable** is **VK_TRUE**. If **anisotropyEnable** is **VK_FALSE**, **maxAnisotropy** is ignored.

• **compareEnable** is **VK_TRUE** to enable comparison against a reference value during lookups, or **VK_FALSE** otherwise.

  ◦ Note: Some implementations will default to shader state if this member does not match.

• **compareOp** is a *VkCompareOp* value specifying the comparison function to apply to fetched data before filtering as described in the Depth Compare Operation section.

• **minLod** 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 anisotropic sampling feature is not enabled, anisotropyEnable must be VK_FALSE

- VUID-VkSamplerCreateInfo-anisotropyEnable-01071
  If anisotropyEnable is VK_TRUE, maxAnisotropy must be between 1.0 and
  VkPhysicalDeviceLimits::maxSamplerAnisotropy, inclusive

- VUID-VkSamplerCreateInfo-minFilter-01645
  If sampler Y'CbCr conversion is enabled and the potential format features of the sampler
  Y'CbCr conversion do not support VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT,
minFilter and magFilter must be equal to the sampler Y’C₆C₈ conversion’s chromaFilter

- VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01072
  If unnormalizedCoordinates is VK_TRUE, minFilter and magFilter must be equal

- VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01073
  If unnormalizedCoordinates is VK_TRUE, mipmapMode must be VK_SAMPLER_MIPMAP_MODE_NEAREST

- VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01074
  If unnormalizedCoordinates is VK_TRUE, minLod and maxLod must be zero

- VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01075
  If unnormalizedCoordinates is VK_TRUE, addressModeU and addressModeV must each be either
  VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE or VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER

- VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01076
  If unnormalizedCoordinates is VK_TRUE, anisotropyEnable must be VK_FALSE

- VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01077
  If unnormalizedCoordinates is VK_TRUE, compareEnable must be VK_FALSE

- VUID-VkSamplerCreateInfo-addressModeU-01078
  If any of addressModeU, addressModeV, or addressModeW are
  VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER, borderColor must be a valid VkBorderColor value

- VUID-VkSamplerCreateInfo-addressModeU-01646
  If sampler Y’C₆C₈ conversion is enabled, addressModeU, addressModeV, and addressModeW
  must be VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE, anisotropyEnable must be VK_FALSE, and
  unnormalizedCoordinates must be VK_FALSE

- VUID-VkSamplerCreateInfo-None-01647
  The sampler reduction mode must be set to VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE
  if sampler Y’C₆C₈ conversion is enabled

- VUID-VkSamplerCreateInfo-addressModeU-01079
  If samplerMirrorClampToEdge is not enabled, and if the
  VK_KHR_sampler_mirror_clamp_to_edge extension is not enabled, addressModeU, addressModeV
  and addressModeW must not be VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE

- VUID-VkSamplerCreateInfo-compareEnable-01080
  If compareEnable is VK_TRUE, compareOp must be a valid VkCompareOp value

- VUID-VkSamplerCreateInfo-magFilter-01081
  If either magFilter or minFilter is VK_FILTER_CUBIC_EXT, anisotropyEnable must be VK_FALSE

- VUID-VkSamplerCreateInfo-compareEnable-01423
  If compareEnable is VK_TRUE, the reductionMode member of
  VkSamplerReductionModeCreateInfo must be
  VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE

- VUID-VkSamplerCreateInfo-borderColor-04011
  If borderColor is one of VK_BORDER_COLOR_FLOAT_CUSTOM_EXT or
  VK_BORDER_COLOR_INT_CUSTOM_EXT, then a VkSamplerCustomBorderColorCreateInfoEXT
  must be included in the pNext chain

- VUID-VkSamplerCreateInfo-customBorderColors-04085
If the `customBorderColors` feature is not enabled, `borderColor` must not be `VK_BORDER_COLOR_FLOAT_CUSTOM_EXT` or `VK_BORDER_COLOR_INT_CUSTOM_EXT`

- **VUID-VkSamplerCreateInfo-borderColor-04442**
  If `borderColor` is one of `VK_BORDER_COLOR_FLOAT_CUSTOM_EXT` or `VK_BORDER_COLOR_INT_CUSTOM_EXT`, and `VkSamplerCustomBorderColorCreateInfoEXT::format` is not `VK_FORMAT_UNDEFINED`, `VkSamplerCustomBorderColorCreateInfoEXT::customBorderColor` must be within the range of values representable in `format`

- **VUID-VkSamplerCreateInfo-None-04012**
  The maximum number of samplers with custom border colors which can be simultaneously created on a device is implementation-dependent and specified by the `maxCustomBorderColorSamplers` member of the `VkPhysicalDeviceCustomBorderColorPropertiesEXT` structure

---

**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 `VkSamplerCustomBorderColorCreateInfoEXT`, `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.
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 the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `reductionMode` is a `VkSamplerReductionMode` value controlling how texture filtering combines texel values.

If the `pNext` chain of `VkSamplerCreateInfo` includes a `VkSamplerReductionModeCreateInfo` structure, then that structure includes a mode controlling how texture filtering combines texel values.

If this structure is not present, `reductionMode` is considered to be `VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE`.

### Valid Usage (Implicit)

- `VUID-VkSamplerReductionModeCreateInfo-sType-sType`  
  `sType` must be `VK_STRUCTURE_TYPE_SAMPLER_REDUCTION_MODE_CREATE_INFO`
- `VUID-VkSamplerReductionModeCreateInfo-reductionMode-parameter`  
  `reductionMode` must be a valid `VkSamplerReductionMode` value

Reduction modes are specified by `VkSamplerReductionMode`, which takes values:
typedef enum VkSamplerReductionMode {
    VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE = 0,
    VK_SAMPLER_REDUCTION_MODE_MIN = 1,
    VK_SAMPLER_REDUCTION_MODE_MAX = 2,
} VkSamplerReductionMode;

- **VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE** specifies that texel values are combined by computing a weighted average of values in the footprint, using weights as specified in the image operations chapter.

- **VK_SAMPLER_REDUCTION_MODE_MIN** specifies that texel values are combined by taking the component-wise minimum of values in the footprint with non-zero weights.

- **VK_SAMPLER_REDUCTION_MODE_MAX** specifies that texel values are combined by taking the component-wise maximum of values in the footprint with non-zero weights.

Possible values of the `VkSamplerCreateInfo::magFilter` and `minFilter` parameters, specifying filters used for texture lookups, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkFilter {
    VK_FILTER_NEAREST = 0,
    VK_FILTER_LINEAR = 1,
    VK_FILTER_CUBIC_IMG = 1000015000,
    // Provided by VK_EXT_filter_cubic
    VK_FILTER_CUBIC_EXT = VK_FILTER_CUBIC_IMG,
} VkFilter;
```

- **VK_FILTER_NEAREST** specifies nearest filtering.
- **VK_FILTER_LINEAR** specifies linear filtering.
- **VK_FILTER_CUBIC_EXT** specifies cubic filtering.

These filters are described in detail in [Texel Filtering](#).

Possible values of the `VkSamplerCreateInfo::mipmapMode`, specifying the mipmap mode used for texture lookups, are:

```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:

```cpp
// 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.

Possible values of `VkSamplerCreateInfo::borderColor`, specifying the border color used for texture lookups, are:

```cpp
// Provided by VK_VERSION_1_0
typedef enum VkBorderColor {
    VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK = 0,
    VK_BORDER_COLOR_INT_TRANSPARENT_BLACK = 1,
    VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK = 2,
    VK_BORDER_COLOR_INT_OPAQUE_BLACK = 3,
    VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE = 4,
    VK_BORDER_COLOR_INT_OPAQUE_WHITE = 5,
    // Provided by VK_EXT_custom_border_color
    VK_BORDER_COLOR_FLOAT_CUSTOM_EXT = 1000287003,
    // Provided by VK_EXT_custom_border_color
    VK_BORDER_COLOR_INT_CUSTOM_EXT = 1000287004,
} 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 a 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.
- `VK_BORDER_COLOR_FLOAT_CUSTOM_EXT` specifies a custom, floating-point format.
- `VK_BORDER_COLOR_INT_CUSTOM_EXT` specifies a custom, integer format.
• **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.
• **VK_BORDER_COLOR_FLOAT_CUSTOM_EXT** indicates that a **VkSamplerCustomBorderColorCreateInfoEXT** structure is included in the **VkSamplerCreateInfo::pNext** chain containing the color data in floating-point format.
• **VK_BORDER_COLOR_INT_CUSTOM_EXT** indicates that a **VkSamplerCustomBorderColorCreateInfoEXT** structure is included in the **VkSamplerCreateInfo::pNext** chain containing the color data in integer format.

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,** must have been created, allocated, or retrieved from **device**
Host Synchronization

- Host access to **sampler** must be externally synchronized

### 13.1. Sampler $Y'C_B'C_R$ conversion

To create a sampler with $Y'C_B'C_R$ conversion enabled, add a `VkSamplerYcbcrConversionInfo` structure to the `pNext` chain of the `VkSamplerCreateInfo` structure. To create a sampler $Y'C_B'C_R$ 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 $Y'C_B'C_R$ conversion**.

The `VkSamplerYcbcrConversionInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkSamplerYcbcrConversionInfo {
    VkStructureType          sType;
    const void*              pNext;
    VkSamplerYcbcrConversion conversion;
} VkSamplerYcbcrConversionInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `conversion` is a `VkSamplerYcbcrConversion` handle created with `vkCreateSamplerYcbcrConversion`.

#### Valid Usage (Implicit)

- **VUID-VkSamplerYcbcrConversionInfo-sType-sType**
  - `sType` **must** be `VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_INFO`

- **VUID-VkSamplerYcbcrConversionInfo-conversion-parameter**
  - `conversion` **must** be a valid `VkSamplerYcbcrConversion` handle

A sampler $Y'C_B'C_R$ conversion is an opaque representation of a device-specific sampler $Y'C_B'C_R$ conversion description, represented as a `VkSamplerYcbcrConversion` handle:

```c
// Provided by VK_VERSION_1_1
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSamplerYcbcrConversion)
```
To create a `VkSamplerYcbcrConversion`, call:

```c
// Provided by VK_VERSION_1_1
VkResult vkCreateSamplerYcbcrConversion(
    VkDevice device,
    const VkSamplerYcbcrConversionCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkSamplerYcbcrConversion* pYcbcrConversion);
```

- `device` is the logical device that creates the sampler $Y'CbCr$ conversion.
- `pCreateInfo` is a pointer to a `VkSamplerYcbcrConversionCreateInfo` structure specifying the requested sampler $Y'CbCr$ conversion.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pYcbcrConversion` is a pointer to a `VkSamplerYcbcrConversion` handle in which the resulting sampler $Y'CbCr$ conversion is returned.

The interpretation of the configured sampler $Y'CbCr$ conversion is described in more detail in the description of sampler $Y'CbCr$ conversion in the Image Operations chapter.

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

---

**Valid Usage**

- **VUID-vkCreateSamplerYcbcrConversion-None-01648**
  The sampler $Y'CbCr$ conversion 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**
  `pCreateInfo` must be a valid pointer to a valid `VkSamplerYcbcrConversionCreateInfo` structure

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

- **VUID-vkCreateSamplerYcbcrConversion-pYcbcrConversion-parameter**
  `pYcbcrConversion` must be a valid pointer to a `VkSamplerYcbcrConversion` handle
The `VkSamplerYcbcrConversionCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkSamplerYcbcrConversionCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkFormat format;
    VkSamplerYcbcrModelConversion ycbcrModel;
    VkSamplerYcbcrRange ycbcrRange;
    VkComponentMapping components;
    VkChromaLocation xChromaOffset;
    VkChromaLocation yChromaOffset;
    VkFilter chromaFilter;
    VkBool32 forceExplicitReconstruction;
} VkSamplerYcbcrConversionCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `format` is the format of the image from which color information will be retrieved.
- `ycbcrModel` describes the color matrix for conversion between color models.
- `ycbcrRange` describes whether the encoded values have headroom and foot room, or whether the encoding uses the full numerical range.
- `components` applies a swizzle based on `VkComponentSwizzle` enums prior to range expansion and color model conversion.
- `xChromaOffset` describes the sample location associated with downsampled chroma components in the x dimension. `xChromaOffset` has no effect for formats in which chroma components are not downsampled horizontally.
- `yChromaOffset` describes the sample location associated with downsampled chroma components in the y dimension. `yChromaOffset` has no effect for formats in which the chroma components are not downsampled vertically.
- `chromaFilter` is the filter for chroma reconstruction.
- `forceExplicitReconstruction` can be used to ensure that reconstruction is done explicitly, if supported.
Setting `forceExplicitReconstruction` to `VK_TRUE` may have a performance penalty on implementations where explicit reconstruction is not the default mode of operation.

If `format` supports `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT` the `forceExplicitReconstruction` value behaves as if it was set to `VK_TRUE`.

Sampler $Y'C_bC_r$ conversion objects do not support external format conversion without additional extensions defining external formats.

### Valid Usage

- **VUID-VkSamplerYcbcrConversionCreateInfo-format-04060**
  `format` must represent unsigned normalized values (i.e. the format must be a `UNORM` format)

- **VUID-VkSamplerYcbcrConversionCreateInfo-format-01650**
  The potential format features of the sampler $Y'C_bC_r$ conversion must support `VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT` or `VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT`

- **VUID-VkSamplerYcbcrConversionCreateInfo-xChromaOffset-01651**
  If the potential format features of the sampler $Y'C_bC_r$ conversion do not support `VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT`, `xChromaOffset` and `yChromaOffset` must not be `VK_CHROMA_LOCATION_COSITED_EVEN` if the corresponding components are downsampled

- **VUID-VkSamplerYcbcrConversionCreateInfo-xChromaOffset-01652**
  If the potential format features of the sampler $Y'C_bC_r$ conversion do not support `VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT`, `xChromaOffset` and `yChromaOffset` must not be `VK_CHROMA_LOCATION_MIDPOINT` if the corresponding components are 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_SWIZZLE_ZERO`

- **VUID-VkSamplerYcbcrConversionCreateInfo-components-02583**
  If the format has a `_422` or `_420` suffix, then `components.r` must be the identity swizzle or `VK_COMPONENT_SWIZZLE_B`

- **VUID-VkSamplerYcbcrConversionCreateInfo-components-02584**
  If the format has a `_422` or `_420` suffix, then `components.b` must be the identity swizzle or `VK_COMPONENT_SWIZZLE_R`

- **VUID-VkSamplerYcbcrConversionCreateInfo-components-02585**
  If the format has a `_422` or `_420` suffix, and if either `components.r` or `components.b` is the identity swizzle, both values must be the identity swizzle

- **VUID-VkSamplerYcbcrConversionCreateInfo-ycbcrModel-01655**
If ycbcrModel is not VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY, then components.r, components.g, and components.b must correspond to components of the format; that is, components.r, components.g, and components.b must not be VK_COMPONENT_SWIZZLE_ZERO or VK_COMPONENT_SWIZZLE_ONE, and must not correspond to a component containing zero or one as a consequence of conversion to RGBA.

- VUID-VkSamplerYcbcrConversionCreateInfo-ycbcrRange-02748
  If ycbcrRange is VK_SAMPLER_YCBCR_RANGE_ITU_NARROW then the R, G and B components obtained by applying the component swizzle to format must each have a bit-depth greater than or equal to 8.

- VUID-VkSamplerYcbcrConversionCreateInfo-forceExplicitReconstruction-01656
  If the potential format features of the sampler Y’C’B’R conversion do not support VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT forceExplicitReconstruction must be VK_FALSE.

- VUID-VkSamplerYcbcrConversionCreateInfo-chromaFilter-01657
  If the potential format features of the sampler Y’C’B’R conversion do not support VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT, chromaFilter must not be VK_FILTER_LINEAR.

### Valid Usage (Implicit)

- VUID-VkSamplerYcbcrConversionCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_CREATE_INFO.

- VUID-VkSamplerYcbcrConversionCreateInfo-pNext-pNext
  pNext must be NULL.

- VUID-VkSamplerYcbcrConversionCreateInfo-format-parameter
  format must be a valid VkFormat value.

- VUID-VkSamplerYcbcrConversionCreateInfo-ycbcrModel-parameter
  ycbcrModel must be a valid VkSamplerYcbcrModelConversion value.

- VUID-VkSamplerYcbcrConversionCreateInfo-ycbcrRange-parameter
  ycbcrRange must be a valid VkSamplerYcbcrRange value.

- VUID-VkSamplerYcbcrConversionCreateInfo-components-parameter
  components must be a valid VkComponentMapping structure.

- VUID-VkSamplerYcbcrConversionCreateInfo-xChromaOffset-parameter
  xChromaOffset must be a valid VkChromaLocation value.

- VUID-VkSamplerYcbcrConversionCreateInfo-yChromaOffset-parameter
  yChromaOffset must be a valid VkChromaLocation value.

- VUID-VkSamplerYcbcrConversionCreateInfo-chromaFilter-parameter
  chromaFilter must be a valid VkFilter value.

If chromaFilter is VK_FILTER_NEAREST, chroma samples are reconstructed to luma component resolution using nearest-neighbour sampling. Otherwise, chroma samples are reconstructed using interpolation. More details can be found in the description of sampler Y’C’B’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 $C_b$ (C$_b$ or “U” blue color difference) component corresponds to the $B$ component of an RGB image.
- the $C_r$ (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:

```c
// Provided by VK_VERSION_1_1
typedef enum VkSamplerYcbcrRange {
    VK_SAMPLER_YCBCR_RANGE_ITU_FULL = 0,
    VK_SAMPLER_YCBCR_RANGE_ITU_NARROW = 1,
} VkSamplerYcbcrRange;
```

- **VK_SAMPLER_YCBCR_RANGE_ITU_FULL** specifies that the full range of the encoded values are valid and interpreted according to the ITU “full range” quantization rules.
- **VK_SAMPLER_YCBCR_RANGE_ITU_NARROW** specifies that headroom and foot room are reserved in the numerical range of encoded values, and the remaining values are expanded according to the ITU “narrow range” quantization rules.

The formulae for these conversions is described in the Sampler Y’C_bC_r Range Expansion section of the Image Operations chapter.

No range modification takes place if ycbcrModel is VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY; the ycbcrRange field of VkSamplerYcbcrConversionCreateInfo is ignored in this case.

The VkChromaLocation enum defines the location of downsampled chroma component samples relative to the luma samples, and is defined as:

```c
// Provided by VK_VERSION_1_1
typedef enum VkChromaLocation {
    VK_CHROMA_LOCATION_COSITED_EVEN = 0,
    VK_CHROMA_LOCATION_MIDPOINT = 1,
} VkChromaLocation;
```

- **VK_CHROMA_LOCATION_COSITED_EVEN** specifies that downsampled chroma samples are aligned with luma samples with even coordinates.
- **VK_CHROMA_LOCATION_MIDPOINT** specifies that downsampled chroma samples are located half way between each even luma sample and the nearest higher odd luma sample.

To destroy a sampler Y’C_bC_r conversion, call:

```c
// Provided by VK_VERSION_1_1
void vkDestroySamplerYcbcrConversion(
```
- **device** is the logical device that destroys the \( Y'C_B'C_R \) conversion.
- **ycbcrConversion** is the conversion to destroy.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage (Implicit)

- VUID-\( \text{vkDestroySamplerYcbcrConversion-device-parameter} \)
  *device* must be a valid *VkDevice* handle

- VUID-\( \text{vkDestroySamplerYcbcrConversion-ycbcrConversion-parameter} \)
  If *ycbcrConversion* is not \( VK\_NULL\_HANDLE \), *ycbcrConversion* must be a valid *VkSamplerYcbcrConversion* handle

- VUID-\( \text{vkDestroySamplerYcbcrConversion-pAllocator-null} \)
  *pAllocator* must be NULL

- VUID-\( \text{vkDestroySamplerYcbcrConversion-ycbcrConversion-parent} \)
  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

In addition to the predefined border color values, applications can provide a custom border color value by including the *VkSamplerCustomBorderColorCreateInfoEXT* structure in the *VkSamplerCreateInfo*::*pNext* chain.

The **VkSamplerCustomBorderColorCreateInfoEXT** structure is defined as:

```c
// Provided by VK_EXT_custom_border_color
typedef struct VkSamplerCustomBorderColorCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkClearColorValue customBorderColor;
    VkFormat format;
} VkSamplerCustomBorderColorCreateInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **customBorderColor** is a *VkClearColorValue* representing the desired custom sampler border
color.

- **format** is a *VkFormat* representing the format of the sampled image view(s). This field may be *VK_FORMAT_UNDEFINED* if the *customBorderColorWithoutFormat* feature is enabled.

### Valid Usage

- **VUID-VkSamplerCustomBorderColorCreateInfoEXT-format-04013**
  If provided *format* is not *VK_FORMAT_UNDEFINED* then the *VkSamplerCreateInfo::borderColor* type must match the sampled type of the provided *format*, as shown in the SPIR-V Sampled Type column of the Interpretation of Numeric Format table.

- **VUID-VkSamplerCustomBorderColorCreateInfoEXT-format-04014**
  If the *customBorderColorWithoutFormat* feature is not enabled then *format* must not be *VK_FORMAT_UNDEFINED*.

- **VUID-VkSamplerCustomBorderColorCreateInfoEXT-format-04015**
  If the sampler is used to sample an image view of *VK_FORMAT_B4G4R4A4_UNORM_PACK16*, *VK_FORMAT_B5G6R5_UNORM_PACK16*, or *VK_FORMAT_B5G5R5A1_UNORM_PACK16* format then *format* must not be *VK_FORMAT_UNDEFINED*.

### Valid Usage (Implicit)

- **VUID-VkSamplerCustomBorderColorCreateInfoEXT-sType-sType**
  *sType* must be *VK_STRUCTURE_TYPE_SAMPLER_CUSTOM_BORDER_COLOR_CREATE_INFO_EXT*.

- **VUID-VkSamplerCustomBorderColorCreateInfoEXT-format-parameter**
  *format* must be a valid *VkFormat* value.
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 organised 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_SHARED_PRESENT_KHR or 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.

The image subresources for a sampled image *must* be in the VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL, or VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL, or VK_IMAGE_LAYOUT_GENERAL layout in order to access its data in a shader.

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’CbCr 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.

The image subresources for a combined image sampler *must* be in the VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL, or VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL, or VK_IMAGE_LAYOUT_GENERAL layout in order to access its data in a shader.

*Note*  
On some implementations, it *may* be more efficient to sample from an image using a combination of sampler and sampled image that are stored together in the descriptor set in a combined descriptor.

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 *formatted load operations* *can* be performed on.
Uniform texel buffers define a tightly-packed 1-dimensional linear array of texels, with texels going through format conversion when read in a shader in the same way as they are for an image.

Load operations from uniform texel buffers are supported in all shader stages for image formats which report support for the `VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT` feature bit via `vkGetPhysicalDeviceFormatProperties` in `VkFormatProperties::bufferFeatures`.

### 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 formatted load, store, and atomic operations can be performed on.

Storage texel buffers define a tightly-packed 1-dimensional linear array of texels, with texels going through format conversion when read in a shader in the same way as they are for an image. Unlike uniform texel buffers, these buffers can also be written to in the same way as for storage images.

Storage texel buffer loads are supported in all shader stages for texel buffer formats which report support for the `VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT` feature bit via `vkGetPhysicalDeviceFormatProperties` in `VkFormatProperties::bufferFeatures`.

Stores to storage texel buffers are supported in compute shaders for texel buffer formats which report support for the `VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT` feature via `vkGetPhysicalDeviceFormatProperties` in `VkFormatProperties::bufferFeatures`.

Atomic operations on storage texel buffers are supported in compute shaders for texel buffer formats which report support for the `VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT` feature via `vkGetPhysicalDeviceFormatProperties` in `VkFormatProperties::bufferFeatures`.

When the `fragmentStoresAndAtomics` feature is enabled, stores and atomic operations are also supported for storage texel buffers in fragment shaders with the same set of texel buffer formats as supported in compute shaders. When the `vertexPipelineStoresAndAtomics` feature is enabled, stores and atomic operations are also supported in vertex, tessellation, and geometry shaders with the same set of texel buffer formats as supported in compute shaders.

### 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
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.

The image subresources for an input attachment must be in a valid image layout in order to access its data in a shader.

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:
To create descriptor set layout objects, call:

```c
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorSetLayout)

// 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.
pAllocator must be NULL

pSetLayout must be a valid pointer to a VkDescriptorSetLayout handle

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

Information about the descriptor set layout is passed in a VkDescriptorSetLayoutCreateInfo structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDescriptorSetLayoutCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkDescriptorSetLayoutCreateFlags flags;
    uint32_t bindingCount;
    const VkDescriptorSetLayoutBinding* pBindings;
} VkDescriptorSetLayoutCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is a bitmask specifying options for descriptor set layout creation.
- **bindingCount** is the number of elements in **pBindings**.
- **pBindings** is a pointer to an array of VkDescriptorSetLayoutBinding structures.

Valid Usage

- **VUID-VkDescriptorSetLayoutCreateInfo-binding-00279**
  The VkDescriptorSetLayoutBinding::binding members of the elements of the pBindings array must each have different values

- **VUID-VkDescriptorSetLayoutCreateInfo-flags-03000**
  If any binding has the VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT bit set, flags must include VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT

- **VUID-VkDescriptorSetLayoutCreateInfo-descriptorType-03001**
  If any binding has the VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT bit set, then all
bindings **must** not have **descriptorType** of **VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC** or **VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC**

- **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** to specify options for descriptor set layout are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkDescriptorSetLayoutCreateFlagBits {
  // Provided by VK_VERSION_1_2
  VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT = 0x00000002,
} VkDescriptorSetLayoutCreateFlagBits;
```

- **VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT** specifies that descriptor sets using this layout **must** be allocated from a descriptor pool created with the **VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT** bit set. Descriptor set layouts created with this bit set have alternate limits for the maximum number of descriptors per-stage and per-pipeline layout. The non-UpdateAfterBind limits only count descriptors in sets created without this flag. The UpdateAfterBind limits count all descriptors, but the limits **may** be higher than the non-UpdateAfterBind limits.

---

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

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

`VkDescriptorSetLayoutCreateFlags` is a bitmask type for setting a mask of zero or more `VkDescriptorSetLayoutCreateFlagBits`.

The `VkDescriptorSetLayoutBinding` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDescriptorSetLayoutBinding {
    uint32_t binding;
    VkDescriptorType descriptorType;
    uint32_t descriptorCount;
    VkShaderStageFlags stageFlags;
    const VkSampler* pImmutableSamplers;
} VkDescriptorSetLayoutBinding;
```

- **binding** is the binding number of this entry and corresponds to a resource of the same binding number in the shader stages.
- **descriptorType** is a `VkDescriptorType` specifying which type of resource descriptors are used for this binding.
- **descriptorCount** is the number of descriptors contained in the binding, accessed in a shader as an array. 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-pImmutableSamplers-04009
  The sampler objects indicated by pImmutableSamplers must not have a borderColor with one of the values VK_BORDER_COLOR_FLOAT_CUSTOM_EXT or VK_BORDER_COLOR_INT_CUSTOM_EXT

- 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
The `VkDescriptorSetLayoutBindingFlagsCreateInfo` structure, then that structure includes an array of flags, one for each descriptor set layout binding.

The `VkDescriptorSetLayoutBindingFlagsCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkDescriptorSetLayoutBindingFlagsCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t bindingCount;
    const VkDescriptorBindingFlags* pBindingFlags;
} VkDescriptorSetLayoutBindingFlagsCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `bindingCount` is zero or the number of elements in `pBindingFlags`.
- `pBindingFlags` is a pointer to an array of `VkDescriptorBindingFlags` bitfields, one for each descriptor set layout binding.

If `bindingCount` is zero or if this structure is not included in the `pNext` chain, the `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-descriptorBindingUniformBufferUpdateAfterBind-03005**
  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`
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

- VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingPartiallyBound-03013
  If VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingPartiallyBound is not enabled, all elements of pBindingFlags must not include VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT

- VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingVariableDescriptorCount-03014
  If VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingVariableDescriptorCount is not enabled, all elements of pBindingFlags must not include VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT
\* VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-pBindingFlags-03015

If an element of \( p\text{BindingFlags} \) includes \( VK\_\text{DESCRIBER\_BINDING\_VARIABLE\_DESCRIPTOR\_COUNT\_BIT} \), that element's descriptorType must not be \( VK\_\text{DESCRIPTION\_TYPE\_UNIFORM\_BUFFER\_DYNAMIC} \) or \( VK\_\text{DESCRIPTION\_TYPE\_STORAGE\_BUFFER\_DYNAMIC} \).

### Valid Usage (Implicit)

\* VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-sType-sType

\( s\text{Type} \) must be \( VK\_\text{STRUCTURE\_TYPE\_DESCRIPTOR\_SET\_LAYOUT\_BINDING\_FLAGS\_CREATE\_INFO} \).

\* VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-pBindingFlags-parameter

If \( binding\text{Count} \) is not 0, \( p\text{BindingFlags} \) must be a valid pointer to an array of \( binding\text{Count} \) valid combinations of \( Vk\text{DescriptorBindingFlagBits} \) values.

Bits which can be set in each element of \( Vk\text{DescriptorSetLayoutBindingFlagsCreateInfo} :: p\text{BindingFlags} \) to specify 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;
```

\* \( VK\_\text{DESCRIBER\_BINDING\_UPDATE\_AFTER\_BIND\_BIT} \) 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 \( vk\text{UpdateDescriptorSets} \). 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.

\* \( VK\_\text{DESCRIBER\_BINDING\_PARTIALLY\_BOUND\_BIT} \) 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.

\* \( VK\_\text{DESCRIBER\_BINDING\_UPDATE\_UNUSED\_WHILE\_PENDING\_BIT} \) 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. If \( VK\_\text{DESCRIBER\_BINDING\_PARTIALLY\_BOUND\_BIT} \) is also set, then descriptors can be updated as long

---

648
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 descriptor binding has a variable size that will be specified when a descriptor set is allocated using this layout. The value of `descriptorCount` is treated as an upper bound on the size of the binding. This must only be used for the last binding in the descriptor set layout (i.e. the binding with the largest value of `binding`). For the purposes of counting against limits such as `maxDescriptorSet` and `maxPerStageDescriptor`, the full value of `descriptorCount` is counted.

**Note**

Note that while `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` and `VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT` both involve updates to descriptor sets after they are bound, `VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT` is a weaker requirement since it is only about descriptors that are not used, whereas `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` requires the implementation to observe updates to descriptors that are used.

```cpp
// 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:

```cpp
// Provided by VK_VERSION_1_1
void vkGetDescriptorSetLayoutSupport(
    VkDevice device,
    const VkDescriptorSetLayoutCreateInfo* pCreateInfo,
    VkDescriptorSetLayoutSupport* pSupport);
```

- `device` is the logical device that would create the descriptor set layout.
- `pCreateInfo` is a pointer to a `VkDescriptorSetLayoutCreateInfo` structure specifying the state of the descriptor set layout object.
- `pSupport` is a pointer to a `VkDescriptorSetLayoutSupport` structure, in which information about support for the descriptor set layout object is returned.

Some implementations have limitations on what fits in a descriptor set which are not easily expressible in terms of existing limits like `maxDescriptorSet`, for example if all descriptor types share a limited space in memory but each descriptor is a different size or alignment. This command returns information about whether a descriptor set satisfies this limit. If the descriptor set layout satisfies the `VkPhysicalDeviceMaintenance3Properties::maxPerSetDescriptors` limit, this command is guaranteed to return `VK_TRUE` in `VkDescriptorSetLayoutSupport::supported`. If the descriptor set
layout exceeds the VkPhysicalDeviceMaintenance3Properties::maxPerSetDescriptors limit, whether the descriptor set layout is supported is implementation-dependent and may depend on whether the descriptor sizes and alignments cause the layout to exceed an internal limit.

This command does not consider other limits such as maxPerStageDescriptor*, and so a descriptor set layout that is supported according to this command must still satisfy the pipeline layout limits such as maxPerStageDescriptor* in order to be used in a pipeline layout.

**Note**

This is a VkDevice query rather than VkPhysicalDevice because the answer may depend on enabled features.

**Valid Usage (Implicit)**

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

- VUID-vkGetDescriptorSetLayoutSupport-pCreateInfo-parameter
  
  pCreateInfo must be a valid pointer to a valid VkDescriptorSetLayoutCreateInfo structure

- VUID-vkGetDescriptorSetLayoutSupport-pSupport-parameter
  
  pSupport must be a valid pointer to a VkDescriptorSetLayoutSupport structure

Information about support for the descriptor set layout is returned in a VkDescriptorSetLayoutSupport structure:

```c
// Provided by VK_VERSION_1_1
typedef struct VkDescriptorSetLayoutSupport {
    VkStructureType sType;
    void* pNext;
    VkBool32 supported;
} VkDescriptorSetLayoutSupport;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- supported specifies whether the descriptor set layout can be created.

supported is set to VK_TRUE if the descriptor set can be created, or else is set to VK_FALSE.

**Valid Usage (Implicit)**

- VUID-VkDescriptorSetLayoutSupport-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_SUPPORT

- VUID-VkDescriptorSetLayoutSupport-pNext-pNext
  
  pNext must be NULL or a pointer to a valid instance of VkDescriptorSetVariableDescriptorCountLayoutSupport
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 the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `maxVariableDescriptorCount` indicates the maximum number of descriptors supported in the highest numbered binding of the layout, if that binding is variable-sized.

If the `VkDescriptorSetLayoutCreateInfo` structure specified in `vkGetDescriptorSetLayoutSupport::pCreateInfo` includes a variable-sized descriptor, then `supported` is determined assuming the requested size of the variable-sized descriptor, and `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 if the `descriptorCount` is one, and thus the binding is not ignored and the maximum descriptor count will be returned. If the layout is not supported, then the value written to `maxVariableDescriptorCount` is undefined.

### Valid Usage (Implicit)

- **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

... %1 = OpExtInstImport "GLSL.std.450"
... OpName %9 "mySampledImage"
OpName %14 "myArrayOfSampledImages"
OpName %18 "myUniformBuffer"
OpMemberName %18 0 "myElement"
OpName %20 ""
OpDecorate %9 DescriptorSet 0
OpDecorate %9 Binding 0
OpDecorate %14 DescriptorSet 0
OpDecorate %14 Binding 1
OpDecorate %17 ArrayStride 16
OpMemberDecorate %18 0 Offset 0
OpDecorate %18 Block
OpDecorate %20 DescriptorSet 1
OpDecorate %20 Binding 0
%2 = OpTypeVoid
%3 = OpTypeFunction %2
%6 = OpTypeFloat 32
%7 = OpTypeImage %6 2D 0 0 0 1 Unknown
%8 = OpTypePointer UniformConstant %7
%9 = OpVariable %8 UniformConstant
%10 = OpTypeInt 32 0
%11 = OpConstant %10 12
%12 = OpTypeArray %7 %11
%13 = OpTypePointer UniformConstant %12
%14 = OpVariable %13 UniformConstant
%15 = OpTypeVector %6 4
%16 = OpConstant %10 32
%17 = OpTypeArray %15 %16
%18 = OpTypeStruct %17
%19 = OpTypePointer Uniform %18
%20 = OpVariable %19 Uniform
API example

```c
VkResult myResult;

const VkDescriptorSetLayoutBinding myDescriptorSetLayoutBinding[] = {
    // binding to a single image descriptor
    
    // binding
    0,            // binding
    VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE,        // descriptorType
    1,            // descriptorCount
    VK_SHADER_STAGE_FRAGMENT_BIT, // stageFlags
    NULL,         // pImmutableSamplers

    // binding to an array of image descriptors
    
    // binding
    1,            // binding
    VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE,        // descriptorType
    12,           // descriptorCount
    VK_SHADER_STAGE_FRAGMENT_BIT,            // stageFlags
    NULL,        // pImmutableSamplers

    // binding to a single uniform buffer descriptor
    
    // binding
    0,             // binding
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,        // descriptorType
    1,             // descriptorCount
    VK_SHADER_STAGE_FRAGMENT_BIT,            // stageFlags
    NULL,          // pImmutableSamplers

};

const VkDescriptorSetLayoutCreateInfo myDescriptorSetLayoutCreateInfo[] = {
    // Information for first descriptor set with two descriptor bindings
    
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO,        // sType
    NULL,        // pNext
    0,             // flags
    2,             // bindingCount
    &myDescriptorSetLayoutBinding[0] // pBindings

    // Information for second descriptor set with one descriptor binding
    
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO,        // sType
    NULL,        // pNext
    0,             // flags
    1,             // bindingCount
    
};
```
null,
0,
1,
&myDescriptorSetLayoutBinding[2]
}
};

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
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,
    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:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineLayoutCreateInfo {
  VkStructureType sType;
  const void* pNext;
  VkPipelineLayoutCreateFlags flags;
  uint32_t setLayoutCount;
  const VkDescriptorSetLayout* pSetLayouts;
  uint32_t pushConstantRangeCount;
  const VkPushConstantRange* pPushConstantRanges;
} VkPipelineLayoutCreateInfo;
```

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• flags is reserved for future use.
• setLayoutCount is the number of descriptor sets included in the pipeline layout.
• pSetLayouts is a pointer to an array of VkDescriptorSetLayout objects.
• pushConstantRangeCount is the number of push constant ranges included in the pipeline layout.
• pPushConstantRanges is a pointer to an array of VkPushConstantRange structures defining a set of push constant ranges for use in a single pipeline layout. In addition to descriptor set layouts, a pipeline layout also describes how many push constants can be accessed by each stage of the pipeline.

Note
Push constants represent a high speed path to modify constant data in pipelines that is expected to outperform memory-backed resource updates.

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_STORAGE_BUFFER_DYNAMIC accessible to any given shader stage across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceLimits::maxPerStageDescriptorStorageBuffers

• VUID-VkPipelineLayoutCreateInfo-descriptorType-03019
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 in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set with a descriptorType of VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT accessible to any given shader stage across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceLimits::maxPerStageDescriptorInputAttachments

- 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::maxPerStageDescriptorUpdateAfterBindStorageBuffers

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03025
  The total number of descriptors with a descriptorType of VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, and VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER accessible to any given shader stage across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxPerStageDescriptorUpdateAfterBindSampledImages

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03026
  The total number of descriptors with a descriptorType of VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, and VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER accessible to any given shader stage across all elements of pSetLayouts must be less than
or equal to \( \text{VkPhysicalDeviceDescriptorIndexingProperties::maxPerStageDescriptorUpdateAfterBindStorageImages} \)

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-03027**
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::maxPerStageDescriptorUpdateAfterBindInputAttachments`

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-03028**
The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_SAMPLER` and `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` accessible across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetSamplers`

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-03029**
The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` accessible across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetUniformBuffers`

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-03030**
The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` accessible across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetUniformBuffersDynamic`

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-03031**
The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` accessible across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetStorageBuffers`

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-03032**
The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` accessible across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetStorageBuffersDynamic`

- **VUID-VkPipelineLayoutCreateInfo-descriptorType-03033**
The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, and `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` accessible across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetSampledImages`
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.

The total number of descriptors of the type VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindStorageBuffersDynamic.

The total number of descriptors of the type VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, and VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindSampledImages.

The total number of descriptors of the type VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, and
**VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER** accessible across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindStorageImages`

- **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`

---

**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-zero bitmask**
  `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 `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

---

```
// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineLayoutCreateFlags;
```

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

The `VkPushConstantRange` structure is defined as:

```
// 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**
  
  **stageFlags** **must** be a valid combination of **VkShaderStageFlagBits** values

- **VUID-VkPushConstantRange-stageFlags-requiredbitmask**
  
  **stageFlags** **must** not be 0

---

Once created, pipeline layouts are used as part of pipeline creation (see [Pipelines](#)), as part of binding descriptor sets (see [Descriptor Set Binding](#)), and as part of setting push constants (see [Push Constant Updates](#)). Pipeline creation accepts a pipeline layout as input, and the layout **may** be used to map (set, binding, arrayElement) tuples to implementation resources or memory locations within a descriptor set. The assignment of implementation resources depends only on the bindings defined in the descriptor sets that comprise the pipeline layout, and not on any shader source.

All resource variables **statically used** in all shaders in a pipeline **must** be declared with a (set,binding,arrayElement) that exists in the corresponding descriptor set layout and is of an appropriate descriptor type and includes the set of shader stages it is used by in **stageFlags**. The pipeline layout **can** include entries that are not used by a particular pipeline, or that are dead-code eliminated from any of the shaders. The pipeline layout allows the application to provide a consistent set of bindings across multiple pipeline compiles, which enables those pipelines to be compiled in a way that the implementation **may** cheaply switch pipelines without reprogramming the bindings.
Similarly, the push constant block declared in each shader (if present) must only place variables at offsets that are each included in a push constant range with stageFlags including the bit corresponding to the shader stage that uses it. The pipeline layout can include ranges or portions of ranges that are not used by a particular pipeline, or for which the variables have been dead-code eliminated from any of the shaders.

There is a limit on the total number of resources of each type that can be included in bindings in all descriptor set layouts in a pipeline layout as shown in Pipeline Layout Resource Limits. The “Total Resources Available” column gives the limit on the number of each type of resource that can be included in bindings in all descriptor sets in the pipeline layout. Some resource types count against multiple limits. Additionally, there are limits on the total number of each type of resource that can be used in any pipeline stage as described in Shader Resource Limits.

Table 14. 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>maxDescriptorSetUniformBuffers or maxDescriptorSetUpdateAfterBindUniformBuffers</td>
<td>combined image sampler</td>
</tr>
<tr>
<td>maxDescriptorSetUniformBuffersDynamic or maxDescriptorSetUpdateAfterBindUniformBuffersDynamic</td>
<td>uniform texel buffer</td>
</tr>
<tr>
<td>maxDescriptorSetStorageBuffers or maxDescriptorSetUpdateAfterBindStorageBuffers</td>
<td>uniform buffer</td>
</tr>
<tr>
<td>maxDescriptorSetStorageBuffersDynamic or maxDescriptorSetUpdateAfterBindStorageBuffersDynamic</td>
<td>uniform buffer dynamic</td>
</tr>
<tr>
<td>maxDescriptorSetInputAttachments or maxDescriptorSetUpdateAfterBindInputAttachments</td>
<td>storage image</td>
</tr>
<tr>
<td>maxDescriptorSetUniformBuffersDynamic or maxDescriptorSetUpdateAfterBindUniformBuffersDynamic</td>
<td>storage texel buffer</td>
</tr>
<tr>
<td>maxDescriptorSetStorageBuffersDynamic or maxDescriptorSetUpdateAfterBindStorageBuffersDynamic</td>
<td>storage buffer dynamic</td>
</tr>
<tr>
<td>maxDescriptorSetInputAttachments or maxDescriptorSetUpdateAfterBindInputAttachments</td>
<td>storage buffer</td>
</tr>
</tbody>
</table>

To destroy a pipeline layout, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyPipelineLayout(
    VkDevice device,
    VkPipelineLayout pipelineLayout,
    const VkAllocationCallbacks* pAllocator);
```
device is the logical device that destroys the pipeline layout.

pipelineLayout is the pipeline layout to destroy.

pAllocator controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- VUID-vkDestroyPipelineLayout-pipelineLayout-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

- VUID-vkDestroyPipelineLayout-pipelineLayout-parent
  If pipelineLayout is a valid handle, it must have been created, allocated, or retrieved from device

Host Synchronization

- Host access to pipelineLayout must be externally synchronized

Pipeline Layout Compatibility

Two pipeline layouts are defined to be “compatible for push constants” if they were created with identical push constant ranges. Two pipeline layouts are defined to be “compatible for set N” if they were created with identically defined descriptor set layouts for sets zero through N, and if they were created with identical push constant ranges.

When binding a descriptor set (see Descriptor Set Binding) to set number N, if the previously bound descriptor sets for sets zero through N-1 were all bound using compatible pipeline layouts, then performing this binding does not disturb any of the lower numbered sets. If, additionally, the previously bound descriptor set for set N was bound using a pipeline layout compatible for set N, then the bindings in sets numbered greater than N are also not disturbed.

Similarly, when binding a pipeline, the pipeline can correctly access any previously bound descriptor sets which were bound with compatible pipeline layouts, as long as all lower numbered sets were also bound with compatible layouts.
Layout compatibility means that descriptor sets can be bound to a command buffer for use by any pipeline created with a compatible pipeline layout, and without having bound a particular pipeline first. It also means that descriptor sets can remain valid across a pipeline change, and the same resources will be accessible to the newly bound pipeline.

**Implementor’s Note**

A consequence of layout compatibility is that when the implementation compiles a pipeline layout and maps pipeline resources to implementation resources, the mechanism for set N should only be a function of sets [0..N].

**Note**

Place the least frequently changing descriptor sets near the start of the pipeline layout, and place the descriptor sets representing the most frequently changing resources near the end. When pipelines are switched, only the descriptor set bindings that have been invalidated will need to be updated and the remainder of the descriptor set bindings will remain in place.

The maximum number of descriptor sets that can be bound to a pipeline layout is queried from physical device properties (see maxBoundDescriptorSets in Limits).

**API example**

```cpp
const VkDescriptorSetLayout layouts[] = { layout1, layout2 };

const VkPushConstantRange ranges[] =
{
  {
    VK_SHADER_STAGE_VERTEX_BIT, // stageFlags
    0, // offset
    4 // size
  },

  {
    VK_SHADER_STAGE_FRAGMENT_BIT, // stageFlags
    4, // offset
    4 // size
  },
};

const VkPipelineLayoutCreateInfo createInfo =
{
  VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO, // sType
  NULL, // pNext
  0, // flags
  2, // setLayoutCount
  layouts, // pSetLayouts
  2, // pushConstantRangeCount
};
```


```cpp
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)
```

To create a descriptor pool object, call:

```cpp
// Provided by VK_VERSION_1_0
VkResult vkCreateDescriptorPool(
    VkDevice device,
    const VkDescriptorPoolCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkDescriptorPool* pDescriptorPool);
```

- *device* is the logical device that creates the descriptor pool.
- *pCreateInfo* is a pointer to a *VkDescriptorPoolCreateInfo* structure specifying the state of the descriptor pool object.
- *pAllocator* controls host memory allocation as described in the *Memory Allocation* chapter.
- *pDescriptorPool* is a pointer to a *VkDescriptorPool* handle in which the resulting descriptor pool object is returned.

The created descriptor pool is returned in *pDescriptorPool*.

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

**Return Codes**

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

Additional information about the pool is passed in a VkDescriptorPoolCreateInfo structure:

```c
// 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;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is a bitmask of VkDescriptorPoolCreateFlagBits specifying certain supported operations on the pool.
- **maxSets** is the maximum number of descriptor sets that can be allocated from the pool.
- **poolSizeCount** is the number of elements in pPoolSizes.
• `pPoolSizes` is a pointer to an array of `VkDescriptorPoolSize` structures, each containing a descriptor type and number of descriptors of that type to be allocated in the pool.

If multiple `VkDescriptorPoolSize` structures containing the same descriptor type appear in the `pPoolSizes` array then the pool will be created with enough storage for the total number of descriptors of each type.

Fragmentation of a descriptor pool is possible and **may** lead to descriptor set allocation failures. A failure due to fragmentation is defined as failing a descriptor set allocation despite the sum of all outstanding descriptor set allocations from the pool plus the requested allocation requiring no more than the total number of descriptors requested at pool creation. Implementations provide certain guarantees of when fragmentation **must** not cause allocation failure, as described below.

If a descriptor pool has not had any descriptor sets freed since it was created or most recently reset then fragmentation **must** not cause an allocation failure (note that this is always the case for a pool created without the `VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT` bit set). Additionally, if all sets allocated from the pool since it was created or most recently reset use the same number of descriptors (of each type) and the requested allocation also uses that same number of descriptors (of each type), then fragmentation **must** not cause an allocation failure.

If an allocation failure occurs due to fragmentation, an application **can** create an additional descriptor pool to perform further descriptor set allocations.

If `flags` has the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT` bit set, descriptor pool creation **may** fail with the error `VK_ERROR_FRAGMENTATION` if the total number of descriptors across all pools (including this one) created with this bit set exceeds `maxUpdateAfterBindDescriptorsInAllPools`, or if fragmentation of the underlying hardware resources occurs.

### Valid Usage

- **VUID-VkDescriptorPoolCreateInfo-maxSets-00301**
  - `maxSets` **must** be greater than 0

### Valid Usage (Implicit)

- **VUID-VkDescriptorPoolCreateInfo-sType-sType**
  - `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` to enable 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_POOL_CREATE_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:

```c
// Provided by VK_VERSION_1_0
VkResult vkAllocateDescriptorSets(
    VkDevice device,
    const VkDescriptorSetAllocateInfo* pAllocateInfo,
    VkDescriptorSet* pDescriptorSets);
```

- device is the logical device that owns the descriptor pool.
- pAllocateInfo is a pointer to a VkDescriptorSetAllocateInfo structure describing parameters of the allocation.
- pDescriptorSets is a pointer to an array of VkDescriptorSet handles in which the resulting descriptor set objects are returned.

The allocated descriptor sets are returned in pDescriptorSets.

When a descriptor set is allocated, the initial state is largely uninitialized and all descriptors are undefined. Descriptors also become undefined if the underlying resource is destroyed. Descriptor sets containing undefined descriptors can still be bound and used, subject to the following conditions:

- For descriptor set bindings created with the VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT 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;
    VkDescriptorPool descriptorPool;
    uint32_t descriptorSetCount;
    const VkDescriptorSetLayout* pSetLayouts;
} VkDescriptorSetAllocateInfo;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `descriptorPool` is the pool which the sets will be allocated from.
- `descriptorSetCount` determines the number of descriptor sets to be allocated from the pool.
- `pSetLayouts` is a pointer to an array of descriptor set layouts, with each member specifying how the corresponding descriptor set is allocated.

**Valid Usage**

- VUID-VkDescriptorSetAllocateInfo-pSetLayouts-03044
  If any element of `pSetLayouts` was created with the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set, `descriptorPool` must have been created with the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT` flag set

**Valid Usage (Implicit)**

- VUID-VkDescriptorSetAllocateInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO`
- VUID-VkDescriptorSetAllocateInfo-pNext-pNext
  `pNext` must be NULL or a pointer to a valid instance of
**VkDescriptorSetVariableDescriptorCountAllocateInfo**

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

- **VUID-VkDescriptorSetAllocateInfo-descriptorPool-parameter**
  `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 descriptor count bindings, one for each descriptor set being allocated.

The `VkDescriptorSetVariableDescriptorCountAllocateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkDescriptorSetVariableDescriptorCountAllocateInfo {
    VkStructureType                 sType;
    const void*                     pNext;
    uint32_t                         descriptorSetCount;
    const uint32_t*                  pDescriptorCounts;
} VkDescriptorSetVariableDescriptorCountAllocateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `descriptorSetCount` is zero or the number of elements in `pDescriptorCounts`.
- `pDescriptorCounts` is a pointer to an array of descriptor counts, with each member specifying the number of descriptors in a variable descriptor count binding in the corresponding descriptor set being allocated.

If `descriptorSetCount` is zero or this structure is not included in the `pNext` chain, then the variable lengths are considered to be zero. Otherwise, `pDescriptorCounts[i]` is the number of descriptors in the variable count descriptor binding in the corresponding descriptor set layout. If `VkDescriptorSetAllocateInfo::pSetLayouts[i]` does not include a variable count 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 descriptor count binding, then `pDescriptorCounts[i]` must be less than or equal to the descriptor count specified for that binding when the descriptor set layout was created

Valid Usage (Implicit)

- **VUID-VkDescriptorSetVariableDescriptorCountAllocateInfo-sType-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 \texttt{pDescriptorSets} must have completed execution

- VUID-vkFreeDescriptorSets-pDescriptorSets-00310 \texttt{pDescriptorSets} must be a valid pointer to an array of \texttt{descriptorSetCount} \texttt{VkDescriptorSet} handles, each element of which must either be a valid handle or \texttt{VK_NULL_HANDLE}

- VUID-vkFreeDescriptorSets-descriptorPool-00312 \texttt{descriptorPool} must have been created with the \texttt{VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT} flag

### Valid Usage (Implicit)

- VUID-vkFreeDescriptorSets-device-parameter \texttt{device} must be a valid \texttt{VkDevice} handle

- VUID-vkFreeDescriptorSets-descriptorPool-parameter \texttt{descriptorPool} must be a valid \texttt{VkDescriptorPool} handle

- VUID-vkFreeDescriptorSets-descriptorSetCount-arraylength \texttt{descriptorSetCount} must be greater than 0

- VUID-vkFreeDescriptorSets-descriptorPool-parent \texttt{descriptorPool} must have been created, allocated, or retrieved from \texttt{device}

- VUID-vkFreeDescriptorSets-pDescriptorSets-parent Each element of \texttt{pDescriptorSets} that is a valid handle must have been created, allocated, or retrieved from \texttt{descriptorPool}

### Host Synchronization

- Host access to \texttt{descriptorPool} must be externally synchronized

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

### Return Codes

**Success**
- \texttt{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:

```c
// 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 pDescriptorWrites[i].descriptorType is VK_DESCRIPTOR_TYPE_SAMPLER or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, and dstSet was not allocated with a layout that included immutable samplers for dstBinding with descriptorType, the sampler member of any element of the pImageInfo member of pDescriptorWrites[i] must have been created on device

- VUID-vkUpdateDescriptorSets-pDescriptorWrites-06239
  For each element i where pDescriptorWrites[i].descriptorType is VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT, or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER the imageView member of any element of pDescriptorWrites[i] must have been created on device

- VUID-vkUpdateDescriptorSets-None-03047
  Descriptor bindings updated by this command 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

Valid Usage (Implicit)

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

- VUID-vkUpdateDescriptorSets-pDescriptorWrites-parameter
  If descriptorWriteCount is not 0, pDescriptorWrites must be a valid pointer to an array of descriptorWriteCount valid VkWriteDescriptorSet structures

- VUID-vkUpdateDescriptorSets-pDescriptorCopies-parameter
  If descriptorCopyCount is not 0, pDescriptorCopies must be a valid pointer to an array of descriptorCopyCount valid VkCopyDescriptorSet structures

Host Synchronization

- Host access to pDescriptorWrites[].dstSet must be externally synchronized
- Host access to pDescriptorCopies[].dstSet must be externally synchronized
The `VkWriteDescriptorSet` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkWriteDescriptorSet {
    VkStructureType          sType;
    const void*              pNext;
    VkDescriptorSet          dstSet;
    uint32_t                 dstBinding;
    uint32_t                 dstArrayElement;
    uint32_t                 descriptorCount;
    VkDescriptorType         descriptorType;
    const VkDescriptorImageInfo* pImageInfo;
    const VkDescriptorBufferInfo* pBufferInfo;
    const VkBufferView*       pTexelBufferView;
} VkWriteDescriptorSet;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to 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 **nullDescriptor** feature is enabled, the buffer, imageView, or bufferView **can** be `VK_NULL_HANDLE`. Loads from a null descriptor return zero values and stores and atomics to a null descriptor are discarded.
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**
  
  dstBinding must be less than or equal to the maximum value of binding of all VkDescriptorSetLayoutBinding structures specified when dstSet's descriptor set layout was created.

- **VUID-VkWriteDescriptorSet-dstBinding-00316**
  
  dstBinding must be a binding with a non-zero descriptorCount.

- **VUID-VkWriteDescriptorSet-descriptorCount-00317**
  
  All consecutive bindings updated via a single VkWriteDescriptorSet structure, except those with a descriptorCount of zero, must have identical descriptorType and stageFlags.

- **VUID-VkWriteDescriptorSet-descriptorCount-00318**
  
  All consecutive bindings updated via a single VkWriteDescriptorSet structure, except those with a descriptorCount of zero, must all either use immutable samplers or must all not use immutable samplers.

- **VUID-VkWriteDescriptorSet-descriptorType-00319**
  
  descriptorType must match the type of dstBinding within dstSet.

- **VUID-VkWriteDescriptorSet-dstSet-00320**
  
  dstSet must be a valid VkDescriptorSet handle.

- **VUID-VkWriteDescriptorSet-dstArrayElement-00321**
  
  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-VkWriteDescriptorSet-descriptorType-00322**
  
  If descriptorType is VK_DESCRIPTOR_TYPE_SAMPLER, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, or VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT, pImageInfo must be a valid pointer to an array of descriptorCount valid VkDescriptorImageInfo structures.

- **VUID-VkWriteDescriptorSet-descriptorType-02994**
  
  If descriptorType is VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, each element of pTexelBufferView must be either a valid VkBufferView handle or VK_NULL_HANDLE.

- **VUID-VkWriteDescriptorSet-descriptorType-02995**
  
  If descriptorType is VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER and the nullDescriptor feature is not enabled, each element of pTexelBufferView must not be VK_NULL_HANDLE.
If `descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`, `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER`, `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`, or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`, `pBufferInfo` must be a valid pointer to an array of `descriptorCount` valid `VkDescriptorBufferInfo` structures.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLER` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, and `dstSet` was not allocated with a layout that included immutable samplers for `dstBinding` with `descriptorType`, the `sampler` member of each element of `pImageInfo` must be a valid `VkSampler` object.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_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`, the buffer member of any element of `pBufferInfo` is the handle of a non-sparse buffer, then that buffer must be bound completely and contiguously to a single `VkDeviceMemory` object.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`, the `buffer` member of each element of `pBufferInfo` must have been created with `VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT` set.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`, the `buffer` member of each element of `pBufferInfo` must have been created with `VK_BUFFER_USAGE_STORAGE_BUFFER_BIT` set.
If `descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`, the `range` member of each element of `pBufferInfo`, or the effective range if `range` is `VK_WHOLE_SIZE`, must be less than or equal to `VkPhysicalDeviceLimits::maxUniformBufferRange`.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`, the `range` member of each element of `pBufferInfo`, or the effective range if `range` is `VK_WHOLE_SIZE`, must be less than or equal to `VkPhysicalDeviceLimits::maxStorageBufferRange`.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER`, the `VkBuffer` that each element of `pTexelBufferView` was created from must have been created with `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT` set.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER`, the `VkBuffer` that each element of `pTexelBufferView` was created from must have been created with `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT` set.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE` or `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`, the `imageView` member of each element of `pImageInfo` must have been created with the identity swizzle.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, the `imageView` member of each element of `pImageInfo` must have been created with `VK_IMAGE_USAGE_SAMPLED_BIT` set.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, the `imageLayout` member of each element of `pImageInfo` must be a member of the list given in Sampled Image.

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.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`, the `imageView` member of each element of `pImageInfo` must have been created with `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT` set. 

If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, the `imageLayout` member of each element of `pImageInfo` must be a member of the list given in Storage Image. 

If `descriptorType` is `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`, the `imageView` member of each element of `pImageInfo` must have been created with `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT` set.
If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, the `imageView` member of each element of `pImageInfo` must have been created with `VK_IMAGE_USAGE_STORAGE_BIT` set.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLER`, then `dstSet` must not have been allocated with a layout that included immutable samplers for `dstBinding`.

**Valid Usage (Implicit)**

- `sType` must be `VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET`
- `pNext` must be `NULL`
- `descriptorType` must be a valid `VkDescriptorType` value
- `descriptorCount` must be greater than `0`
- Both of `dstSet`, and the elements of `pTexelBufferView` that are valid handles 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 **VK_NULL_HANDLE** or 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 (\texttt{maxUniformBufferRange} or \texttt{maxStorageBufferRange}). This means that \texttt{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 \texttt{maxUniformBufferRange}.

For \texttt{VK\_DESCRIPTOR\_TYPE\_UNIFORM\_BUFFER\_DYNAMIC} and \texttt{VK\_DESCRIPTOR\_TYPE\_STORAGE\_BUFFER\_DYNAMIC} descriptor types, \textit{offset} is the base offset from which the dynamic offset is applied and \textit{range} is the static size used for all dynamic offsets.

### Valid Usage

- VUID-VkDescriptorBufferInfo-offset-00340
  - \textit{offset} must be less than the size of \textit{buffer}

- VUID-VkDescriptorBufferInfo-range-00341
  - If \textit{range} is not equal to \texttt{VK\_WHOLE\_SIZE}, \textit{range} must be greater than 0

- VUID-VkDescriptorBufferInfo-range-00342
  - If \textit{range} is not equal to \texttt{VK\_WHOLE\_SIZE}, \textit{range} must be less than or equal to the size of \textit{buffer} minus \textit{offset}

- VUID-VkDescriptorBufferInfo-buffer-02998
  - If the nullDescriptor feature is not enabled, \textit{buffer} must not be \texttt{VK\_NULL\_HANDLE}

- VUID-VkDescriptorBufferInfo-buffer-02999
  - If \textit{buffer} is \texttt{VK\_NULL\_HANDLE}, \textit{offset} must be zero and \textit{range} must be \texttt{VK\_WHOLE\_SIZE}

### Valid Usage (Implicit)

- VUID-VkDescriptorBufferInfo-buffer-parameter
  - If \textit{buffer} is not \texttt{VK\_NULL\_HANDLE}, \textit{buffer} must be a valid \texttt{VkBuffer} handle

The \texttt{VkDescriptorImageInfo} structure is defined as:

```c
// Provided by VK\_VERSION\_1\_0
typedef struct VkDescriptorImageInfo {
    VkSampler sampler;
    VkImageView imageView;
    VkImageLayout imageLayout;
} VkDescriptorImageInfo;
```

- \texttt{sampler} is a sampler handle, and is used in descriptor updates for types \texttt{VK\_DESCRIPTOR\_TYPE\_SAMPLED\_IMAGE}, \texttt{VK\_DESCRIPTOR\_TYPE\_STORAGE\_IMAGE}, \texttt{VK\_DESCRIPTOR\_TYPE\_COMBINED\_IMAGE\_SAMPLER}, and \texttt{VK\_DESCRIPTOR\_TYPE\_INPUT\_ATTACHMENT}.

- \texttt{imageView} is \texttt{VK\_NULL\_HANDLE} or an image view handle, and is used in descriptor updates for types \texttt{VK\_DESCRIPTOR\_TYPE\_SAMPLED\_IMAGE}, \texttt{VK\_DESCRIPTOR\_TYPE\_STORAGE\_IMAGE}, \texttt{VK\_DESCRIPTOR\_TYPE\_COMBINED\_IMAGE\_SAMPLER}, and \texttt{VK\_DESCRIPTOR\_TYPE\_INPUT\_ATTACHMENT}. 

685
- **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-00343
  **imageView** must not be 2D or 2D array image view created from a 3D image

- VUID-VkDescriptorImageInfo-imageView-01976
  If **imageView** is created from a depth/stencil image, the **aspectMask** used to create the **imageView** must include either `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT` but not both

- VUID-VkDescriptorImageInfo-imageLayout-00344
  **imageLayout** must match the actual **VkImageLayout** of each subresource accessible from **imageView** at the time this descriptor is accessed as defined by the **image layout matching rules**

- VUID-VkDescriptorImageInfo-sampler-01564
  If **sampler** is used and the **VkFormat** of the image is a **multi-planar format**, the image must have been created with `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT`, and the **aspectMask** of the **imageView** must be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT` or (for three-plane formats only) `VK_IMAGE_ASPECT_PLANE_2_BIT`

### Valid Usage (Implicit)

- 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;
    VkDescriptorSet dstSet;
    uint32_t dstBinding;
    uint32_t dstArrayElement;
    uint32_t descriptorCount;
}';
```
VkCopyDescriptorSet;

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcSet`, `srcBinding`, and `srcArrayElement` are the source set, binding, and array element, respectively.
- `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-04886
  - If `srcSet`’s layout was created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` flag set, then `dstSet`’s layout must also have been created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` flag set
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.

If the descriptor pool from which `srcSet` was allocated was created without the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT` flag set, then the descriptor pool from which `dstSet` was allocated must also have been created without the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT` flag set.

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)**

1. **VUID-VkCopyDescriptorSet-sType-sType**
   - `sType` must be `VK_STRUCTURE_TYPE_COPY_DESCRIPTOR_SET`.
2. **VUID-VkCopyDescriptorSet-pNext-pNext**
   - `pNext` must be `NULL`.
3. **VUID-VkCopyDescriptorSet-srcSet-parameter**
   - `srcSet` must be a valid `VkDescriptorSet` handle.
4. **VUID-VkCopyDescriptorSet-dstSet-parameter**
   - `dstSet` must be a valid `VkDescriptorSet` handle.
5. **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
void vkCmdBindDescriptorSets(
    VkCommandBuffer commandBuffer,
    VkPipelineBindPoint pipelineBindPoint,
    VkPipelineLayout layout,
    uint32_t firstSet,
    uint32_t descriptorSetCount,
    const VkDescriptorSet* pDescriptorSets,
    uint32_t dynamicOffsetCount,
    const uint32_t* pDynamicOffsets);
```
• `commandBuffer` is the command buffer that the descriptor sets will be bound to.

• `pipelineBindPoint` is a `VkPipelineBindPoint` indicating 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` causes the sets numbered `[firstSet..firstSet+descriptorSetCount-1]` to use the bindings stored in `pDescriptorSets[0..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.

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, as defined above, and the range of the binding must be less than or equal to the size of the buffer
Valid Usage (Implicit)

- VUID-vkCmdBindDescriptorSets-commandBuffer-parameter
  \texttt{commandBuffer must} be a valid \texttt{VkCommandBuffer} handle

- VUID-vkCmdBindDescriptorSets-pipelineBindPoint-parameter
  \texttt{pipelineBindPoint must} be a valid \texttt{VkPipelineBindPoint} value

- VUID-vkCmdBindDescriptorSets-layout-parameter
  \texttt{layout must} be a valid \texttt{VkPipelineLayout} handle

- VUID-vkCmdBindDescriptorSets-pDescriptorSets-parameter
  \texttt{pDescriptorSets must} be a valid pointer to an array of \texttt{descriptorSetCount} valid \texttt{VkDescriptorSet} handles

- VUID-vkCmdBindDescriptorSets-pDynamicOffsets-parameter
  If \texttt{dynamicOffsetCount} is not 0, \texttt{pDynamicOffsets must} be a valid pointer to an array of \texttt{dynamicOffsetCount \texttt{uint32_t} values}

- VUID-vkCmdBindDescriptorSets-commandBuffer-recording
  \texttt{commandBuffer must} be in the \texttt{recording state}

- VUID-vkCmdBindDescriptorSets-commandBuffer-cmdpool
  The \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from \texttt{must} support graphics, or compute operations

- VUID-vkCmdBindDescriptorSets-descriptorSetCount-arraylength
  \texttt{descriptorSetCount must} be greater than 0

- VUID-vkCmdBindDescriptorSets-commonparent
  Each of \texttt{commandBuffer}, \texttt{layout}, and the elements of \texttt{pDescriptorSets must} have been created, allocated, or retrieved from the same \texttt{VkDevice}

Host Synchronization

- Host access to \texttt{commandBuffer must} be externally synchronized

- Host access to the \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from \texttt{must} be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</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.
- `pValues` is a pointer to an array of `size` bytes containing the new push constant values.

When a command buffer begins recording, all push constant values are undefined.

Push constant values _can_ be updated incrementally, causing shader stages in `stageFlags` to read the new data from `pValues` for push constants modified by this command, while still reading the previous data for push constants not modified by this command. When a _bound pipeline command_ is issued, the bound pipeline's layout _must_ be compatible with the layouts used to set the values of all push constants in the pipeline layout's push constant ranges, as described in _Pipeline Layout Compatibility_. Binding a pipeline with a layout that is not compatible with the push constant layout does not disturb the push constant values.

**Note**
As `stageFlags` needs to include all flags the relevant push constant ranges were created with, any flags that are not supported by the queue family that the `VkCommandPool` used to allocate `commandBuffer` was created on are ignored.

---

**Valid Usage**

- VUID-vkCmdPushConstants-offset-01795
  For each byte in the range specified by `offset` and `size` and for each shader stage in `stageFlags`, there _must_ be a push constant range in `layout` that includes that byte and that
stage
• VUID-vkCmdPushConstants-offset-01796
  For each byte in the range specified by offset and size and for each push constant range that overlaps that byte, stageFlags must include all stages in that push constant range’s VkPushConstantRange::stageFlags

• VUID-vkCmdPushConstants-offset-00368
  offset must be a multiple of 4

• VUID-vkCmdPushConstants-size-00369
  size must be a multiple of 4

• VUID-vkCmdPushConstants-offset-00370
  offset must be less than VkPhysicalDeviceLimits::maxPushConstantsSize

• VUID-vkCmdPushConstants-size-00371
  size must be less than or equal to VkPhysicalDeviceLimits::maxPushConstantsSize minus offset

Valid Usage (Implicit)
• VUID-vkCmdPushConstants-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

• VUID-vkCmdPushConstants-layout-parameter
  layout must be a valid VkPipelineLayout handle

• VUID-vkCmdPushConstants-stageFlags-parameter
  stageFlags must be a valid combination of VkShaderStageFlagBits values

• VUID-vkCmdPushConstants-stageFlags-requiredbitmask
  stageFlags must not be 0

• VUID-vkCmdPushConstants-pValues-parameter
  pValues must be a valid pointer to an array of size bytes

• VUID-vkCmdPushConstants-commandBuffer-recording
  commandBuffer must be in the recording state

• VUID-vkCmdPushConstants-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations

• VUID-vkCmdPushConstants-size-arraylength
  size must be greater than 0

• VUID-vkCmdPushConstants-commonparent
  Both of commandBuffer, and layout must have been created, allocated, or retrieved from the same VkDevice
Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</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.

Valid Usage

- VUID-vkGetBufferDeviceAddress-None-06542
  The `bufferDeviceAddress` feature must be enabled
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;
    const void* pNext;
    VkBuffer buffer;
} VkBufferDeviceAddressInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `buffer` specifies the buffer whose address is being queried.

**Valid Usage**

- VUID-VkBufferDeviceAddressInfo-buffer-02600
  If `buffer` is non-sparse and was not created with the `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT` flag, then it must be bound completely and contiguously to a single `VkDeviceMemory` object

- VUID-VkBufferDeviceAddressInfo-buffer-02601
  `buffer` must have been created with `VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT`

**Valid Usage (Implicit)**

- VUID-VkBufferDeviceAddressInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO`

- VUID-VkBufferDeviceAddressInfo-pNext-pNext
  `pNext` must be `NULL`

- VUID-VkBufferDeviceAddressInfo-buffer-parameter
  `buffer` must be a valid `VkBuffer` handle
To query a 64-bit buffer opaque capture address, call:

```c
// Provided by VK_VERSION_1_2
uint64_t vkGetBufferOpaqueCaptureAddress(
    VkDevice device,
    const VkBufferDeviceAddressInfo* pInfo);
```

- `device` is the logical device that the buffer was created on.
- `pInfo` is a pointer to a `VkBufferDeviceAddressInfo` structure specifying the buffer to retrieve an address for.

The 64-bit return value is an opaque capture address of the start of `pInfo->buffer`.

If the buffer was created with a non-zero value of `VkBufferOpaqueCaptureAddressCreateInfo::opaqueCaptureAddress` the return value **must** be the same address.

### Valid Usage

- VUID-vkGetBufferOpaqueCaptureAddress-None-03326
  The `bufferDeviceAddress` feature **must** be enabled
- VUID-vkGetBufferOpaqueCaptureAddress-device-03327
  If `device` was created with multiple physical devices, then the `bufferDeviceAddressMultiDevice` feature **must** be enabled

### Valid Usage (Implicit)

- VUID-vkGetBufferOpaqueCaptureAddress-device-parameter
  `device` **must** be a valid `VkDevice` handle
- VUID-vkGetBufferOpaqueCaptureAddress-pInfo-parameter
  `pInfo` **must** be a valid pointer to a valid `VkBufferDeviceAddressInfo` structure
Chapter 15. Shader Interfaces

When a pipeline is created, the set of shaders specified in the corresponding \texttt{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.

Interface definitions make use of the following SPIR-V decorations:

- \texttt{DescriptorSet} and \texttt{Binding}
- \texttt{Location, Component, and Index}
- \texttt{Flat, NoPerspective, Centroid, and Sample}
- \texttt{Block and BufferBlock}
- \texttt{InputAttachmentIndex}
- \texttt{Offset, ArrayStride, and MatrixStride}
- \texttt{BuiltIn}

This specification describes valid uses for Vulkan of these decorations. Any other use of one of these decorations is invalid, with the exception that, when using SPIR-V versions 1.4 and earlier: \texttt{Block}, \texttt{BufferBlock}, \texttt{Offset}, \texttt{ArrayStride}, and \texttt{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 \texttt{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 \textit{interfaces} are listed as operands to the
OpEntryPoint instruction and are declared with the Input or Output storage classes, respectively, in the SPIR-V module. These generally form the interfaces between consecutive shader stages, regardless of any non-shader stages between the consecutive shader stages.

There are two classes of variables that can be matched between shader stages, built-in variables and user-defined variables. Each class has a different set of matching criteria.

Output variables of a shader stage have undefined values until the shader writes to them or uses the Initializer operand when declaring the variable.

### 15.1.1. Built-in Interface Block

Shader built-in variables meeting the following requirements define the built-in interface block. They must

- be explicitly declared (there are no implicit built-ins),
- be identified with a BuiltIn decoration,
- form object types as described in the Built-in Variables section, and
- be declared in a block whose top-level members are the built-ins.

There must be no more than one built-in interface block per shader per interface.

Built-ins must not have any Location or Component decorations.

### 15.1.2. User-defined Variable Interface

The non-built-in variables listed by OpEntryPoint with the Input or Output storage class form the user-defined variable interface. These must have SPIR-V numerical types or, recursively, composite types of such types. By default, the components of such types have a width of 32 or 64 bits. If an implementation supports storageInputOutput16, components can also have a width of 16 bits. These variables must be identified with a Location decoration and can also be identified with a Component decoration.

### 15.1.3. Interface Matching

An output variable, block, or structure member in a given shader stage has an interface match with an input variable, block, or structure member in a subsequent shader stage if they both adhere to the following conditions:

- They have equivalent decorations, other than:
  - Interpolation decorations
  - one is not decorated with Component and the other is declared with a Component of 0
- 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 locations are consumed by a given user-variable type. As mentioned above, some inputs and outputs have an additional level of arrayness relative to other shader inputs and outputs. This outer array level is removed from the type before considering how many locations the type consumes.

The `Location` value specifies an interface slot comprised of a 32-bit four-component vector conveyed between stages. The `Component` specifies components within these vector locations. Only types with widths of 16, 32 or 64 are supported in shader interfaces.

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

- 16-bit scalar and vector types, and
- 32-bit scalar and vector types, and
- 64-bit scalar and 2-component vector types.

64-bit three- and four-component vectors consume two consecutive locations.

If a declared input or output is an array of size `n` and each element takes `m` locations, it will be assigned `m × n` consecutive locations starting with the location specified.

If the declared input or output is an `n × m` 16-, 32- or 64-bit matrix, it will be assigned multiple locations starting with the location specified. The number of locations assigned for each matrix will be the same as for an `n`-element array of `m`-component vectors.

An `OpVariable` with a structure type that is not a block **must** be decorated with a `Location`.

When an `OpVariable` with a structure type (either block or non-block) is decorated with a `Location`, the members in the structure type **must** not be decorated with a `Location`. The `OpVariable`'s members are assigned consecutive locations in declaration order, starting from the first member, which is assigned the location decoration from the `OpVariable`.

When a block-type `OpVariable` is declared without a `Location` decoration, each member in its structure type **must** be decorated with a `Location`. Types nested deeper than the top-level members
must not have Location decorations.

The locations consumed by block and structure members are determined by applying the rules above in a depth-first traversal of the instantiated members as though the structure or block member were declared as an input or output variable of the same type.

Any two inputs listed as operands on the same OpEntryPoint must not be assigned the same location, either explicitly or implicitly. Any two outputs listed as operands on the same OpEntryPoint must not be assigned the same location, either explicitly or implicitly.

The number of input and output locations available for a shader input or output interface are limited, and dependent on the shader stage as described in Shader Input and Output Locations. All variables in both the built-in interface block and the user-defined variable interface count against these limits. Each effective Location must have a value less than the number of locations available for the given interface, as specified in the "Locations Available" column in Shader Input and Output Locations.

<table>
<thead>
<tr>
<th>Shader Interface</th>
<th>Locations Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex input</td>
<td>maxVertexInputAttributes</td>
</tr>
<tr>
<td>vertex output</td>
<td>maxVertexOutputComponents / 4</td>
</tr>
<tr>
<td>tessellation control input</td>
<td>maxTessellationControlPerVertexInputComponents / 4</td>
</tr>
<tr>
<td>tessellation control output</td>
<td>maxTessellationControlPerVertexOutputComponents / 4</td>
</tr>
<tr>
<td>tessellation evaluation input</td>
<td>maxTessellationEvaluationInputComponents / 4</td>
</tr>
<tr>
<td>tessellation evaluation output</td>
<td>maxTessellationEvaluationOutputComponents / 4</td>
</tr>
<tr>
<td>geometry input</td>
<td>maxGeometryInputComponents / 4</td>
</tr>
<tr>
<td>geometry output</td>
<td>maxGeometryOutputComponents / 4</td>
</tr>
<tr>
<td>fragment input</td>
<td>maxFragmentInputComponents / 4</td>
</tr>
<tr>
<td>fragment output</td>
<td>maxFragmentOutputAttachments</td>
</tr>
</tbody>
</table>

15.1.5. Component Assignment

The Component decoration allows the Location to be more finely specified for scalars and vectors, down to the individual components within a location that are consumed. The components within a location are 0, 1, 2, and 3. A variable or block member starting at component N will consume components N, N+1, N+2, ... up through its size. For 16-, and 32-bit types, it is invalid if this sequence of components gets larger than 3. A scalar 64-bit type will consume two of these components in sequence, and a two-component 64-bit vector type will consume all four components available within a location. A three- or four-component 64-bit vector type must not specify a Component decoration. A three-component 64-bit vector type will consume all four components of the first location and components 0 and 1 of the second location. This leaves components 2 and 3 available for other component-qualified declarations.
A scalar or two-component 64-bit data type must not specify a Component decoration of 1 or 3. A Component decoration must not be specified for any type that is not a scalar or vector.

15.2. Vertex Input Interface

When the vertex stage is present in a pipeline, the vertex shader input variables form an interface with the vertex input attributes. The vertex shader input variables are matched by the Location and Component decorations to the vertex input attributes specified in the pVertexInputState member of the VkGraphicsPipelineCreateInfo structure.

The vertex shader input variables listed by OpEntryPoint with the Input storage class form the vertex input interface. These variables must be identified with a Location decoration and can also be identified with a Component decoration.

For the purposes of interface matching: variables declared without a Component decoration are considered to have a Component decoration of zero. The number of available vertex input locations is given by the maxVertexInputAttributes member of the VkPhysicalDeviceLimits structure.

See Attribute Location and Component Assignment for details.

All vertex shader inputs declared as above must have a corresponding attribute and binding in the pipeline.

15.3. Fragment Output Interface

When the fragment stage is present in a pipeline, the fragment shader outputs form an interface with the output attachments defined by a render pass instance. The fragment shader output variables are matched by the Location and Component decorations to specified color attachments.

The fragment shader output variables listed by OpEntryPoint with the Output storage class form the fragment output interface. These variables must be identified with a Location decoration. They can also be identified with a Component decoration and/or an Index decoration. For the purposes of interface matching: variables declared without a Component decoration are considered to have a Component decoration of zero, and variables declared without an Index decoration are considered to have an Index decoration of zero.

A fragment shader output variable identified with a Location decoration of i is associated with the color attachment indicated by pColorAttachments[i]. Values are written to those attachments after passing through the blending unit as described in Blending, if enabled. Locations are consumed as described in Location Assignment. The number of available fragment output locations is given by the maxFragmentOutputAttachments member of the VkPhysicalDeviceLimits structure.

Components of the output variables are assigned as described in Component Assignment. Output components identified as 0, 1, 2, and 3 will be directed to the R, G, B, and A inputs to the blending unit, respectively, or to the output attachment if blending is disabled. If two variables are placed within the same location, they must have the same underlying type (floating-point or integer). The input values to blending or color attachment writes are undefined for components which do not correspond to a fragment shader output.
Fragment outputs identified with an Index of zero are directed to the first input of the blending unit associated with the corresponding Location. Outputs identified with an Index of one are directed to the second input of the corresponding blending unit.

No component aliasing of output variables is allowed, that is there must not be two output variables which have the same location, component, and index, either explicitly declared or implied.

Output values written by a fragment shader must be declared with either OpTypeFloat or OpTypeInt, and a Width of 32. If storageInputOutput16 is supported, output values written by a fragment shader can be also declared with either OpTypeFloat or OpTypeInt and a Width of 16. Composites of these types are also permitted. If the color attachment has a signed or unsigned normalized fixed-point format, color values are assumed to be floating-point and are converted to fixed-point as described in Conversion from Floating-Point to Normalized Fixed-Point; If the color attachment has an integer format, color values are assumed to be integers and converted to the bit-depth of the target. Any value that cannot be represented in the attachment's format is undefined. For any other attachment format no conversion is performed. If the type of the values written by the fragment shader do not match the format of the corresponding color attachment, the resulting values are undefined for those components.

15.4. Fragment Input Attachment Interface

When a fragment stage is present in a pipeline, the fragment shader subpass inputs form an interface with the input attachments of the current subpass. The fragment shader subpass input variables are matched by InputAttachmentIndex decorations to the input attachments specified in the pInputAttachments array of the VkSubpassDescription structure describing the subpass that the fragment shader is executed in.

The fragment shader subpass input variables with the UniformConstant storage class and a decoration of InputAttachmentIndex that are statically used by OpEntryPoint form the fragment input attachment interface. These variables must be declared with a type of OpTypeImage, a Dim operand of SubpassData, an Arrayed operand of 0, and a Sampled operand of 2. The MS operand of the OpTypeImage must be 0 if the samples field of the corresponding VkAttachmentDescription is VK_SAMPLE_COUNT_1_BIT and 1 otherwise.

A subpass input variable identified with an InputAttachmentIndex decoration of i reads from the input attachment indicated by pInputAttachments[i] member of VkSubpassDescription. If the subpass input variable is declared as an array of size N, it consumes N consecutive input attachments, starting with the index specified. There must not be more than one input variable with the same InputAttachmentIndex whether explicitly declared or implied by an array declaration. The number of available input attachment indices is given by the maxPerStageDescriptorInputAttachments member of the VkPhysicalDeviceLimits structure.

Variables identified with the InputAttachmentIndex must only be used by a fragment stage. The basic data type (floating-point, integer, unsigned integer) of the subpass input must match the basic format of the corresponding input attachment, or the values of subpass loads from these variables are undefined.

See Input Attachment for more details.
15.5. Shader Resource Interface

When a shader stage accesses buffer or image resources, as described in the Resource Descriptors section, the shader resource variables must be matched with the pipeline layout that is provided at pipeline creation time.

The set of shader variables that form the shader resource interface for a stage are the variables statically used by that stage’s OpEntryPoint with a storage class of Uniform, UniformConstant, StorageBuffer, or PushConstant. For the fragment shader, this includes the fragment input attachment interface.

The shader resource interface consists of two sub-interfaces: the push constant interface and the descriptor set interface.

15.5.1. Push Constant Interface

The shader variables defined with a storage class of PushConstant that are statically used by the shader entry points for the pipeline define the push constant interface. They must be:

- typed as OpTypeStruct,
- identified with a Block decoration, and
- laid out explicitly using the Offset, ArrayStride, and MatrixStride decorations as specified in Offset and Stride Assignment.

There must be no more than one push constant block statically used per shader entry point.

Each statically used member of a push constant block must be placed at an Offset such that the entire member is entirely contained within the VkPushConstantRange for each OpEntryPoint that uses it, and the stageFlags for that range must specify the appropriate VkShaderStageFlagBits for that stage. The Offset decoration for any member of a push constant block must not cause the space required for that member to extend outside the range [0, maxPushConstantsSize).

Any member of a push constant block that is declared as an array must only be accessed with dynamically uniform indices.

15.5.2. Descriptor Set Interface

The descriptor set interface is comprised of the shader variables with the storage class of StorageBuffer, Uniform or UniformConstant (including the variables in the fragment input attachment interface) that are statically used by the shader entry points for the pipeline.

These variables must have DescriptorSet and Binding decorations specified, which are assigned and matched with the VkDescriptorSetLayout objects in the pipeline layout as described in DescriptorSet and Binding Assignment.

The Image Format of an OpTypeImage declaration must not be Unknown, for variables which are used for OpImageRead, OpImageSparseRead, or OpImageWrite operations, except under the following conditions:
• For `OpImageWrite`, if the image format is listed in the `storage without format` list and if the `shaderStorageImageWriteWithoutFormat` feature is enabled and the shader module declares the `StorageImageWriteWithoutFormat` capability.

• For `OpImageRead` or `OpImageSparseRead`, if the image format is listed in the `storage without format` list and if the `shaderStorageImageReadWithoutFormat` feature is enabled and the shader module declares the `StorageImageReadWithoutFormat` capability.

• For `OpImageRead`, if `Dim` is `SubpassData` (indicating a read from an input attachment).

The `Image Format` of an `OpTypeImage` declaration must not be `Unknown`, for variables which are used for `OpAtomic*` operations.

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

• typed as `OpTypeStruct`, or an array of this type,
• identified with a `Block` or `BufferBlock` decoration, and
• laid out explicitly using the `Offset`, `ArrayStride`, and `MatrixStride` decorations as specified in `Offset and Stride Assignment`.

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

• typed as `OpTypeStruct`, or an array of this type,
• identified with a `Block` decoration, and
• laid out explicitly using the `Offset`, `ArrayStride`, and `MatrixStride` decorations as specified in `Offset and Stride Assignment`.

The `Offset` decoration for any member of a `Block`-decorated variable in the `Uniform` storage class must not cause the space required for that variable to extend outside the range `[0, maxUniformBufferRange)`. The `Offset` decoration for any member of a `Block`-decorated variable in the `StorageBuffer` storage class must not cause the space required for that variable to extend outside the range `[0, maxStorageBufferRange)`.  

Variables identified with 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_bC_a conversion, it must be indexed only by
constant integral expressions when aggregated into arrays in shader code, irrespective of the
shaderSampledImageArrayDynamicIndexing feature.

Table 16. 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 17. Shader Resource and Storage Class Correspondence

<table>
<thead>
<tr>
<th>Resource type</th>
<th>Storage Class</th>
<th>Type¹</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</td>
<td>OpTypeSampler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OpTypeImage (Sampled=1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OpTypeImage</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>storage buffer</td>
<td>Uniform</td>
<td>OpTypeStruct</td>
<td>BufferBlock, Offset, (ArrayStride), (MatrixStride)</td>
</tr>
<tr>
<td></td>
<td>StorageBuffer</td>
<td>OpTypeStruct</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>
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.

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 18. 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>uniform buffer dynamic</td>
</tr>
<tr>
<td>maxPerStageDescriptorStorageBuffers or maxPerStageDescriptorUpdateAfterBindStorageBuffers</td>
<td>storage buffer</td>
</tr>
<tr>
<td>maxPerStageDescriptorStorageBuffers or maxPerStageDescriptorUpdateAfterBindStorageBuffers</td>
<td>storage buffer dynamic</td>
</tr>
<tr>
<td>maxPerStageDescriptorInputAttachments or 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.

The scalar alignment of the type of an OpTypeStruct member is defined recursively as follows:

- A scalar of size N has a scalar alignment of N.
- A vector or matrix type has a scalar alignment equal to that of its component type.
- An array type has a scalar alignment equal to that of its element type.
- A structure has a scalar alignment equal to the largest scalar alignment of any of its members.

The base alignment of the type of an OpTypeStruct member is defined recursively as follows:

- A scalar has a base alignment equal to its scalar alignment.
- A two-component vector has a base alignment equal to twice its scalar alignment.
- A three- or four-component vector has a base alignment equal to four times its scalar alignment.
- An array has a base alignment equal to the base alignment of its element type.
- A structure has a base alignment equal to the largest base alignment of any of its members. An empty structure has a base alignment equal to the size of the smallest scalar type permitted by the capabilities declared in the SPIR-V module. (e.g., for a 1 byte aligned empty struct in the StorageBuffer storage class, StorageBuffer8BitAccess or UniformAndStorageBuffer8BitAccess must be declared in the SPIR-V module.)
- A row-major matrix of C columns has a base alignment equal to the base alignment of a vector of C matrix components.
- A column-major matrix has a base alignment equal to the base alignment of the matrix column type.

The extended alignment of the type of an OpTypeStruct member is similarly defined as follows:

- A scalar, vector or matrix type has an extended alignment equal to its base alignment.
- An array or structure type has an extended alignment equal to the largest extended alignment of any of its members, rounded up to a multiple of 16.

A member is defined to improperly straddle if either of the following are true:

- It is a vector with total size less than or equal to 16 bytes, and has Offset decorations placing its first byte at F and its last byte at L, where 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 or an array and the next multiple of the alignment of that structure or array.

```
Note
The std430 layout in GLSL satisfies these rules for types using the base alignment. The std140 layout satisfies the rules for types using the extended alignment.
```
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 `MeshNV, Vertex, Fragment, TessellationControl, TessellationEvaluation, or Geometry Execution Model`.

- **VUID-ClipDistance-ClipDistance-04188**
  The variable decorated with `ClipDistance` within the `MeshNV or Vertex Execution Model` must be declared using the `Output Storage Class`.

- **VUID-ClipDistance-ClipDistance-04189**
  The variable decorated with `ClipDistance` within the `Fragment Execution Model` must be declared using the `Input Storage Class`.

- **VUID-ClipDistance-ClipDistance-04190**
  The variable decorated with `ClipDistance` within the `TessellationControl, TessellationEvaluation, or Geometry Execution Model` must not be declared in a `Storage Class` other than `Input` or `Output`.

- **VUID-ClipDistance-ClipDistance-04191**
  The variable decorated with `ClipDistance` must be declared as an array of 32-bit floating-point values.

**CullDistance**

Decorating a variable with the `CullDistance` built-in decoration will make that variable contain the mechanism for controlling user culling. If any member of this array is assigned a negative value for all vertices belonging to a primitive, then the primitive is discarded before rasterization.
**Note**

In fragment shaders, the values of the `CullDistance` array are linearly interpolated across each primitive.

**Note**

If `CullDistance` decorates an input variable, that variable will contain the corresponding value from the `CullDistance` decorated output variable from the previous shader stage.

### Valid Usage

- **VUID-CullDistance-CullDistance-04196**
  - The `CullDistance` decoration **must** be used only within the MeshNV, Vertex, Fragment, TessellationControl, TessellationEvaluation, or Geometry Execution Model.

- **VUID-CullDistance-CullDistance-04197**
  - The variable decorated with `CullDistance` within the MeshNV or Vertex Execution Model **must** be declared using the Output Storage Class.

- **VUID-CullDistance-CullDistance-04198**
  - The variable decorated with `CullDistance` within the Fragment Execution Model **must** be declared using the Input Storage Class.

- **VUID-CullDistance-CullDistance-04199**
  - The variable decorated with `CullDistance` within the TessellationControl, TessellationEvaluation, or Geometry Execution Model **must** not be declared using a Storage Class other than Input or Output.

- **VUID-CullDistance-CullDistance-04200**
  - The variable decorated with `CullDistance` **must** be declared as an array of 32-bit floating-point values.

### DeviceIndex

The `DeviceIndex` decoration **can** be applied to a shader input which will be filled with the device index of the physical device that is executing the current shader invocation. This value will be in the range \(0, \max(1, \text{physicalDeviceCount})\), where `physicalDeviceCount` is the `physicalDeviceCount` member of `VkDeviceGroupDeviceCreateInfo`.

### Valid Usage

- **VUID-DeviceIndex-DeviceIndex-04205**
  - The variable decorated with `DeviceIndex` **must** be declared using the Input Storage Class.

- **VUID-DeviceIndex-DeviceIndex-04206**
  - The variable decorated with `DeviceIndex` **must** be declared as a scalar 32-bit integer value.
DrawIndex

Decorating a variable with the DrawIndex built-in will make that variable contain the integer value corresponding to the zero-based index of the drawing command that invoked the current vertex shader invocation. For indirect drawing commands, DrawIndex begins at zero and increments by one for each drawing command executed. The number of drawing commands is given by the drawCount parameter. For direct drawing commands, DrawIndex is always zero. DrawIndex is dynamically uniform.

Valid Usage

• VUID-DrawIndex-DrawIndex-04207
  The DrawIndex decoration must be used only within the Vertex, 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

FragStencilRefEXT

Decorating a variable with the FragStencilRefEXT built-in decoration will make that variable contain the new stencil reference value for all samples covered by the fragment. This value will be used as the stencil reference value used in stencil testing.

To write to FragStencilRefEXT, a shader must declare the StencilRefReplacingEXT execution mode. If a shader declares the StencilRefReplacingEXT execution mode and there is an execution path through the shader that does not set FragStencilRefEXT, then the fragment's stencil reference value is undefined for executions of the shader that take that path.

Only the least significant $s$ bits of the integer value of the variable decorated with FragStencilRefEXT are considered for stencil testing, where $s$ is the number of bits in the stencil framebuffer attachment, and higher order bits are discarded.

See fragment shader stencil reference replacement for more details.

Valid Usage

- VUID-FragStencilRefEXT-FragStencilRefEXT-04223
  The FragStencilRefEXT decoration must be used only within the Fragment Execution Model

- VUID-FragStencilRefEXT-FragStencilRefEXT-04224
  The variable decorated with FragStencilRefEXT must be declared using the Output Storage Class

- VUID-FragStencilRefEXT-FragStencilRefEXT-04225
The variable decorated with `FragStencilRefEXT` must be declared as a scalar integer value

**FrontFacing**

Decorating a variable with the `FrontFacing` built-in decoration will make that variable contain whether the fragment is front or back facing. This variable is non-zero if the current fragment is considered to be part of a front-facing polygon primitive or of a non-polygon primitive and is zero if the fragment is considered to be part of a back-facing polygon primitive.

**Valid Usage**

- VUID-FrontFacing-FrontFacing-04229
  The `FrontFacing` decoration must be used only within the Fragment Execution Model
- VUID-FrontFacing-FrontFacing-04230
  The variable decorated with `FrontFacing` must be declared using the Input Storage Class
- VUID-FrontFacing-FrontFacing-04231
  The variable decorated with `FrontFacing` must be declared as a boolean value

**FullyCoveredEXT**

Decorating a variable with the `FullyCoveredEXT` built-in decoration will make that variable indicate whether the fragment area is fully covered by the generating primitive. This variable is non-zero if conservative rasterization is enabled and the current fragment area is fully covered by the generating primitive, and is zero if the fragment is not covered or partially covered, or conservative rasterization is disabled.

**Valid Usage**

- VUID-FullyCoveredEXT-FullyCoveredEXT-04232
  The `FullyCoveredEXT` decoration must be used only within the Fragment Execution Model
- VUID-FullyCoveredEXT-FullyCoveredEXT-04233
  The variable decorated with `FullyCoveredEXT` must be declared using the Input Storage Class
- VUID-FullyCoveredEXT-FullyCoveredEXT-04234
  The variable decorated with `FullyCoveredEXT` must be declared as a boolean value
- VUID-FullyCoveredEXT-conservativeRasterizationPostDepthCoverage-04235
  If `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::conservativeRasterizationPostDepthCoverage` is not supported the PostDepthCoverage Execution Mode must not be declared, when a variable with the `FullyCoveredEXT` decoration is declared

**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, 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 `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 `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.

---

### 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, 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**
  
  (4283)
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, 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, 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 Execution Model`

- VUID-NumWorkgroups-NumWorkgroups-04297
  The variable decorated with `NumWorkgroups` **must** be declared using the `Input Storage Class`

- VUID-NumWorkgroups-NumWorkgroups-04298
  The variable decorated with `NumWorkgroups` **must** be declared as a three-component vector of 32-bit integer values

**PatchVertices**

Decorating a variable with the `PatchVertices` built-in decoration will make that variable contain the number of vertices in the input patch being processed by the shader. In a Tessellation Control Shader, this is the same as the name:patchControlPoints member of `VkPipelineTessellationStateCreateInfo`. In a Tessellation Evaluation Shader, `PatchVertices` is equal to the tessellation control output patch size. When the same shader is used in different pipelines where the patch sizes are configured differently, the value of the `PatchVertices` variable will also differ.

**Valid Usage**

- VUID-PatchVertices-PatchVertices-04308
  The `PatchVertices` decoration **must** be used only within the `TessellationControl` or `TessellationEvaluation Execution Model`

- VUID-PatchVertices-PatchVertices-04309
  The variable decorated with `PatchVertices` **must** be declared using the `Input Storage Class`

- VUID-PatchVertices-PatchVertices-04310
  The variable decorated with `PatchVertices` **must** be declared as a scalar 32-bit integer value

**PointCoord**

Decorating a variable with the `PointCoord` built-in decoration will make that variable contain the coordinate of the current fragment within the point being rasterized, normalized to the size of the point with origin in the upper left corner of the point, as described in Basic Point Rasterization. If the primitive the fragment shader invocation belongs to is not a point, then the variable decorated with `PointCoord` contains an undefined value.

**Note**

Depending on how the point is rasterized, `PointCoord may` never reach (0,0) or (1,1).
Valid Usage

- VUID-PointCoord-PointCoord-04311
  The PointCoord decoration must be used only within the Fragment Execution Model

- VUID-PointCoord-PointCoord-04312
  The variable decorated with PointCoord must be declared using the Input Storage Class

- VUID-PointCoord-PointCoord-04313
  The variable decorated with PointCoord must be declared as a two-component vector of 32-bit floating-point values

PointSize

Decorating a variable with the PointSize built-in decoration will make that variable contain the size of point primitives. The value written to the variable decorated with PointSize by the last pre-rasterization shader stage in the pipeline is used as the framebuffer-space size of points produced by rasterization.

Note

When PointSize decorates a variable in the Input Storage Class, it contains the data written to the output variable decorated with PointSize from the previous shader stage.

Valid Usage

- VUID-PointSize-PointSize-04314
  The PointSize decoration must be used only within the MeshNV, Vertex, TessellationControl, TessellationEvaluation, or Geometry Execution Model

- VUID-PointSize-PointSize-04315
  The variable decorated with PointSize within the 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 `MeshNV, Vertex, TessellationControl, TessellationEvaluation, or Geometry Execution Model`.

- **VUID-Position-Position-04319**
  
  The variable decorated with `Position` within `MeshNV` or `Vertex Execution Model` **must** be declared using the Output Storage Class.

- **VUID-Position-Position-04320**
  
  The variable decorated with `Position` within `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 `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 `MeshNV` Execution Model and a variable decorated with `PrimitiveId` is read from `Fragment` shader, then the `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 `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.

- **VUID-PrimitiveId-PrimitiveId-04336**
  The variable decorated with `PrimitiveId` within the `MeshNV` Execution Model **must** be declared using the `Output Storage` Class.

- **VUID-PrimitiveId-PrimitiveId-04337**
  The variable decorated with `PrimitiveId` **must** be declared as a scalar 32-bit integer value.

**PrimitiveShadingRateKHR**

Decorating a variable with the `PrimitiveShadingRateKHR` built-in decoration will make that variable contain the **primitive fragment shading rate**.

The value written to the variable decorated with `PrimitiveShadingRateKHR` by the last **pre-rasterization shader stage** in the pipeline is used as the **primitive fragment shading rate**. Outputs...
in previous shader stages are ignored.

If the last active pre-rasterization shader stage shader entry point’s interface does not include a variable decorated with `PrimitiveShadingRateKHR`, then it is as if the shader specified a fragment shading rate value of 0, indicating a horizontal and vertical rate of 1 pixel.

If a shader has `PrimitiveShadingRateKHR` in the output interface and there is an execution path through the shader that does not write to it, its value is undefined for executions of the shader that take that path.

**Valid Usage**

- VUID-PrimitiveShadingRateKHR-PrimitiveShadingRateKHR-04484
  The `PrimitiveShadingRateKHR` decoration must be used only within the MeshNV, Vertex, or Geometry Execution Model

- VUID-PrimitiveShadingRateKHR-PrimitiveShadingRateKHR-04485
  The variable decorated with `PrimitiveShadingRateKHR` must be declared using the Output Storage Class

- VUID-PrimitiveShadingRateKHR-PrimitiveShadingRateKHR-04486
  The variable decorated with `PrimitiveShadingRateKHR` must be declared as a scalar 32-bit integer value

- VUID-PrimitiveShadingRateKHR-PrimitiveShadingRateKHR-04487
  The value written to `PrimitiveShadingRateKHR` must include no more than one of `Vertical2Pixels` and `Vertical4Pixels`

- VUID-PrimitiveShadingRateKHR-PrimitiveShadingRateKHR-04488
  The value written to `PrimitiveShadingRateKHR` must include no more than one of `Horizontal2Pixels` and `Horizontal4Pixels`

- VUID-PrimitiveShadingRateKHR-PrimitiveShadingRateKHR-04489
  The value written to `PrimitiveShadingRateKHR` must not have any bits set other than those defined by Fragment Shading Rate Flags enumerants in the SPIR-V specification

**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
SampleMask

Decorating a variable with the SampleMask built-in decoration will make any variable contain the sample mask for the current fragment shader invocation.

A variable in the Input storage class decorated with SampleMask will contain a bitmask of the set of samples covered by the primitive generating the fragment during rasterization. It has a sample bit set if and only if the sample is considered covered for this fragment shader invocation. SampleMask[] is an array of integers. Bits are mapped to samples in a manner where bit B of mask M (SampleMask[M]) corresponds to sample $32 \times M + B$.

A variable in the Output storage class decorated with SampleMask is an array of integers forming a bit array in a manner similar to an input variable decorated with SampleMask, but where each bit represents coverage as computed by the shader. This computed SampleMask is combined with the generated coverage mask in the multisample coverage operation.

Variables decorated with SampleMask must be either an unsized array, or explicitly sized to be no larger than the implementation-dependent maximum sample-mask (as an array of 32-bit elements), determined by the maximum number of samples.

If a fragment shader entry point’s interface includes an output variable decorated with SampleMask, the sample mask will be undefined for any array elements of any fragment shader invocations that fail to assign a value. If a fragment shader entry point’s interface does not include an output variable decorated with SampleMask, the sample mask has no effect on the processing of a fragment.

Valid Usage

- VUID-SampleMask-SampleMask-04357
  The SampleMask decoration must be used only within the Fragment Execution Model

- VUID-SampleMask-SampleMask-04358
  The variable decorated with SampleMask must be declared using the Input or Output Storage Class

- VUID-SampleMask-SampleMask-04359
  The variable decorated with SampleMask must be declared as an array of 32-bit integer values

SamplePosition

Decorating a variable with the SamplePosition built-in decoration will make that variable contain the sub-pixel position of the sample being shaded. The top left of the pixel is considered to be at coordinate (0,0) and the bottom right of the pixel is considered to be at coordinate (1,1).

If a fragment shader entry point’s interface includes an input variable decorated with SamplePosition, Sample Shading is considered enabled with a minSampleShading value of 1.0.
If the current pipeline uses custom sample locations the value of any variable decorated with the SamplePosition built-in decoration is undefined.

**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

**ShadingRateKHR**

Decorating a variable with the ShadingRateKHR built-in decoration will make that variable contain the fragment shading rate for the current fragment invocation.

**Valid Usage**

- VUID-ShadingRateKHR-ShadingRateKHR-04490
  The ShadingRateKHR decoration **must** be used only within the Fragment Execution Model

- VUID-ShadingRateKHR-ShadingRateKHR-04491
  The variable decorated with ShadingRateKHR **must** be declared using the Input Storage Class

- VUID-ShadingRateKHR-ShadingRateKHR-04492
  The variable decorated with ShadingRateKHR **must** be declared as a scalar 32-bit integer value

**SubgroupId**

Decorating a variable with the SubgroupId built-in decoration will make that variable contain the index of the subgroup within the local workgroup. This variable is in range [0, NumSubgroups-1].

**Valid Usage**

- VUID-SubgroupId-SubgroupId-04367
  The SubgroupID decoration **must** be used only within the GLCompute, MeshNV, or TaskNV Execution Model

- VUID-SubgroupId-SubgroupId-04368
  The variable decorated with SubgroupId **must** be declared using the Input Storage Class

- VUID-SubgroupId-SubgroupId-04369
  The variable decorated with SubgroupId **must** be declared as a scalar 32-bit integer value
SubgroupEqMask

Decorating a variable with the SubgroupEqMask builtin decoration will make that variable contain the subgroup mask of the current subgroup invocation. The bit corresponding to the SubgroupLocalInvocationId is set in the variable decorated with SubgroupEqMask. All other bits are set to zero.

SubgroupEqMaskKHR is an alias of SubgroupEqMask.

Valid Usage

- VUID-SubgroupEqMask-SubgroupEqMask-04370
  The variable decorated with SubgroupEqMask must be declared using the Input Storage Class

- VUID-SubgroupEqMask-SubgroupEqMask-04371
  The variable decorated with SubgroupEqMask must be declared as a four-component vector of 32-bit integer values

SubgroupGeMask

Decorating a variable with the SubgroupGeMask builtin decoration will make that variable contain the subgroup mask of the current subgroup invocation. The bits corresponding to the invocations greater than or equal to SubgroupLocalInvocationId through SubgroupSize-1 are set in the variable decorated with SubgroupGeMask. All other bits are set to zero.

SubgroupGeMaskKHR is an alias of SubgroupGeMask.

Valid Usage

- VUID-SubgroupGeMask-SubgroupGeMask-04372
  The variable decorated with SubgroupGeMask must be declared using the Input Storage Class

- VUID-SubgroupGeMask-SubgroupGeMask-04373
  The variable decorated with SubgroupGeMask must be declared as a four-component vector of 32-bit integer values

SubgroupGtMask

Decorating a variable with the SubgroupGtMask builtin decoration will make that variable contain the subgroup mask of the current subgroup invocation. The bits corresponding to the invocations greater than SubgroupLocalInvocationId through SubgroupSize-1 are set in the variable decorated with SubgroupGtMask. All other bits are set to zero.

SubgroupGtMaskKHR is an alias of SubgroupGtMask.

Valid Usage

- VUID-SubgroupGtMask-SubgroupGtMask-04374
The variable decorated with `SubgroupGtMask` **must** be declared using the Input Storage Class

- VUID-SubgroupGtMask-SubgroupGtMask-04375
  The variable decorated with `SubgroupGtMask` **must** be declared as a four-component vector of 32-bit integer values

**SubgroupLeMask**

Decorating a variable with the `SubgroupLeMask` builtin decoration will make that variable contain the *subgroup mask* of the current subgroup invocation. The bits corresponding to the invocations less than or equal to `SubgroupLocalInvocationId` are set in the variable decorated with `SubgroupLeMask`. All other bits are set to zero.

`SubgroupLeMaskKHR` is an alias of `SubgroupLeMask`.

### Valid Usage

- VUID-SubgroupLeMask-SubgroupLeMask-04376
  The variable decorated with `SubgroupLeMask` **must** be declared using the Input Storage Class

- VUID-SubgroupLeMask-SubgroupLeMask-04377
  The variable decorated with `SubgroupLeMask` **must** be declared as a four-component vector of 32-bit integer values

**SubgroupLtMask**

Decorating a variable with the `SubgroupLtMask` builtin decoration will make that variable contain the *subgroup mask* of the current subgroup invocation. The bits corresponding to the invocations less than `SubgroupLocalInvocationId` are set in the variable decorated with `SubgroupLtMask`. All other bits are set to zero.

`SubgroupLtMaskKHR` is an alias of `SubgroupLtMask`.

### Valid Usage

- VUID-SubgroupLtMask-SubgroupLtMask-04378
  The variable decorated with `SubgroupLtMask` **must** be declared using the Input Storage Class

- VUID-SubgroupLtMask-SubgroupLtMask-04379
  The variable decorated with `SubgroupLtMask` **must** be declared as a four-component vector of 32-bit integer values

**SubgroupLocalInvocationId**

Decorating a variable with the `SubgroupLocalInvocationId` builtin decoration will make that variable contain the index of the invocation within the subgroup. This variable is in range [0, `SubgroupSize`-1].
**Note**

There is no direct relationship between `SubgroupLocalInvocationId` and `LocalInvocationId` or `LocalInvocationIndex`. If the pipeline was created with `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT`, applications can compute their own local invocation index to serve the same purpose:

\[
\text{index} = \text{SubgroupLocalInvocationId} + \text{SubgroupId} \times \text{SubgroupSize}
\]

If full subgroups are not enabled, some subgroups may be dispatched with inactive invocations that do not correspond to a local workgroup invocation, making the value of `index` unreliable.

---

**Valid Usage**

- **VUID-SubgroupLocalInvocationId-SubgroupLocalInvocationId-04380**
  The variable decorated with `SubgroupLocalInvocationId` **must** be declared using the Input Storage Class.

- **VUID-SubgroupLocalInvocationId-SubgroupLocalInvocationId-04381**
  The variable decorated with `SubgroupLocalInvocationId` **must** be declared as a scalar 32-bit integer value.

**SubgroupSize**

Decorating a variable with the `SubgroupSize` builtin decoration will make that variable contain the implementation-dependent **number of invocations in a subgroup**. This value **must** be a power-of-two integer.

If the pipeline was created with the `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT` flag set, the `SubgroupSize` decorated variable will contain the subgroup size for each subgroup that gets dispatched. This value **must** be between `minSubgroupSize` and `maxSubgroupSize` and **must** be uniform with subgroup scope. The value **may** vary across a single draw call, and for fragment shaders **may** vary across a single primitive. In compute dispatches, `SubgroupSize` **must** be uniform with command scope.

If the pipeline was created with a chained `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure, the `SubgroupSize` decorated variable will match `requiredSubgroupSize`.

If the pipeline was not created with the `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT` flag set and no `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure was chained, the variable decorated with `SubgroupSize` will match `subgroupSize`.

The maximum number of invocations that an implementation can support per subgroup is 128.
**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**
**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** not be used within the `GLCompute Execution Model`

- VUID-ViewIndex-ViewIndex-04402
  The variable decorated with `ViewIndex` **must** be declared using the `Input Storage Class`

- VUID-ViewIndex-ViewIndex-04403
  The variable decorated with `ViewIndex` **must** be declared as a scalar 32-bit integer value

**ViewportIndex**

Decorating a variable with the `ViewportIndex` built-in decoration will make that variable contain the index of the viewport.

In a vertex, tessellation evaluation, or geometry shader, the variable decorated with `ViewportIndex` can be written to with the viewport index to which the primitive produced by that shader will be directed.

The selected viewport index is used to select the viewport transform and scissor rectangle.

The last active **pre-rasterization shader stage** (in pipeline order) controls the `ViewportIndex` that is used. Outputs in previous shader stages are not used, even if the last stage fails to write the `ViewportIndex`.

If the last active **pre-rasterization shader stage** shader entry point's interface does not include a variable decorated with `ViewportIndex`, then the first viewport is used. If a **pre-rasterization shader stage** shader entry point's interface includes a variable decorated with `ViewportIndex`, it **must** write the same value to `ViewportIndex` for all output vertices of a given primitive.
In a fragment shader, the variable decorated with `ViewportIndex` contains the viewport index of the primitive that the fragment invocation belongs to.

## Valid Usage

- VUID-ViewportIndex-ViewportIndex-04404
  The `ViewportIndex` decoration must be used only within the `MeshNV`, `Vertex`, `TessellationEvaluation`, `Geometry`, or `Fragment Execution Model`.

- VUID-ViewportIndex-ViewportIndex-04405
  If the `shaderOutputViewportIndex` feature is not enabled then the `ViewportIndex` decoration must be used only within the `Geometry` or `Fragment Execution Model`.

- VUID-ViewportIndex-ViewportIndex-04406
  The variable decorated with `ViewportIndex` within the `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.

## 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`, `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`, `MeshNV`, or `TaskNV` Execution Model.

- **VUID-WorkgroupSize-WorkgroupSize-04426**
  The variable decorated with `WorkgroupSize` must be a specialization constant or a constant.

- **VUID-WorkgroupSize-WorkgroupSize-04427**
  The variable decorated with `WorkgroupSize` must be declared as a three-component vector of 32-bit integer values.
Chapter 16. Image Operations

16.1. Image Operations Overview

Vulkan Image Operations are operations performed by those SPIR-V Image Instructions which take an OpTypeImage (representing a VkImageView) or OpTypeSampledImage (representing a (VkImageView, VkSampler) pair). Read, write, and atomic operations also take texel coordinates as operands, and return a value based on a neighborhood of texture elements (texels) within the image. Query operations return properties of the bound image or of the lookup itself. The “Depth” operand of OpTypeImage is ignored.

**Note**

Texel is a term which is a combination of the words texture and element. Early interactive computer graphics supported texture operations on textures, a small subset of the image operations on images described here. The discrete samples remain essentially equivalent, however, so we retain the historical term texel to refer to them.

Image Operations include the functionality of the following SPIR-V Image Instructions:

- **OpImageSample** and **OpImageSparseSample** read one or more neighboring texels of the image, and filter the texel values based on the state of the sampler.
  - Instructions with **ImplicitLod** in the name determine the LOD used in the sampling operation based on the coordinates used in neighboring fragments.
  - Instructions with **ExplicitLod** in the name determine the LOD used in the sampling operation based on additional coordinates.
  - Instructions with **Proj** in the name apply homogeneous projection to the coordinates.

- **OpImageFetch** and **OpImageSparseFetch** return a single texel of the image. No sampler is used.

- **OpImage*Gather** and **OpImageSparse*Gather** read neighboring texels and return a single component of each.

- **OpImageRead** (and **OpImageSparseRead**) and **OpImageWrite** read and write, respectively, a texel in the image. No sampler is used.

- Instructions with **Dref** in the name apply depth comparison on the texel values.

- Instructions with **Sparse** in the name additionally return a sparse residency code.


- **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]

SPIR-V `OpImageFetch`, `OpImageSparseFetch`, `OpImageRead`, `OpImageSparseRead`, and `OpImageWrite` instructions use integer texel coordinates. Other image instructions can use either normalized or unnormalized texel coordinates (selected by the `unnormalizedCoordinates` state of the sampler used in the instruction), but there are limitations on what operations, image state, and sampler state is supported. Normalized coordinates are logically converted to unnormalized as part of image operations, and certain steps are only performed on normalized coordinates. The array layer coordinate is always treated as unnormalized even when other coordinates are normalized.

Normalized texel coordinates are referred to as (s,t,r,q,a), with the coordinates having the following meanings:

• s: Coordinate in the first dimension of an image.
• t: Coordinate in the second dimension of an image.
• r: Coordinate in the third dimension of an image.
  ◦ (s,t,r) are interpreted as a direction vector for Cube images.
• q: Fourth coordinate, for homogeneous (projective) coordinates.
• a: Coordinate for array layer.

The coordinates are extracted from the SPIR-V operand based on the dimensionality of the image variable and type of instruction. For Proj instructions, the components are in order (s, [t,] [r,] q), with t and r being conditionally present based on the Dim of the image. For non-`Proj` instructions, the coordinates are (s [,t] [,r] [,a]), with t and r being conditionally present based on the Dim of the image and a being conditionally present based on the Arrayed property of the image. Projective image instructions are not supported on Arrayed images.

Unnormalized texel coordinates are referred to as (u,v,w,a), with the coordinates having the following meanings:

• u: Coordinate in the first dimension of an image.
• v: Coordinate in the second dimension of an image.
• w: Coordinate in the third dimension of an image.
• a: Coordinate for array layer.

Only the u and v coordinates are directly extracted from the SPIR-V operand, because only 1D and 2D (non-Arrayed) dimensionalities support unnormalized coordinates. The components are in order (u [,v]), with v being conditionally present when the dimensionality is 2D. When normalized coordinates are converted to unnormalized coordinates, all four coordinates are used.

Integer texel coordinates are referred to as (i,j,k,l,n), with the coordinates having the following meanings:

• i: Coordinate in the first dimension of an image.
• j: Coordinate in the second dimension of an image.
• k: Coordinate in the third dimension of an image.
• l: Coordinate for array layer.
• n: 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 Dim of the image, and l conditionally present based on the Arrayed property of the image. n is conditionally present and is taken from the Sample image operand.

For all coordinate types, unused coordinates are assigned a value of zero.

Figure 3. Texel Coordinate Systems, Linear Filtering

The Texel Coordinate Systems - For the example shown of an 8×4 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.

Figure 4. Texel Coordinate Systems, Nearest Filtering
The Texel Coordinate Systems - For the example shown of an 8×4 texel two dimensional image.

- Texel coordinates as above. Also shown for nearest filtering:
  - Given the unnormalized coordinates \((u, v)\), the texel selected is \(ij\).
  - Given the offset \(\Delta_i\) and \(\Delta_j\), the texel selected by the offset is \(ij'\).

16.2. Conversion Formulas

16.2.1. RGB to Shared Exponent Conversion
An RGB color \((\text{red}, \text{green}, \text{blue})\) 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 \((\text{red}, \text{green}, \text{blue})\) are clamped to \((\text{red}_{\text{clamped}}, \text{green}_{\text{clamped}}, \text{blue}_{\text{clamped}})\) as:
\[ \text{red}_\text{clamped} = \max(0, \min(\text{sharedexp}_{\text{max}}, \text{red})) \]

\[ \text{green}_\text{clamped} = \max(0, \min(\text{sharedexp}_{\text{max}}, \text{green})) \]

\[ \text{blue}_\text{clamped} = \max(0, \min(\text{sharedexp}_{\text{max}}, \text{blue})) \]

where:

\[
\begin{align*}
N &= 9 & \text{number of mantissa bits per component} \\
B &= 15 & \text{exponent bias} \\
E_{\text{max}} &= 31 & \text{maximum possible biased exponent value} \\
\text{sharedexp}_{\text{max}} &= \frac{(2^N - 1)}{2^N} \times 2^{(E_{\text{max}} - B)}
\end{align*}
\]

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 \( \exp' \) is computed:

\[ \exp' = \begin{cases} 
\lfloor \log_2(\text{max}_\text{clamped}) \rfloor + (B + 1) & \text{for } \text{max}_\text{clamped} > 2^{-(B + 1)} \\
0 & \text{for } \text{max}_\text{clamped} \leq 2^{-(B + 1)} 
\end{cases} \]

The shared exponent \( \exp_{\text{shared}} \) is computed:

\[ \text{max}_\text{shared} = \left\lfloor \frac{\text{max}_\text{clamped}}{2^{(\exp' - B - N)}} + \frac{1}{2} \right\rfloor \]

\[ \exp_{\text{shared}} = \begin{cases} 
\exp' & \text{for } 0 \leq \text{max}_\text{shared} < 2^N \\
\exp' + 1 & \text{for } \text{max}_\text{shared} = 2^N 
\end{cases} \]

Finally, three integer values in the range 0 to \( 2^N \) are computed:
16.2.2. Shared Exponent to RGB

A shared exponent color \((\text{red}_{\text{shared}}, \text{green}_{\text{shared}}, \text{blue}_{\text{shared}}, \text{exp}_{\text{shared}})\) is transformed to an RGB color \((\text{red}, \text{green}, \text{blue})\) as follows:

\[
\begin{align*}
\text{red}_{\text{shared}} &= \left(\frac{\text{red}_{\text{clamped}}}{2^{(\text{exp}_{\text{shared}} - B - N)}} + \frac{1}{2}\right) \\
\text{green}_{\text{shared}} &= \left(\frac{\text{green}_{\text{clamped}}}{2^{(\text{exp}_{\text{shared}} - B - N)}} + \frac{1}{2}\right) \\
\text{blue}_{\text{shared}} &= \left(\frac{\text{blue}_{\text{clamped}}}{2^{(\text{exp}_{\text{shared}} - B - N)}} + \frac{1}{2}\right)
\end{align*}
\]

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

where:

\(N = 9\) (number of mantissa bits per component)

\(B = 15\) (exponent bias)
- Conversion to RGBA
- Component swizzle
- Chroma reconstruction
- Y’CbCr 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’CbCr 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 sampler borderColor is a custom color (VK_BORDER_COLOR_FLOAT_CUSTOM_EXT or VK_BORDER_COLOR_INT_CUSTOM_EXT) and the supplied VkSamplerCustomBorderColorCreateInfoEXT::customBorderColor is outside the bounds of the values representable in the image view’s format.
- The sampler borderColor is a custom color (VK_BORDER_COLOR_FLOAT_CUSTOM_EXT or VK_BORDER_COLOR_INT_CUSTOM_EXT) 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 `VK_IMAGE_ASPECT_DEPTH_BIT`.

• The SPIR-V instruction's image variable's properties are not compatible with the image view:
  ◦ Rules for `viewType`:
    ▪ `VK_IMAGE_VIEW_TYPE_1D` must have `Dim = 1D, Arrayed = 0, MS = 0`.
    ▪ `VK_IMAGE_VIEW_TYPE_2D` must have `Dim = 2D, Arrayed = 0`.
    ▪ `VK_IMAGE_VIEW_TYPE_3D` must have `Dim = 3D, Arrayed = 0, MS = 0`.
    ▪ `VK_IMAGE_VIEW_TYPE_CUBE` must have `Dim = Cube, Arrayed = 0, MS = 0`.
    ▪ `VK_IMAGE_VIEW_TYPE_1D_ARRAY` must have `Dim = 1D, Arrayed = 1, MS = 0`.
    ▪ `VK_IMAGE_VIEW_TYPE_2D_ARRAY` must have `Dim = 2D, Arrayed = 1`.
    ▪ `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY` must have `Dim = Cube, Arrayed = 1, MS = 0`.
  ◦ If the image was created with `VkImageCreateInfo::samples` equal to `VK_SAMPLE_COUNT_1_BIT`, the instruction must have `MS = 0`.
  ◦ If the image was created with `VkImageCreateInfo::samples` not equal to `VK_SAMPLE_COUNT_1_BIT`, the instruction must have `MS = 1`.
  ◦ If the `Sampled Type` of the `OpTypeImage` does not match the numeric format of the image, as shown in the `SPIR-V Sampled Type` column of the `Interpretation of Numeric Format` table.
  ◦ If the signedness of any read or sample operation does not match the signedness of the image's format.

• The sampler was created with a specified `VkSamplerCustomBorderColorCreateInfoEXT::format` which does not match the `VkFormat` of the image view(s) it is sampling.

• The sampler is sampling an image view of `VK_FORMAT_B4G4R4A4_UNORM_PACK16`, `VK_FORMAT_B5G6R5_UNORM_PACK16`, or `VK_FORMAT_B5G5R5A1_UNORM_PACK16` format without a specified `VkSamplerCustomBorderColorCreateInfoEXT::format`.

Only `OpImageSample*` and `OpImageSparseSample*` can be used with a sampler that enables `sampler Y'CbCr` conversion.

`OpImageFetch`, `OpImageSparseFetch`, `OpImage*Gather`, and `OpImageSparse*Gather` must not be used with a sampler that enables `sampler Y'CbCr` conversion.

The `ConstOffset` and `Offset` operands must not be used with a sampler that enables `sampler Y'CbCr` conversion.
**Integer Texel Coordinate Validation**

Integer texel coordinates are validated against the size of the image level, and the number of layers and number of samples in the image. For SPIR-V instructions that use integer texel coordinates, this is performed directly on the integer coordinates. For instructions that use normalized or unnormalized texel coordinates, this is performed on the coordinates that result after conversion to integer texel coordinates.

If the integer texel coordinates do not satisfy all of the conditions

\[
0 \leq i < w_s \\
0 \leq j < h_s \\
0 \leq k < d_s \\
0 \leq l < \text{layers} \\
0 \leq n < \text{samples}
\]

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, the following steps are performed. Note that this does not occur when using VK_FILTER_NEAREST filtering within a mip level, since VK_FILTER_NEAREST is treated as using VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE.

- Cube Map Edge Texel
  - If the texel lies beyond the selected cube map face in either only i or only j, then the coordinates (i,j) and the array layer l are transformed to select the adjacent texel from the appropriate neighboring face.

- Cube Map Corner Texel
  - If the texel lies beyond the selected cube map face in both i and j, then there is no unique neighboring face from which to read that texel. The texel should be replaced by the average of the three values of the adjacent texels in each incident face. However, implementations may replace the cube map corner texel by other methods. The methods are subject to the constraint that for linear filtering if the three available texels have the same value, the resulting filtered texel must have that value, and for cubic filtering if the twelve available samples have the same value, the resulting filtered texel must have that value.

Sparse Validation

If the texel reads from an unbound region of a sparse image, the texel is a sparse unbound texel, and processing continues with texel replacement.

Layout Validation

If all planes of a disjoint multi-planar image are not in the same image layout, the image must not be sampled with sampler Y’C_bC_a 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 borderColor</th>
<th>Corresponding Border Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK</td>
<td>([B_r, B_g, B_b, B_a] = [0.0, 0.0, 0.0, 0.0])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK</td>
<td>([B_r, B_g, B_b, B_a] = [0.0, 0.0, 0.0, 1.0])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE</td>
<td>([B_r, B_g, B_b, B_a] = [1.0, 1.0, 1.0, 1.0])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_INT_TRANSPARENT_BLACK</td>
<td>([B_r, B_g, B_b, B_a] = [0, 0, 0, 0])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_INT_OPAQUE_BLACK</td>
<td>([B_r, B_g, B_b, B_a] = [0, 0, 0, 1])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_INT_OPAQUE_WHITE</td>
<td>([B_r, B_g, B_b, B_a] = [1, 1, 1, 1])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_FLOAT_CUSTOM_EXT</td>
<td>([B_r, B_g, B_b, B_a] = [U_r, U_g, U_b, U_a])</td>
</tr>
</tbody>
</table>
The custom border color (U) may be rounded by implementations prior to texel replacement, but the error introduced by such a rounding must not exceed one ULP of the image's format.

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 20. Border Texel Components After Replacement

<table>
<thead>
<tr>
<th>Texel Aspect or Format</th>
<th>Component Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth aspect</td>
<td>D = B_r</td>
</tr>
<tr>
<td>Stencil aspect</td>
<td>S = B_r</td>
</tr>
<tr>
<td>One component color format</td>
<td>Color_r = B_r</td>
</tr>
<tr>
<td>Two component color format</td>
<td>[Color_r,Color_g] = [B_r,B_g]</td>
</tr>
<tr>
<td>Three component color format</td>
<td>[Color_r,Color_g,Color_b] = [B_r,B_g,B_b]</td>
</tr>
<tr>
<td>Four component color format</td>
<td>[Color_r,Color_g,Color_b,Color_a] = [B_r,B_g,B_b,B_a]</td>
</tr>
</tbody>
</table>

The value returned by a read of an invalid texel is undefined, unless that read operation is from a buffer resource and the robustBufferAccess feature is enabled. In that case, an invalid texel is replaced as described by the robustBufferAccess feature. If the access is to an image resource and the x, y, z, or layer coordinate validation fails and robustImageAccess is enabled then zero must be returned for the R, G, and B components, if present. Either zero or one must be returned for the A component, if present. If robustImageAccess2 is enabled, zero values must be returned. If only the sample index was invalid, the values returned are undefined.

Additionally, if robustImageAccess is enabled, but robustImageAccess2 is not, any invalid texels may be expanded to four components prior to texel replacement. This means that components not present in the image format may be replaced with 0 or may undergo conversion to RGBA as normal.

Loads from a null descriptor return a four component color value of all zeros. However, for storage images and storage texel buffers using an explicit SPIR-V Image Format, loads from a null descriptor may return an alpha value of 1 (float or integer, depending on format) if the format does not include alpha.

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 a `Dref` instruction, a depth comparison is performed. The value of the result `D` is 1.0 if the result of the compare operation is true, and 0.0 otherwise. The compare operation is selected by the `compareOp` member of the sampler.

\[
\begin{align*}
D &= 1.0 & & \text{for LEQUAL} \\
D &\leq D_{\text{tex}} \\
D &= 0.0 & & \text{for GEQUAL} \\
D &\geq D_{\text{tex}} \\
D &= 1.0 & & \text{for LESS} \\
D &< D_{\text{tex}} \\
D &= 0.0 & & \text{for GREATER} \\
D &= D_{\text{tex}} \\
D &= 0.0 & & \text{for EQUAL} \\
D &\neq D_{\text{tex}} \\
D &= 1.0 & & \text{for NOTEQUAL} \\
\text{true} & & & \text{for ALWAYS} \\
\text{false} & & & \text{for NEVER}
\end{align*}
\]

where \(D_{\text{tex}}\) is the texel depth value and \(D_{\text{ref}}\) is the reference value from the SPIR-V operand. If the image being sampled has a fixed-point format then the reference value is clamped to \([0,1]\) before the comparison 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 21. Texel Color After Conversion To RGBA**

<table>
<thead>
<tr>
<th>Texel Aspect or Format</th>
<th>RGBA Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth aspect</td>
<td>([\text{Color}_r,\text{Color}_g,\text{Color}_b,\text{Color}_a] = [D,0,0,\text{one}])</td>
</tr>
<tr>
<td>Stencil aspect</td>
<td>([\text{Color}_r,\text{Color}_g,\text{Color}_b,\text{Color}_a] = [S,0,0,\text{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,\text{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,\text{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}_g,\text{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 \(\text{one} = 1.0f\) for floating-point formats and depth aspects, and \(\text{one} = 1\) 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'CBCA conversion` is not enabled, and
• the `VkComponentSwizzle` enums in the `components` member of the `VkSamplerYcbcrConversionCreateInfo` structure for the sampler Y'CbCr conversion if sampler Y'CbCr conversion is enabled.

The swizzle can rearrange the components of the texel, or substitute zero or one for any components. It is defined as follows for each color component:

\[
\text{Color}_{\text{component}} = \begin{cases} 
\text{Color}_r & \text{for RED swizzle} \\
\text{Color}_g & \text{for GREEN swizzle} \\
\text{Color}_b & \text{for BLUE swizzle} \\
\text{Color}_a & \text{for ALPHA swizzle} \\
0 & \text{for ZERO swizzle} \\
\text{one} & \text{for ONE swizzle} \\
\text{identity} & \text{for IDENTITY swizzle}
\end{cases}
\]

where:

\[
\text{one} = \begin{cases} 
1.0f & \text{for floating point components} \\
1 & \text{for integer components}
\end{cases}
\]

\[
\text{identity} = \begin{cases} 
\text{Color}_r & \text{for component} = r \\
\text{Color}_g & \text{for component} = g \\
\text{Color}_b & \text{for component} = b \\
\text{Color}_a & \text{for component} = a
\end{cases}
\]

If the border color is one of the `VK_BORDER_COLOR_*_OPAQUE_BLACK` enums and the `VkComponentSwizzle` is not the `identity` swizzle for all components, the value of the texel after swizzle is undefined.

16.3.7. Sparse Residency

OpImageSparse* instructions return a structure which includes a `residency code` indicating whether any texels accessed by the instruction are sparse unbound texels. This code can be interpreted by the OpImageSparseTexelsResident instruction which converts the residency code to a boolean value.

16.3.8. Chroma Reconstruction

In some color models, the color representation is defined in terms of monochromatic light intensity (often called “luma”) and color differences relative to this intensity, often called “chroma”. It is common for color models other than RGB to represent the chroma components at lower spatial resolution than the luma component. This approach is used to take advantage of the eye’s lower spatial sensitivity to color compared with its sensitivity to brightness. Less commonly, the same approach is used with additive color, since the green component dominates the eye’s sensitivity to light intensity and the spatial sensitivity to color introduced by red and blue is lower.

Lower-resolution components are “downsampled” by resizing them to a lower spatial resolution than the component representing luminance. This process is also commonly known as “chroma subsampling”. There is one luminance sample in each texture texel, but each chrominance sample
may be shared among several texels in one or both texture dimensions.

- “._444” formats do not spatially downsample chroma values compared with luma: there are unique chroma samples for each texel.
- “._422” formats have downsampling in the x dimension (corresponding to u or s coordinates): they are sampled at half the resolution of luma in that dimension.
- “._420” formats have downsampling in the x dimension (corresponding to u or s coordinates) and the y dimension (corresponding to v or t coordinates): they are sampled at half the resolution of luma in both dimensions.

The process of reconstructing a full color value for texture access involves accessing both chroma and luma values at the same location. To generate the color accurately, the values of the lower-resolution components at the location of the luma samples must be reconstructed from the lower-resolution sample locations, an operation known here as “chroma reconstruction” irrespective of the actual color model.

The location of the chroma samples relative to the luma coordinates is determined by the xChromaOffset and yChromaOffset members of the VkSamplerYcbcrConversionCreateInfo structure used to create the sampler Y′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_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT`, or the `VkSamplerYcbcrConversionCreateInfo`'s `forceExplicitReconstruction` is set to `VK_TRUE`, reconstruction is performed as an explicit step independent of filtering, described in the `Explicit Reconstruction` section.

If the format of the image that is to be sampled does not set `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT` and if the
VkSamplerYcbcrConversionCreateInfo’s `forceExplicitReconstruction` is set to `VK_FALSE`, reconstruction is performed as an implicit part of filtering prior to color model conversion, with no separate post-conversion texel filtering step, as described in the Implicit Reconstruction section.

### Explicit Reconstruction

- If the `chromaFilter` member of the `VkSamplerYcbcrConversionCreateInfo` structure is `VK_FILTER_NEAREST`:
  - If the format’s R and B components are reduced in resolution in just width by a factor of two relative to the G component (i.e. this is a “-422” format), the $\tau_{ijk}[level]$ values accessed by texel filtering are reconstructed as follows:
    \[
    \begin{align*}
    \tau_R'(i, j) &= \tau_R([i \times 0.5], [j])[level] \\
    \tau_B'(i, j) &= \tau_B([i \times 0.5], [j])[level]
    \end{align*}
    \]

  - If the format’s R and B components are reduced in resolution in width and height by a factor of two relative to the G component (i.e. this is a “-420” format), the $\tau_{ijk}[level]$ values accessed by texel filtering are reconstructed as follows:
    \[
    \begin{align*}
    \tau_R'(i, j) &= \tau_R([i \times 0.5], [j \times 0.5])[level] \\
    \tau_B'(i, j) &= \tau_B([i \times 0.5], [j \times 0.5])[level]
    \end{align*}
    \]

  Note: 
  `xChromaOffset` and `yChromaOffset` have no effect if `chromaFilter` is `VK_FILTER_NEAREST` for explicit reconstruction.

- If the `chromaFilter` member of the `VkSamplerYcbcrConversionCreateInfo` structure is `VK_FILTER_LINEAR`:
  - If the format’s R and B components are reduced in resolution in just width by a factor of two relative to the G component (i.e. this is a “-422” format):
    - If `xChromaOffset` is `VK_CHROMA_LOCATION_COSITED_EVEN`:
      \[
      \tau_{RB}'(i, j) = \begin{cases} \\
      \tau_{RB}([i \times 0.5], [j])[level], & 0.5 \times i = [0.5 \times i] \\
      0.5 \times \tau_{RB}([i \times 0.5], [j])[level] + 0.5 \times \tau_{RB}([i \times 0.5] + 1, [j])[level], & 0.5 \times i \neq [0.5 \times i]
      \end{cases}
      \]
    - If `xChromaOffset` is `VK_CHROMA_LOCATION_MIDPOINT`:
      \[
      \tau_{RB}'(i, j) = \begin{cases} \\
      0.25 \times \tau_{RB}([i \times 0.5] - 1, [j])[level] + 0.75 \times \tau_{RB}([i \times 0.5], [j])[level], & 0.5 \times i = [0.5 \times i] \\
      0.75 \times \tau_{RB}([i \times 0.5], [j])[level] + 0.25 \times \tau_{RB}([i \times 0.5] + 1, [j])[level], & 0.5 \times i \neq [0.5 \times i]
      \end{cases}
      \]

  - If the format’s R and B components are reduced in resolution in width and height by a factor of two relative to the G component (i.e. this is a “-420” format), a similar relationship applies. Due to the number of options, these formulae are expressed more concisely as follows:
Note
In the case where the texture itself is bilinearly interpolated as described in Texel Filtering, thus requiring four full-color samples for the filtering operation, and where the reconstruction of these samples uses bilinear interpolation in the chroma components due to \( \text{chromaFilter} = \text{VK_FILTER_LINEAR} \), 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.

Note
This will not have any visible effect if the locations of the luma samples coincide with the location of the samples used for rasterization.

The sample coordinates are adjusted by the downsample factor of the component (such that, for example, the sample coordinates are divided by two if the component has a downsample factor of two relative to the luma component):

\[
\begin{align*}
\mathbf{i}_{RB} &= \begin{cases} 
0.5 \times (i) & \text{xChromaOffset} = \text{COSITED\_EVEN} \\
0.5 \times (i - 0.5) & \text{xChromaOffset} = \text{MIDPOINT}
\end{cases} \\
\mathbf{j}_{RB} &= \begin{cases} 
0.5 \times (j) & \text{yChromaOffset} = \text{COSITED\_EVEN} \\
0.5 \times (j - 0.5) & \text{yChromaOffset} = \text{MIDPOINT}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\mathbf{i}_{floor} &= \lfloor \mathbf{i}_{RB} \rfloor \\
\mathbf{j}_{floor} &= \lfloor \mathbf{j}_{RB} \rfloor \\
\mathbf{i}_{frac} &= \mathbf{i}_{RB} - \mathbf{i}_{floor} \\
\mathbf{j}_{frac} &= \mathbf{j}_{RB} - \mathbf{j}_{floor}
\end{align*}
\]

\[
\begin{align*}
\mathbf{r}_{RB} \cdot (i, j) &= \mathbf{r}_{RB}(\mathbf{i}_{floor}, \mathbf{j}_{floor})[\text{level}] \times (1 - \mathbf{i}_{frac}) \times (1 - \mathbf{j}_{frac}) + \\
& \quad \mathbf{r}_{RB}(1 + \mathbf{i}_{floor}, \mathbf{j}_{floor})[\text{level}] \times \mathbf{i}_{frac} \times (1 - \mathbf{j}_{frac}) + \\
& \quad \mathbf{r}_{RB}(\mathbf{i}_{floor}, 1 + \mathbf{j}_{floor})[\text{level}] \times (1 - \mathbf{i}_{frac}) \times \mathbf{j}_{frac} + \\
& \quad \mathbf{r}_{RB}(1 + \mathbf{i}_{floor}, 1 + \mathbf{j}_{floor})[\text{level}] \times \mathbf{i}_{frac} \times \mathbf{j}_{frac}
\end{align*}
\]

16.3.9. Sampler Y’C_B C_R Conversion

Sampler Y’C_B C_R conversion performs the following operations, which an implementation may
combine into a single mathematical operation:

- Sampler Y’C₈C₉ Range Expansion
- Sampler Y’C₈C₉ Model Conversion

**Sampler Y’C₈C₉ Range Expansion**

Sampler Y’C₈C₉ range expansion is applied to color component values after all texel input operations which are not specific to sampler Y’C₈C₉ conversion. For example, the input values to this stage have been converted using the normal *format conversion* rules.

Sampler Y’C₈C₉ range expansion is not applied if *ycbcrModel* is `VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY`. That is, the shader receives the vector C₉gba 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’C₈C₉ 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’C₈C₉ range expansion.

The range expansion to be applied is defined by the *ycbcrRange* member of the `VkSamplerYcbcrConversionCreateInfo` structure:

- If *ycbcrRange* is `VK_SAMPLER_YCBCR_RANGE_ITU_FULL`, the following transformations are applied:

  \[
  Y' = C'gba [G] \\
  C_B = C'gba [B] - \frac{2^{(n-1)}}{(2^n) - 1} \\
  C_R = C'gba [R] - \frac{2^{(n-1)}}{(2^n) - 1}
  \]

  *Note*
  These formulae correspond to the “full range” encoding in the “Quantization schemes” chapter of the *Khronos Data Format Specification*.

  Should any future amendments be made to the ITU specifications from which these equations are derived, the formulae used by Vulkan *may* also be updated to maintain parity.

- If *ycbcrRange* is `VK_SAMPLER_YCBCR_RANGE_ITU_NARROW`, the following transformations are applied:
\[ Y' = \frac{C'_{rgb} [G] \times (2^n - 1) - 16 \times 2^n - 8}{219 \times 2^n - 8} \]
\[ C_B = \frac{C'_{rgb} [B] \times (2^n - 1) - 128 \times 2^n - 8}{224 \times 2^n - 8} \]
\[ C_R = \frac{C'_{rgb} [R] \times (2^n - 1) - 128 \times 2^n - 8}{224 \times 2^n - 8} \]

**Note**

These formulae correspond to the “narrow range” encoding in the “Quantization schemes” chapter of the Khronos Data Format Specification.

- \( n \) 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'CbCr Model Conversion**

The range-expanded values are converted between color models, according to the color model conversion specified in the `ycbcrModel` member:

**VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY**

The color 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; \( Y' \), \( C_B \), \( C_R \) range expansion is also ignored.

**VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_IDENTITY**

The color components are not modified by the color model conversion and are assumed to be treated as though in \( Y' \), \( C_B \), \( C_R \) form both in memory and in the shader; \( Y' \), \( C_B \), \( C_R \) range expansion is applied to the components as for other \( Y' \), \( C_B \), \( C_R \) models, with the vector \((C_R', Y', C_B', A)\) provided to the shader.

**VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_709**

The color components are transformed from a \( Y' \), \( C_B \), \( C_R \) representation to an R'G'B' representation as 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**

The color components are transformed from a \( Y' \), \( C_B \), \( C_R \) representation to an R'G'B' representation as 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**

The color components are transformed from a \( Y' \), \( C_B \), \( C_R \) representation to an R'G'B' representation as described in the “BT.2020 Y’C_B'C_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:

1. Calculate the unnormalized texel coordinates corresponding to the desired sample position.
2. 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.

3. For a `minFilter` or `magFilter` of `VK_FILTER_LINEAR`:
   1. Calculate \((i_{[0,1]},j_{[0,1]})\) for the sample location as described under the “linear filtering” formulae in \((u,v,w,a)\) to \((i,j,k,l,n)\) Transformation And Array Layer Selection
   2. Calculate the normalized texel coordinates corresponding to these integer coordinates.
   3. Sample using sampler Y'CbCr conversion at each of these locations.
   4. Convert the non-linear A'R'G'B' outputs of the YCbCr 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'CBCr conversion at the desired sample coordinates will produce the “correct” results without further processing.

16.4. Texel Output Operations

*Texel output instructions* are SPIR-V image instructions that write to an image. *Texel output operations* are a set of steps that are performed on state, coordinates, and texel values while processing a texel output instruction, and which are common to some or all texel output instructions. They include the following steps, which are performed in the listed order:

- Validation operations
  - Format validation
  - Type validation
  - Coordinate validation
  - Sparse validation
- Texel output format conversion

16.4.1. Texel Output Validation Operations

*Texel output validation operations* inspect instruction/image state or coordinates, and in certain circumstances cause the write to have no effect. There are a series of validations that the texel undergoes.

**Texel Format Validation**

If the image format of the `OpTypeImage` is not compatible with the `VkImageView`'s format, the write causes the contents of the image’s memory to become undefined.

**Texel Type Validation**

If the *Sampled Type* of the `OpTypeImage` does not match the type defined for the format, as specified in the SPIR-V *Sampled Type* column of the Interpretation of Numeric Format table, the write causes the value of the texel to become undefined. For integer types, if the *signedness of the access* does not match the signedness of the accessed resource, the write causes the value of the texel to become undefined.

16.4.2. Integer Texel Coordinate Validation

The integer texel coordinates are validated according to the same rules as for texel input *coordinate validation*.

If the texel fails integer texel coordinate validation, then the write has no effect.
16.4.3. Sparse Texel Operation

If the texel attempts to write to an unbound region of a sparse image, the texel is a sparse unbound texel. In such a case, if the \texttt{VkPhysicalDeviceSparseProperties::residencyNonResidentStrict} property is \texttt{VK_TRUE}, the sparse unbound texel write has no effect. If \texttt{residencyNonResidentStrict} is \texttt{VK_FALSE}, the write \textbf{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 \textit{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 \texttt{VkFormat} of the image view. Any unused components are ignored.

Each component is converted based on its type and size (as defined in the \textit{Format Definition} section for each \texttt{VkFormat}). Floating-point outputs are converted as described in \textit{Floating-Point Format Conversions} and \textit{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 \texttt{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 \texttt{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:

\[
\begin{align*}
\frac{\partial s}{\partial x} &= \frac{dPd}{dx}(s), & \frac{\partial s}{\partial y} &= \frac{dPd}{dy}(s), & \text{for 1D, 2D, Cube, or 3D image} \\
\frac{\partial t}{\partial x} &= \frac{dPd}{dx}(t), & \frac{\partial t}{\partial y} &= \frac{dPd}{dy}(t), & \text{for 2D, Cube, or 3D image} \\
\frac{\partial r}{\partial x} &= \frac{dPd}{dx}(r), & \frac{\partial r}{\partial y} &= \frac{dPd}{dy}(r), & \text{for Cube or 3D image}
\end{align*}
\]

Partial derivatives not defined above for certain image dimensionalities are set to zero.

For explicit LOD image instructions, if the optional SPIR-V operand Grad is provided, then the operand values are used for the derivatives. The number of components present in each derivative for a given image dimensionality matches the number of partial derivatives computed above.

If the optional SPIR-V operand Lod is provided, then derivatives are set to zero, the cube map derivative transformation is skipped, and the scale factor operation is skipped. Instead, the floating point scalar coordinate is directly assigned to \(\lambda_{\text{base}}\) as described in Level-of-Detail Operation.

If the image or sampler object used by an implicit derivative image instruction is not uniform across the quad and quadDivergentImplicitLod is not supported, then the derivative and LOD values are undefined. Implicit derivatives are well-defined when the image and sampler and control flow are uniform across the quad, even if they diverge between different quads.

If quadDivergentImplicitLod is supported, then derivatives and implicit LOD values are well-defined even if the image or sampler object are not uniform within a quad. The derivatives are computed as specified above, and the implicit LOD calculation proceeds for each shader invocation using its respective image and sampler object.

### 16.5.3. Cube Map Face Selection and Transformations

For cube map image instructions, the \((s,t,r)\) coordinates are treated as a direction vector \((r_x,r_y,r_z)\). The direction vector is used to select a cube map face. The direction vector is transformed to a per-face texel coordinate system \((s_{\text{face}},t_{\text{face}},r_{\text{face}})\). The direction vector is also used to transform the derivatives to per-face derivatives.

### 16.5.4. Cube Map Face Selection

The direction vector selects one of the cube map’s faces based on the largest magnitude coordinate direction (the major axis direction). Since two or more coordinates can have identical magnitude, the implementation must have rules to disambiguate this situation.

The rules should have as the first rule that \(r_z\) wins over \(r_y\) and \(r_x\), and the second rule that \(r_y\) wins over \(r_x\). An implementation may choose other rules, but the rules must be deterministic and depend only on \((r_x,r_y,r_z)\).

The layer number (corresponding to a cube map face), the coordinate selections for \(s_c, t_c, r_c\), and the selection of derivatives, are determined by the major axis direction as specified in the following two tables.

*Table 22. Cube map face and coordinate selection*
<table>
<thead>
<tr>
<th>Major Axis Direction</th>
<th>Layer Number</th>
<th>Cube Map Face</th>
<th>( s_c )</th>
<th>( t_c )</th>
<th>( r_c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+r_x)</td>
<td>0</td>
<td>Positive X</td>
<td>(-r_z)</td>
<td>(-r_y)</td>
<td>( r_x)</td>
</tr>
<tr>
<td>(-r_x)</td>
<td>1</td>
<td>Negative X</td>
<td>(+r_z)</td>
<td>(-r_y)</td>
<td>( r_x)</td>
</tr>
<tr>
<td>(+r_y)</td>
<td>2</td>
<td>Positive Y</td>
<td>(+r_z)</td>
<td>(+r_y)</td>
<td>( r_y)</td>
</tr>
<tr>
<td>(-r_y)</td>
<td>3</td>
<td>Negative Y</td>
<td>(+r_z)</td>
<td>(-r_y)</td>
<td>( r_y)</td>
</tr>
<tr>
<td>(+r_z)</td>
<td>4</td>
<td>Positive Z</td>
<td>(+r_z)</td>
<td>(-r_y)</td>
<td>( r_z)</td>
</tr>
<tr>
<td>(-r_z)</td>
<td>5</td>
<td>Negative Z</td>
<td>(-r_z)</td>
<td>(-r_y)</td>
<td>( r_z)</td>
</tr>
</tbody>
</table>

**Table 23. Cube map derivative selection**

<table>
<thead>
<tr>
<th>Major Axis Direction</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>(-\partial r_z / \partial x)</td>
<td>(-\partial r_z / \partial y)</td>
<td>(-\partial r_y / \partial x)</td>
<td>(-\partial r_y / \partial y)</td>
<td>(+\partial r_x / \partial x)</td>
<td>(+\partial r_x / \partial y)</td>
</tr>
<tr>
<td>(-r_x)</td>
<td>(+\partial r_z / \partial x)</td>
<td>(+\partial r_z / \partial y)</td>
<td>(-\partial r_y / \partial x)</td>
<td>(-\partial r_y / \partial y)</td>
<td>(-\partial r_x / \partial x)</td>
<td>(-\partial r_x / \partial y)</td>
</tr>
<tr>
<td>(+r_y)</td>
<td>(+\partial r_z / \partial x)</td>
<td>(+\partial r_z / \partial y)</td>
<td>(+\partial r_z / \partial x)</td>
<td>(+\partial r_z / \partial y)</td>
<td>(+\partial r_y / \partial x)</td>
<td>(+\partial r_y / \partial y)</td>
</tr>
<tr>
<td>(-r_y)</td>
<td>(+\partial r_z / \partial x)</td>
<td>(+\partial r_z / \partial y)</td>
<td>(-\partial r_z / \partial x)</td>
<td>(-\partial r_z / \partial y)</td>
<td>(-\partial r_y / \partial x)</td>
<td>(-\partial r_y / \partial y)</td>
</tr>
<tr>
<td>(+r_z)</td>
<td>(+\partial r_z / \partial x)</td>
<td>(+\partial r_z / \partial y)</td>
<td>(-\partial r_y / \partial x)</td>
<td>(-\partial r_y / \partial y)</td>
<td>(+\partial r_z / \partial x)</td>
<td>(+\partial r_z / \partial y)</td>
</tr>
<tr>
<td>(-r_z)</td>
<td>(-\partial r_z / \partial x)</td>
<td>(-\partial r_z / \partial y)</td>
<td>(-\partial r_y / \partial x)</td>
<td>(-\partial r_y / \partial y)</td>
<td>(-\partial r_z / \partial x)</td>
<td>(-\partial r_z / \partial y)</td>
</tr>
</tbody>
</table>

16.5.5. Cube Map Coordinate Transformation

\[
s_{face} = \frac{1}{2} \times \frac{s_c}{|r_c|} + \frac{1}{2}
\]
\[
t_{face} = \frac{1}{2} \times \frac{t_c}{|r_c|} + \frac{1}{2}
\]

16.5.6. Cube Map Derivative Transformation

\[
\frac{\partial s_{face}}{\partial x} = \frac{\partial}{\partial x} \left( \frac{1}{2} \times \frac{s_c}{|r_c|} + \frac{1}{2} \right)
\]
\[
\frac{\partial s_{face}}{\partial x} = \frac{1}{2} \times \frac{\partial}{\partial x} \left( \frac{s_c}{|r_c|} \right)
\]
\[
\frac{\partial s_{face}}{\partial x} = \frac{1}{2} \times \frac{|r_c| \times \partial s_c / \partial x - s_c \times \partial r_c / \partial x}{(r_c)^2}
\]
16.5.7. Scale Factor Operation, Level-of-Detail Operation and Image Level(s) Selection

LOD selection can be either explicit (provided explicitly by the image instruction) or implicit (determined from a scale factor calculated from the derivatives). The LOD must be computed with `mipmapPrecisionBits` of accuracy.

Scale Factor Operation

The magnitude of the derivatives are calculated by:

\[
\begin{align*}
\frac{\partial s_{face}}{\partial y} &= \frac{1}{2} \times \left( \frac{|r_c| \times \partial s / \partial y - s_c \times \partial r / \partial y}{(r_c)^2} \right) \\
\frac{\partial t_{face}}{\partial x} &= \frac{1}{2} \times \left( \frac{|r_c| \times \partial t / \partial x - t_c \times \partial r / \partial x}{(r_c)^2} \right) \\
\frac{\partial t_{face}}{\partial y} &= \frac{1}{2} \times \left( \frac{|r_c| \times \partial t / \partial y - t_c \times \partial r / \partial y}{(r_c)^2} \right)
\end{align*}
\]

where:

\[
\frac{\partial t}{\partial x} = \frac{\partial t}{\partial y} = 0 \text{ (for 1D images)}
\]

\[
\frac{\partial r}{\partial x} = \frac{\partial r}{\partial y} = 0 \text{ (for 1D, 2D or Cube images)}
\]

and:

\[
\text{w}_{\text{base}} = \text{image.w}
\]
h_base = image.h

d_base = image.d

(for the baseMipLevel, from the image descriptor).

A point sampled in screen space has an elliptical footprint in texture space. The minimum and maximum scale factors \((\rho_{\text{min}}, \rho_{\text{max}})\) 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 \(m_{ux}, m_{vx}, \text{ and } m_{wx}\)
- \(f_y\) is continuous and monotonically increasing in each of \(m_{uy}, m_{vy}, \text{ and } m_{wy}\)

\[
\begin{align*}
\max(|m_{ux}|, |m_{vx}|, |m_{wx}|) & \leq f_x \leq \sqrt{2(\max(|m_{ux}|, |m_{vx}|, |m_{wx}|))} \\
\max(|m_{uy}|, |m_{vy}|, |m_{wy}|) & \leq f_y \leq \sqrt{2(\max(|m_{uy}|, |m_{vy}|, |m_{wy}|))}
\end{align*}
\]

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:

\[
\eta = \min(\rho_{\text{max}}/\rho_{\text{min}}, \max_{\text{Aniso}})
\]

where:

- \(\text{sampler.max}_{\text{Aniso}} = \max_{\text{Anisotropy}}\) (from sampler descriptor)
- \(\text{limits.max}_{\text{Aniso}} = \max_{\text{SamplerAnisotropy}}\) (from physical device limits)

\[
\max_{\text{Aniso}} = \min(\text{sampler.max}_{\text{Aniso}}, \text{limits.max}_{\text{Aniso}})
\]

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 \( \rho_{\text{min}} \) 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 \( \eta \).

### Level-of-Detail Operation

The LOD parameter \( \lambda \) is computed as follows:

\[
\lambda_{\text{base}}(x, y) = \begin{cases} \text{shaderOp.Lod} & \text{(from optional SPIR-V operand)} \\ \log_2 \left( \frac{\rho_{\text{max}}}{\eta} \right) & \text{otherwise} \end{cases}
\]

\[
\lambda'(x, y) = \lambda_{\text{base}} + \text{clamp}(\text{sampler.bias} + \text{shaderOp.bias}, -\text{maxSamplerLodBias}, \text{maxSamplerLodBias})
\]

\[
\lambda = \begin{cases} \text{lod}_{\text{max}}, & \lambda' > \text{lod}_{\text{max}} \\ \lambda', & \text{lod}_{\text{min}} \leq \lambda' \leq \text{lod}_{\text{max}} \\ \text{lod}_{\text{min}}, & \lambda' < \text{lod}_{\text{min}} \\ \text{undefined}, & \text{lod}_{\text{min}} > \text{lod}_{\text{max}} \end{cases}
\]

where:

- \( \text{sampler.bias} = \text{mipLodBias} \) (from sampler descriptor)
- \( \text{shaderOp.bias} = \begin{cases} \text{Bias} & \text{(from optional SPIR-V operand)} \\ 0 & \text{otherwise} \end{cases} \)
- \( \text{sampler.lod}_{\text{min}} = \text{minLod} \) (from sampler descriptor)
- \( \text{shaderOp.lod}_{\text{min}} = \begin{cases} \text{MinLod} & \text{(from optional SPIR-V operand)} \\ 0 & \text{otherwise} \end{cases} \)
- \( \text{lod}_{\text{min}} = \text{max}(\text{sampler.lod}_{\text{min}}, \text{shaderOp.lod}_{\text{min}}) \)
- \( \text{lod}_{\text{max}} = \text{maxLod} \) (from sampler descriptor)

and \text{maxSamplerLodBias} is the value of the \texttt{VkPhysicalDeviceLimits} feature \texttt{maxSamplerLodBias}.

### Image Level(s) Selection

The image level(s) \( d, d_{\text{hi}}, \) and \( d_{\text{lo}} \) which texels are read from are determined by an image-level parameter \( d_{\text{b}} \), which is computed based on the LOD parameter, as follows:

\[
d_{j} = \begin{cases} \text{nearest}(d'), & \text{mipmapMode is VK_SAMPLER_MIPMAP_MODE_NEAREST} \\ d', & \text{otherwise} \end{cases}
\]

where:

\[
d' = \text{level}_{\text{base}} + \text{clamp}(\lambda, 0, q)
\]
\[
\text{nearest}(d^{'}) = \begin{cases} 
[d^{'} + 0.5] - 1, & \text{preferred} \\
[d^{'} + 0.5], & \text{alternative}
\end{cases}
\]

and:

\[\text{level}_{\text{base}} = \text{baseMipLevel}\]

\[q = \text{levelCount} - 1\]

\text{baseMipLevel} and \text{levelCount} are taken from the \text{subresourceRange} of the image view.

If the sampler's \text{mipmapMode} is \text{VK_SAMPLER_MIPMAP_MODE_NEAREST}, then the level selected is \(d = d_i\).

If the sampler's \text{mipmapMode} is \text{VK_SAMPLER_MIPMAP_MODE_LINEAR}, two neighboring levels are selected:

\[
\begin{align*}
\delta_i &= d_i - d_{hi} \\
d_{hi} &= \lfloor d_i \rfloor \\
d_{lo} &= \min(d_{hi} + 1, q)
\end{align*}
\]

\(\delta\) is the fractional value, quantized to the number of mipmap precision bits, used for linear filtering between levels.

**16.5.8. (s,t,r,q,a) to (u,v,w,a) Transformation**

The normalized texel coordinates are scaled by the image level dimensions and the array layer is selected.

This transformation is performed once for each level used in 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 \text{ConstOffset} or \text{Offset} operand, otherwise they are taken to be zero.

Operations then proceed to Unnormalized Texel Coordinate Operations.

### 16.6. Unnormalized Texel Coordinate Operations

#### 16.6.1. \((u,v,w,a)\) to \((i,j,k,l,n)\) Transformation And Array Layer Selection

The unnormalized texel coordinates are transformed to integer texel coordinates relative to the selected mipmap level.

The layer index \(l\) is computed as:

\[
l = \text{clamp}(\text{RNE}(a), 0, \text{layerCount} - 1) + \text{baseArrayLayer}
\]

where \text{layerCount} is the number of layers in the image subresource range of the image view, \text{baseArrayLayer} is the first layer from the subresource range, and where:

\[
\text{RNE}(a) = \begin{cases} 
\text{roundTiesToEven}(a) & \text{preferred, from IEEE Std 754-2008 Floating-Point Arithmetic} \\
\lfloor a + 0.5 \rfloor & \text{alternative}
\end{cases}
\]

The sample index \(n\) is assigned the value 0.

Nearest filtering (\text{VK_FILTER_NEAREST}) computes the integer texel coordinates that the unnormalized coordinates lie within:

\[
i = \lfloor u + \text{shift} \rfloor \\
j = \lfloor v + \text{shift} \rfloor \\
k = \lfloor w + \text{shift} \rfloor
\]

where:

\[
\text{shift} = 0.0
\]

Linear filtering (\text{VK_FILTER_LINEAR}) computes a set of neighboring coordinates which bound the unnormalized coordinates. The integer texel coordinates are combinations of \(i_0\) or \(i_1\), \(j_0\) or \(j_1\), \(k_0\) or \(k_1\), as well as weights \(\alpha\), \(\beta\), and \(\gamma\).
where:

\[
\begin{align*}
    i_0 &= \lfloor u - \text{shift} \rfloor \\
    i_1 &= i_0 + 1 \\
    j_0 &= \lfloor v - \text{shift} \rfloor \\
    j_1 &= j_0 + 1 \\
    k_0 &= \lfloor w - \text{shift} \rfloor \\
    k_1 &= k_0 + 1
\end{align*}
\]

\[
\begin{align*}
    \alpha &= \text{frac}(u - \text{shift}) \\
    \beta &= \text{frac}(v - \text{shift}) \\
    \gamma &= \text{frac}(w - \text{shift})
\end{align*}
\]

where:

\[
\text{shift} = 0.5
\]

and where:

\[
\text{frac}(x) = x - \lfloor x \rfloor
\]

where the number of fraction bits retained is specified by `VkPhysicalDeviceLimits::subTexelPrecisionBits`.

Cubic filtering (`VK_FILTER_CUBIC_EXT`) computes a set of neighboring coordinates which bound the unnormalized coordinates. The integer texel coordinates are combinations of \(i_0, i_1, i_2\) or \(i_3\), \(j_0, j_1, j_2\) or \(j_3\), \(k_0, k_1, k_2\) or \(k_3\), as well as weights \(\alpha, \beta, \) and \(\gamma\).

\[
\begin{align*}
    i_0 &= \lfloor u - \frac{3}{2} \rfloor \\
    i_1 &= i_0 + 1 \\
    i_2 &= i_1 + 1 \\
    i_3 &= i_2 + 1 \\
    j_0 &= \lfloor v - \frac{3}{2} \rfloor \\
    j_1 &= j_0 + 1 \\
    j_2 &= j_1 + 1 \\
    j_3 &= j_2 + 1 \\
    k_0 &= \lfloor w - \frac{3}{2} \rfloor \\
    k_1 &= k_0 + 1 \\
    k_2 &= k_1 + 1 \\
    k_3 &= k_2 + 1
\end{align*}
\]

\[
\begin{align*}
    \alpha &= \text{frac} \left( u - \frac{1}{2} \right) \\
    \beta &= \text{frac} \left( v - \frac{1}{2} \right) \\
    \gamma &= \text{frac} \left( w - \frac{1}{2} \right)
\end{align*}
\]

where:

\[
\text{frac}(x) = x - \lfloor x \rfloor
\]

where the number of fraction bits retained is specified by `VkPhysicalDeviceLimits`.
16.7. Integer Texel Coordinate Operations

The `OpImageFetch` and `OpImageFetchSparse` SPIR-V instructions may supply a LOD from which texels are to be fetched using the optional SPIR-V operand `Lod`. Other integer-coordinate operations must not. If the `Lod` is provided then it must be an integer.

The image level selected is:

\[
d = level_{base} + \begin{cases} 
Lod & \text{(from optional SPIR-V operand)} \\
0 & \text{otherwise}
\end{cases}
\]

If \(d\) does not lie in the range \([baseMipLevel, baseMipLevel + levelCount)\) then any values fetched are zero if `robustImageAccess2` is enabled, otherwise are undefined, and any writes (if supported) are discarded.

16.8. Image Sample Operations

16.8.1. Wrapping Operation

Cube images ignore the wrap modes specified in the sampler. Instead, if `VK_FILTER_NEAREST` is used within a mip level then `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE` is used, and if `VK_FILTER_LINEAR` is used within a mip level then sampling at the edges is performed as described earlier in the Cube map edge handling section.

The first integer texel coordinate \(i\) is transformed based on the `addressModeU` parameter of the sampler.

\[
i = \begin{cases} 
\text{mirror}((i \mod (2 \times size)) - size) & \text{for repeat} \\
\text{clamp}(i, 0, size - 1) & \text{for mirrored repeat} \\
\text{clamp}(i, size - 1, 0) & \text{for clamp to edge} \\
\text{clamp}(\text{mirror}(i), 0, size - 1) & \text{for clamp to border} \\
\text{clamp}(\text{mirror}(i), size - 1, 0) & \text{for mirror clamp to edge}
\end{cases}
\]

where:

\[
\text{mirror}(n) = \begin{cases} 
n & \text{for } n \geq 0 \\
-1 - n & \text{otherwise}
\end{cases}
\]

\(j\) (for 2D and Cube image) and \(k\) (for 3D image) are similarly transformed based on the `addressModeV` and `addressModeW` parameters of the sampler, respectively.

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:

$$
\tau[R] = \tau_{i0j0}[level_{base},comp] \\
\tau[G] = \tau_{i0j0}[level_{base},comp] \\
\tau[B] = \tau_{i0j0}[level_{base},comp] \\
\tau[A] = \tau_{i0j0}[level_{base},comp]
$$

If the operation does use the ConstOffsets image operand then the offsets allow a custom filter to be defined:

$$
\tau[R] = \tau_{i0j0} + \Delta_0[level_{base},comp] \\
\tau[G] = \tau_{i0j0} + \Delta_1[level_{base},comp] \\
\tau[B] = \tau_{i0j0} + \Delta_2[level_{base},comp] \\
\tau[A] = \tau_{i0j0} + \Delta_3[level_{base},comp]
$$

where:

$$
\tau[level_{base},comp] = \begin{cases} 
\tau[level_{base},R], & \text{for } comp = 0 \\
\tau[level_{base},G], & \text{for } comp = 1 \\
\tau[level_{base},B], & \text{for } comp = 2 \\
\tau[level_{base},A], & \text{for } comp = 3 
\end{cases}
$$

OpImage*Gather must not be used on a sampled image with sampler Y′C′C′C′R 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:

\[
\begin{align*}
    \tau_{3D} &= \sum_{k = k_0}^{k_1} \sum_{j = j_0}^{j_1} \sum_{i = i_0}^{i_1} (w_i)(w_j)(w_k)\tau_{ijk} \\
    \tau_{2D} &= \sum_{j = j_0}^{j_1} \sum_{i = i_0}^{i_1} (w_i)(w_j)\tau_{ij} \\
    \tau_{1D} &= \sum_{i = i_0}^{i_1} (w_i)\tau_i
\end{align*}
\]

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 Cubic Filtering

Within a mip level, `VK_FILTER_CUBIC_EXT`, filtering computes a weighted average of 64 (for 3D), 16 (for 2D), or 4 (for 1D) texel values, together with their Catmull-Rom weights.

Catmull-Rom weights are derived from the fractions computed earlier.
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:

\[
\begin{align*}
T_{3D} &= \sum_{k = j_0}^{k_3} \sum_{j = j_0}^{j_3} \sum_{i = i_0}^{i_3} (w_i)(w_j)(w_k)T_{ijk} \\
T_{2D} &= \sum_{j = j_0}^{j_3} \sum_{i = i_0}^{i_3} (w_i)(w_j)T_{ij} \\
T_{1D} &= \sum_{i = i_0}^{i_3} (w_i)T_i
\end{align*}
\]

However, if the reduction mode is `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX`, the process operates on the above set of multiple texels, together with their weights, computing a component-wise minimum or maximum, respectively, of the components of the set of texels with non-zero weights.

**Texel Mipmap Filtering**

`VK_SAMPLER_MIPMAP_MODE_NEAREST` filtering returns the value of a single mipmap level,

\[\tau = \tau[d].\]

`VK_SAMPLER_MIPMAP_MODE_LINEAR` filtering combines the values of multiple mipmap levels (\(\tau[hi]\) and \(\tau[lo]\)), together with their linear weights.

The linear weights are derived from the fraction computed earlier:

\[w_{hi} = (1 - \delta)\]
\[w_{lo} = (\delta)\]
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:

\[ \tau = (w_h)i + (w_l)o \]

**Texel Anisotropic Filtering**

Anisotropic filtering is enabled by the `anisotropyEnable` in the sampler. When enabled, the image filtering scheme accounts for a degree of anisotropy.

The particular scheme for anisotropic texture filtering is implementation-dependent. Implementations should consider the `magFilter`, `minFilter` and `mipmapMode` of the sampler to control the specifics of the anisotropic filtering scheme used. In addition, implementations should consider `minLod` and `maxLod` of the sampler.

The following describes one particular approach to implementing anisotropic filtering for the 2D Image case, implementations may choose other methods:

Given a `magFilter`, `minFilter` of `VK_FILTER_LINEAR` and a `mipmapMode` of `VK_SAMPLER_MIPMAP_MODE_NEAREST`:

Instead of a single isotropic sample, N isotropic samples are sampled within the image footprint of the image level d to approximate an anisotropic filter. The sum \( \tau_{2D_{aniso}} \) is defined using the single isotropic \( \tau_{2D}(u,v) \) at level d.

\[
\begin{align*}
\tau_{2D_{aniso}} &= \frac{1}{N} \sum_{i=1}^{N} \tau_{2D} \left( u \left( x - \frac{1}{2} + \frac{i}{N+1}, y \right), \left( v \left( x - \frac{1}{2} + \frac{i}{N+1}, y \right) \right) ) \right), \quad \text{when } \rho_x > \rho_y \\
\tau_{2D_{aniso}} &= \frac{1}{N} \sum_{i=1}^{N} \tau_{2D} \left( u \left( x, y - \frac{1}{2} + \frac{i}{N+1}, \right), \left( v \left( x, y - \frac{1}{2} + \frac{i}{N+1} \right) \right) ) \right), \quad \text{when } \rho_y \geq \rho_x
\end{align*}
\]

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 \texttt{OpImage*Dref} instructions.

• All Texel output operations: Performed by \texttt{OpImageWrite}.

• Projection: Performed by all \texttt{OpImage*Proj} instructions.

• Derivative Image Operations, Cube Map Operations, Scale Factor Operation, Level-of-Detail Operation and Image Level(s) Selection, and Texel Anisotropic Filtering: Performed by all \texttt{OpImageSample*} and \texttt{OpImageSparseSample*} instructions.

• (s,t,r,q,a) to (u,v,w,a) Transformation, Wrapping, and (u,v,w,a) to (i,j,k,l,n) Transformation And Array Layer Selection: Performed by all \texttt{OpImageSample}, \texttt{OpImageSparseSample}, and \texttt{OpImage*Gather} instructions.

• Texel Gathering: Performed by \texttt{OpImage*Gather} instructions.

• Texel Filtering: Performed by all \texttt{OpImageSample*} and \texttt{OpImageSparseSample*} instructions.

• Sparse Residency: Performed by all \texttt{OpImageSparse*} instructions.

16.10. Image Query Instructions

16.10.1. Image Property Queries

\texttt{OpImageQuerySize}, \texttt{OpImageQuerySizeLod}, \texttt{OpImageQueryLevels}, and \texttt{OpImageQuerySamples} query properties of the image descriptor that would be accessed by a shader image operation. They return 0 if the bound descriptor is a null descriptor.

\texttt{OpImageQuerySizeLod} returns the size of the image level identified by the \texttt{Level of Detail} operand. If that level does not exist in the image, and the descriptor is not null, then the value returned is undefined.

16.10.2. Lod Query

\texttt{OpImageQueryLod} returns the Lod parameters that would be used in an image operation with the given image and coordinates. If the descriptor that would be accessed is a null descriptor then (0, 0) is returned. Otherwise, the steps described in this chapter are performed as if for \texttt{OpImageSampleImplicitLod}, up to \texttt{Scale Factor Operation, Level-of-Detail Operation and Image Level(s) Selection}. The return value is the vector \((\lambda', d)\). 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. Performance Queries are supported if the associated extension is available.

17.1. Query Pools

Queries are managed using query pool objects. Each query pool is a collection of a specific number of queries of a particular type.

Query pools are represented by VkQueryPool handles:

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

To create a query pool, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateQueryPool(
    VkDevice device,
    const VkQueryPoolCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkQueryPool* pQueryPool);
```

- `device` is the logical device that creates the query pool.
- `pCreateInfo` is a pointer to a VkQueryPoolCreateInfo structure containing the number and type of queries to be managed by the pool.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pQueryPool` is a pointer to a VkQueryPool handle in which the resulting query pool object is returned.

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_OFDEVICE_MEMORY

The VkQueryPoolCreateInfo structure is defined as:

```cpp
// Provided by VK_VERSION_1_0
typedef struct VkQueryPoolCreateInfo {
  VkStructureType sType;
  const void* pNext;
  VkQueryPoolCreateFlags flags;
  VkQueryType queryType;
  uint32_t queryCount;
  VkQueryPipelineStatisticFlags pipelineStatistics;
} VkQueryPoolCreateInfo;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- flags is reserved for future use.
- queryType is a VkQueryType value specifying the type of queries managed by the pool.
- queryCount is the number of queries managed by the pool.
- pipelineStatistics is a bitmask of VkQueryPipelineStatisticFlagBits specifying which counters will be returned in queries on the new pool, as described below in Pipeline Statistics Queries.
pipelineStatistics is ignored if queryType is not VK_QUERY_TYPE_PIPELINE_STATISTICS.

Valid Usage

- VUID-VkQueryPoolCreateInfo-queryType-00791
  If the pipeline statistics queries 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-queryType-03222
  If queryType is VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR, the pNext chain must include a VkQueryPoolPerformanceCreateInfoKHR structure

- 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

- VUID-VkQueryPoolCreateInfo-queryType-05049
  If queryType is VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR then queryCount must be less than or equal to the maximum of all VkPerformanceQueryReservationInfoKHR::maxPerformanceQueriesPerPool 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 or a pointer to a valid instance of VkQueryPoolPerformanceCreateInfoKHR

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

- VUID-VkQueryPoolCreateInfo-flags-zerobitmask

777
**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.

The VkQueryPoolPerformanceCreateInfoKHR structure is defined as:

```c
// Provided by VK_KHR_performance_query
typedef struct VkQueryPoolPerformanceCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    uint32_t queueFamilyIndex;
    uint32_t counterIndexCount;
    const uint32_t* pCounterIndices;
} VkQueryPoolPerformanceCreateInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **queueFamilyIndex** is the queue family index to create this performance query pool for.
- **counterIndexCount** is the length of the **pCounterIndices** array.
- **pCounterIndices** is a pointer to an array of indices into the vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR::*pCounters to enable in this performance query pool.

### Valid Usage

- VUID-VkQueryPoolPerformanceCreateInfoKHR-queueFamilyIndex-03236
  queueFamilyIndex must be a valid queue family index of the device

- VUID-VkQueryPoolPerformanceCreateInfoKHR-performanceCounterQueryPools-03237
  The performanceCounterQueryPools feature must be enabled

- VUID-VkQueryPoolPerformanceCreateInfoKHR-pCounterIndices-03321
  Each element of **pCounterIndices** must be in the range of counters reported by vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR for the queue family specified in queueFamilyIndex

### Valid Usage (Implicit)

- VUID-VkQueryPoolPerformanceCreateInfoKHR-sType-sType
sType must be VK_STRUCTURE_TYPE_QUERY_POOL_PERFORMANCE_CREATE_INFO_KHR

- VUID-VkQueryPoolPerformanceCreateInfoKHR-pCounterIndices-parameter
pCounterIndices must be a valid pointer to an array of counterIndexCount uint32_t values
- VUID-VkQueryPoolPerformanceCreateInfoKHR-counterIndexCount-arraylength
counterIndexCount must be greater than 0

To query the number of passes required to query a performance query pool on a physical device, call:

```c
// Provided by VK_KHR_performance_query
void vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR(
    VkPhysicalDevice physicalDevice,
    const VkQueryPoolPerformanceCreateInfoKHR* pPerformanceQueryCreateInfo,
    uint32_t* pNumPasses);
```

- physicalDevice is the handle to the physical device whose queue family performance query counter properties will be queried.
- pPerformanceQueryCreateInfo is a pointer to a VkQueryPoolPerformanceCreateInfoKHR of the performance query that is to be created.
- pNumPasses is a pointer to an integer related to the number of passes required to query the performance query pool, as described below.

The pPerformanceQueryCreateInfo member VkQueryPoolPerformanceCreateInfoKHR::queueFamilyIndex must be a queue family of physicalDevice. The number of passes required to capture the counters specified in the pPerformanceQueryCreateInfo member VkQueryPoolPerformanceCreateInfoKHR::pCounters is returned in pNumPasses.

Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR-physicalDevice-parameter
  physicalDevice must be a valid VkPhysicalDevice handle
- VUID-vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR-pPerformanceQueryCreateInfo-parameter
  pPerformanceQueryCreateInfo must be a valid pointer to a valid VkQueryPoolPerformanceCreateInfoKHR structure
- VUID-vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR-pNumPasses-parameter
  pNumPasses must be a valid pointer to a uint32_t value

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 \texttt{VkQueryPoolCreateInfo::queryType}, specifying the type of queries managed by the pool, are:

\begin{verbatim}
// 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,
    // Provided by VK_KHR_performance_query
    VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR = 1000116000,
} VkQueryType;
\end{verbatim}

- \texttt{VK_QUERY_TYPE_OCCLUSION} specifies an \textit{occlusion query}.
- \texttt{VK_QUERY_TYPE_PIPELINE_STATISTICS} specifies a \textit{pipeline statistics query}.
- \texttt{VK_QUERY_TYPE_TIMESTAMP} specifies a \textit{timestamp query}.
- \texttt{VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR} specifies a \textit{performance query}.

## 17.2. Query Operation

The operation of queries is controlled by the commands \texttt{vkCmdBeginQuery}, \texttt{vkCmdEndQuery}, \texttt{vkCmdResetQueryPool}, \texttt{vkCmdCopyQueryPoolResults}, \texttt{vkCmdWriteTimestamp2KHR}, and \texttt{vkCmdWriteTimestamp}.

In order for a \texttt{VkCommandBuffer} to record query management commands, the queue family for which its \texttt{VkCommandPool} was created \textbf{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 \textit{unavailable} or \textit{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 \texttt{vkCmdResetQueryPool} or \texttt{vkResetQueryPool} sets the status to unavailable and makes the numerical results undefined. Performing a query operation with \texttt{vkCmdBeginQuery} and \texttt{vkCmdEndQuery} changes the status to available when the query \textit{finishes}, and updates the numerical results. Both the availability status and numerical results are retrieved by calling either \texttt{vkGetQueryPoolResults} or \texttt{vkCmdCopyQueryPoolResults}.

Query commands, for the same query and submitted to the same queue, execute in their entirety in \textit{submission order}, relative to each other. In effect there is an implicit execution dependency from each such query command to all query commands previously submitted to the same queue. There is one significant exception to this; if the \texttt{flags} parameter of \texttt{vkCmdCopyQueryPoolResults} does not include \texttt{VK_QUERY_RESULT_WAIT_BIT}, execution of \texttt{vkCmdCopyQueryPoolResults} \textbf{may} happen-before the results of \texttt{vkCmdEndQuery} are available.

After query pool creation, each query \textbf{must} be reset before it is used. Queries \textbf{must} also be reset between uses.

If a logical device includes multiple physical devices, then each command that writes a query \textbf{must} execute on a single physical device, and any call to \texttt{vkCmdBeginQuery} \textbf{must} execute the
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 `[firstQuery, firstQuery + queryCount - 1]` to unavailable.

If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, this command sets the status of query indices `[firstQuery, firstQuery + queryCount - 1]` to unavailable for each pass of `queryPool`, as indicated by a call to `vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR`.

**Note**

Because `vkCmdResetQueryPool` resets all the passes of the indicated queries, applications must not record a `vkCmdResetQueryPool` command for a `queryPool` created with `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` in a command buffer that needs to be submitted multiple times as indicated by a call to `vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR`. Otherwise applications will never be able to complete the recorded queries.

**Valid Usage**

- VUID-vkCmdResetQueryPool-firstQuery-00796
  `firstQuery` must be less than the number of queries in `queryPool`

- VUID-vkCmdResetQueryPool-firstQuery-00797
  The sum of `firstQuery` and `queryCount` must be less than or equal to the number of queries in `queryPool`

- VUID-vkCmdResetQueryPool-None-02841
  All queries used by the command must not be active

- VUID-vkCmdResetQueryPool-firstQuery-02862
  If `queryPool` was created with `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, this command must not be recorded in a command buffer that, either directly or through secondary command buffers, also contains begin commands for a query from the set of queries `[firstQuery,`
Valid Usage (Implicit)

- VUID-vkCmdResetQueryPool-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle
- VUID-vkCmdResetQueryPool-queryPool-parameter
  queryPool must be a valid VkQueryPool handle
- VUID-vkCmdResetQueryPool-commandBuffer-recording
  commandBuffer must be in the recording state
- VUID-vkCmdResetQueryPool-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations
- VUID-vkCmdResetQueryPool-renderpass
  This command must only be called outside of a render pass instance
- VUID-vkCmdResetQueryPool-commonparent
  Both of commandBuffer, and queryPool must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
</tr>
</tbody>
</table>

To reset a range of queries in a query pool on the host, call:

```c
// Provided by VK_VERSION_1_2
def 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.

If queryPool is VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR this command sets the status of query indices [firstQuery, firstQuery + queryCount - 1] to unavailable for each pass.

Valid Usage

- VUID-vkResetQueryPool-None-02665
  The hostQueryReset feature must be enabled
- VUID-vkResetQueryPool-firstQuery-02666
  firstQuery must be less than the number of queries in queryPool
- VUID-vkResetQueryPool-firstQuery-02667
  The sum of firstQuery and queryCount must be less than or equal to the number of queries in queryPool
- VUID-vkResetQueryPool-firstQuery-02741
  Submitted commands that refer to the range specified by firstQuery and queryCount in queryPool must have completed execution
- VUID-vkResetQueryPool-firstQuery-02742
  The range of queries specified by firstQuery and queryCount in queryPool must not be in use by calls to vkGetQueryPoolResults or vkResetQueryPool in other threads

Valid Usage (Implicit)

- VUID-vkResetQueryPool-device-parameter
  device must be a valid VkDevice handle
- VUID-vkResetQueryPool-queryPool-parameter
  queryPool must be a valid VkQueryPool handle
- VUID-vkResetQueryPool-queryPool-parent
  queryPool must have been created, allocated, or retrieved from device

Once queries are reset and ready for use, query commands can be issued to a command buffer. Occlusion queries and pipeline statistics queries count events - drawn samples and pipeline stage invocations, respectively - resulting from commands that are recorded between a vkCmdBeginQuery command and a vkCmdEndQuery command within a specified command buffer, effectively scoping a set of drawing and/or dispatching commands. Timestamp queries write timestamps to a query pool. Performance queries record performance counters to a query pool.
A query **must** begin and end in the same command buffer, although if it is a primary command buffer, and the inherited queries feature is enabled, it **can** execute secondary command buffers during the query operation. For a secondary command buffer to be executed while a query is active, it **must** set the occlusionQueryEnable, queryFlags, and/or pipelineStatistics members of VkCommandBufferInheritanceInfo to conservative values, as described in the Command Buffer Recording section. A query **must** either begin and end inside the same subpass of a render pass instance, or **must** both begin and end outside of a render pass instance (i.e. contain entire render pass instances).

If queries are used while executing a render pass instance that has multiview enabled, the query uses N consecutive query indices in the query pool (starting at query) where N is the number of bits set in the view mask in the subpass the query is used in. How the numerical results of the query are distributed among the queries is implementation-dependent. For example, some implementations may write each view's results to a distinct query, while other implementations may write the total result to the first query and write zero to the other queries. However, the sum of the results in all the queries **must** accurately reflect the total result of the query summed over all views. Applications **can** sum the results from all the queries to compute the total result.

Queries used with multiview rendering **must** not span subpasses, i.e. they **must** begin and end in the same subpass.

To begin a query, call:

```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.

**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 precise occlusion queries feature is not enabled, or the queryType used to create queryPool was not VKQUERY_TYPE_OCCLUSION, flags must not contain VK_QUERY_CONTROL_PRECISE_BIT

- VUID-vkCmdBeginQuery-queryType-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 VKQUERY_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-queryType-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

- VUID-vkCmdBeginQuery-queryPool-03223
  If queryPool was created with a queryType of VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR, the profiling lock must have been held before vkBeginCommandBuffer was called on commandBuffer

- VUID-vkCmdBeginQuery-queryPool-03224
  If queryPool was created with a queryType of VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR and one of the counters used to create queryPool was VKPERFORMANCE_COUNTER_SCOPE_COMMAND_BUFFER_KHR, the query begin must be the first recorded command in commandBuffer

- VUID-vkCmdBeginQuery-queryPool-03225
  If queryPool was created with a queryType of VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR and one of the counters used to create queryPool was VKPERFORMANCE_COUNTER_SCOPE_RENDER_PASS_KHR, the begin command must not be recorded within a render pass instance
If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` and another query pool with a `queryType` `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` has been used within `commandBuffer`, its parent primary command buffer or secondary command buffer recorded within the same parent primary command buffer as `commandBuffer`, the `performanceCounterMultipleQueryPools` feature must be enabled.

If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, this command must not be recorded in a command buffer that, either directly or through secondary command buffers, also contains a `vkCmdResetQueryPool` command affecting the same query.

**Valid Usage (Implicit)**

- `commandBuffer` must be a valid `VkCommandBuffer` handle.
- `queryPool` must be a valid `VkQueryPool` handle.
- `flags` must be a valid combination of `VkQueryControlFlagBits` values.
- `commandBuffer` must be in the recording state.
- The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations.
- Both of `commandBuffer`, and `queryPool` must have been created, allocated, or retrieved from the same `VkDevice`.

**Host Synchronization**

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

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</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.

As queries operate asynchronously, ending a query does not immediately set the query's status to available. A query is considered finished when the final results of the query are ready to be retrieved by `vkGetQueryPoolResults` and `vkCmdCopyQueryPoolResults`, and this is when the query's status is set to available.

Once a query is ended the query must finish in finite time, unless the state of the query is changed using other commands, e.g. by issuing a reset of the query.

### Valid Usage

- **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-queryPool-03227**
  If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` and one or more of the counters used to create `queryPool` was `VK_PERFORMANCE_COUNTER_SCOPE_COMMAND_BUFFER_KHR`, the `vkCmdEndQuery` must be the last recorded command in `commandBuffer`.

- **VUID-vkCmdEndQuery-queryPool-03228**
  If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` and one or more of the counters used to create `queryPool` was `VK_PERFORMANCE_COUNTER_SCOPE_RENDER_PASS_KHR`, the `vkCmdEndQuery` must not be recorded within 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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</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.

- Performance queries store results in a tightly packed array whose type is determined by the unit member of the corresponding VkPerformanceCounterKHR.

- If VK_QUERY_RESULT_WITH_AVAILABILITY_BIT is used, the final element of each query’s result is an integer indicating whether the query’s result is available, with any non-zero value indicating that it is available.

- Occlusion queries write one integer value - the number of samples passed. Pipeline statistics queries write one integer value for each bit that is enabled in the pipelineStatistics when the pool is created, and the statistics values are written in bit order starting from the least significant bit. Timestamp queries write one integer value. Performance queries write one VkPerformanceCounterResultKHR value for each VkPerformanceCounterKHR in the query.

- 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,
    VkQueryResultFlags flags);
```

- device is the logical device that owns the query pool.
- queryPool is the query pool managing the queries containing the desired results.
- firstQuery is the initial query index.
- queryCount is the number of queries to read.
- dataSize is the size in bytes of the buffer pointed to by pData.
- pData is a pointer to a user-allocated buffer where the results will be written
- stride is the stride in bytes between results for individual queries within pData.
- flags is a bitmask of VkQueryResultFlagBits specifying how and when results are returned.
The range of queries read is defined by \([\text{firstQuery}, \text{firstQuery} + \text{queryCount} - 1]\). For pipeline statistics queries, each query index in the pool contains one integer value for each bit that is enabled in \(\text{VkQueryPoolCreateInfo::pipelineStatistics}\) when the pool is created.

If no bits are set in \(\text{flags}\), and all requested queries are in the available state, results are written as an array of 32-bit unsigned integer values. The behavior when not all queries are available, is described below.

If \(\text{VK_QUERY_RESULT_64_BIT}\) is not set and the result overflows a 32-bit value, the value may either wrap or saturate. Similarly, if \(\text{VK_QUERY_RESULT_64_BIT}\) is set and the result overflows a 64-bit value, the value may either wrap or saturate.

If \(\text{VK_QUERY_RESULT_WAIT_BIT}\) is set, Vulkan will wait for each query to be in the available state before retrieving the numerical results for that query. In this case, \(\text{vkGetQueryPoolResults}\) is guaranteed to succeed and return \(\text{VK_SUCCESS}\) if the queries become available in a finite time (i.e. if they have been issued and not reset). If queries will never finish (e.g. due to being reset but not issued), then \(\text{vkGetQueryPoolResults}\) may not return in finite time.

If \(\text{VK_QUERY_RESULT_WAIT_BIT}\) and \(\text{VK_QUERY_RESULT_PARTIAL_BIT}\) are both not set then no result values are written to \(\text{pData}\) for queries that are in the unavailable state at the time of the call, and \(\text{vkGetQueryPoolResults}\) returns \(\text{VK_NOT_READY}\). However, availability state is still written to \(\text{pData}\) for those queries if \(\text{VK_QUERY_RESULT_WITH_AVAILABILITY_BIT}\) is set.

If \(\text{VK_QUERY_RESULT_WAIT_BIT}\) is not set, \(\text{vkGetQueryPoolResults}\) may return \(\text{VK_NOT_READY}\) if there are queries in the unavailable state.

**Note**

Applications must take care to ensure that use of the \(\text{VK_QUERY_RESULT_WAIT_BIT}\) bit has the desired effect.

For example, if a query has been used previously and a command buffer records the commands \(\text{vkCmdResetQueryPool}, \text{vkCmdBeginQuery},\) and \(\text{vkCmdEndQuery}\) for that query, then the query will remain in the available state until \(\text{vkResetQueryPool}\) is called or the \(\text{vkCmdResetQueryPool}\) 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 \(\text{VK_QUERY_RESULT_WAIT_BIT}\) is used in combination with \(\text{VK_QUERY_RESULT_WITH_AVAILABILITY_BIT}\). In this case, the returned availability status may reflect the result of a previous use of the query unless \(\text{vkResetQueryPool}\) is called or the \(\text{vkCmdResetQueryPool}\) 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 \(\text{VK_QUERY_RESULT_PARTIAL_BIT}\) is set, \(\text{VK_QUERY_RESULT_WAIT_BIT}\) is not set, and the query’s status is...
unavailable, an intermediate result value between zero and the final result value is written to \( pData \) for that query.

If \( VK_QUERY_RESULT_WITH_AVAILABILITY_BIT \) is set, the final integer value written for each query is non-zero if the query's status was available or zero if the status was unavailable. When \( VK_QUERY_RESULT_WITH_AVAILABILITY_BIT \) is used, implementations must guarantee that if they return a non-zero availability value then the numerical results must be valid, assuming the results are not reset by a subsequent command.

Note

Satisfying this guarantee may require careful ordering by the application, e.g. to read the availability status before reading the results.

If \( \text{VkPhysicalDeviceVulkanSC10Properties}::\text{deviceNoDynamicHostAllocations} \) is \( VK_TRUE \), \( \text{vkGetQueryPoolResults} \) must not return \( VK_ERROR_OUT_OF_HOST_MEMORY \).

Valid Usage

- VUID-vkGetQueryPoolResults-firstQuery-00813
  \( \text{firstQuery} \) must be less than the number of queries in \( \text{queryPool} \)

- VUID-vkGetQueryPoolResults-flags-02828
  If \( VK_QUERY_RESULT_64_BIT \) is not set in \( \text{flags} \) and the \( \text{queryType} \) used to create \( \text{queryPool} \) was not \( VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR \), then \( pData \) and \( \text{stride} \) must be multiples of 4

- VUID-vkGetQueryPoolResults-flags-00815
  If \( VK_QUERY_RESULT_64_BIT \) is set in \( \text{flags} \) then \( pData \) and \( \text{stride} \) must be multiples of 8

- VUID-vkGetQueryPoolResults-queryType-03229
  If the \( \text{queryType} \) used to create \( \text{queryPool} \) was \( VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR \), then \( pData \) and \( \text{stride} \) must be multiples of the size of \( \text{VkPerformanceCounterResultKHR} \)

- VUID-vkGetQueryPoolResults-queryType-04519
  If the \( \text{queryType} \) used to create \( \text{queryPool} \) was \( VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR \), then \( \text{stride} \) must be large enough to contain \( \text{VkQueryPoolPerformanceCreateInfoKHR} ::\text{counterIndexCount} \) used to create \( \text{queryPool} \) times the size of \( \text{VkPerformanceCounterResultKHR} \)

- VUID-vkGetQueryPoolResults-firstQuery-00816
  The sum of \( \text{firstQuery} \) and \( \text{queryCount} \) must be less than or equal to the number of queries in \( \text{queryPool} \)

- VUID-vkGetQueryPoolResults-dataSize-00817
  \( \text{dataSize} \) must be large enough to contain the result of each query, as described here

- VUID-vkGetQueryPoolResults-queryType-00818
  If the \( \text{queryType} \) used to create \( \text{queryPool} \) was \( VK_QUERY_TYPE_TIMESTAMP \), \( \text{flags} \) must not contain \( VK_QUERY_RESULT_PARTIAL_BIT \)

- VUID-vkGetQueryPoolResults-queryType-03230
  If the \( \text{queryType} \) used to create \( \text{queryPool} \) was \( VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR \), \( \text{flags} \) must not contain \( VK_QUERY_RESULT_WITH_AVAILABILITY_BIT \), \( VK_QUERY_RESULT_PARTIAL_BIT \) or...
VK_QUERY_RESULT_64_BIT

• VUID-vkGetQueryPoolResults-queryType-03231
  If the queryType used to create queryPool was VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR, the queryPool must have been recorded once for each pass as retrieved via a call to vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR

Valid Usage (Implicit)

• 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:

// 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.

**vkCmdCopyQueryPoolResults** is guaranteed to see the effect of previous uses of **vkCmdResetQueryPool** in the same queue, without any additional synchronization. Thus, the results will always reflect the
most recent use of the query.

flags has the same possible values described above for the flags parameter of vkGetQueryPoolResults, but the different style of execution causes some subtle behavioral differences. Because vkCmdCopyQueryPoolResults executes in order with respect to other query commands, there is less ambiguity about which use of a query is being requested.

Results for all requested occlusion queries, pipeline statistics queries, and timestamp queries are written as 64-bit unsigned integer values if VK_QUERY_RESULT_64_BIT is set or 32-bit unsigned integer values otherwise. Performance queries store results in a tightly packed array whose type is determined by the unit member of the corresponding VkPerformanceCounterKHR.

If neither of VK_QUERY_RESULT_WAIT_BIT and VK_QUERY_RESULT_WITH_AVAILABILITY_BIT are set, results are only written out for queries in the available state.

If VK_QUERY_RESULT_WAIT_BIT is set, the implementation will wait for each query’s status to be in the available state before retrieving the numerical results for that query. This is guaranteed to reflect the most recent use of the query on the same queue, assuming that the query is not being simultaneously used by other queues. If the query does not become available in a finite amount of time (e.g. due to not issuing a query since the last reset), a VK_ERROR_DEVICE_LOST error may occur.

Similarly, if VK_QUERY_RESULT_WITH_AVAILABILITY_BIT is set and VK_QUERY_RESULT_WAIT_BIT is not set, the availability is guaranteed to reflect the most recent use of the query on the same queue, assuming that the query is not being simultaneously used by other queues. As with vkGetQueryPoolResults, implementations must guarantee that if they return a non-zero availability value, then the numerical results are valid.

If VK_QUERY_RESULT_PARTIAL_BIT is set, VK_QUERY_RESULT_WAIT_BIT is not set, and the query’s status is unavailable, an intermediate result value between zero and the final result value is written for that query.

VK_QUERY_RESULT_PARTIAL_BIT must not be used if the pool’s queryType is VK_QUERY_TYPE_TIMESTAMP.

vkCmdCopyQueryPoolResults is considered to be a transfer operation, and its writes to buffer memory must be synchronized using VK_PIPELINE_STAGE_TRANSFER_BIT and VK_ACCESS_TRANSFER_WRITE_BIT before using the results.

---

**Valid Usage**

- VUID-vkCmdCopyQueryPoolResults-dstOffset-00819
  - dstOffset must be less than the size of dstBuffer
- VUID-vkCmdCopyQueryPoolResults-firstQuery-00820
  - firstQuery must be less than the number of queries in queryPool
- VUID-vkCmdCopyQueryPoolResults-firstQuery-00821
  - The sum of firstQuery and queryCount must be less than or equal to the number of queries in queryPool
- VUID-vkCmdCopyQueryPoolResults-flags-00822
  - If VK_QUERY_RESULT_64_BIT is not set in flags then dstOffset and stride must be multiples
• VUID-vkCmdCopyQueryPoolResults-flags-00823
If VK_QUERY_RESULT_64_BIT is set in flags then dstOffset and stride must be multiples of 8

• VUID-vkCmdCopyQueryPoolResults-dstBuffer-00824
dstBuffer must have enough storage, from dstOffset, to contain the result of each query, as described here

• VUID-vkCmdCopyQueryPoolResults-dstBuffer-00825
dstBuffer must have been created with VK_BUFFER_USAGE_TRANSFER_DST_BIT usage flag

• VUID-vkCmdCopyQueryPoolResults-dstBuffer-00826
If dstBuffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

• VUID-vkCmdCopyQueryPoolResults-queryType-00827
If the queryType used to create queryPool was VK_QUERY_TYPE_TIMESTAMP, flags must not contain VK_QUERY_RESULT_PARTIAL_BIT

• VUID-vkCmdCopyQueryPoolResults-queryType-03232
If the queryType used to create queryPool was VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR, VkPhysicalDevicePerformanceQueryPropertiesKHR::allowCommandBufferQueryCopies must be VK_TRUE

• VUID-vkCmdCopyQueryPoolResults-queryType-03233
If the queryType used to create queryPool was VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR, flags must not contain VK_QUERY_RESULT_WITH_AVAILABILITY_BIT, VK_QUERY_RESULT_PARTIAL_BIT or VK_QUERY_RESULT_64_BIT

• VUID-vkCmdCopyQueryPoolResults-queryType-03234
If the queryType used to create queryPool was VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR, the queryPool must have been submitted once for each pass as retrieved via a call to vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR

Valid Usage (Implicit)

• VUID-vkCmdCopyQueryPoolResults-commandBuffer-parameter
commandBuffer must be a valid VkCommandBuffer handle

• VUID-vkCmdCopyQueryPoolResults-queryPool-parameter
queryPool must be a valid VkQueryPool handle

• VUID-vkCmdCopyQueryPoolResults-dstBuffer-parameter
dstBuffer must be a valid VkBuffer handle

• VUID-vkCmdCopyQueryPoolResults-flags-parameter
flags must be a valid combination of VkQueryResultFlagBits values

• VUID-vkCmdCopyQueryPoolResults-commandBuffer-recording
commandBuffer must be in the recording state

• VUID-vkCmdCopyQueryPoolResults-commandBuffer-cmdpool
The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations
Rendering operations such as clears, MSAA resolves, attachment load/store operations, and blits may count towards the results of queries. This behavior is implementation-dependent and may vary depending on the path used within an implementation. For example, some implementations have several types of clears, some of which may include vertices and some not.

### 17.3. Occlusion Queries

Occlusion queries track the number of samples that pass the per-fragment tests for a set of drawing commands. As such, occlusion queries are only available on queue families supporting graphics operations. The application can then use these results to inform future rendering decisions. An occlusion query is begun and ended by calling `vkCmdBeginQuery` and `vkCmdEndQuery`, respectively. When an occlusion query begins, the count of passing samples always starts at zero. For each drawing command, the count is incremented as described in Sample Counting. If `flags` does not contain `VK_QUERY_CONTROL_PRECISE_BIT` an implementation may generate any non-zero result value for the query if the count of passing samples is non-zero.

#### Note

Not setting `VK_QUERY_CONTROL_PRECISE_BIT` mode may be more efficient on some implementations, and should be used where it is sufficient to know a boolean result on whether any samples passed the per-fragment tests. In this case, some implementations may only return zero or one, indifferent to the actual number of samples passing the per-fragment tests.

When an occlusion query finishes, the result for that query is marked as available. The application can then either copy the result to a buffer (via `vkCmdCopyQueryPoolResults`) or request it be put into
host memory (via \texttt{vkGetQueryPoolResults}).

\begin{itemize}
\item If occluding geometry is not drawn first, samples \textbf{can} pass the depth test, but still not be visible in a final image.
\end{itemize}

### 17.4. Pipeline Statistics Queries

Pipeline statistics queries allow the application to sample a specified set of \texttt{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 \texttt{pipelineStatisticsQuery} member of the \texttt{VkPhysicalDeviceFeatures} object (see \texttt{vkGetPhysicalDeviceFeatures} and \texttt{vkCreateDevice} for detecting and requesting this query type on a \texttt{VkDevice}).

A pipeline statistics query is begun and ended by calling \texttt{vkCmdBeginQuery} and \texttt{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 \textbf{must} be configured on the query pool when it is created. If a statistic counter is issued on a command buffer that does not support the corresponding operation, the value of that counter is undefined after the query has finished. At least one statistic counter relevant to the operations supported on the recording command buffer \textbf{must} be enabled.

Bits which \textbf{can} be set to individually enable pipeline statistics counters for query pools with \texttt{VkQueryPoolCreateInfo::pipelineStatistics}, and for secondary command buffers with \texttt{VkCommandBufferInheritanceInfo::pipelineStatistics}, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkQueryPipelineStatisticFlagBits {
   VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_VERTICES_BIT = 0x00000001,
   VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_PRIMITIVES_BIT = 0x00000002,
   VK_QUERY_PIPELINE_STATISTIC_VERTEX_SHADER_INVOCATIONS_BIT = 0x00000004,
   VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_INVOCATIONS_BIT = 0x00000008,
   VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_PRIMITIVES_BIT = 0x00000010,
   VK_QUERY_PIPELINE_STATISTIC_CLIPPING_INVOCATIONS_BIT = 0x00000020,
   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;
```

- \texttt{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 \textbf{may} contribute to the count.
• **VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_PRIMITIVES_BIT** specifies that queries managed by the pool will count the number of primitives processed by the **input assembly** stage. If primitive restart is enabled, restarting the primitive topology has no effect on the count. Incomplete primitives may be counted.

• **VK_QUERY_PIPELINE_STATISTIC_VERTEX_SHADER_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of vertex shader invocations. This counter’s value is incremented each time a vertex shader is invoked.

• **VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of geometry shader invocations. This counter’s value is incremented each time a geometry shader is invoked. In the case of **instanced geometry shaders**, the geometry shader invocations count is incremented for each separate instanced invocation.

• **VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_PRIMITIVES_BIT** specifies that queries managed by the pool will count the number of primitives generated by geometry shader invocations. The counter’s value is incremented each time the geometry shader emits a primitive. Restarting primitive topology using the SPIR-V instructions `OpEndPrimitive` or `OpEndStreamPrimitive` has no effect on the geometry shader output primitives count.

• **VK_QUERY_PIPELINE_STATISTIC_CLIPPING_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of primitives processed by the **Primitive Clipping** stage of the pipeline. The counter’s value is incremented each time a primitive reaches the primitive clipping stage.

• **VK_QUERY_PIPELINE_STATISTIC_CLIPPING_PRIMITIVES_BIT** specifies that queries managed by the pool will count the number of primitives output by the **Primitive Clipping** stage of the pipeline. The counter’s value is incremented each time a primitive passes the primitive clipping stage. The actual number of primitives output by the primitive clipping stage for a particular input primitive is implementation-dependent but must satisfy the following conditions:
  ◦ If at least one vertex of the input primitive lies inside the clipping volume, the counter is incremented by one or more.
  ◦ Otherwise, the counter is incremented by zero or more.

• **VK_QUERY_PIPELINE_STATISTIC_FRAGMENT_SHADER_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of fragment shader invocations. The counter’s value is incremented each time the fragment shader is invoked.

• **VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_CONTROL_SHADER_PATCHES_BIT** specifies that queries managed by the pool will count the number of patches processed by the tessellation control shader. The counter’s value is incremented once for each patch for which a tessellation control shader is invoked.

• **VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_EVALUATION_SHADER_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of invocations of the tessellation evaluation shader. The counter’s value is incremented each time the tessellation evaluation shader is invoked.

• **VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of compute shader invocations. The counter’s value is incremented every time the compute shader is invoked. Implementations may skip the
execution of certain compute shader invocations or execute additional compute shader invocations for implementation-dependent reasons as long as the results of rendering otherwise remain unchanged.

These values are intended to measure relative statistics on one implementation. Various device architectures will count these values differently. Any or all counters may be affected by the issues described in Query Operation.

### Note

For example, tile-based rendering devices may need to replay the scene multiple times, affecting some of the counts.

If a pipeline has rasterizerDiscardEnable enabled, implementations may discard primitives after the final 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, call:

```c
// Provided by VK_KHR_synchronization2
void vkCmdWriteTimestamp2KHR(
    VkCommandBuffer commandBuffer,
    VkPipelineStageFlags2KHR stage,
    VkQueryPool queryPool,
    uint32_t query);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `stage` specifies a stage of the pipeline.
- `queryPool` is the query pool that will manage the timestamp.
- `query` is the query within the query pool that will contain the timestamp.

When `vkCmdWriteTimestamp2KHR` is submitted to a queue, it defines an execution dependency on commands that were submitted before it, and writes a timestamp to a query pool.

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

The second synchronization scope includes only the timestamp write operation.

When the timestamp value is written, the availability status of the query is set to available.

**Note**

If an implementation is unable to detect completion and latch the timer at any specific stage of the pipeline, it may instead do so at any logically later stage.

Comparisons between timestamps are not meaningful if the timestamps are written by commands submitted to different queues.

**Note**

An example of such a comparison is subtracting an older timestamp from a newer one to determine the execution time of a sequence of commands.

If `vkCmdWriteTimestamp2KHR` 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-vkCmdWriteTimestamp2KHR-stage-03929**
  If the geometry shaders feature is not enabled, stage must not contain `VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR`.

- **VUID-vkCmdWriteTimestamp2KHR-stage-03930**
  If the tessellation shaders feature is not enabled, stage must not contain `VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR` or `VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR`.

- **VUID-vkCmdWriteTimestamp2KHR-synchronization2-03858**
  The synchronization2 feature must be enabled.

- **VUID-vkCmdWriteTimestamp2KHR-stage-03859**
  stage must only include a single pipeline stage.

- **VUID-vkCmdWriteTimestamp2KHR-stage-03860**
  stage must only include stages valid for the queue family that was used to create the command pool that commandBuffer was allocated from.

- **VUID-vkCmdWriteTimestamp2KHR-queryPool-03861**
  queryPool must have been created with a queryType of `VK_QUERY_TYPE_TIMESTAMP`.

- **VUID-vkCmdWriteTimestamp2KHR-queryPool-03862**
  The query identified by queryPool and query must be unavailable.

- **VUID-vkCmdWriteTimestamp2KHR-timestampValidBits-03863**
  The command pool's queue family must support a non-zero timestampValidBits.

- **VUID-vkCmdWriteTimestamp2KHR-query-04903**
  query must be less than the number of queries in queryPool.

- **VUID-vkCmdWriteTimestamp2KHR-None-03864**
  All queries used by the command must be unavailable.

- **VUID-vkCmdWriteTimestamp2KHR-query-03865**
  If `vkCmdWriteTimestamp2KHR` 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-vkCmdWriteTimestamp2KHR-commandBuffer-parameter**
  commandBuffer must be a valid `VkCommandBuffer` handle.
- VUID-vkCmdWriteTimestamp2KHR-stage-parameter
  stage must be a valid combination of VkPipelineStageFlagBits2KHR values
- VUID-vkCmdWriteTimestamp2KHR-queryPool-parameter
  queryPool must be a valid VkQueryPool handle
- VUID-vkCmdWriteTimestamp2KHR-commandBuffer-recording
  commandBuffer must be in the recording state
- VUID-vkCmdWriteTimestamp2KHR-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support transfer, graphics, or compute operations
- VUID-vkCmdWriteTimestamp2KHR-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Transfer, Graphics, Compute</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To request a timestamp, 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.

vkCmdWriteTimestamp latches the value of the timer when all previous commands have completed
executing as far as the specified pipeline stage, and writes the timestamp value to memory. When
the timestamp value is written, the availability status of the query is set to available.

**Note**
If an implementation is unable to detect completion and latch the timer at any
specific stage of the pipeline, it **may** instead do so at any logically later stage.

Comparisons between timestamps are not meaningful if the timestamps are written by commands
submitted to different queues.

**Note**
An example of such a comparison is subtracting an older timestamp from a newer
one to determine the execution time of a sequence of commands.

If `vkCmdWriteTimestamp` is called while executing a render pass instance that has multiview enabled,
the timestamp uses N consecutive query indices in the query pool (starting at `query`) where N is the
number of bits set in the view mask of the subpass the command is executed in. The resulting query
values are determined by an implementation-dependent choice of one of the following behaviors:

- The first query is a timestamp value and (if more than one bit is set in the view mask) zero is
  written to the remaining queries. If two timestamps are written in the same subpass, the sum of
  the execution time of all views between those commands is the difference between the first
  query written by each command.

- All N queries are timestamp values. If two timestamps are written in the same subpass, the sum
  of the execution time of all views between those commands is the sum of the difference
  between corresponding queries written by each command. The difference between
  corresponding queries **may** be the execution time of a single view.

In either case, the application can sum the differences between all N queries to determine the total
execution time.

### Valid Usage

- **VUID-vkCmdWriteTimestamp-pipelineStage-04074**
  
  `pipelineStage` **must** be a valid stage for the queue family that was used to create the
  command pool that `commandBuffer` was allocated from.

- **VUID-vkCmdWriteTimestamp-pipelineStage-04075**
  
  If the geometry shaders feature is not enabled, `pipelineStage` **must** not be
  `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`

- **VUID-vkCmdWriteTimestamp-pipelineStage-04076**
  
  If the tessellation shaders feature is not enabled, `pipelineStage` **must** not be
  `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT`
  or
  `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`

- **VUID-vkCmdWriteTimestamp-synchronization2-06489**
  
  If the synchronization2 feature is not enabled, `pipelineStage` **must** not be
  `VK_PIPELINE_STAGE_NONE_KHR`
queryPool must have been created with a queryType of **VK_QUERY_TYPE_TIMESTAMP**

The query identified by queryPool and query must be unavailable

The command pool's queue family must support a non-zero timestampValidBits

query must be less than the number of queries in queryPool

All queries used by the command must be unavailable

If vkCmdWriteTimestamp is called within a render pass instance, the sum of query and the number of bits set in the current subpass's view mask must be less than or equal to the number of queries in queryPool

**Valid Usage (Implicit)**

commandBuffer must be a valid VkCommandBuffer handle

pipelineStage must be a valid VkPipelineStageFlagBits value

queryPool must be a valid VkQueryPool handle

commandBuffer must be in the recording state

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

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

**Host Synchronization**

Host access to commandBuffer must be externally synchronized

Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized
### 17.6. Performance Queries

Performance queries provide applications with a mechanism for getting performance counter information about the execution of command buffers, render passes, and commands.

Each queue family advertises the performance counters that can be queried on a queue of that family via a call to `vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR`. Implementations may limit access to performance counters based on platform requirements or only to specialized drivers for development purposes.

**Note**

This may include no performance counters being enumerated, or a reduced set. Please refer to platform-specific documentation for guidance on any such restrictions.

Performance queries use the existing `vkCmdBeginQuery` and `vkCmdEndQuery` to control what command buffers, render passes, or commands to get performance information for.

Implementations may require multiple passes where the command buffer, render passes, or commands being recorded are the same and are executed on the same queue to record performance counter data. This is achieved by submitting the same batch and providing a `VkPerformanceQuerySubmitInfoKHR` structure containing a counter pass index. The number of passes required for a given performance query pool can be queried via a call to `vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR`.

**Note**

Command buffers created with `VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT` must not be re-submitted. Changing command buffer usage bits may affect performance. To avoid this, the application should re-record any command buffers with the `VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT` when multiple counter passes are required.

Performance counter results from a performance query pool can be obtained with the command `vkGetQueryPoolResults`.

Performance query results are returned in an array of `VkPerformanceCounterResultKHR` unions containing the data associated with each counter in the query, stored in the same order as the counters supplied in `pCounterIndices` when creating the performance query. The
VkPerformanceCounterKHR::unit enumeration specifies how to parse the counter data.

```c
// Provided by VK_KHR_performance_query
typedef union VkPerformanceCounterResultKHR {
    int32_t     int32;
    int64_t     int64;
    uint32_t    uint32;
    uint64_t    uint64;
    float       float32;
    double      float64;
} VkPerformanceCounterResultKHR;
```

### 17.6.1. Profiling Lock

To record and submit a command buffer containing a performance query pool the profiling lock must be held. The profiling lock must be acquired prior to any call to `vkBeginCommandBuffer` that will be using a performance query pool. The profiling lock must be held while any command buffer containing a performance query pool is in the recording, executable, or pending state. To acquire the profiling lock, call:

```c
// Provided by VK_KHR_performance_query
VkResult vkAcquireProfilingLockKHR(
    VkDevice device,
    const VkAcquireProfilingLockInfoKHR* pInfo);
```

- device is the logical device to profile.
- pInfo is a pointer to a `VkAcquireProfilingLockInfoKHR` structure containing information about how the profiling is to be acquired.

Implementations may allow multiple actors to hold the profiling lock concurrently.

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

### Valid Usage (Implicit)

- VUID-vkAcquireProfilingLockKHR-device-parameter device must be a valid `VkDevice` handle
- VUID-vkAcquireProfilingLockKHR-pInfo-parameter pInfo must be a valid pointer to a valid `VkAcquireProfilingLockInfoKHR` structure

### Return Codes

**Success**
- `VK_SUCCESS`
The VkAcquireProfilingLockInfoKHR structure is defined as:

```c
// Provided by VK_KHR_performance_query
typedef struct VkAcquireProfilingLockInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkAcquireProfilingLockFlagsKHR flags;
    uint64_t timeout;
} VkAcquireProfilingLockInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `timeout` indicates how long the function waits, in nanoseconds, if the profiling lock is not available.

### Valid Usage (Implicit)

- VUID-VkAcquireProfilingLockInfoKHR-sType-sType `sType` must be `VK_STRUCTURE_TYPE_ACQUIRE_PROFILING_LOCK_INFO_KHR`
- VUID-VkAcquireProfilingLockInfoKHR-pNext-pNext `pNext` must be `NULL`
- VUID-VkAcquireProfilingLockInfoKHR-flags-zerobitmask `flags` must be `0`

If `timeout` is 0, `vkAcquireProfilingLockKHR` will not block while attempting to acquire the profiling lock. If `timeout` is `UINT64_MAX`, the function will not return until the profiling lock was acquired.

```c
// Provided by VK_KHR_performance_query
typedef enum VkAcquireProfilingLockFlagBitsKHR {
} VkAcquireProfilingLockFlagBitsKHR;
```

```c
// Provided by VK_KHR_performance_query
typedef VkFlags VkAcquireProfilingLockFlagsKHR;
```

`VkAcquireProfilingLockFlagsKHR` is a bitmask type for setting a mask, but is currently reserved for future use.
To release the profiling lock, call:

```c
// Provided by VK_KHR_performance_query
void vkReleaseProfilingLockKHR(
    VkDevice device);
```

- `device` is the logical device to cease profiling on.

### Valid Usage

- VUID-vkReleaseProfilingLockKHR-device-03235
  The profiling lock of `device` must have been held via a previous successful call to `vkAcquireProfilingLockKHR`

### Valid Usage (Implicit)

- VUID-vkReleaseProfilingLockKHR-device-parameter
device must be a valid VkDevice handle
Chapter 18. Clear Commands

18.1. Clearing Images Outside A Render Pass Instance

Color and depth/stencil images can be cleared outside a render pass instance using `vkCmdClearColorImage` or `vkCmdClearDepthStencilImage`, respectively. These commands are only allowed outside of a render pass instance.

To clear one or more subranges of a color image, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdClearColorImage(
    VkCommandBuffer commandBuffer,
    VkImage image,
    VkImageLayout imageLayout,
    const VkClearColorValue* pColor,
    uint32_t rangeCount,
    const VkImageSubresourceRange* pRanges);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `image` is the image to be cleared.
- `imageLayout` specifies the current layout of the image subresource ranges to be cleared, and must be `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_GENERAL` or `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`.
- `pColor` is a pointer to a `VkClearColorValue` structure containing the values that the image subresource ranges will be cleared to (see Clear Values below).
- `rangeCount` is the number of image subresource range structures in `pRanges`.
- `pRanges` is a pointer to an array of `VkImageSubresourceRange` structures describing a range of mipmap levels, array layers, and aspects to be cleared, as described in Image Views.

Each specified range in `pRanges` is cleared to the value specified by `pColor`.

Valid Usage

- **VUID-vkCmdClearColorImage-image-01993** The format features of `image` must contain `VK_FORMAT_FEATURE_TRANSFER_DST_BIT`
- **VUID-vkCmdClearColorImage-image-00002** `image` must have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag
- **VUID-vkCmdClearColorImage-image-01545** `image` must not use any of the formats that require a sampler Y'CbCr conversion
- **VUID-vkCmdClearColorImage-image-00003** If `image` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object
• VUID-vkCmdClearColorImage-imageLayout-00004
  imageLayout must specify the layout of the image subresource ranges of image specified in pRanges at the time this command is executed on a VkDevice

• VUID-vkCmdClearColorImage-imageLayout-01394
  imageLayout must be VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, VK_IMAGE_LAYOUT_GENERAL, or VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR

• 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 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 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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</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 VkClearColorDepthStencilValue* 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 **VkImageStencelUsageCreateInfo::stencilUsage** used to create **image**`
- **VUID-vkCmdClearDepthStencilImage-pRanges-02659**
  If the **aspect** member of any element of **pRanges** includes `VK_IMAGE_ASPECT_STENCIL_BIT`, and **image** was not created with separate stencil usage, `VK_IMAGE_USAGE_TRANSFER_DST_BIT **must** have been included in the **VkImageCreateInfo::usage** used to create **image**`
- **VUID-vkCmdClearDepthStencilImage-pRanges-02660**
  If the **aspect** member of any element of **pRanges** includes `VK_IMAGE_ASPECT_DEPTH_BIT`, `VK_IMAGE_USAGE_TRANSFER_DST_BIT **must** have been included in the **VkImageCreateInfo::usage** used to create **image**`
- **VUID-vkCmdClearDepthStencilImage-image-00010**
  If **image** is non-sparse then it **must** be bound completely and contiguously to a single **VkDeviceMemory** object
- **VUID-vkCmdClearDepthStencilImage-imageLayout-00011**
  **imageLayout** **must** specify the layout of the image subresource ranges of **image** specified in **pRanges** at the time this command is executed on a **VkDevice**
- **VUID-vkCmdClearDepthStencilImage-imageLayout-00012**
  **imageLayout** **must** be either of `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`
- **VUID-vkCmdClearDepthStencilImage-aspectMask-02824**
  The **VkImageSubresourceRange::aspectMask** member of each element of the **pRanges** array **must** not include bits other than `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT`
- **VUID-vkCmdClearDepthStencilImage-image-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

- VUID-vkCmdClearDepthStencilImage-image-02826
  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**

- VUID-vkCmdClearDepthStencilImage-baseMipLevel-01474
  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-vkCmdClearDepthStencilImage-pRanges-01694
  For each `VkImageSubresourceRange` element of `pRanges`, if the `levelCount` member is not `VK_REMAINING_MIP_LEVELS`, then `baseMipLevel + levelCount` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when image was created

- VUID-vkCmdClearDepthStencilImage-baseArrayLayer-01476
  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-vkCmdClearDepthStencilImage-pRanges-01695
  For each `VkImageSubresourceRange` element of `pRanges`, if the `layerCount` member is not `VK_REMAINING_ARRAY_LAYERS`, then `baseArrayLayer + layerCount` **must** be less than the `arrayLayers` specified in `VkImageCreateInfo` when image was created

- VUID-vkCmdClearDepthStencilImage-image-00014
  image **must** have a depth/stencil format

- VUID-vkCmdClearDepthStencilImage-commandBuffer-01807
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, image **must** not be a protected image

- VUID-vkCmdClearDepthStencilImage-commandBuffer-01808
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, image **must** not be an unprotected image

---

**Valid Usage (Implicit)**

- VUID-vkCmdClearDepthStencilImage-commandBuffer-parameter
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle

- VUID-vkCmdClearDepthStencilImage-image-parameter
  `image` **must** be a valid `VkImage` handle

- VUID-vkCmdClearDepthStencilImage-imageLayout-parameter
  `imageLayout` **must** be a valid `VkImageLayout` value

- VUID-vkCmdClearDepthStencilImage-pDepthStencil-parameter
  `pDepthStencil` **must** be a valid pointer to a valid `VkClearDepthStencilValue` structure

- VUID-vkCmdClearDepthStencilImage-pRanges-parameter
  `pRanges` **must** be a valid pointer to an array of `rangeCount` valid `VkImageSubresourceRange`
structures

- VUID-vkCmdClearDepthStencilImage-commandBuffer-recording
commandBuffer must be in the recording state

- VUID-vkCmdClearDepthStencilImage-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

- VUID-vkCmdClearDepthStencilImage-renderpass
  This command must only be called outside of a render pass instance

- VUID-vkCmdClearDepthStencilImage-rangeCount-arraylength
  rangeCount must be greater than 0

- VUID-vkCmdClearDepthStencilImage-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</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. If any attachment index to be cleared is not backed by an image view, then the clear has no effect.

• `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` executes as a drawing command, rather than a transfer command, with writes performed by it executing in rasterization order. Clears to color attachments are executed as color attachment writes, by the `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` stage. Clears to depth/stencil attachments are executed as depth writes and writes by the `VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT` and `VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT` stages.

`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.

### Valid Usage

- **VUID-vkCmdClearAttachments-aspectMask-02501**
  If the `aspectMask` member of any element of `pAttachments` contains `VK_IMAGE_ASPECT_COLOR_BIT`, then the `colorAttachment` member of that element must either refer to a color attachment which is `VK_ATTACHMENT_UNUSED`, or must be a valid color attachment.

- **VUID-vkCmdClearAttachments-aspectMask-02502**
  If the `aspectMask` member of any element of `pAttachments` contains `VK_IMAGE_ASPECT_DEPTH_BIT`, then the current subpass' depth/stencil attachment must either be `VK_ATTACHMENT_UNUSED`, or must have a depth component.

- **VUID-vkCmdClearAttachments-aspectMask-02503**
  If the `aspectMask` member of any element of `pAttachments` contains `VK_IMAGE_ASPECT_STENCIL_BIT`, then the current subpass' depth/stencil attachment must either be `VK_ATTACHMENT_UNUSED`, or must have a stencil component.

- **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
• VUID-vkCmdClearAttachments-pRects-00017
  The layers specified by each element of \textit{pRects} must be contained within every attachment that \textit{pAttachments} refers to

• VUID-vkCmdClearAttachments-layerCount-01934
  The \textit{layerCount} member of each element of \textit{pRects} must not be 0

• VUID-vkCmdClearAttachments-commandBuffer-02504
  If \textit{commandBuffer} is an unprotected command buffer and \textit{protectedNoFault} is not supported, each attachment to be cleared must not be a protected image

• VUID-vkCmdClearAttachments-commandBuffer-02505
  If \textit{commandBuffer} is a protected command buffer and \textit{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 \textit{baseArrayLayer} must be zero and \textit{layerCount} must be one

---

**Valid Usage (Implicit)**

• VUID-vkCmdClearAttachments-commandBuffer-parameter
  \textit{commandBuffer} must be a valid \texttt{VkCommandBuffer} handle

• VUID-vkCmdClearAttachments-pAttachments-parameter
  \textit{pAttachments} must be a valid pointer to an array of \textit{attachmentCount} valid \texttt{VkClearAttachment} structures

• VUID-vkCmdClearAttachments-pRects-parameter
  \textit{pRects} must be a valid pointer to an array of \textit{rectCount} \texttt{VkClearRect} structures

• VUID-vkCmdClearAttachments-commandBuffer-recording
  \textit{commandBuffer} must be in the \textit{recording state}

• VUID-vkCmdClearAttachments-commandBuffer-cmdpool
  The \texttt{VkCommandPool} that \textit{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
  \textit{attachmentCount} must be greater than 0

• VUID-vkCmdClearAttachments-rectCount-arraylength
  \textit{rectCount} must be greater than 0

---

**Host Synchronization**

• Host access to \textit{commandBuffer} must be externally synchronized
• Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</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 VkClearColorAttachment structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkClearColorAttachment {
    VkImageAspectFlags aspectMask;
    uint32_t colorAttachment;
    VkClearValue clearValue;
} VkClearColorAttachment;
```

- `aspectMask` is a mask selecting the color, depth and/or stencil aspects of the attachment to be cleared.
- `colorAttachment` is only meaningful if VK_IMAGE_ASPECT_COLOR_BIT is set in `aspectMask`, in which case it is an index into the currently bound color attachments.
- `clearValue` is the color or depth/stencil value to clear the attachment to, as described in Clear Values below.
Valid Usage

- VUID-VkClearAttachment-aspectMask-00019
  If `aspectMask` includes `VK_IMAGE_ASPECT_COLOR_BIT`, it must not include `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT`

- VUID-VkClearAttachment-aspectMask-00020
  `aspectMask` must not include `VK_IMAGE_ASPECT_METADATA_BIT`

- VUID-VkClearAttachment-aspectMask-02246
  `aspectMask` must not include `VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT` for any index `i`

- VUID-VkClearAttachment-clearValue-00021
  `clearValue` must be a valid `VkClearColorValue` union

Valid Usage (Implicit)

- VUID-VkClearAttachment-aspectMask-parameter
  `aspectMask` must be a valid combination of `VkImageAspectFlagBits` values

- VUID-VkClearAttachment-aspectMask-requiredbitmask
  `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 formats in the Interpretation of Numeric Format table other than signed integer (`SINT`) or unsigned integer (`UINT`). Floating point values are automatically converted to the format of the image, with the clear value being treated as linear if the image is sRGB.

- `int32` are the color clear values when the format of the image or attachment is signed integer (`SINT`). Signed integer values are converted to the format of the image by casting to the smaller type (with negative 32-bit values mapping to negative values in the smaller type). If the integer clear value is not representable in the target type (e.g. would overflow in conversion to that type), the clear value is undefined.

- `uint32` are the color clear values when the format of the image or attachment is unsigned integer (`UINT`). Unsigned integer values are converted to the format of the image by casting to the integer type with fewer bits.
The four array elements of the clear color map to R, G, B, and A components of image formats, in order.

If the image has more than one sample, the same value is written to all samples for any pixels being cleared.

The \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

- VUID-VkClearDepthStencilValue-depth-00022
  Unless the \texttt{VK_EXT_depth_range_unrestricted} extension is enabled \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:
// Provided by VK_VERSION_1_0

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

- **commandBuffer** is the command buffer into which the command will be recorded.
- **dstBuffer** is the buffer to be filled.
- **dstOffset** is the byte offset into the buffer at which to start filling, and **must** be a multiple of 4.
- **size** is the number of bytes to fill, and **must** be either a multiple of 4, or **VK_WHOLE_SIZE** to fill the range from **offset** to the end of the buffer. If **VK_WHOLE_SIZE** is used and the remaining size of the buffer is not a multiple of 4, then the nearest smaller multiple is used.
- **data** is the 4-byte word written repeatedly to the buffer to fill **size** bytes of data. The data word is written to memory according to the host endianness.

`vkCmdFillBuffer` is treated as “transfer” operation for the purposes of synchronization barriers. The **VK_BUFFER_USAGE_TRANSFER_DST_BIT** must be specified in **usage** of **VkBufferCreateInfo** in order for the buffer to be compatible with `vkCmdFillBuffer`.

### Valid Usage

- **VUID-vkCmdFillBuffer-dstOffset-00024**
  - **dstOffset** must be less than the size of **dstBuffer**
- **VUID-vkCmdFillBuffer-dstOffset-00025**
  - **dstOffset** must be a multiple of 4
- **VUID-vkCmdFillBuffer-size-00026**
  - If **size** is not equal to **VK_WHOLE_SIZE**, **size** must be greater than **0**
- **VUID-vkCmdFillBuffer-size-00027**
  - If **size** is not equal to **VK_WHOLE_SIZE**, **size** must be less than or equal to the size of **dstBuffer** minus **dstOffset**
- **VUID-vkCmdFillBuffer-size-00028**
  - If **size** is not equal to **VK_WHOLE_SIZE**, **size** must be a multiple of **4**
- **VUID-vkCmdFillBuffer-dstBuffer-00029**
  - **dstBuffer** must have been created with **VK_BUFFER_USAGE_TRANSFER_DST_BIT** usage flag
- **VUID-vkCmdFillBuffer-dstBuffer-00031**
  - If **dstBuffer** is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object
- **VUID-vkCmdFillBuffer-commandBuffer-01811**
  - If **commandBuffer** is an unprotected command buffer and **protectedNoFault** is not supported, **dstBuffer** must not be a protected buffer
• VUID-vkCmdFillBuffer-commandBuffer-01812
If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `dstBuffer` must not be an unprotected buffer

Valid Usage (Implicit)

• VUID-vkCmdFillBuffer-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

• VUID-vkCmdFillBuffer-dstBuffer-parameter
  `dstBuffer` must be a valid `VkBuffer` handle

• VUID-vkCmdFillBuffer-commandBuffer-recording
  `commandBuffer` must be in the recording state

• VUID-vkCmdFillBuffer-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics or compute operations

• VUID-vkCmdFillBuffer-renderpass
  This command must only be called outside of a render pass instance

• VUID-vkCmdFillBuffer-commonparent
  Both of `commandBuffer`, and `dstBuffer` must have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

• Host access to `commandBuffer` must be externally synchronized

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</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, 
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 “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
**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

- VUID-vkCmdUpdateBuffer-commonparent
  Both of `commandBuffer`, and *dstBuffer* must have been created, allocated, or retrieved from the same `VkDevice`
Host Synchronization

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
</tr>
</tbody>
</table>
Chapter 19. Copy Commands

An application can copy buffer and image data using several methods depending on the type of data transfer. Data can be copied between buffer objects with vkCmdCopyBuffer2KHR and vkCmdCopyBuffer and a portion of an image can be copied to another image with vkCmdCopyImage2KHR and vkCmdCopyImage. Image data can also be copied to and from buffer memory using vkCmdCopyImageToBuffer2KHR, vkCmdCopyImageToBuffer, VkCmdCopyBufferToImage2KHR, and VkCmdCopyBufferToImage. Image data can be blitted (with or without scaling and filtering) with vkCmdBlitImage2KHR and vkCmdBlitImage. Multisampled images can be resolved to a non-multisampled image with vkCmdResolveImage2KHR and vkCmdResolveImage.

19.1. Common Operation

The following valid usage rules apply to all copy commands:

- Copy commands **must** be recorded outside of a render pass instance.
- The set of all bytes bound to all the source regions **must** not overlap the set of all bytes bound to the destination regions.
- The set of all bytes bound to each destination region **must** not overlap the set of all bytes bound to another destination region.
- Copy regions **must** be non-empty.
- Regions **must** not extend outside the bounds of the buffer or image level, except that regions of compressed images can extend as far as the dimension of the image level rounded up to a complete compressed texel block.
- Source image subresources **must** be in either the VK_IMAGE_LAYOUT_GENERAL or VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL layout. Destination image subresources **must** be in the VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR, VK_IMAGE_LAYOUT_GENERAL or VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL layout. As a consequence, if an image subresource is used as both source and destination of a copy, it **must** be in the VK_IMAGE_LAYOUT_GENERAL layout.
- Source images **must** have VK_FORMAT_FEATURE_TRANSFER_SRC_BIT in their format features.
- Destination images **must** have VK_FORMAT_FEATURE_TRANSFER_DST_BIT in their format features.
- Source buffers **must** have been created with the VK_BUFFER_USAGE_TRANSFER_SRC_BIT usage bit enabled and destination buffers **must** have been created with the VK_BUFFER_USAGE_TRANSFER_DST_BIT usage bit enabled.
- If the stencil aspect of source image is accessed, and the source image was not created with separate stencil usage, the source image **must** have been created with VK_IMAGE_USAGE_TRANSFER_SRC_BIT set in VkImageCreateInfo::usage.
- If the stencil aspect of destination image is accessed, and the destination image was not created with separate stencil usage, the destination image **must** have been created with VK_IMAGE_USAGE_TRANSFER_DST_BIT set in VkImageCreateInfo::usage.
- If the stencil aspect of source image is accessed, and the source image was created with separate stencil usage, the source image **must** have been created with VK_IMAGE_USAGE_TRANSFER_SRC_BIT set in VkImageStencilUsageCreateInfo::stencilUsage.
• If the stencil aspect of destination image is accessed, and the destination image was created with separate stencil usage, the destination image must have been created with VK_IMAGE_USAGE_TRANSFER_DST_BIT set in VkImageStencilUsageCreateInfo::stencilUsage

• If non-stencil aspects of a source image are accessed, the source image must have been created with VK_IMAGE_USAGE_TRANSFER_SRC_BIT set in VkImageCreateInfo::usage

• If non-stencil aspects of a destination image are accessed, the destination image must have been created with VK_IMAGE_USAGE_TRANSFER_DST_BIT set in VkImageCreateInfo::usage

All copy commands are treated as “transfer” operations for the purposes of synchronization barriers.

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.

### 19.2. 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 region in **pRegions** is copied from the source buffer to the same region of the destination buffer. **srcBuffer** and **dstBuffer** can be the same buffer or alias the same memory, but the resulting values are undefined if the copy regions overlap in memory.

### Valid Usage

- 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
The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations.

This command must only be called outside of a render pass instance.

regionCount must be greater than 0.

Each of `commandBuffer`, `dstBuffer`, and `srcBuffer` must have been created, allocated, or retrieved from the same `VkDevice`.

Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
</tr>
</tbody>
</table>

The `VkBufferCopy` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkBufferCopy {
    VkDeviceSize srcOffset;
    VkDeviceSize dstOffset;
    VkDeviceSize size;
} VkBufferCopy;
```

- `srcOffset` is the starting offset in bytes from the start of `srcBuffer`.
- `dstOffset` is the starting offset in bytes from the start of `dstBuffer`.
- `size` is the number of bytes to copy.

Valid Usage

- VUID-VkBufferCopy-size-01988
  The `size` must be greater than 0.
A more extensible version of the copy buffer command is defined below.

To copy data between buffer objects, call:

```c
// Provided by VK_KHR_copy_commands2
void vkCmdCopyBuffer2KHR(
    VkCommandBuffer commandBuffer,
    const VkCopyBufferInfo2KHR* pCopyBufferInfo);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pCopyBufferInfo` is a pointer to a `VkCopyBufferInfo2KHR` structure describing the copy parameters.

This command is functionally identical to `vkCmdCopyBuffer`, but includes extensible sub-structures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

**Valid Usage**

- VUID-vkCmdCopyBuffer2KHR-commandBuffer-01822
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `srcBuffer` must not be a protected buffer

- VUID-vkCmdCopyBuffer2KHR-commandBuffer-01823
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `dstBuffer` must not be a protected buffer

- VUID-vkCmdCopyBuffer2KHR-commandBuffer-01824
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `dstBuffer` must not be an unprotected buffer

**Valid Usage (Implicit)**

- VUID-vkCmdCopyBuffer2KHR-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdCopyBuffer2KHR-pCopyBufferInfo-parameter
  `pCopyBufferInfo` must be a valid pointer to a valid `VkCopyBufferInfo2KHR` structure

- VUID-vkCmdCopyBuffer2KHR-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdCopyBuffer2KHR-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations

- VUID-vkCmdCopyBuffer2KHR-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer, Graphics, Compute</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkCopyBufferInfo2KHR` structure is defined as:

```c
// Provided by VK_KHR_copy_commands2
typedef struct VkCopyBufferInfo2KHR {
    VkStructureType sType;
    const void* pNext;
    VkBuffer srcBuffer;
    VkBuffer dstBuffer;
    uint32_t regionCount;
    const VkBufferCopy2KHR* pRegions;
} VkCopyBufferInfo2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcBuffer` is the source buffer.
- `dstBuffer` is the destination buffer.
- `regionCount` is the number of regions to copy.
- `pRegions` is a pointer to an array of `VkBufferCopy2KHR` structures specifying the regions to copy.

Members defined by this structure with the same name as parameters in `vkCmdCopyBuffer` have the identical effect to those parameters; the child structure `VkBufferCopy2KHR` is a variant of `VkBufferCopy` which includes `sType` and `pNext` parameters, allowing it to be extended.

Valid Usage

- VUID-VkCopyBufferInfo2KHR-srcOffset-00113
  The `srcOffset` member of each element of `pRegions` must be less than the size of `srcBuffer`
- VUID-VkCopyBufferInfo2KHR-dstOffset-00114
  The `dstOffset` member of each element of `pRegions` must be less than the size of `dstBuffer`
Invalid Usage:

- **VUID-VkCopyBufferInfo2KHR-size-00115**
  The size member of each element of pRegions must be less than or equal to the size of srcBuffer minus srcOffset.

- **VUID-VkCopyBufferInfo2KHR-size-00116**
  The size member of each element of pRegions must be less than or equal to the size of dstBuffer minus dstOffset.

- **VUID-VkCopyBufferInfo2KHR-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-VkCopyBufferInfo2KHR-srcBuffer-00118**
  srcBuffer must have been created with VK_BUFFER_USAGE_TRANSFER_SRC_BIT usage flag.

- **VUID-VkCopyBufferInfo2KHR-srcBuffer-00119**
  If srcBuffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object.

- **VUID-VkCopyBufferInfo2KHR-dstBuffer-00120**
  dstBuffer must have been created with VK_BUFFER_USAGE_TRANSFER_DST_BIT usage flag.

- **VUID-VkCopyBufferInfo2KHR-dstBuffer-00121**
  If dstBuffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object.

Valid Usage (Implicit):

- **VUID-VkCopyBufferInfo2KHR-sType-sType**
  sType must be VK_STRUCTURE_TYPE_COPY_BUFFER_INFO_2_KHR.

- **VUID-VkCopyBufferInfo2KHR-pNext-pNext**
  pNext must be NULL.

- **VUID-VkCopyBufferInfo2KHR-srcBuffer-parameter**
  srcBuffer must be a valid VkBuffer handle.

- **VUID-VkCopyBufferInfo2KHR-dstBuffer-parameter**
  dstBuffer must be a valid VkBuffer handle.

- **VUID-VkCopyBufferInfo2KHR-pRegions-parameter**
  pRegions must be a valid pointer to an array of regionCount valid VkBufferCopy2KHR structures.

- **VUID-VkCopyBufferInfo2KHR-regionCount-arraylength**
  regionCount must be greater than 0.

- **VUID-VkCopyBufferInfo2KHR-commonparent**
  Both of dstBuffer, and srcBuffer must have been created, allocated, or retrieved from the same VkDevice.

The VkBufferCopy2KHR structure is defined as:
typedef struct VkBufferCopy2KHR {
    VkStructureType sType;
    const void* pNext;
    VkDeviceSize srcOffset;
    VkDeviceSize dstOffset;
    VkDeviceSize size;
} VkBufferCopy2KHR;

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• srcOffset is the starting offset in bytes from the start of srcBuffer.
• dstOffset is the starting offset in bytes from the start of dstBuffer.
• size is the number of bytes to copy.

Valid Usage

• VUID-VkBufferCopy2KHR-size-01988
  The size must be greater than 0

Valid Usage (Implicit)

• VUID-VkBufferCopy2KHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_BUFFER_COPY_2_KHR
• VUID-VkBufferCopy2KHR-pNext-pNext
  pNext must be NULL

19.3. Copying Data Between Images

vkCmdCopyImage performs image copies in a similar manner to a host memcpy. It does not perform general-purpose conversions such as scaling, resizing, blending, color-space conversion, or format conversions. Rather, it simply copies raw image data. vkCmdCopyImage can copy between images with different formats, provided the formats are compatible as defined below.

To copy data between image objects, call:

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

- **commandBuffer** is the command buffer into which the command will be recorded.
- **srcImage** is the source image.
- **srcImageLayout** is the current layout of the source image subresource.
- **dstImage** is the destination image.
- **dstImageLayout** is the current layout of the destination image subresource.
- **regionCount** is the number of regions to copy.
- **pRegions** is a pointer to an array of VkImageCopy structures specifying the regions to copy.

Each region in **pRegions** is copied from the source image to the same region of the destination image. **srcImage** and **dstImage** can be the same image or alias the same memory.

The formats of **srcImage** and **dstImage** must be compatible. Formats are compatible if they share the same class, as shown in the Compatible Formats table. Depth/stencil formats must match exactly.

If either **srcImage** or **dstImage** has a multi-planar format, regions of each plane to be copied must be specified separately using the **srcSubresource** and **dstSubresource** members of the VkImageCopy structure. In this case, the **aspectMask** of the **srcSubresource** or **dstSubresource** that refers to the multi-planar image must be **VK_IMAGE_ASPECT_PLANE_0_BIT**, **VK_IMAGE_ASPECT_PLANE_1_BIT**, or **VK_IMAGE_ASPECT_PLANE_2_BIT**. For the purposes of **vkCmdCopyImage**, each plane of a multi-planar image is treated as having the format listed in Compatible formats of planes of multi-planar formats for the plane identified by the **aspectMask** of the corresponding subresource. This applies both to **VkFormat** and to coordinates used in the copy, which correspond to texels in the plane rather than how these texels map to coordinates in the image as a whole.

**Note**

For example, the **VK_IMAGE_ASPECT_PLANE_1_BIT** plane of a **VK_FORMAT_G8 B8R8 2PLANE_420 UNORM** image is compatible with an image of format **VK_FORMAT_R8G8 UNORM** and (less usefully) with the **VK_IMAGE_ASPECT_PLANE_0_BIT** plane of an image of format **VK_FORMAT_G10X6 B10X6 R10X6 3PLANE_420 UNORM 3PACK16**, as each texel is 2 bytes in size.

**vkCmdCopyImage** allows copying between size-compatible compressed and uncompressed internal formats. Formats are size-compatible if the texel block size of the uncompressed format is equal to the texel block size of the compressed format. Such a copy does not perform on-the-fly compression or decompression. When copying from an uncompressed format to a compressed format, each texel of uncompressed data of the source image is copied as a raw value to the corresponding compressed texel block of the destination image. When copying from a compressed format to an uncompressed format, each compressed texel block of the source image is copied as a raw value to the corresponding texel of uncompressed data in the destination image. Thus, for example, it is legal to copy between a 128-bit uncompressed format and a compressed format which has a 128-bit sized compressed texel block representing 4×4 texels (using 8 bits per texel), or between a 64-bit
uncompressed format and a compressed format which has a 64-bit sized compressed texel block representing $4 \times 4$ texels (using 4 bits per texel).

When copying between compressed and uncompressed formats the extent members represent the texel dimensions of the source image and not the destination. When copying from a compressed image to an uncompressed image the image texel dimensions written to the uncompressed image will be source extent divided by the compressed texel block dimensions. When copying from an uncompressed image to a compressed image the image texel dimensions written to the compressed image will be the source extent multiplied by the compressed texel block dimensions. In both cases the number of bytes read and the number of bytes written will be identical.

Copying to or from block-compressed images is typically done in multiples of the compressed texel block size. For this reason the extent must be a multiple of the compressed texel block dimension. There is one exception to this rule which is required to handle compressed images created with dimensions that are not a multiple of the compressed texel block dimensions: if the srcImage is compressed, then:

- If extent.width is not a multiple of the compressed texel block width, then $(\text{extent.width} + \text{srcOffset.x})$ must equal the image subresource width.
- If extent.height is not a multiple of the compressed texel block height, then $(\text{extent.height} + \text{srcOffset.y})$ must equal the image subresource height.
- If extent.depth is not a multiple of the compressed texel block depth, then $(\text{extent.depth} + \text{srcOffset.z})$ must equal the image subresource depth.

Similarly, if the dstImage is compressed, then:

- If extent.width is not a multiple of the compressed texel block width, then $(\text{extent.width} + \text{dstOffset.x})$ must equal the image subresource width.
- If extent.height is not a multiple of the compressed texel block height, then $(\text{extent.height} + \text{dstOffset.y})$ must equal the image subresource height.
- If extent.depth is not a multiple of the compressed texel block depth, then $(\text{extent.depth} + \text{dstOffset.z})$ must equal the image subresource depth.

This allows the last compressed texel block of the image in each non-multiple dimension to be included as a source or destination of the copy.

“422” image formats that are not multi-planar are treated as having a $2 \times 1$ compressed texel block for the purposes of these rules.

vkCmdCopyImage can be used to copy image data between multisample images, but both images must have the same number of samples.

**Valid Usage**

- VUID-vkCmdCopyImage-commandBuffer-01825
  If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, srcImage must not be a protected image

834
If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `dstImage` must not be a protected image.

If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `dstImage` must not be an unprotected image.

The union of all source regions, and the union of all destination regions, specified by the elements of `pRegions`, must not overlap in memory.

The `format features` of `srcImage` must contain `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT`.

`srcImage` must have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage flag.

If `srcImage` is non-sparse then the image or `disjoint` plane to be copied must be bound completely and contiguously to a single `VkDeviceMemory` object.

`srcImageLayout` must specify the layout of the image subresources of `srcImage` specified in `pRegions` at the time this command is executed on a `VkDevice`.

`srcImageLayout` must be `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL`, `VK_IMAGE_LAYOUT_GENERAL`, or `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`.

The `format features` of `dstImage` must contain `VK_FORMAT_FEATURE_TRANSFER_DST_BIT`.

`dstImage` must have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag.

If `dstImage` is non-sparse then the image or `disjoint` plane that is the destination of the copy must be bound completely and contiguously to a single `VkDeviceMemory` object.

`dstImageLayout` must specify the layout of the image subresources of `dstImage` specified in `pRegions` at the time this command is executed on a `VkDevice`.

`dstImageLayout` must be `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`, `VK_IMAGE_LAYOUT_GENERAL`, or `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`.

If the `VkFormat` of each of `srcImage` and `dstImage` is not a `multi-planar format`, the `VkFormat` of each of `srcImage` and `dstImage` must be compatible, as defined above.

In a copy to or from a plane of a `multi-planar image`, the `VkFormat` of the image and plane must be compatible according to the description of compatible planes for the plane being copied.
• VUID-vkCmdCopyImage-srcImage-00136
  The sample count of srcImage and dstImage must match

• VUID-vkCmdCopyImage-srcSubresource-01696
  The srcSubresource.mipLevel member of each element of pRegions must be less than the mipLevels specified in VkImageCreateInfo when srcImage was created

• VUID-vkCmdCopyImage-dstSubresource-01697
  The dstSubresource.mipLevel member of each element of pRegions must be less than the mipLevels specified in VkImageCreateInfo when dstImage was created

• VUID-vkCmdCopyImage-srcSubresource-01698
  The srcSubresource.baseArrayLayer + srcSubresource.layerCount of each element of pRegions must be less than or equal to the arrayLayers specified in VkImageCreateInfo when srcImage was created

• VUID-vkCmdCopyImage-dstSubresource-01699
  The dstSubresource.baseArrayLayer + dstSubresource.layerCount of each element of pRegions must be less than or equal to the arrayLayers specified in VkImageCreateInfo when dstImage was created

• VUID-vkCmdCopyImage-srcOffset-01783
  The srcOffset and extent members of each element of pRegions must respect the image transfer granularity requirements of commandBuffer’s command pool’s queue family, as described in VkQueueFamilyProperties

• VUID-vkCmdCopyImage-dstOffset-01784
  The dstOffset and extent members of each element of pRegions must respect the image transfer granularity requirements of commandBuffer’s command pool’s queue family, as described in VkQueueFamilyProperties

• VUID-vkCmdCopyImage-srcImage-01551
  If neither srcImage nor dstImage has a multi-planar image format then for each element of pRegions, srcSubresource.aspectMask and dstSubresource.aspectMask must match

• VUID-vkCmdCopyImage-srcImage-01552
  If srcImage has a VkFormat with two planes then for each element of pRegions, srcSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT or VK_IMAGE_ASPECT_PLANE_1_BIT

• VUID-vkCmdCopyImage-srcImage-01553
  If srcImage has a VkFormat with three planes then for each element of pRegions, srcSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or VK_IMAGE_ASPECT_PLANE_2_BIT

• VUID-vkCmdCopyImage-dstImage-01554
  If dstImage has a VkFormat with two planes then for each element of pRegions, dstSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT or VK_IMAGE_ASPECT_PLANE_1_BIT

• VUID-vkCmdCopyImage-dstImage-01555
  If dstImage has a VkFormat with three planes then for each element of pRegions, dstSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or VK_IMAGE_ASPECT_PLANE_2_BIT
If `srcImage` has a multi-planar image format and the `dstImage` does not have a multi-planar image format, then for each element of `pRegions`, `dstSubresource.aspectMask` must be `VK_IMAGE_ASPECT_COLOR_BIT`.

If `dstImage` has a multi-planar image format and the `srcImage` does not have a multi-planar image format, then for each element of `pRegions`, `srcSubresource.aspectMask` must be `VK_IMAGE_ASPECT_COLOR_BIT`.

If `srcImage` has a type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions`, `srcSubresource.baseArrayLayer` must be 0 and `srcSubresource.layerCount` must be 1.

If `dstImage` has a type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions`, `dstSubresource.baseArrayLayer` must be 0 and `dstSubresource.layerCount` must be 1.

For each element of `pRegions`, `srcSubresource.aspectMask` must specify aspects present in `srcImage`.

For each element of `pRegions`, `dstSubresource.aspectMask` must specify aspects present in `dstImage`.

For each element of `pRegions`, `srcOffset.x` and `(extent.width + srcOffset.x)` must both be greater than or equal to 0 and less than or equal to the width of the specified `srcSubresource` of `srcImage`.

For each element of `pRegions`, `srcOffset.y` and `(extent.height + srcOffset.y)` must both be greater than or equal to 0 and less than or equal to the height of the specified `srcSubresource` of `srcImage`.

If `srcImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `srcOffset.y` must be 0 and `extent.height` must be 1.

If `dstImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `dstOffset.z` must be 0 and `extent.depth` must be 1.

If `srcImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `srcOffset.z` must be 0 and `extent.depth` must be 1.

If `srcImage` is of type `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `srcOffset.z` must be 0 and `extent.depth` must be 1.
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-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`

- **VUID-vkCmdCopyImage-dstImage-00152**
  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-vkCmdCopyImage-dstOffset-00153**
  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-vkCmdCopyImage-srcImage-01727**
  If `srcImage` is a blocked image, then for each element of `pRegions`, all members of `srcOffset` must be a multiple of the corresponding dimensions of the compressed texel block

- **VUID-vkCmdCopyImage-srcImage-01728**
  If `srcImage` is a blocked image, then for each element of `pRegions`, `extent.width` must be a multiple of the compressed texel block width or `(extent.width + srcOffset.x)` must equal the width of the specified `srcSubresource` of `srcImage`

- **VUID-vkCmdCopyImage-srcImage-01729**
  If `srcImage` is a blocked image, then for each element of `pRegions`, `extent.height` must be a multiple of the compressed texel block height or `(extent.height + srcOffset.y)` must equal the height of the specified `srcSubresource` of `srcImage`

- **VUID-vkCmdCopyImage-srcImage-01730**
  If `srcImage` is a blocked image, then for each element of `pRegions`, `extent.depth` must be a multiple of the compressed texel block depth or `(extent.depth + srcOffset.z)` must equal the depth of the specified `srcSubresource` of `srcImage"
• VUID-vkCmdCopyImage-dstImage-01731
  If \textit{dstImage} is a \textit{blocked image}, then for each element of \textit{pRegions}, all members of \textit{dstOffset} must be a multiple of the corresponding dimensions of the compressed texel block.

• VUID-vkCmdCopyImage-dstImage-01732
  If \textit{dstImage} is a \textit{blocked image}, then for each element of \textit{pRegions}, \textit{extent.width} must be a multiple of the compressed texel block width or (\textit{extent.width + dstOffset.x}) must equal the width of the specified \textit{dstSubresource} of \textit{dstImage}.

• VUID-vkCmdCopyImage-dstImage-01733
  If \textit{dstImage} is a \textit{blocked image}, then for each element of \textit{pRegions}, \textit{extent.height} must be a multiple of the compressed texel block height or (\textit{extent.height + dstOffset.y}) must equal the height of the specified \textit{dstSubresource} of \textit{dstImage}.

• VUID-vkCmdCopyImage-dstImage-01734
  If \textit{dstImage} is a \textit{blocked image}, then for each element of \textit{pRegions}, \textit{extent.depth} must be a multiple of the compressed texel block depth or (\textit{extent.depth + dstOffset.z}) must equal the depth of the specified \textit{dstSubresource} of \textit{dstImage}.

Valid Usage (Implicit)

• VUID-vkCmdCopyImage-commandBuffer-parameter
  \textit{commandBuffer} must be a valid \textit{VkCommandBuffer} handle.

• VUID-vkCmdCopyImage-srcImage-parameter
  \textit{srcImage} must be a valid \textit{VkImage} handle.

• VUID-vkCmdCopyImage-srcImageLayout-parameter
  \textit{srcImageLayout} must be a valid \textit{VkImageLayout} value.

• VUID-vkCmdCopyImage-dstImage-parameter
  \textit{dstImage} must be a valid \textit{VkImage} handle.

• VUID-vkCmdCopyImage-dstImageLayout-parameter
  \textit{dstImageLayout} must be a valid \textit{VkImageLayout} value.

• VUID-vkCmdCopyImage-pRegions-parameter
  \textit{pRegions} must be a valid pointer to an array of \textit{regionCount} valid \textit{VkImageCopy} structures.

• VUID-vkCmdCopyImage-commandBuffer-recording
  \textit{commandBuffer} must be in the \textit{recording state}.

• VUID-vkCmdCopyImage-commandBuffer-cmdpool
  The \textit{VkCommandPool} that \textit{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
  \textit{regionCount} must be greater than 0.

• VUID-vkCmdCopyImage-commonparent
  Each of \textit{commandBuffer}, \textit{dstImage}, and \textit{srcImage} must have been created, allocated, or retrieved from the same \textit{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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer, Graphics, Compute</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkImageCopy` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkImageCopy {
    VkImageSubresourceLayers srcSubresource;
    VkOffset3D srcOffset;
    VkImageSubresourceLayers dstSubresource;
    VkOffset3D dstOffset;
    VkExtent3D extent;
} VkImageCopy;
```

- `srcSubresource` and `dstSubresource` are `VkImageSubresourceLayers` structures specifying the image subresources of the images used for the source and destination image data, respectively.
- `srcOffset` and `dstOffset` select the initial x, y, and z offsets in texels of the sub-regions of the source and destination image data.
- `extent` is the size in texels of the image to copy in width, height and depth.

For `VK_IMAGE_TYPE_3D` images, copies are performed slice by slice starting with the z member of the `srcOffset` or `dstOffset`, and copying `depth` slices. For images with multiple layers, copies are performed layer by layer starting with the `baseArrayLayer` member of the `srcSubresource` or `dstSubresource` and copying `layerCount` layers. Image data can be copied between images with different image types. If one image is `VK_IMAGE_TYPE_3D` and the other image is `VK_IMAGE_TYPE_2D` with multiple layers, then each slice is copied to or from a different layer.

Copies involving a multi-planar image format specify the region to be copied in terms of the `plane` to be copied, not the coordinates of the multi-planar image. This means that copies accessing the R/B planes of “.422” format images must fit the copied region within half the `width` of the parent image, and that copies accessing the R/B planes of “.420” format images must fit the copied region within half the `width` and `height` of the parent image.
Valid Usage

- VUID-VkImageCopy-extent-00140
  The number of slices of the *extent* (for 3D) or layers of the *srcSubresource* (for non-3D) must match the number of slices of the *extent* (for 3D) or layers of the *dstSubresource* (for non-3D)

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-aspectMask-02247
  *aspectMask* must not include *VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT* for any index *i*

- VUID-VkImageSubresourceLayers-layerCount-01700
  *layerCount* must be greater than 0
A more extensible version of the copy image command is defined below.

To copy data between image objects, call:

```c
// Provided by VK_KHR_copy_commands2
void vkCmdCopyImage2KHR(
    VkCommandBuffer commandBuffer,
    const VkCopyImageInfo2KHR* pCopyImageInfo);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pCopyImageInfo` is a pointer to a `VkCopyImageInfo2KHR` structure describing the copy parameters.

This command is functionally identical to `vkCmdCopyImage`, but includes extensible sub-structures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

---

**Valid Usage**

- VUID-vkCmdCopyImage2KHR-commandBuffer-01825
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `srcImage` must not be a protected image

- VUID-vkCmdCopyImage2KHR-commandBuffer-01826
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `dstImage` must not be a protected image

- VUID-vkCmdCopyImage2KHR-commandBuffer-01827
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `dstImage` must not be an unprotected image

---

**Valid Usage (Implicit)**

- VUID-vkCmdCopyImage2KHR-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdCopyImage2KHR-pCopyImageInfo-parameter
  `pCopyImageInfo` must be a valid pointer to a valid `VkCopyImageInfo2KHR` structure

- VUID-vkCmdCopyImage2KHR-commandBuffer-recording
  `commandBuffer` must be in the recording state
- VUID-vkCmdCopyImage2KHR-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support transfer, graphics, or compute operations

- VUID-vkCmdCopyImage2KHR-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
</tr>
</tbody>
</table>

The VkCopyImageInfo2KHR structure is defined as:

```c
// Provided by VK_KHR_copy_commands2
typedef struct VkCopyImageInfo2KHR {
    VkStructureType sType;
    const void* pNext;
    VkImage srcImage;
    VkImageLayout srcImageLayout;
    VkImage dstImage;
    VkImageLayout dstImageLayout;
    uint32_t regionCount;
    const VkImageCopy2KHR* pRegions;
} VkCopyImageInfo2KHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **srcImage** is the source image.
- **srcImageLayout** is the current layout of the source image subresource.
- **dstImage** is the destination image.
- **dstImageLayout** is the current layout of the destination image subresource.
- **regionCount** is the number of regions to copy.
• **pRegions** is a pointer to an array of **VkImageCopy2KHR** structures specifying the regions to copy.

### Valid Usage

- **VUID-VkCopyImageInfo2KHR-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-VkCopyImageInfo2KHR-srcImage-01995**
  The format features of **srcImage** **must** contain **VK_FORMAT_FEATURE_TRANSFER_SRC_BIT**

- **VUID-VkCopyImageInfo2KHR-srcImage-00126**
  **srcImage** **must** have been created with **VK_IMAGE_USAGE_TRANSFER_SRC_BIT** usage flag

- **VUID-VkCopyImageInfo2KHR-srcImage-01546**
  If **srcImage** is non-sparse then the image or disjoint plane to be copied **must** be bound completely and contiguously to a single **VkDeviceMemory** object

- **VUID-VkCopyImageInfo2KHR-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-VkCopyImageInfo2KHR-srcImageLayout-01917**
  **srcImageLayout** **must** be **VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL**, **VK_IMAGE_LAYOUT_GENERAL**, or **VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR**

- **VUID-VkCopyImageInfo2KHR-dstImage-01996**
  The format features of **dstImage** **must** contain **VK_FORMAT_FEATURE_TRANSFER_DST_BIT**

- **VUID-VkCopyImageInfo2KHR-dstImage-00131**
  **dstImage** **must** have been created with **VK_IMAGE_USAGE_TRANSFER_DST_BIT** usage flag

- **VUID-VkCopyImageInfo2KHR-dstImage-01547**
  If **dstImage** is non-sparse then the image or disjoint plane that is the destination of the copy **must** be bound completely and contiguously to a single **VkDeviceMemory** object

- **VUID-VkCopyImageInfo2KHR-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-VkCopyImageInfo2KHR-dstImageLayout-01395**
  **dstImageLayout** **must** be **VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL**, **VK_IMAGE_LAYOUT_GENERAL**, or **VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR**

- **VUID-VkCopyImageInfo2KHR-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 compatible, as defined above

- **VUID-VkCopyImageInfo2KHR-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-VkCopyImageInfo2KHR-srcImage-00136**
  The sample count of **srcImage** and **dstImage** **must** match
• **VUID-VkCopyImageInfo2KHR-srcSubresource-01696**
The `srcSubresource.mipLevel` member of each element of `pRegions` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `srcImage` was created.

• **VUID-VkCopyImageInfo2KHR-dstSubresource-01697**
The `dstSubresource.mipLevel` member of each element of `pRegions` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `dstImage` was created.

• **VUID-VkCopyImageInfo2KHR-srcSubresource-01698**
The `srcSubresource.baseArrayLayer + srcSubresource.layerCount` of each element of `pRegions` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `srcImage` was created.

• **VUID-VkCopyImageInfo2KHR-dstSubresource-01699**
The `dstSubresource.baseArrayLayer + dstSubresource.layerCount` of each element of `pRegions` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `dstImage` was created.

• **VUID-VkCopyImageInfo2KHR-srcOffset-01783**
The `srcOffset` and `extent` members of each element of `pRegions` **must** respect the image transfer granularity requirements of `commandBuffer`'s command pool's queue family, as described in `VkQueueFamilyProperties`.

• **VUID-VkCopyImageInfo2KHR-dstOffset-01784**
The `dstOffset` and `extent` members of each element of `pRegions` **must** respect the image transfer granularity requirements of `commandBuffer`'s command pool's queue family, as described in `VkQueueFamilyProperties`.

• **VUID-VkCopyImageInfo2KHR-srcImage-01551**
If neither `srcImage` nor `dstImage` has a **multi-planar image format** then for each element of `pRegions`, `srcSubresource.aspectMask` and `dstSubresource.aspectMask` **must** match.

• **VUID-VkCopyImageInfo2KHR-srcImage-01552**
If `srcImage` has a `VkFormat` with **two planes** then for each element of `pRegions`, `srcSubresource.aspectMask` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT` or `VK_IMAGE_ASPECT_PLANE_1_BIT`.

• **VUID-VkCopyImageInfo2KHR-srcImage-01553**
If `srcImage` has a `VkFormat` with **three planes** then for each element of `pRegions`, `srcSubresource.aspectMask` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT`.

• **VUID-VkCopyImageInfo2KHR-dstImage-01554**
If `dstImage` has a `VkFormat` with **two planes** then for each element of `pRegions`, `dstSubresource.aspectMask` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT` or `VK_IMAGE_ASPECT_PLANE_1_BIT`.

• **VUID-VkCopyImageInfo2KHR-dstImage-01555**
If `dstImage` has a `VkFormat` with **three planes** then for each element of `pRegions`, `dstSubresource.aspectMask` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT`.

• **VUID-VkCopyImageInfo2KHR-srcImage-01556**
If `srcImage` has a **multi-planar image format** and the `dstImage` does not have a multi-planar
image format, then for each element of `pRegions, dstSubresource.aspectMask` must be `VK_IMAGE_ASPECT_COLOR_BIT`

- VUID-VkCopyImageInfo2KHR-dstImage-01557
  If `dstImage` has a multi-planar image format and the `srcImage` does not have a multi-planar image format, then for each element of `pRegions, srcSubresource.aspectMask` must be `VK_IMAGE_ASPECT_COLOR_BIT`

- VUID-VkCopyImageInfo2KHR-srcImage-04443
  If `srcImage` is of type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions, srcSubresource.baseArrayLayer` must be 0 and `srcSubresource.layerCount` must be 1

- VUID-VkCopyImageInfo2KHR-dstImage-04444
  If `dstImage` is of type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions, dstSubresource.baseArrayLayer` must be 0 and `dstSubresource.layerCount` must be 1

- VUID-VkCopyImageInfo2KHR-aspectMask-00142
  For each element of `pRegions, srcSubresource.aspectMask` must specify aspects present in `srcImage`

- VUID-VkCopyImageInfo2KHR-aspectMask-00143
  For each element of `pRegions, dstSubresource.aspectMask` must specify aspects present in `dstImage`

- VUID-VkCopyImageInfo2KHR-srcOffset-00144
  For each element of `pRegions, srcOffset.x` and `(extent.width + srcOffset.x)` must both be greater than or equal to 0 and less than or equal to the width of the specified `srcSubresource` of `srcImage`

- VUID-VkCopyImageInfo2KHR-srcOffset-00145
  For each element of `pRegions, srcOffset.y` and `(extent.height + srcOffset.y)` must both be greater than or equal to 0 and less than or equal to the height of the specified `srcSubresource` of `srcImage`

- VUID-VkCopyImageInfo2KHR-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-VkCopyImageInfo2KHR-srcOffset-00147
  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-VkCopyImageInfo2KHR-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-VkCopyImageInfo2KHR-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-VkCopyImageInfo2KHR-srcImage-01787
  If `srcImage` is of type `VK_IMAGE_TYPE_2D`, then for each element of `pRegions, srcOffset.z` must be 0
If \( \text{dstImage} \) is of type \( \text{VK_IMAGE_TYPE_2D} \), then for each element of \( \text{pRegions} \), \( \text{dstOffset}.z \) must be 0.

- VUID-VkCopyImageInfo2KHR-srcImage-01790
  If \( \text{srcImage} \) and \( \text{dstImage} \) are both of type \( \text{VK_IMAGE_TYPE_2D} \), then for each element of \( \text{pRegions} \), \( \text{extent}.depth \) must be 1.

- VUID-VkCopyImageInfo2KHR-srcImage-01791
  If \( \text{srcImage} \) is of type \( \text{VK_IMAGE_TYPE_2D} \), and \( \text{dstImage} \) is of type \( \text{VK_IMAGE_TYPE_3D} \), then for each element of \( \text{pRegions} \), \( \text{extent}.depth \) must equal \( \text{srcSubresource}.layerCount \).

- VUID-VkCopyImageInfo2KHR-dstImage-01792
  If \( \text{srcImage} \) is of type \( \text{VK_IMAGE_TYPE_2D} \), and \( \text{dstImage} \) is of type \( \text{VK_IMAGE_TYPE_3D} \), then for each element of \( \text{pRegions} \), \( \text{extent}.depth \) must equal \( \text{dstSubresource}.layerCount \).

- VUID-VkCopyImageInfo2KHR-dstImage-00150
  For each element of \( \text{pRegions} \), \( \text{dstOffset}.x \) and (\( \text{extent}.width + \text{dstOffset}.x \)) must both be greater than or equal to 0 and less than or equal to the width of the specified \( \text{dstSubresource} \) of \( \text{dstImage} \).

- VUID-VkCopyImageInfo2KHR-dstImage-00151
  For each element of \( \text{pRegions} \), \( \text{dstOffset}.y \) and (\( \text{extent}.height + \text{dstOffset}.y \)) must both be greater than or equal to 0 and less than or equal to the height of the specified \( \text{dstSubresource} \) of \( \text{dstImage} \).

- VUID-VkCopyImageInfo2KHR-dstImage-00152
  If \( \text{dstImage} \) is of type \( \text{VK_IMAGE_TYPE_1D} \), then for each element of \( \text{pRegions} \), \( \text{dstOffset}.y \) must be 0 and \( \text{extent}.height \) must be 1.

- VUID-VkCopyImageInfo2KHR-dstImage-00153
  For each element of \( \text{pRegions} \), \( \text{dstOffset}.z \) and (\( \text{extent}.depth + \text{dstOffset}.z \)) must both be greater than or equal to 0 and less than or equal to the depth of the specified \( \text{dstSubresource} \) of \( \text{dstImage} \).

- VUID-VkCopyImageInfo2KHR-dstImage-01731
  If \( \text{dstImage} \) is a blocked image, then for each element of \( \text{pRegions} \), all members of \( \text{srcOffset} \) must be a multiple of the corresponding dimensions of the compressed texel block.

- VUID-VkCopyImageInfo2KHR-srcImage-01727
  If \( \text{srcImage} \) is a blocked image, then for each element of \( \text{pRegions} \), \( \text{extent}.width \) must be a multiple of the compressed texel block width or (\( \text{extent}.width + \text{srcOffset}.x \)) must equal the width of the specified \( \text{srcSubresource} \) of \( \text{srcImage} \).

- VUID-VkCopyImageInfo2KHR-srcImage-01728
  If \( \text{srcImage} \) is a blocked image, then for each element of \( \text{pRegions} \), \( \text{extent}.height \) must be a multiple of the compressed texel block height or (\( \text{extent}.height + \text{srcOffset}.y \)) must equal the height of the specified \( \text{srcSubresource} \) of \( \text{srcImage} \).

- VUID-VkCopyImageInfo2KHR-srcImage-01729
  If \( \text{srcImage} \) is a blocked image, then for each element of \( \text{pRegions} \), \( \text{extent}.depth \) must be a multiple of the compressed texel block depth or (\( \text{extent}.depth + \text{srcOffset}.z \)) must equal the depth of the specified \( \text{srcSubresource} \) of \( \text{srcImage} \).

- VUID-VkCopyImageInfo2KHR-srcImage-01730
  If \( \text{srcImage} \) is a blocked image, then for each element of \( \text{pRegions} \), all members of \( \text{dstOffset} \).
must be a multiple of the corresponding dimensions of the compressed texel block

- VUID-VkCopyImageInfo2KHR-dstImage-01732
  If \( \text{dstImage} \) is a \textit{blocked image}, then for each element of \textit{pRegions}, \( \text{extent.width} \) \textbf{must} be a multiple of the compressed texel block width or \((\text{extent.width} + \text{dstOffset.x}) \) \textbf{must} equal the width of the specified \( \text{dstSubresource} \) of \( \text{dstImage} \)

- VUID-VkCopyImageInfo2KHR-dstImage-01733
  If \( \text{dstImage} \) is a \textit{blocked image}, then for each element of \textit{pRegions}, \( \text{extent.height} \) \textbf{must} be a multiple of the compressed texel block height or \((\text{extent.height} + \text{dstOffset.y}) \) \textbf{must} equal the height of the specified \( \text{dstSubresource} \) of \( \text{dstImage} \)

- VUID-VkCopyImageInfo2KHR-dstImage-01734
  If \( \text{dstImage} \) is a \textit{blocked image}, then for each element of \textit{pRegions}, \( \text{extent.depth} \) \textbf{must} be a multiple of the compressed texel block depth or \((\text{extent.depth} + \text{dstOffset.z}) \) \textbf{must} equal the depth of the specified \( \text{dstSubresource} \) of \( \text{dstImage} \)

Valid Usage (Implicit)

- VUID-VkCopyImageInfo2KHR-sType-sType
  \( \text{sType} \) \textbf{must} be \text{VK_STRUCTURE_TYPE_COPY_IMAGE_INFO_2_KHR}

- VUID-VkCopyImageInfo2KHR-pNext-pNext
  \( \text{pNext} \) \textbf{must} be \text{NULL}

- VUID-VkCopyImageInfo2KHR-srcImage-parameter
  \( \text{srcImage} \) \textbf{must} be a valid \text{VkImage} handle

- VUID-VkCopyImageInfo2KHR-srcImageLayout-parameter
  \( \text{srcImageLayout} \) \textbf{must} be a valid \text{VkImageLayout} value

- VUID-VkCopyImageInfo2KHR-dstImage-parameter
  \( \text{dstImage} \) \textbf{must} be a valid \text{VkImage} handle

- VUID-VkCopyImageInfo2KHR-dstImageLayout-parameter
  \( \text{dstImageLayout} \) \textbf{must} be a valid \text{VkImageLayout} value

- VUID-VkCopyImageInfo2KHR-pRegions-parameter
  \( \text{pRegions} \) \textbf{must} be a valid pointer to an array of \text{regionCount} valid \text{VkImageCopy2KHR} structures

- VUID-VkCopyImageInfo2KHR-regionCount-arraylength
  \( \text{regionCount} \) \textbf{must} be greater than 0

- VUID-VkCopyImageInfo2KHR-commonparent
  Both of \( \text{dstImage} \), and \( \text{srcImage} \) \textbf{must} have been created, allocated, or retrieved from the same \text{VkDevice}

The \text{VkImageCopy2KHR} structure is defined as:

```c
// Provided by VK_KHR_copy_commands2
typedef struct VkImageCopy2KHR {
    VkStructureType sType;
```
const void* pNext;
VkImageSubresourceLayers srcSubresource;
VkOffset3D srcOffset;
VkImageSubresourceLayers dstSubresource;
VkOffset3D dstOffset;
VkExtent3D extent;
} VkImageCopy2KHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **srcSubresource** and **dstSubresource** are `VkImageSubresourceLayers` structures specifying the image subresources of the images used for the source and destination image data, respectively.
- **srcOffset** and **dstOffset** select the initial x, y, and z offsets in texels of the sub-regions of the source and destination image data.
- **extent** is the size in texels of the image to copy in width, height and depth.

### Valid Usage

- VUID-VkImageCopy2KHR-extent-00140
  The number of slices of the **extent** (for 3D) or layers of the **srcSubresource** (for non-3D) must match the number of slices of the **extent** (for 3D) or layers of the **dstSubresource** (for non-3D)

### Valid Usage (Implicit)

- VUID-VkImageCopy2KHR-sType-sType
  **sType** must be `VK_STRUCTURE_TYPE_IMAGE_COPY_2_KHR`

- VUID-VkImageCopy2KHR-pNext-pNext
  **pNext** must be NULL

- VUID-VkImageCopy2KHR-srcSubresource-parameter
  **srcSubresource** must be a valid `VkImageSubresourceLayers` structure

- VUID-VkImageCopy2KHR-dstSubresource-parameter
  **dstSubresource** must be a valid `VkImageSubresourceLayers` structure

### 19.4. Copying Data Between Buffers and Images

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 region in `pRegions` is copied from the specified region of the source buffer to the specified region of the destination image.

If `dstImage` has a multi-planar format, regions of each plane to be a target of a copy must be specified separately using the `pRegions` member of the `VkBufferImageCopy` structure. In this case, the `aspectMask` of `imageSubresource` must be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT`. For the purposes of `vkCmdCopyBufferToImage`, each plane of a multi-planar image is treated as having the format listed in `Compatible formats of planes of multi-planar formats` for the plane identified by the `aspectMask` of the corresponding subresource. This applies both to `VkFormat` and to coordinates used in the copy, which correspond to texels in the `plane` rather than how these texels map to coordinates in the image as a whole.

Valid Usage

- 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-pRegions-06217
  The image region specified by each element of `pRegions` must be contained within the specified `imageSubresource` of `dstImage`

- 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-srcBuffer-00176**
  
  If `srcBuffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-vkCmdCopyBufferToImage-dstImage-00177**
  
  `dstImage` **must** have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag

- **VUID-vkCmdCopyBufferToImage-dstImage-00178**
  
  If `dstImage` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-vkCmdCopyBufferToImage-dstImage-00179**
  
  `dstImage` **must** have a sample count equal to `VK_SAMPLE_COUNT_1_BIT`

- **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`, `VK_IMAGE_LAYOUT_GENERAL`, or `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`

- **VUID-vkCmdCopyBufferToImage-imageSubresource-01701**
  
  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-01702**
  
  The `imageSubresource.baseArrayLayer` + `imageSubresource.layerCount` of each element of `pRegions` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `dstImage` was created

- **VUID-vkCmdCopyBufferToImage-imageOffset-01793**
  
  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-04477**
  
  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-06218**
  
  For each element of `pRegions`, `imageOffset.x` and `(imageExtent.width + imageOffset.x)` **must** both be greater than or equal to 0 and less than or equal to the width of the specified `imageSubresource` of `dstImage`
For each element of $pRegions$, $imageOffset.y$ and $(imageExtent.height + imageOffset.y)$ must both be greater than or equal to 0 and less than or equal to the height of the specified imageSubresource of $dstImage$.

If $dstImage$ does not have either a depth/stencil or a multi-planar format, then for each element of $pRegions$, $bufferOffset$ must be a multiple of the format's texel block size.

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 $dstImage$ 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 $dstImage$.

If $dstImage$ 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.

If $dstImage$ is a blocked image, for each element of $pRegions$, $bufferRowLength$ must be a multiple of the compressed texel block width.

If $dstImage$ is a blocked image, for each element of $pRegions$, $bufferImageHeight$ must be a multiple of the compressed texel block height.

If $dstImage$ is a blocked image, for each element of $pRegions$, all members of $imageOffset$ must be a multiple of the corresponding dimensions of the compressed texel block.

If $dstImage$ is a blocked image, for each element of $pRegions$, $bufferOffset$ must be a multiple of the compressed texel block size in bytes.

If $dstImage$ is a blocked image, for each element of $pRegions$, $imageExtent.width$ must be a multiple of the compressed texel block width or $(imageExtent.width + imageOffset.x)$ must equal the width of the specified imageSubresource of $dstImage$.

If $dstImage$ is a blocked image, for each element of $pRegions$, $imageExtent.height$ must be a multiple of the compressed texel block height or $(imageExtent.height + imageOffset.y)$ must equal the height of the specified imageSubresource of $dstImage$.

If $dstImage$ is a blocked image, for each element of $pRegions$, $imageExtent.width$ must be a multiple of the compressed texel block width or $(imageExtent.width + imageOffset.x)$ must equal the width of the specified imageSubresource of $dstImage$.
If `dstImage` is a blocked image, for each element of `pRegions`, `imageExtent.depth` must be a multiple of the compressed texel block depth or `(imageExtent.depth + imageOffset.z)` must equal the depth of the specified `imageSubresource` of `dstImage`.

- **VUID-vkCmdCopyBufferToImage-aspectMask-00211**
  For each element of `pRegions`, `imageSubresource.aspectMask` must specify aspects present in `dstImage`.

- **VUID-vkCmdCopyBufferToImage-aspectMask-01560**
  If `dstImage` has a multi-planar format, then for each element of `pRegions`, `imageSubresource.aspectMask` must be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT` (with `VK_IMAGE_ASPECT_PLANE_2_BIT` valid only for image formats with three planes).

- **VUID-vkCmdCopyBufferToImage-baseArrayLayer-00213**
  If `dstImage` has a type `VK_IMAGE_TYPE_3D`, for each element of `pRegions`, `imageSubresource.baseArrayLayer` must be 0 and `imageSubresource.layerCount` must be 1.

- **VUID-vkCmdCopyBufferToImage-pRegions-04725**
  If `dstImage` is not a blocked image, for each element of `pRegions`, `bufferRowLength` multiplied by the texel block size of `dstImage` must be less than or equal to $2^{31}$-1.

- **VUID-vkCmdCopyBufferToImage-pRegions-04726**
  If `dstImage` is a blocked image, for each element of `pRegions`, `bufferRowLength` divided by the compressed texel block width and then multiplied by the texel block size of `dstImage` must be less than or equal to $2^{31}$-1.

- **VUID-vkCmdCopyBufferToImage-commandBuffer-04052**
  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-srcImage-04053**
  If `dstImage` has a depth/stencil format, the `bufferOffset` member of any element of `pRegions` must be a multiple of 4.

**Valid Usage (Implicit)**

- **VUID-vkCmdCopyBufferToImage-commandBuffer-parameter**
  `commandBuffer` must be a valid `VkCommandBuffer` handle.

- **VUID-vkCmdCopyBufferToImage-srcBuffer-parameter**
  `srcBuffer` must be a valid `VkBuffer` handle.

- **VUID-vkCmdCopyBufferToImage-dstImage-parameter**
  `dstImage` must be a valid `VkImage` handle.

- **VUID-vkCmdCopyBufferToImage-dstImageLayout-parameter**
  `dstImageLayout` must be a valid `VkImageLayout` value.

- **VUID-vkCmdCopyBufferToImage-pRegions-parameter**
  `pRegions` must be a valid pointer to an array of `regionCount` valid `VkBufferImageCopy` structures.
• VUID-vkCmdCopyBufferToImage-commandBuffer-recording
commandBuffer must be in the recording state

• VUID-vkCmdCopyBufferToImage-commandBuffer-cmdpool
The VkCommandPool that commandBuffer was allocated from must support transfer, graphics, or compute operations

• VUID-vkCmdCopyBufferToImage-renderpass
This command must only be called outside of a render pass instance

• VUID-vkCmdCopyBufferToImage-regionCount-arraylength
regionCount must be greater than 0

• VUID-vkCmdCopyBufferToImage-commonparent
Each of commandBuffer, dstImage, and srcBuffer must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
</tr>
</tbody>
</table>

To copy data from an image object to a buffer object, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdCopyImageToBuffer(
    VkCommandBuffer commandBuffer,
    VkImage srcImage,
    VkImageLayout srcImageLayout,
    VkBuffer dstBuffer,
    uint32_t regionCount,
    const VkBufferImageCopy* pRegions);
```

- commandBuffer is the command buffer into which the command will be recorded.
- srcImage is the source image.
- srcImageLayout is the layout of the source image subresources for the copy.
- dstBuffer is the destination buffer.
- `regionCount` is the number of regions to copy.
- `pRegions` is a pointer to an array of `VkBufferImageCopy` structures specifying the regions to copy.

Each region in `pRegions` is copied from the specified region of the source image to the specified region of the destination buffer.

If `srcImage` has a multi-planar format, regions of each plane to be a source of a copy must be specified separately using the `pRegions` member of the `VkBufferImageCopy` structure. In this case, the `aspectMask` of `imageSubresource` must be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT`. For the purposes of `vkCmdCopyBufferToImage`, each plane of a multi-planar image is treated as having the format listed in Compatible formats of planes of multi-planar formats for the plane identified by the `aspectMask` of the corresponding subresource. This applies both to `VkFormat` and to coordinates used in the copy, which correspond to texels in the `plane` rather than how these texels map to coordinates in the image as a whole.

**Valid Usage**

- 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-pRegions-06220
  The image region specified by each element of `pRegions` must be contained within the specified `imageSubresource` of `srcImage`

- VUID-vkCmdCopyImageToBuffer-pRegions-00183
  `dstBuffer` must be large enough to contain all buffer locations that are accessed according to Buffer and Image Addressing, for each element of `pRegions`

- VUID-vkCmdCopyImageToBuffer-pRegions-00184
  The union of all source regions, and the union of all destination regions, specified by the elements of `pRegions`, must not overlap in memory

- VUID-vkCmdCopyImageToBuffer-srcImage-00186
  `srcImage` must have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage flag

- VUID-vkCmdCopyImageToBuffer-srcImage-01998
  The format features of `srcImage` must contain `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT`

- VUID-vkCmdCopyImageToBuffer-srcImage-00187
  If `srcImage` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object
VUID-vkCmdCopyImageToBuffer-dstBuffer-00191

**dstBuffer** must have been created with **VK_BUFFER_USAGE_TRANSFER_DST_BIT** usage flag.

VUID-vkCmdCopyImageToBuffer-dstBuffer-00192

If **dstBuffer** is non-sparse then it **must** be bound completely and contiguously to a single **VkDeviceMemory** object.

VUID-vkCmdCopyImageToBuffer-srcImage-00188

**srcImage** must have a sample count equal to **VK_SAMPLE_COUNT_1_BIT**.

VUID-vkCmdCopyImageToBuffer-srcImageLayout-00189

**srcImageLayout** must specify the layout of the image subresources of **srcImage** specified in **pRegions** at the time this command is executed on a **VkDevice**.

VUID-vkCmdCopyImageToBuffer-srcImageLayout-01397

**srcImageLayout** must be **VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL**, **VK_IMAGE_LAYOUT_GENERAL**, or **VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR**.

VUID-vkCmdCopyImageToBuffer-imageSubresource-01703

The **imageSubresource.mipLevel** member of each element of **pRegions** must be less than the **mipLevels** specified in **VkImageCreateInfo** when **srcImage** was created.

VUID-vkCmdCopyImageToBuffer-imageSubresource-01704

The **imageSubresource.baseArrayLayer** + **imageSubresource.layerCount** of each element of **pRegions** must be less than or equal to the **arrayLayers** specified in **VkImageCreateInfo** when **srcImage** was created.

VUID-vkCmdCopyImageToBuffer-imageOffset-01794

The **imageOffset** and **imageExtent** members of each element of **pRegions** must respect the image transfer granularity requirements of **commandBuffer**’s command pool’s queue family, as described in **VkQueueFamilyProperties**.

VUID-vkCmdCopyImageToBuffer-pRegions-06221

For each element of **pRegions**, **imageOffset.x** and (**imageExtent.width** + **imageOffset.x**) must both be greater than or equal to 0 and less than or equal to the width of the specified **imageSubresource** of **srcImage**.

VUID-vkCmdCopyImageToBuffer-pRegions-06222

For each element of **pRegions**, **imageOffset.y** and (**imageExtent.height** + **imageOffset.y**) must both be greater than or equal to 0 and less than or equal to the height of the specified **imageSubresource** of **srcImage**.

VUID-vkCmdCopyImageToBuffer-bufferOffset-01558

If **srcImage** does not have either a depth/stencil or a multi-planar format, then for each element of **pRegions**, **bufferOffset** must be a multiple of the format’s texel block size.

VUID-vkCmdCopyImageToBuffer-bufferOffset-01559

If **srcImage** has a multi-planar format, then for each element of **pRegions**, **bufferOffset** must be a multiple of the element size of the compatible format for the format and the **aspectMask** of the **imageSubresource** as defined in **Compatible formats of planes of multi-planar formats**.

VUID-vkCmdCopyImageToBuffer-srcImage-00199

If **srcImage** is of type **VK_IMAGE_TYPE_1D**, then for each element of **pRegions**, **imageOffset.y** must be 0 and **imageExtent.height** must be 1.
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.

If \( srcImage \) is a blocked image, for each element of \( pRegions, bufferRowLength \) must be a multiple of the compressed texel block width.

If \( srcImage \) is a blocked image, for each element of \( pRegions, bufferImageHeight \) must be a multiple of the compressed texel block height.

If \( srcImage \) is a blocked image, for each element of \( pRegions, all members of imageOffset \) must be a multiple of the corresponding dimensions of the compressed texel block.

If \( srcImage \) is a blocked image, for each element of \( pRegions, bufferOffset \) must be a multiple of the compressed texel block size in bytes.

If \( srcImage \) is a blocked image, for each element of \( pRegions, imageExtent.width \) must be a multiple of the compressed texel block width or \((imageExtent.width + imageOffset.x)\) must equal the width of the specified imageSubresource of srcImage.

If \( srcImage \) is a blocked image, for each element of \( pRegions, imageExtent.height \) must be a multiple of the compressed texel block height or \((imageExtent.height + imageOffset.y)\) must equal the height of the specified imageSubresource of srcImage.

If \( srcImage \) is a blocked image, for each element of \( pRegions, imageExtent.depth \) must be a multiple of the compressed texel block depth or \((imageExtent.depth + imageOffset.z)\) must equal the depth of the specified imageSubresource of srcImage.

For each element of \( pRegions, imageSubresource.aspectMask \) must specify aspects present in srcImage.

If \( srcImage \) has a multi-planar format, then for each element of \( pRegions, imageSubresource.aspectMask \) must be \( VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, \) or \( VK_IMAGE_ASPECT_PLANE_2_BIT \) (with \( VK_IMAGE_ASPECT_PLANE_2_BIT \) valid only for image formats with three planes).

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.

For each element of \( pRegions \), \( imageSubresource.aspectMask \) must specify aspects present in srcImage.

If \( srcImage \) has a multi-planar format, then for each element of \( pRegions, imageSubresource.aspectMask \) must be \( VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, \) or \( VK_IMAGE_ASPECT_PLANE_2_BIT \) (with \( VK_IMAGE_ASPECT_PLANE_2_BIT \) valid only for image formats with three planes).

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.
If `srcImage` is not a blocked image, for each element of `pRegions`, `bufferRowLength` multiplied by the texel block size of `srcImage` must be less than or equal to $2^{31}-1$

- VUID-vkCmdCopyImageToBuffer-pRegions-04726
  If `srcImage` is a blocked image, for each element of `pRegions`, `bufferRowLength` divided by the compressed texel block width and then multiplied by the texel block size of `srcImage` must be less than or equal to $2^{31}-1$

- VUID-vkCmdCopyImageToBuffer-commandBuffer-04052
  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-srcImage-04053
  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-srcImage-parameter
  `srcImage` must be a valid `VkImage` handle

- VUID-vkCmdCopyImageToBuffer-srcImageLayout-parameter
  `srcImageLayout` must be a valid `VkImageLayout` value

- VUID-vkCmdCopyImageToBuffer-dstBuffer-parameter
  `dstBuffer` must be a valid `VkBuffer` handle

- VUID-vkCmdCopyImageToBuffer-pRegions-parameter
  `pRegions` must be a valid pointer to an array of `regionCount` valid `VkBufferImageCopy` structures

- VUID-vkCmdCopyImageToBuffer-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdCopyImageToBuffer-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations

- VUID-vkCmdCopyImageToBuffer-renderpass
  This command must only be called outside of a render pass instance

- VUID-vkCmdCopyImageToBuffer-regionCount-arraylength
  `regionCount` must be greater than 0

- VUID-vkCmdCopyImageToBuffer-commonparent
  Each of `commandBuffer`, `dstBuffer`, and `srcImage` must have been created, allocated, or retrieved from the same `VkDevice`
Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
</tr>
</tbody>
</table>

For both `vkCmdCopyBufferToImage` and `vkCmdCopyImageToBuffer`, each element of `pRegions` is a structure defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkBufferImageCopy {
    VkDeviceSize bufferOffset;
    uint32_t bufferRowLength;
    uint32_t bufferImageHeight;
    VkImageSubresourceLayers imageSubresource;
    VkOffset3D imageOffset;
    VkExtent3D imageExtent;
} VkBufferImageCopy;
```

- **bufferOffset** is the offset in bytes from the start of the buffer object where the image data is copied from or to.
- **bufferRowLength** and **bufferImageHeight** specify in texels a subregion of a larger two- or three-dimensional image in buffer memory, and control the addressing calculations. If either of these values is zero, that aspect of the buffer memory is considered to be tightly packed according to the **imageExtent**.
- **imageSubresource** is a `VkImageSubresourceLayers` used to specify the specific image subresources of the image used for the source or destination image data.
- **imageOffset** selects the initial x, y, z offsets in texels of the sub-region of the source or destination image data.
- **imageExtent** is the size in texels of the image to copy in width, height and depth.

When copying to or from a depth or stencil aspect, the data in buffer memory uses a layout that is a (mostly) tightly packed representation of the depth or stencil data. Specifically:

- data copied to or from the stencil aspect of any depth/stencil format is tightly packed with one `VK_FORMAT_S8_UINT` value per texel.
• data copied to or from the depth aspect of a `VK_FORMAT_D16_UNORM` or `VK_FORMAT_D16_UNORM_S8_UINT` format is tightly packed with one `VK_FORMAT_D16_UNORM` value per texel.

• data copied to or from the depth aspect of a `VK_FORMAT_D32_SFLOAT` or `VK_FORMAT_D32_SFLOAT_S8_UINT` format is tightly packed with one `VK_FORMAT_D32_SFLOAT` value per texel.

• data copied to or from the depth aspect of a `VK_FORMAT_X8_D24_UNORM_PACK32` or `VK_FORMAT_D24_UNORM_S8_UINT` format is packed with one 32-bit word per texel with the D24 value in the LSBs of the word, and undefined values in the eight MSBs.

Note

To copy both the depth and stencil aspects of a depth/stencil format, two entries in `pRegions` can be used, where one specifies the depth aspect in `imageSubresource`, and the other specifies the stencil aspect.

Because depth or stencil aspect buffer to image copies may require format conversions on some implementations, they are not supported on queues that do not support graphics.

When copying to a depth aspect, and the `VK_EXT_depth_range_unrestricted` extension is not enabled, the data in buffer memory must be in the range [0,1], or the resulting values are undefined.

Copies are done layer by layer starting with image layer `baseArrayLayer` member of `imageSubresource`. `layerCount` layers are copied from the source image or to the destination image.

For purpose of valid usage statements here and in related copy commands, a blocked image is defined as:

• an image with a single-plane, “_422” format, which is treated as a format with a $2 \times 1$ compressed texel block, or

• a compressed image.

Valid Usage

• VUID-VkBufferImageCopy-bufferRowLength-00195
  `bufferRowLength` must be 0, or greater than or equal to the `width` member of `imageExtent`

• VUID-VkBufferImageCopy-bufferImageHeight-00196
  `bufferImageHeight` must be 0, or greater than or equal to the `height` member of `imageExtent`

• VUID-VkBufferImageCopy-aspectMask-00212
  The `aspectMask` member of `imageSubresource` must only have a single bit set

Valid Usage (Implicit)

• VUID-VkBufferImageCopy-imageSubresource-parameter
  `imageSubresource` must be a valid `VkImageSubresourceLayers` structure
More extensible versions of the commands to copy between buffers and images are defined below.

To copy data from a buffer object to an image object, call:

```c
// Provided by VK_KHR_copy_commands2
void vkCmdCopyBufferToImage2KHR(
    VkCommandBuffer commandBuffer,
    const VkCopyBufferToImageInfo2KHR* pCopyBufferToImageInfo);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pCopyBufferToImageInfo` is a pointer to a `VkCopyBufferToImageInfo2KHR` structure describing the copy parameters.

This command is functionally identical to `vkCmdCopyBufferToImage`, but includes extensible sub-structures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

### Valid Usage

- VUID-vkCmdCopyBufferToImage2KHR-commandBuffer-01828
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `srcBuffer` must not be a protected buffer

- VUID-vkCmdCopyBufferToImage2KHR-commandBuffer-01829
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `dstImage` must not be a protected image

- VUID-vkCmdCopyBufferToImage2KHR-commandBuffer-01830
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `dstImage` must not be an unprotected image

### Valid Usage (Implicit)

- VUID-vkCmdCopyBufferToImage2KHR-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdCopyBufferToImage2KHR-pCopyBufferToImageInfo-parameter
  `pCopyBufferToImageInfo` must be a valid pointer to a valid `VkCopyBufferToImageInfo2KHR` structure

- VUID-vkCmdCopyBufferToImage2KHR-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdCopyBufferToImage2KHR-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations

- VUID-vkCmdCopyBufferToImage2KHR-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
</tr>
</tbody>
</table>

The `VkCopyBufferToImageInfo2KHR` structure is defined as:

```c
// Provided by VK_KHR_copy_commands2
typedef struct VkCopyBufferToImageInfo2KHR {
    VkStructureType          sType;
    const void*              pNext;
    VkBuffer                 srcBuffer;
    VkImage                  dstImage;
    VkImageLayout            dstImageLayout;
    uint32_t                 regionCount;
    const VkBufferImageCopy2KHR* pRegions;
} VkCopyBufferToImageInfo2KHR;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `srcBuffer` is the source buffer.
- `dstImage` is the destination image.
- `dstImageLayout` is the layout of the destination image subresources for the copy.
- `regionCount` is the number of regions to copy.
- `pRegions` is a pointer to an array of `VkBufferImageCopy2KHR` structures specifying the regions to copy.

Valid Usage

- VUID-VkCopyBufferToImageInfo2KHR-pRegions-00172
  The image region specified by each element of `pRegions` must be contained within the specified `imageSubresource` of `dstImage`.
- VUID-VkCopyBufferToImageInfo2KHR-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-VkCopyBufferToImageInfo2KHR-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-VkCopyBufferToImageInfo2KHR-srcBuffer-00174
  srcBuffer must have been created with **VK_BUFFER_USAGE_TRANSFER_SRC_BIT** usage flag

- VUID-VkCopyBufferToImageInfo2KHR-dstImage-01997
  The format features of **dstImage** must contain **VK_FORMAT_FEATURE_TRANSFER_DST_BIT**

- VUID-VkCopyBufferToImageInfo2KHR-srcBuffer-00176
  If srcBuffer is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object

- VUID-VkCopyBufferToImageInfo2KHR-dstImage-00177
  dstImage must have been created with **VK_IMAGE_USAGE_TRANSFER_DST_BIT** usage flag

- VUID-VkCopyBufferToImageInfo2KHR-dstImage-00178
  If dstImage is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object

- VUID-VkCopyBufferToImageInfo2KHR-dstImage-00179
  dstImage must have a sample count equal to **VK_SAMPLE_COUNT_1_BIT**

- VUID-VkCopyBufferToImageInfo2KHR-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-VkCopyBufferToImageInfo2KHR-dstImageLayout-01396
  dstImageLayout must be **VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL**, **VK_IMAGE_LAYOUT_GENERAL**, or **VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR**

- VUID-VkCopyBufferToImageInfo2KHR-imageSubresource-01701
  The imageSubresource.mipLevel member of each element of **pRegions** must be less than the **mipLevels** specified in **VkImageCreateInfo** when **dstImage** was created

- VUID-VkCopyBufferToImageInfo2KHR-imageSubresource-01702
  The imageSubresource.baseArrayLayer + imageSubresource.layerCount of each element of **pRegions** must be less than or equal to the **arrayLayers** specified in **VkImageCreateInfo** when **dstImage** was created

- VUID-VkCopyBufferToImageInfo2KHR-imageOffset-01793
  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-VkCopyBufferToImageInfo2KHR-commandBuffer-04477
  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-VkCopyBufferToImageInfo2KHR-pRegions-06223
For each element of $pRegions$ not containing $VkCopyCommandTransformInfoQCOM$ in its $pNext$ chain, $imageOffset.x$ and $(imageExtent.width + imageOffset.x)$ must both be greater than or equal to 0 and less than or equal to the width of the specified $imageSubresource$ of $dstImage$

- VUID-VkCopyBufferToImageInfo2KHR-pRegions-06224
  For each element of $pRegions$ not containing $VkCopyCommandTransformInfoQCOM$ in its $pNext$ chain, $imageOffset.y$ and $(imageExtent.height + imageOffset.y)$ must both be greater than or equal to 0 and less than or equal to the height of the specified $imageSubresource$ of $dstImage$

- VUID-VkCopyBufferToImageInfo2KHR-bufferOffset-01558
  If $dstImage$ does not have either a depth/stencil or a multi-planar format, then for each element of $pRegions$, $bufferOffset$ must be a multiple of the format's texel block size

- VUID-VkCopyBufferToImageInfo2KHR-bufferOffset-01559
  If $dstImage$ has a multi-planar format, then for each element of $pRegions$, $bufferOffset$ must be a multiple of the element size of the compatible format for the format and the $aspectMask$ of the $imageSubresource$ as defined in Compatible formats of planes of multi-planar formats

- VUID-VkCopyBufferToImageInfo2KHR-srcImage-00199
  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-VkCopyBufferToImageInfo2KHR-imageOffset-00200
  For each element of $pRegions$, $imageOffset.z$ and $(imageExtent.depth + imageOffset.z)$ must both be greater than or equal to 0 and less than or equal to the depth of the specified $imageSubresource$ of $dstImage$

- VUID-VkCopyBufferToImageInfo2KHR-srcImage-00201
  If $dstImage$ is of type $VK_IMAGE_TYPE_1D$ or $VK_IMAGE_TYPE_2D$, then for each element of $pRegions$, $imageOffset.z$ must be 0 and $imageExtent.depth$ must be 1

- VUID-VkCopyBufferToImageInfo2KHR-bufferRowLength-00203
  If $dstImage$ is a blocked image, for each element of $pRegions$, $bufferRowLength$ must be a multiple of the compressed texel block width

- VUID-VkCopyBufferToImageInfo2KHR-bufferImageHeight-00204
  If $dstImage$ is a blocked image, for each element of $pRegions$, $bufferImageHeight$ must be a multiple of the compressed texel block height

- VUID-VkCopyBufferToImageInfo2KHR-imageOffset-00205
  If $dstImage$ is a blocked image, for each element of $pRegions$, all members of $imageOffset$ must be a multiple of the corresponding dimensions of the compressed texel block

- VUID-VkCopyBufferToImageInfo2KHR-bufferOffset-00206
  If $dstImage$ is a blocked image, for each element of $pRegions$, $bufferOffset$ must be a multiple of the compressed texel block size in bytes

- VUID-VkCopyBufferToImageInfo2KHR-imageExtent-00207
  If $dstImage$ is a blocked image, for each element of $pRegions$, $imageExtent.width$ must be a multiple of the compressed texel block width or $(imageExtent.width + imageOffset.x)$ must equal the width of the specified $imageSubresource$ of $dstImage$

- VUID-VkCopyBufferToImageInfo2KHR-imageExtent-00208

If \( \text{dstImage} \) is a **blocked image**, for each element of \( \text{pRegions} \), \( \text{imageExtent}.\text{height} \) **must** be a multiple of the compressed texel block height or \((\text{imageExtent}.\text{height} + \text{imageOffset}.y)\) **must** equal the height of the specified \( \text{imageSubresource} \) of \( \text{dstImage} \)

- **VUID-VkCopyBufferToImageInfo2KHR-imageExtent-00209**
  If \( \text{dstImage} \) is a **blocked image**, for each element of \( \text{pRegions} \), \( \text{imageExtent}.\text{depth} \) **must** be a multiple of the compressed texel block depth or \((\text{imageExtent}.\text{depth} + \text{imageOffset}.z)\) **must** equal the depth of the specified \( \text{imageSubresource} \) of \( \text{dstImage} \)

- **VUID-VkCopyBufferToImageInfo2KHR-aspectMask-00211**
  For each element of \( \text{pRegions} \), \( \text{imageSubresource}.\text{aspectMask} \) **must** specify aspects present in \( \text{dstImage} \)

- **VUID-VkCopyBufferToImageInfo2KHR-aspectMask-01560**
  If \( \text{dstImage} \) has a **multi-planar format**, then for each element of \( \text{pRegions} \), \( \text{imageSubresource}.\text{aspectMask} \) **must** be \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \), \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \), or \( \text{VK_IMAGE_ASPECT_PLANE_2_BIT} \) (with \( \text{VK_IMAGE_ASPECT_PLANE_2_BIT} \) valid only for image formats with three planes)

- **VUID-VkCopyBufferToImageInfo2KHR-baseArrayLayer-00213**
  If \( \text{dstImage} \) is of type \( \text{VK_IMAGE_TYPE_3D} \), for each element of \( \text{pRegions} \), \( \text{imageSubresource}.\text{baseArrayLayer} \) **must** be \( 0 \) and \( \text{imageSubresource}.\text{layerCount} \) **must** be \( 1 \)

- **VUID-VkCopyBufferToImageInfo2KHR-pRegions-04725**
  If \( \text{dstImage} \) is not a **blocked image**, for each element of \( \text{pRegions} \), \( \text{bufferRowLength} \) multiplied by the texel block size of \( \text{dstImage} \) **must** be less than or equal to \( 2^{31}-1 \)

- **VUID-VkCopyBufferToImageInfo2KHR-pRegions-04726**
  If \( \text{dstImage} \) is a **blocked image**, for each element of \( \text{pRegions} \), \( \text{bufferRowLength} \) divided by the compressed texel block width and then multiplied by the texel block size of \( \text{dstImage} \) **must** be less than or equal to \( 2^{31}-1 \)

- **VUID-VkCopyBufferToImageInfo2KHR-commandBuffer-04052**
  If the queue family used to create the \( \text{VkCommandPool} \) which \( \text{commandBuffer} \) was allocated from does not support \( \text{VK_QUEUE_GRAPHICS_BIT} \) or \( \text{VK_QUEUE_COMPUTE_BIT} \), the \( \text{bufferOffset} \) member of any element of \( \text{pRegions} \) **must** be a multiple of \( 4 \)

- **VUID-VkCopyBufferToImageInfo2KHR-srcImage-04053**
  If \( \text{dstImage} \) has a depth/stencil format, the \( \text{bufferOffset} \) member of any element of \( \text{pRegions} \) **must** be a multiple of \( 4 \)

### Valid Usage (Implicit)

- **VUID-VkCopyBufferToImageInfo2KHR-sType-sType**
  \( \text{sType} \) **must** be \( \text{VK_STRUCTURE_TYPE_COPY_BUFFER_TO_IMAGE_INFO_2_KHR} \)

- **VUID-VkCopyBufferToImageInfo2KHR-pNext-pNext**
  \( \text{pNext} \) **must** be \( \text{NULL} \)

- **VUID-VkCopyBufferToImageInfo2KHR-srcBuffer-parameter**
  \( \text{srcBuffer} \) **must** be a valid \( \text{VkBuffer} \) handle

- **VUID-VkCopyBufferToImageInfo2KHR-dstImage-parameter**

---
dstImage must be a valid VkImage handle

- VUID-VkCopyBufferToImageInfo2KHR-dstImageLayout-parameter
dstImageLayout must be a valid VkImageLayout value

- VUID-VkCopyBufferToImageInfo2KHR-pRegions-parameter
pRegions must be a valid pointer to an array of regionCount valid VkBufferImageCopy2KHR structures

- VUID-VkCopyBufferToImageInfo2KHR-regionCount-arraylength
regionCount must be greater than 0

- VUID-VkCopyBufferToImageInfo2KHR-commonparent
Both of dstImage, and srcBuffer must have been created, allocated, or retrieved from the same VkDevice

To copy data from an image object to a buffer object, call:

```c
// Provided by VK_KHR_copy_commands2
void vkCmdCopyImageToBuffer2KHR(
    VkCommandBuffer commandBuffer,
    const VkCopyImageToBufferInfo2KHR* pCopyImageToBufferInfo);
```

- commandBuffer is the command buffer into which the command will be recorded.

- pCopyImageToBufferInfo is a pointer to a VkCopyImageToBufferInfo2KHR structure describing the copy parameters.

This command is functionally identical to vkCmdCopyImageToBuffer, but includes extensible substructures that include sType and pNext parameters, allowing them to be more easily extended.

Valid Usage

- VUID-vkCmdCopyImageToBuffer2KHR-commandBuffer-01831
  If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, srcImage must not be a protected image

- VUID-vkCmdCopyImageToBuffer2KHR-commandBuffer-01832
  If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, dstBuffer must not be a protected buffer

- VUID-vkCmdCopyImageToBuffer2KHR-commandBuffer-01833
  If commandBuffer is a protected command buffer and protectedNoFault is not supported, dstBuffer must not be an unprotected buffer

Valid Usage (Implicit)

- VUID-vkCmdCopyImageToBuffer2KHR-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle
• **VUID-vkCmdCopyImageToBuffer2KHR-pCopyImageToBufferInfo-parameter**
  pCopyImageToBufferInfo **must** be a valid pointer to a valid VkCopyImageToBufferInfo2KHR structure
  
• **VUID-vkCmdCopyImageToBuffer2KHR-commandBuffer-recording**
  commandBuffer **must** be in the recording state
  
• **VUID-vkCmdCopyImageToBuffer2KHR-commandBuffer-cmdpool**
  The VkCommandPool that commandBuffer was allocated from **must** support transfer, graphics, or compute operations

• **VUID-vkCmdCopyImageToBuffer2KHR-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
</tr>
</tbody>
</table>

The VkCopyImageToBufferInfo2KHR structure is defined as:

```c
// Provided by VK_KHR_copy_commands2
typedef struct VkCopyImageToBufferInfo2KHR {
    VkStructureType sType;
    const void* pNext;
    VkImage srcImage;
    VkImageLayout srcImageLayout;
    VkBuffer dstBuffer;
    uint32_t regionCount;
    const VkBufferImageCopy2KHR* pRegions;
} VkCopyImageToBufferInfo2KHR;
```

• sType **is** the type of this structure.

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

• srcImage **is** the source image.

• srcImageLayout **is** the layout of the source image subresources for the copy.
• **dstBuffer** is the destination buffer.

• **regionCount** is the number of regions to copy.

• **pRegions** is a pointer to an array of **VkBufferImageCopy2KHR** structures specifying the regions to copy.

### Valid Usage

- **VUID-VkCopyImageToBufferInfo2KHR-pRegions-00182**
  The image region specified by each element of **pRegions** must be contained within the specified **imageSubresource** of **srcImage**

- **VUID-VkCopyImageToBufferInfo2KHR-pRegions-00183**
  **dstBuffer** must be large enough to contain all buffer locations that are accessed according to **Buffer and Image Addressing**, for each element of **pRegions**

- **VUID-VkCopyImageToBufferInfo2KHR-pRegions-00184**
  The union of all source regions, and the union of all destination regions, specified by the elements of **pRegions**, must not overlap in memory

- **VUID-VkCopyImageToBufferInfo2KHR-srcImage-00186**
  **srcImage** must have been created with **VK_IMAGE_USAGE_TRANSFER_SRC_BIT** usage flag

- **VUID-VkCopyImageToBufferInfo2KHR-srcImage-01998**
  The **format features** of **srcImage** must contain **VK_FORMAT_FEATURE_TRANSFER_SRC_BIT**

- **VUID-VkCopyImageToBufferInfo2KHR-srcImage-00187**
  If **srcImage** is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object

- **VUID-VkCopyImageToBufferInfo2KHR-dstBuffer-00191**
  **dstBuffer** must have been created with **VK_BUFFER_USAGE_TRANSFER_DST_BIT** usage flag

- **VUID-VkCopyImageToBufferInfo2KHR-dstBuffer-00192**
  If **dstBuffer** is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object

- **VUID-VkCopyImageToBufferInfo2KHR-srcImage-00188**
  **srcImage** must have a sample count equal to **VK_SAMPLE_COUNT_1_BIT**

- **VUID-VkCopyImageToBufferInfo2KHR-srcImageLayout-00189**
  **srcImageLayout** must specify the layout of the image subresources of **srcImage** specified in **pRegions** at the time this command is executed on a **VkDevice**

- **VUID-VkCopyImageToBufferInfo2KHR-srcImageLayout-01397**
  **srcImageLayout** must be **VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL**, **VK_IMAGE_LAYOUT_GENERAL**, or **VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR**

- **VUID-VkCopyImageToBufferInfo2KHR-imageSubresource-01703**
  The **imageSubresource.mipLevel** member of each element of **pRegions** must be less than the **mipLevels** specified in **VkImageCreateInfo** when **srcImage** was created

- **VUID-VkCopyImageToBufferInfo2KHR-imageSubresource-01704**
  The **imageSubresource.baseArrayLayer + imageSubresource.layerCount** of each element of **pRegions** must be less than or equal to the **arrayLayers** specified in **VkImageCreateInfo**
when `srcImage` was created

- **VUID-VkCopyImageToBufferInfo2KHR-imageOffset-01794**
  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-VkCopyImageToBufferInfo2KHR-imageOffset-00197**
  For each element of `pRegions` not containing `VkCopyCommandTransformInfoQCOM` in its `pNext` chain, `imageOffset.x` and `(imageExtent.width + imageOffset.x)` must both be greater than or equal to 0 and less than or equal to the width of the specified `imageSubresource` of `srcImage`.

- **VUID-VkCopyImageToBufferInfo2KHR-imageOffset-00198**
  For each element of `pRegions` not containing `VkCopyCommandTransformInfoQCOM` in its `pNext` chain, `imageOffset.y` and `(imageExtent.height + imageOffset.y)` must both be greater than or equal to 0 and less than or equal to the height of the specified `imageSubresource` of `srcImage`.

- **VUID-VkCopyImageToBufferInfo2KHR-bufferOffset-01558**
  If `srcImage` does not have either a depth/stencil or a multi-planar format, then for each element of `pRegions`, `bufferOffset` must be a multiple of the format's texel block size.

- **VUID-VkCopyImageToBufferInfo2KHR-bufferOffset-01559**
  If `srcImage` has a multi-planar format, then for each element of `pRegions`, `bufferOffset` must be a multiple of the element size of the compatible format for the format and the `aspectMask` of the `imageSubresource` as defined in `Compatible formats of planes of multi-planar formats`.

- **VUID-VkCopyImageToBufferInfo2KHR-srcImage-00199**
  If `srcImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `imageOffset.y` must be 0 and `imageExtent.height` must be 1.

- **VUID-VkCopyImageToBufferInfo2KHR-bufferRowLength-00203**
  If `srcImage` is a blocked image, for each element of `pRegions`, `bufferRowLength` must be a multiple of the compressed texel block width.

- **VUID-VkCopyImageToBufferInfo2KHR-bufferImageHeight-00204**
  If `srcImage` is a blocked image, for each element of `pRegions`, `bufferImageHeight` must be a multiple of the compressed texel block height.

- **VUID-VkCopyImageToBufferInfo2KHR-imageOffset-00205**
  If `srcImage` is a blocked image, for each element of `pRegions`, all members of `imageOffset` must be a multiple of the corresponding dimensions of the compressed texel block.

- **VUID-VkCopyImageToBufferInfo2KHR-bufferOffset-00206**
  If `srcImage` is a blocked image, for each element of `pRegions`, `bufferOffset` must be a
multiple of the compressed texel block size in bytes

• VUID-VkCopyImageToBufferInfo2KHR-imageExtent-00207
  If srcImage is a blocked image, for each element of pRegions, imageExtent.width must be a multiple of the compressed texel block width or (imageExtent.width + imageOffset.x) must equal the width of the specified imageSubresource of srcImage.

• VUID-VkCopyImageToBufferInfo2KHR-imageExtent-00208
  If srcImage is a blocked image, for each element of pRegions, imageExtent.height must be a multiple of the compressed texel block height or (imageExtent.height + imageOffset.y) must equal the height of the specified imageSubresource of srcImage.

• VUID-VkCopyImageToBufferInfo2KHR-imageExtent-00209
  If srcImage is a blocked image, for each element of pRegions, imageExtent.depth must be a multiple of the compressed texel block depth or (imageExtent.depth + imageOffset.z) must equal the depth of the specified imageSubresource of srcImage.

• VUID-VkCopyImageToBufferInfo2KHR-aspectMask-00211
  For each element of pRegions, imageSubresource.aspectMask must specify aspects present in srcImage.

• VUID-VkCopyImageToBufferInfo2KHR-aspectMask-01560
  If srcImage has a multi-planar format, then for each element of pRegions, imageSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or VK_IMAGE_ASPECT_PLANE_2_BIT (with VK_IMAGE_ASPECT_PLANE_2_BIT valid only for image formats with three planes).

• VUID-VkCopyImageToBufferInfo2KHR-baseArrayLayer-00213
  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-VkCopyImageToBufferInfo2KHR-pRegions-04725
  If srcImage is not a blocked image, for each element of pRegions, bufferRowLength multiplied by the texel block size of srcImage must be less than or equal to $2^{31} - 1$.

• VUID-VkCopyImageToBufferInfo2KHR-pRegions-04726
  If srcImage is a blocked image, for each element of pRegions, bufferRowLength divided by the compressed texel block width and then multiplied by the texel block size of srcImage must be less than or equal to $2^{31} - 1$.

• VUID-VkCopyImageToBufferInfo2KHR-commandBuffer-04052
  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-VkCopyImageToBufferInfo2KHR-srcImage-04053
  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-VkCopyImageToBufferInfo2KHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_COPY_IMAGE_TO_BUFFER_INFO_2_KHR.
• VUID-VkCopyImageToBufferInfo2KHR-pNext-pNext  
  pNext must be NULL

• VUID-VkCopyImageToBufferInfo2KHR-srcImage-parameter  
  srcImage must be a valid VkImage handle

• VUID-VkCopyImageToBufferInfo2KHR-srcImageLayout-parameter  
  srcImageLayout must be a valid VkImageLayout value

• VUID-VkCopyImageToBufferInfo2KHR-dstBuffer-parameter  
  dstBuffer must be a valid VkBuffer handle

• VUID-VkCopyImageToBufferInfo2KHR-pRegions-parameter  
  pRegions must be a valid pointer to an array of regionCount valid VkBufferImageCopy2KHR structures

• VUID-VkCopyImageToBufferInfo2KHR-regionCount-arraylength  
  regionCount must be greater than 0

• VUID-VkCopyImageToBufferInfo2KHR-commonparent  
  Both of dstBuffer, and srcImage must have been created, allocated, or retrieved from the same VkDevice

For both vkCmdCopyBufferToImage2KHR and vkCmdCopyImageToBuffer2KHR, each element of pRegions is a structure defined as:

```c
// Provided by VK_KHR_copy_commands2
typedef struct VkBufferImageCopy2KHR {
    VkStructureType sType;
    const void* pNext;
    VkDeviceSize bufferOffset;
    uint32_t bufferRowLength;
    uint32_t bufferImageHeight;
    VkImageSubresourceLayers imageSubresource;
    VkOffset3D imageOffset;
    VkExtent3D imageExtent;
} VkBufferImageCopy2KHR;
```

• sType is the type of this structure.

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

• 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.

This structure is functionally identical to VkBufferImageCopy, but adds sType and pNext parameters, allowing it to be more easily extended.

### Valid Usage

- VUID-VkBufferImageCopy2KHR-bufferRowLength-00195
  
  bufferRowLength must be 0, or greater than or equal to the width member of imageExtent

- VUID-VkBufferImageCopy2KHR-bufferImageHeight-00196
  
  bufferImageHeight must be 0, or greater than or equal to the height member of imageExtent

- VUID-VkBufferImageCopy2KHR-aspectMask-00212
  
  The aspectMask member of imageSubresource must only have a single bit set

### Valid Usage (Implicit)

- VUID-VkBufferImageCopy2KHR-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_BUFFER_IMAGE_COPY_2_KHR

- VUID-VkBufferImageCopy2KHR-pNext-pNext
  
  pNext must be NULL

- VUID-VkBufferImageCopy2KHR-imageSubresource-parameter
  
  imageSubresource must be a valid VkImageSubresourceLayers structure

### 19.4.1. Buffer and Image Addressing

Pseudocode for image/buffer addressing of uncompressed formats is:

```c
rowLength = region->bufferRowLength;
if (rowLength == 0)
    rowLength = region->imageExtent.width;

imageHeight = region->bufferImageHeight;
if (imageHeight == 0)
    imageHeight = region->imageExtent.height;

texelBlockSize = <texel block size of the format of the src/dstImage>;

address of (x,y,z) = region->bufferOffset + (((z * imageHeight) + y) * rowLength + x) * texelBlockSize;

where x,y,z range from (0,0,0) to region->imageExtent.{width,height,depth}.
```

Note that imageOffset does not affect addressing calculations for buffer memory. Instead,
bufferOffset can be used to select the starting address in buffer memory.

For block-compressed formats, all parameters are still specified in texels rather than compressed texel blocks, but the addressing math operates on whole compressed texel blocks. Pseudocode for compressed copy addressing is:

```c
rowLength = region->bufferRowLength;
if (rowLength == 0)
    rowLength = region->imageExtent.width;

imageHeight = region->bufferImageHeight;
if (imageHeight == 0)
    imageHeight = region->imageExtent.height;

compressedTexelBlockSizeInBytes = <compressed texel block size taken from the src /dstImage>;
rowLength = (rowLength + compressedTexelBlockWidth - 1) / compressedTexelBlockWidth;
imageHeight = (imageHeight + compressedTexelBlockHeight - 1) / compressedTexelBlockHeight;

address of (x,y,z) = region->bufferOffset + ((z * imageHeight + y) * rowLength + x) * compressedTexelBlockSizeInBytes;
```

where x,y,z range from (0,0,0) to region->imageExtent.{width/compressedTexelBlockWidth, height/compressedTexelBlockHeight, depth/compressedTexelBlockDepth}.

Copying to or from block-compressed images is typically done in multiples of the compressed texel block size. For this reason the imageExtent must be a multiple of the compressed texel block dimension. There is one exception to this rule which is required to handle compressed images created with dimensions that are not a multiple of the compressed texel block dimensions:

- If imageExtent.width is not a multiple of the compressed texel block width, then (imageExtent.width + imageOffset.x) must equal the image subresource width.
- If imageExtent.height is not a multiple of the compressed texel block height, then (imageExtent.height + imageOffset.y) must equal the image subresource height.
- If imageExtent.depth is not a multiple of the compressed texel block depth, then (imageExtent.depth + imageOffset.z) must equal the image subresource depth.

This allows the last compressed texel block of the image in each non-multiple dimension to be included as a source or destination of the copy.

### 19.5. Image Copies with Scaling

To copy regions of a source image into a destination image, potentially performing format conversion, arbitrary scaling, and filtering, call:
void vkCmdBlitImage(
  VkCommandBuffer commandBuffer,
  VkImage srcImage,
  VkImageLayout srcImageLayout,
  VkImage dstImage,
  VkImageLayout dstImageLayout,
  uint32_t regionCount,
  const VkImageBlit* pRegions,
  VkFilter filter);

- `commandBuffer` is the command buffer into which the command will be recorded.
- `srcImage` is the source image.
- `srcImageLayout` is the layout of the source image subresources for the blit.
- `dstImage` is the destination image.
- `dstImageLayout` is the layout of the destination image subresources for the blit.
- `regionCount` is the number of regions to blit.
- `pRegions` is a pointer to an array of `VkImageBlit` structures specifying the regions to blit.
- `filter` is a `VkFilter` specifying the filter to apply if the blits require scaling.

`vkCmdBlitImage` must not be used for multisampled source or destination images. Use `vkCmdResolveImage` for this purpose.

As the sizes of the source and destination extents can differ in any dimension, texels in the source extent are scaled and filtered to the destination extent. Scaling occurs via the following operations:

- For each destination texel, the integer coordinate of that texel is converted to an unnormalized texture coordinate, using the effective inverse of the equations described in unnormalized to integer conversion:

  \[
  u_{\text{base}} = i + \frac{1}{2} \\
  v_{\text{base}} = j + \frac{1}{2} \\
  w_{\text{base}} = k + \frac{1}{2}
  \]

- These base coordinates are then offset by the first destination offset:

  \[
  u_{\text{offset}} = u_{\text{base}} - x_{\text{dst0}} \\
  v_{\text{offset}} = v_{\text{base}} - y_{\text{dst0}}
  \]
\[ w_{\text{offset}} = w_{\text{base}} - z_{\text{dst}0} \]

\[ a_{\text{offset}} = a - \text{baseArrayCount}_{\text{dst}} \]

- The scale is determined from the source and destination regions, and applied to the offset coordinates:

\[ \text{scale}_u = \frac{x_{\text{src}1} - x_{\text{src}0}}{x_{\text{dst}1} - x_{\text{dst}0}} \]

\[ \text{scale}_v = \frac{y_{\text{src}1} - y_{\text{src}0}}{y_{\text{dst}1} - y_{\text{dst}0}} \]

\[ \text{scale}_w = \frac{z_{\text{src}1} - z_{\text{src}0}}{z_{\text{dst}1} - z_{\text{dst}0}} \]

\[ u_{\text{scaled}} = u_{\text{offset}} \times \text{scale}_u \]

\[ v_{\text{scaled}} = v_{\text{offset}} \times \text{scale}_v \]

\[ w_{\text{scaled}} = w_{\text{offset}} \times \text{scale}_w \]

- Finally the source offset is added to the scaled coordinates, to determine the final unnormalized coordinates used to sample from \textit{srcImage}:

\[ u = u_{\text{scaled}} + x_{\text{src}0} \]

\[ v = v_{\text{scaled}} + y_{\text{src}0} \]

\[ w = w_{\text{scaled}} + z_{\text{src}0} \]

\[ q = \text{mipLevel} \]

\[ a = a_{\text{offset}} + \text{baseArrayCount}_{\text{src}} \]

These coordinates are used to sample from the source image, as described in \textit{Image Operations} chapter, with the filter mode equal to that of \textit{filter}, a mipmap mode of \textit{VK_SAMPLER_MIPMAP_MODE_NEAREST} and an address mode of \textit{VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE}. Implementations \textbf{must} clamp at the edge of the source image, and \textbf{may} additionally clamp to the
edge of the source region.

**Note**
Due to allowable rounding errors in the generation of the source texture coordinates, it is not always possible to guarantee exactly which source texels will be sampled for a given blit. As rounding errors are implementation-dependent, the exact results of a blitting operation are also implementation-dependent.

Blits are done layer by layer starting with the `baseArrayLayer` member of `srcSubresource` for the source and `dstSubresource` for the destination. `layerCount` layers are blitted to the destination image.

When blitting 3D textures, slices in the destination region bounded by `dstOffsets[0].z` and `dstOffsets[1].z` are sampled from slices in the source region bounded by `srcOffsets[0].z` and `srcOffsets[1].z`. If the `filter` parameter is `VK_FILTER_LINEAR` then the value sampled from the source image is taken by doing linear filtering using the interpolated `z` coordinate represented by `w` in the previous equations. If the `filter` parameter is `VK_FILTER_NEAREST` then the value sampled from the source image is taken from the single nearest slice, with an implementation-dependent arithmetic rounding mode.

The following filtering and conversion rules apply:

- Integer formats can only be converted to other integer formats with the same signedness.
- No format conversion is supported between depth/stencil images. The formats must match.
- Format conversions on unorm, snorm, unscaled and packed float formats of the copied aspect of the image are performed by first converting the pixels to float values.
- For sRGB source formats, nonlinear RGB values are converted to linear representation prior to filtering.
- After filtering, the float values are first clamped and then cast to the destination image format. In case of sRGB destination format, linear RGB values are converted to nonlinear representation before writing the pixel to the image.

Signed and unsigned integers are converted by first clamping to the representable range of the destination format, then casting the value.

### Valid Usage

- **VUID-vkCmdBlitImage-commandBuffer-01834**
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `srcImage` must not be a protected image
- **VUID-vkCmdBlitImage-commandBuffer-01835**
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `dstImage` must not be a protected image
- **VUID-vkCmdBlitImage-commandBuffer-01836**
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `dstImage` must not be an unprotected image
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'\textsubscript{u}C\textsubscript{r} 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_SHARED_PRESENT_KHR}, \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'\textsubscript{u}C\textsubscript{r} 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}.

\texttt{dstImageLayout} must be \texttt{VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR}, \texttt{VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL} or \texttt{VK_IMAGE_LAYOUT_GENERAL}.

If either of \texttt{srcImage} or \texttt{dstImage} was created with a signed integer \texttt{VkFormat}, the other must also have been created with a signed integer \texttt{VkFormat}. 

If either of `srcImage` or `dstImage` was created with an unsigned integer `VkFormat`, the other must also have been created with an unsigned integer `VkFormat`.

If either of `srcImage` or `dstImage` was created with a depth/stencil format, the other must have exactly the same format.

If `srcImage` was created with a depth/stencil format, `filter` must be `VK_FILTER_NEAREST`.

`srcImage` must have been created with a `samples` value of `VK_SAMPLE_COUNT_1_BIT`.

`dstImage` must have been created with a `samples` value of `VK_SAMPLE_COUNT_1_BIT`.

If `filter` is `VK_FILTER_LINEAR`, then the format features of `srcImage` must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.

If `filter` is `VK_FILTER_CUBIC_EXT`, then the format features of `srcImage` must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`.

If `filter` is `VK_FILTER_CUBIC_EXT`, `srcImage` must be of type `VK_IMAGE_TYPE_2D`.

The `srcSubresource.mipLevel` member of each element of `pRegions` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `srcImage` was created.

The `dstSubresource.mipLevel` member of each element of `pRegions` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `dstImage` was created.

The `srcSubresource.baseArrayLayer + srcSubresource.layerCount` of each element of `pRegions` must be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `srcImage` was created.

The `dstSubresource.baseArrayLayer + dstSubresource.layerCount` of each element of `pRegions` must be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `dstImage` was created.

If either `srcImage` or `dstImage` is of type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions`, `srcSubresource.baseArrayLayer` and `dstSubresource.baseArrayLayer` must each be 0, and `srcSubresource.layerCount` and `dstSubresource.layerCount` must each be 1.

For each element of `pRegions`, `srcSubresource.aspectMask` must specify aspects present in `srcImage`.

For each element of `pRegions`, `dstSubresource.aspectMask` must specify aspects present in `dstImage`.
For each element of `pRegions`, `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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The VkImageBlit structure is defined as:

```c
// Provided by VK_VERSION_1_0
```
typedef struct VkImageBlit {
    VkImageSubresourceLayers srcSubresource;
    VkOffset3D             srcOffsets[2];
    VkImageSubresourceLayers dstSubresource;
    VkOffset3D             dstOffsets[2];
} VkImageBlit;

- `srcSubresource` is the subresource to blit from.
- `srcOffsets` is a pointer to an array of two `VkOffset3D` structures specifying the bounds of the source region within `srcSubresource`.
- `dstSubresource` is the subresource to blit into.
- `dstOffsets` is a pointer to an array of two `VkOffset3D` structures specifying the bounds of the destination region within `dstSubresource`.

For each element of the `pRegions` array, a blit operation is performed for the specified source and destination regions.

### Valid Usage

- VUID-VkImageBlit-aspectMask-00238
  The `aspectMask` member of `srcSubresource` and `dstSubresource` **must** match
- VUID-VkImageBlit-layerCount-00239
  The `layerCount` member of `srcSubresource` and `dstSubresource` **must** match

### 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

A more extensible version of the blit image command is defined below.

To copy regions of a source image into a destination image, potentially performing format conversion, arbitrary scaling, and filtering, call:

```c
// Provided by VK_KHR_copy_commands2
void vkCmdBlitImage2KHR(
    VkCommandBuffer commandBuffer,
    const VkBlitImageInfo2KHR* pBlitImageInfo);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pBlitImageInfo` is a pointer to a `VkBlitImageInfo2KHR` structure describing the blit parameters.
This command is functionally identical to `vkCmdBlitImage`, but includes extensible sub-structures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

### Valid Usage

- **VUID-vkCmdBlitImage2KHR-commandBuffer-01834**
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `srcImage` must not be a protected image.

- **VUID-vkCmdBlitImage2KHR-commandBuffer-01835**
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `dstImage` must not be a protected image.

- **VUID-vkCmdBlitImage2KHR-commandBuffer-01836**
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `dstImage` must not be an unprotected image.

### Valid Usage (Implicit)

- **VUID-vkCmdBlitImage2KHR-commandBuffer-parameter**
  `commandBuffer` must be a valid `VkCommandBuffer` handle.

- **VUID-vkCmdBlitImage2KHR-pBlitImageInfo-parameter**
  `pBlitImageInfo` must be a valid pointer to a valid `VkBlitImageInfo2KHR` structure.

- **VUID-vkCmdBlitImage2KHR-commandBuffer-recording**
  `commandBuffer` must be in the recording state.

- **VUID-vkCmdBlitImage2KHR-commandBuffer-cmdpool**
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations.

- **VUID-vkCmdBlitImage2KHR-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

882
The *VkBlitImageInfo2KHR* structure is defined as:

```c
// Provided by VK_KHR_copy_commands2
typedef struct VkBlitImageInfo2KHR {
    VkStructureType sType;
    const void* pNext;
    VkImage srcImage;
    VkImageLayout srcImageLayout;
    VkImage dstImage;
    VkImageLayout dstImageLayout;
    uint32_t regionCount;
    const VkImageBlit2KHR* pRegions;
    VkFilter filter;
} VkBlitImageInfo2KHR;
```

- *sType* is the type of this structure.
- *pNext* is *NULL* or a pointer to a structure extending this structure.
- *srcImage* is the source image.
- *srcImageLayout* is the layout of the source image subresources for the blit.
- *dstImage* is the destination image.
- *dstImageLayout* is the layout of the destination image subresources for the blit.
- *regionCount* is the number of regions to blit.
- *pRegions* is a pointer to an array of *VkImageBlit2KHR* structures specifying the regions to blit.
- *filter* is a *VkFilter* specifying the filter to apply if the blits require scaling.

## Valid Usage

- **VUID-VkBlitImageInfo2KHR-pRegions-00215**
  The source region specified by each element of *pRegions* must be a region that is contained within *srcImage*
- **VUID-VkBlitImageInfo2KHR-pRegions-00216**
  The destination region specified by each element of *pRegions* must be a region that is contained within *dstImage*
- **VUID-VkBlitImageInfo2KHR-pRegions-00217**
  The union of all destination regions, specified by the elements of *pRegions*, must not overlap in memory with any texel that may be sampled during the blit operation
- **VUID-VkBlitImageInfo2KHR-srcImage-01999**
  The format features of *srcImage* must contain `VK_FORMAT_FEATURE_BLIT_SRC_BIT`
- **VUID-VkBlitImageInfo2KHR-srcImage-06421**
  *srcImage* must not use a format that requires a sampler Y’C_bC_r conversion
- **VUID-VkBlitImageInfo2KHR-srcImage-00219**
  *srcImage* must have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage flag
• VUID-VkBlitImageInfo2KHR-srcImage-00220
  If srcImage is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

• VUID-VkBlitImageInfo2KHR-srcImageLayout-00221
  srcImageLayout must specify the layout of the image subresources of srcImage specified in pRegions at the time this command is executed on a VkDevice

• VUID-VkBlitImageInfo2KHR-srcImageLayout-01398
  srcImageLayout must be VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL

• VUID-VkBlitImageInfo2KHR-dstImage-02000
  The format features of dstImage must contain VK_FORMAT_FEATURE_BLIT_DST_BIT

• VUID-VkBlitImageInfo2KHR-dstImage-06422
  dstImage must not use a format that requires a sampler Y'CbCr conversion

• VUID-VkBlitImageInfo2KHR-dstImage-00224
  dstImage must have been created with VK_IMAGE_USAGE_TRANSFER_DST_BIT usage flag

• VUID-VkBlitImageInfo2KHR-dstImage-00225
  If dstImage is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

• VUID-VkBlitImageInfo2KHR-dstImageLayout-00226
  dstImageLayout must specify the layout of the image subresources of dstImage specified in pRegions at the time this command is executed on a VkDevice

• VUID-VkBlitImageInfo2KHR-dstImageLayout-01399
  dstImageLayout must be VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL

• VUID-VkBlitImageInfo2KHR-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-VkBlitImageInfo2KHR-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-VkBlitImageInfo2KHR-srcImage-00231
  If either of srcImage or dstImage was created with a depth/stencil format, the other must have exactly the same format

• VUID-VkBlitImageInfo2KHR-srcImage-00232
  If srcImage was created with a depth/stencil format, filter must be VK_FILTER_NEAREST

• VUID-VkBlitImageInfo2KHR-dstImage-00234
  dstImage must have been created with a samples value of VK_SAMPLE_COUNT_1_BIT

• VUID-VkBlitImageInfo2KHR-dstImage-00235
  dstImage must have been created with a samples value of VK_SAMPLE_COUNT_1_BIT

• VUID-VkBlitImageInfo2KHR-filter-02001
  If filter is VK_FILTER_LINEAR, then the format features of srcImage must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT
- VUID-VkBlitImageInfo2KHR-filter-02002
  If `filter` is `VK_FILTER_CUBIC_EXT`, then the format features of `srcImage` must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`.

- VUID-VkBlitImageInfo2KHR-filter-00237
  If `filter` is `VK_FILTER_CUBIC_EXT`, `srcImage` must be of type `VK_IMAGE_TYPE_2D`.

- VUID-VkBlitImageInfo2KHR-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-VkBlitImageInfo2KHR-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-VkBlitImageInfo2KHR-srcSubresource-01707
  The `srcSubresource.baseArrayLayer + srcSubresource.layerCount` of each element of `pRegions` must be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `srcImage` was created.

- VUID-VkBlitImageInfo2KHR-dstSubresource-01708
  The `dstSubresource.baseArrayLayer + dstSubresource.layerCount` of each element of `pRegions` must be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `dstImage` was created.

- VUID-VkBlitImageInfo2KHR-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-VkBlitImageInfo2KHR-aspectMask-00241
  For each element of `pRegions`, `srcSubresource.aspectMask` must specify aspects present in `srcImage`.

- VUID-VkBlitImageInfo2KHR-aspectMask-00242
  For each element of `pRegions`, `dstSubresource.aspectMask` must specify aspects present in `dstImage`.

- VUID-VkBlitImageInfo2KHR-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-VkBlitImageInfo2KHR-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-VkBlitImageInfo2KHR-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-VkBlitImageInfo2KHR-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-VkBlitImageInfo2KHR-srcImage-00247
  If `srcImage` is of type `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `srcOffsets[0].y` must be `0` and `srcOffsets[1].y` must be `1`.

- VUID-VkBlitImageInfo2KHR-srcOffset-00248
  For each element of `pRegions`, `srcOffsets[0].z` and `srcOffsets[1].z` must both be greater than or equal to `0` and less than or equal to the depth of the specified `srcSubresource` of `srcImage`.
If `srcImage` is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `srcOffsets[0].z` must be 0 and `srcOffsets[1].z` must be 1.

For each element of `pRegions`, `dstOffsets[0].x` and `dstOffsets[1].x` must both be greater than or equal to 0 and less than or equal to the width of the specified `dstSubresource` of `dstImage`.

For each element of `pRegions`, `dstOffsets[0].y` and `dstOffsets[1].y` must both be greater than or equal to 0 and less than or equal to the height of the specified `dstSubresource` of `dstImage`.

If `dstImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `dstOffsets[0].y` must be 0 and `dstOffsets[1].y` must be 1.

For each element of `pRegions`, `dstOffsets[0].z` and `dstOffsets[1].z` must both be greater than or equal to 0 and less than or equal to the depth of the specified `dstSubresource` of `dstImage`.

If `dstImage` is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `dstOffsets[0].z` must be 0 and `dstOffsets[1].z` must be 1.

**Valid Usage (Implicit)**

- `sType` must be `VK_STRUCTURE_TYPE_BLIT_IMAGE_INFO_2_KHR`
- `pNext` must be `NULL`
- `srcImage` must be a valid `VkImage` handle
- `srcImageLayout` must be a valid `VkImageLayout` value
- `dstImage` must be a valid `VkImage` handle
- `dstImageLayout` must be a valid `VkImageLayout` value
- `pRegions` must be a valid pointer to an array of `regionCount` valid `VkImageBlit2KHR` structures
- `filter` must be a valid `VkFilter` value
The `VkImageBlit2KHR` structure is defined as:

```c
// Provided by VK_KHR_copy_commands2
typedef struct VkImageBlit2KHR {
    VkStructureType sType;
    const void* pNext;
    VkImageSubresourceLayers srcSubresource;
    VkOffset3D srcOffsets[2];
    VkImageSubresourceLayers dstSubresource;
    VkOffset3D dstOffsets[2];
} VkImageBlit2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcSubresource` is the subresource to blit from.
- `srcOffsets` is a pointer to an array of two `VkOffset3D` structures specifying the bounds of the source region within `srcSubresource`.
- `dstSubresource` is the subresource to blit into.
- `dstOffsets` is a pointer to an array of two `VkOffset3D` structures specifying the bounds of the destination region within `dstSubresource`.

For each element of the `pRegions` array, a blit operation is performed for the specified source and destination regions.

### Valid Usage

- VUID-VkImageBlit2KHR-aspectMask-00238
  The `aspectMask` member of `srcSubresource` and `dstSubresource` must match

- VUID-VkImageBlit2KHR-layerCount-00239
  The `layerCount` member of `srcSubresource` and `dstSubresource` must match

### Valid Usage (Implicit)

- VUID-VkImageBlit2KHR-sType-sType
  The `sType` must be `VK_STRUCTURE_TYPE_IMAGE_BLIT_2_KHR`

- VUID-VkImageBlit2KHR-pNext-pNext
  The `pNext` must be `NULL`
19.6. 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 \textit{commandBuffer} is an unprotected command buffer and \textit{protectedNoFault} is not supported, \textit{srcImage} must not be a protected image

- VUID-vkCmdResolveImage-commandBuffer-01838
  If \textit{commandBuffer} is an unprotected command buffer and \textit{protectedNoFault} is not supported, \textit{dstImage} must not be a protected image

- VUID-vkCmdResolveImage-commandBuffer-01839
  If \textit{commandBuffer} is a protected command buffer and \textit{protectedNoFault} is not supported, \textit{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 \textit{pRegions}, must not overlap in memory

- VUID-vkCmdResolveImage-srcImage-00256
  If \textit{srcImage} is non-sparse then it must be bound completely and contiguously to a single \textit{VkDeviceMemory} object

- VUID-vkCmdResolveImage-srcImage-00257
  \textit{srcImage} must have a sample count equal to any valid sample count value other than \textit{VK_SAMPLE_COUNT_1_BIT}

- VUID-vkCmdResolveImage-dstImage-00258
  If \textit{dstImage} is non-sparse then it must be bound completely and contiguously to a single \textit{VkDeviceMemory} object

- VUID-vkCmdResolveImage-dstImage-00259
  \textit{dstImage} must have a sample count equal to \textit{VK_SAMPLE_COUNT_1_BIT}

- VUID-vkCmdResolveImage-srcImageLayout-00260
  \textit{srcImageLayout} must specify the layout of the image subresources of \textit{srcImage} specified in \textit{pRegions} at the time this command is executed on a \textit{VkDevice}

- VUID-vkCmdResolveImage-dstImageLayout-00262
  \textit{dstImageLayout} must specify the layout of the image subresources of \textit{dstImage} specified in \textit{pRegions} at the time this command is executed on a \textit{VkDevice}

- VUID-vkCmdResolveImage-dstImageLayout-01401
  \textit{dstImageLayout} must be \textit{VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR}, \textit{VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL} or \textit{VK_IMAGE_LAYOUT_GENERAL}

- VUID-vkCmdResolveImage-dstImage-02003
  The \textit{format features} of \textit{dstImage} must contain \textit{VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT}

- VUID-vkCmdResolveImage-srcImage-01386
  \textit{srcImage} and \textit{dstImage} must have been created with the same image format

- VUID-vkCmdResolveImage-srcSubresource-01709
  The \textit{srcSubresource.mipLevel} member of each element of \textit{pRegions} must be less than the \textit{mipLevels} specified in \textit{VkImageCreateInfo} when \textit{srcImage} was created
The `dstSubresource.mipLevel` member of each element of `pRegions` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `dstImage` was created.

The `srcSubresource.baseArrayLayer + srcSubresource.layerCount` of each element of `pRegions` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `srcImage` was created.

The `dstSubresource.baseArrayLayer + dstSubresource.layerCount` of each element of `pRegions` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `dstImage` was created.

If either `srcImage` or `dstImage` are of type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions`, `srcSubresource.baseArrayLayer` **must** be 0 and `srcSubresource.layerCount` **must** be 1.

If either `srcImage` or `dstImage` are of type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions`, `dstSubresource.baseArrayLayer` **must** be 0 and `dstSubresource.layerCount` **must** be 1.

For each element of `pRegions`, `srcOffset.x` and `(extent.width + srcOffset.x)` **must** both be greater than or equal to 0 and less than or equal to the width of the specified `srcSubresource` of `srcImage`.

For each element of `pRegions`, `srcOffset.y` and `(extent.height + srcOffset.y)` **must** both be greater than or equal to 0 and less than or equal to the height of the specified `srcSubresource` of `srcImage`.

If `srcImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `srcOffset.y` **must** be 0 and `extent.height` **must** be 1.

For each element of `pRegions`, `srcOffset.z` and `(extent.depth + srcOffset.z)` **must** both be greater than or equal to 0 and less than or equal to the depth of the specified `srcSubresource` of `srcImage`.

If `srcImage` is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `srcOffset.z` **must** be 0 and `extent.depth` **must** be 1.

For each element of `pRegions`, `dstOffset.x` and `(extent.width + dstOffset.x)` **must** both be greater than or equal to 0 and less than or equal to the width of the specified `dstSubresource` of `dstImage`.

For each element of `pRegions`, `dstOffset.y` and `(extent.height + dstOffset.y)` **must** both be
greater than or equal to 0 and less than or equal to the height of the specified dstSubresource of dstImage

• VUID-vkCmdResolveImage-dstImage-00276
  If dstImage is of type VK_IMAGE_TYPE_1D, then for each element of pRegions, dstOffset.y must be 0 and extent.height must be 1

• VUID-vkCmdResolveImage-dstOffset-00277
  For each element of pRegions, dstOffset.z and (extent.depth + dstOffset.z) must both be greater than or equal to 0 and less than or equal to the depth of the specified dstSubresource of dstImage

• VUID-vkCmdResolveImage-dstImage-00278
  If dstImage is of type VK_IMAGE_TYPE_1D or VK_IMAGE_TYPE_2D, then for each element of pRegions, dstOffset.z must be 0 and extent.depth must be 1

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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkImageResolve` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkImageResolve {
    VkImageSubresourceLayers srcSubresource;
    VkOffset3D srcOffset;
    VkImageSubresourceLayers dstSubresource;
    VkOffset3D dstOffset;
    VkExtent3D extent;
} VkImageResolve;
```

- `srcSubresource` and `dstSubresource` are `VkImageSubresourceLayers` structures specifying the image subresources of the images used for the source and destination image data, respectively. Resolve of depth/stencil images is not supported.
- `srcOffset` and `dstOffset` select the initial x, y, and z offsets in texels of the sub-regions of the source and destination image data.
- `extent` is the size in texels of the source image to resolve in width, height and depth.

Valid Usage

- VUID-VkImageResolve-aspectMask-00266
  The `aspectMask` member of `srcSubresource` and `dstSubresource` must only contain `VK_IMAGE_ASPECT_COLOR_BIT`
- VUID-VkImageResolve-layerCount-00267
  The `layerCount` member of `srcSubresource` and `dstSubresource` must match

Valid Usage (Implicit)

- VUID-VkImageResolve-srcSubresource-parameter
A more extensible version of the resolve image command is defined below.

To resolve a multisample image to a non-multisample image, call:

```c
// Provided by VK_KHR_copy_commands2
donot partnered void vkCmdResolveImage2KHR(
    VkCommandBuffer commandBuffer,
    const VkResolveImageInfo2KHR* pResolveImageInfo);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pResolveImageInfo` is a pointer to a `VkResolveImageInfo2KHR` structure describing the resolve parameters.

This command is functionally identical to `vkCmdResolveImage`, but includes extensible substructures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

### Valid Usage

- VUID-vkCmdResolveImage2KHR-commandBuffer-01837
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `srcImage` must not be a protected image
- VUID-vkCmdResolveImage2KHR-commandBuffer-01838
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, `dstImage` must not be a protected image
- VUID-vkCmdResolveImage2KHR-commandBuffer-01839
  If `commandBuffer` is a protected command buffer and `protectedNoFault` is not supported, `dstImage` must not be an unprotected image

### Valid Usage (Implicit)

- VUID-vkCmdResolveImage2KHR-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdResolveImage2KHR-pResolveImageInfo-parameter
  `pResolveImageInfo` must be a valid pointer to a valid `VkResolveImageInfo2KHR` structure
- VUID-vkCmdResolveImage2KHR-commandBuffer-recording
  `commandBuffer` must be in the `recording` state
- VUID-vkCmdResolveImage2KHR-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations
• VUID-vkCmdResolveImage2KHR-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkResolveImageInfo2KHR` structure is defined as:

```c
// Provided by VK_KHR_copy_commands2
typedef struct VkResolveImageInfo2KHR {
    VkStructureType sType;
    const void* pNext;
    VkImage srcImage;
    VkImageLayout srcImageLayout;
    VkImage dstImage;
    VkImageLayout dstImageLayout;
    uint32_t regionCount;
    const VkImageResolve2KHR* pRegions;
} VkResolveImageInfo2KHR;
```

• `sType` is the type of this structure.
• `pNext` is **NULL** or a pointer to a structure extending this structure.
• `srcImage` is the source image.
• `srcImageLayout` is the layout of the source image subresources for the resolve.
• `dstImage` is the destination image.
• `dstImageLayout` is the layout of the destination image subresources for the resolve.
• `regionCount` is the number of regions to resolve.
• `pRegions` is a pointer to an array of `VkImageResolve2KHR` structures specifying the regions to resolve.
Valid Usage

- **VUID-VkResolveImageInfo2KHR-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-VkResolveImageInfo2KHR-srcImage-00256**
  If `srcImage` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-VkResolveImageInfo2KHR-srcImage-00257**
  `srcImage` must have a sample count equal to any valid sample count value other than `VK_SAMPLE_COUNT_1_BIT`

- **VUID-VkResolveImageInfo2KHR-dstImage-00258**
  If `dstImage` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-VkResolveImageInfo2KHR-dstImage-00259**
  `dstImage` must have a sample count equal to `VK_SAMPLE_COUNT_1_BIT`

- **VUID-VkResolveImageInfo2KHR-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-VkResolveImageInfo2KHR-srcImageLayout-01400**
  `srcImageLayout` must be `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`

- **VUID-VkResolveImageInfo2KHR-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-VkResolveImageInfo2KHR-dstImageLayout-01401**
  `dstImageLayout` must be `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`

- **VUID-VkResolveImageInfo2KHR-dstImage-02003**
  The format features of `dstImage` must contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`

- **VUID-VkResolveImageInfo2KHR-srcImage-01386**
  `srcImage` and `dstImage` must have been created with the same image format

- **VUID-VkResolveImageInfo2KHR-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-VkResolveImageInfo2KHR-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-VkResolveImageInfo2KHR-srcSubresource-01711**
  The `srcSubresource.baseArrayLayer + srcSubresource.layerCount` of each element of `pRegions` must be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `srcImage` was created
• VUID-VkResolveImageInfo2KHR-dstSubresource-01712
The dstSubresource.baseArrayLayer + dstSubresource.layerCount of each element of pRegions must be less than or equal to the arrayLayers specified in VkImageCreateInfo when dstImage was created

• VUID-VkResolveImageInfo2KHR-srcImage-04446
If either srcImage or dstImage are of type VK_IMAGE_TYPE_3D, then for each element of pRegions, srcSubresource.baseArrayLayer must be 0 and srcSubresource.layerCount must be 1

• VUID-VkResolveImageInfo2KHR-srcImage-04447
If either srcImage or dstImage are of type VK_IMAGE_TYPE_3D, then for each element of pRegions, dstSubresource.baseArrayLayer must be 0 and dstSubresource.layerCount must be 1

• VUID-VkResolveImageInfo2KHR-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-VkResolveImageInfo2KHR-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-VkResolveImageInfo2KHR-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-VkResolveImageInfo2KHR-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-VkResolveImageInfo2KHR-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-VkResolveImageInfo2KHR-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-VkResolveImageInfo2KHR-dstOffset-00275
For each element of pRegions, dstOffset.y and (extent.height + dstOffset.y) must both be greater than or equal to 0 and less than or equal to the height of the specified dstSubresource of dstImage

• VUID-VkResolveImageInfo2KHR-dstImage-00276
If dstImage is of type VK_IMAGE_TYPE_1D, then for each element of pRegions, dstOffset.y must be 0 and extent.height must be 1

• VUID-VkResolveImageInfo2KHR-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-VkResolveImageInfo2KHR-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

Valid Usage (Implicit)

• VUID-VkResolveImageInfo2KHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_RESOLVE_IMAGE_INFO_2_KHR

• VUID-VkResolveImageInfo2KHR-pNext-pNext
  pNext must be NULL

• VUID-VkResolveImageInfo2KHR-srcImage-parameter
  srcImage must be a valid VkImage handle

• VUID-VkResolveImageInfo2KHR-srcImageLayout-parameter
  srcImageLayout must be a valid VkImageLayout value

• VUID-VkResolveImageInfo2KHR-dstImage-parameter
  dstImage must be a valid VkImage handle

• VUID-VkResolveImageInfo2KHR-dstImageLayout-parameter
  dstImageLayout must be a valid VkImageLayout value

• VUID-VkResolveImageInfo2KHR-pRegions-parameter
  pRegions must be a valid pointer to an array of regionCount valid VkImageResolve2KHR structures

• VUID-VkResolveImageInfo2KHR-regionCount-arraylength
  regionCount must be greater than 0

• VUID-VkResolveImageInfo2KHR-commonparent
  Both of dstImage, and srcImage must have been created, allocated, or retrieved from the same VkDevice

The VkImageResolve2KHR structure is defined as:

```c
// Provided by VK_KHR_copy_commands2
typedef struct VkImageResolve2KHR {
    VkStructureType sType;
    const void* pNext;
    VkImageSubresourceLayers srcSubresource;
    VkOffset3D srcOffset;
    VkImageSubresourceLayers dstSubresource;
    VkOffset3D dstOffset;
    VkExtent3D extent;
} VkImageResolve2KHR;
```
• **sType** is the type of this structure.

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

• **srcSubresource** and **dstSubresource** are **VkImageSubresourceLayers** structures specifying the image subresources of the images used for the source and destination image data, respectively. Resolve of depth/stencil images is not supported.

• **srcOffset** and **dstOffset** select the initial x, y, and z offsets in texels of the sub-regions of the source and destination image data.

• **extent** is the size in texels of the source image to resolve in **width**, **height** and **depth**.

### Valid Usage

- **VUID-VkImageResolve2KHR-aspectMask-00266**
  The aspectMask member of **srcSubresource** and **dstSubresource** **must** only contain **VK_IMAGE_ASPECT_COLOR_BIT**

- **VUID-VkImageResolve2KHR-layerCount-00267**
  The layerCount member of **srcSubresource** and **dstSubresource** **must** match

### Valid Usage (Implicit)

- **VUID-VkImageResolve2KHR-sType-sType**
  sType **must** be **VK_STRUCTURE_TYPE_IMAGE_RESOLVE_2_KHR**

- **VUID-VkImageResolve2KHR-pNext-pNext**
  pNext **must** be **NULL**

- **VUID-VkImageResolve2KHR-srcSubresource-parameter**
  srcSubresource **must** be a valid **VkImageSubresourceLayers** structure

- **VUID-VkImageResolve2KHR-dstSubresource-parameter**
  dstSubresource **must** be a valid **VkImageSubresourceLayers** structure

## 19.7. Object Refreshes

Safety critical applications **may** need to contend with single event upsets (SEUs). For a Vulkan object explicitly backed by device memory, such as a **VkImage** or **VkBuffer**, an application **can** bind its backing memory to a SEU-safe heap with the **VK_MEMORY_HEAP_SEU_SAFE_BIT** bit set. Alternatively, an application **can** also periodically reload the non-SEU-safe device memory contents from a known SEU-safe portion of host memory, or otherwise periodically regenerate or refresh the contents of non-SEU-safe device memory.

However, an implementation **may** store implementation-specific internal object data in non-SEU-safe memory, and Base Vulkan provides no method to determine which object types this applies to or how to refresh their data. An application **can** query the list of object types that have implementation internal object data stored in non-SEU-safe memory using **vkGetPhysicalDeviceRefreshableObjectTypesKHR**, and **can** instruct the implementation to refresh
the internal data of specific objects from a backup in SEU-safe memory using the `vkCmdRefreshObjectsKHR` command.

To refresh a list of objects as a pipelined operation, call:

```c
// Provided by VK_KHR_object_refresh
void vkCmdRefreshObjectsKHR(
    VkCommandBuffer commandBuffer,
    const VkRefreshObjectListKHR* pRefreshObjects);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pRefreshObjects` is a pointer to a `VkRefreshObjectListKHR` structure specifying the list of objects to refresh.

The access scope for object refreshes falls under the `VK_ACCESS_TRANSFER_WRITE_BIT`, and the pipeline stages for identifying the synchronization scope must include `VK_PIPELINE_STAGE_TRANSFER_BIT`.

**Note**
If an implementation does not store a supplied object's internal data in SEU-susceptible memory, it may ignore the refresh command for that object.

### Valid Usage (Implicit)

- VUID-vkCmdRefreshObjectsKHR-commandBuffer-parameter
  - `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdRefreshObjectsKHR-pRefreshObjects-parameter
  - `pRefreshObjects` must be a valid pointer to a valid `VkRefreshObjectListKHR` structure
- VUID-vkCmdRefreshObjectsKHR-commandBuffer-recording
  - `commandBuffer` must be in the recording state
- VUID-vkCmdRefreshObjectsKHR-commandBuffer-cmdpool
  - The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, compute, or transfer operations
- VUID-vkCmdRefreshObjectsKHR-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
The `VkRefreshObjectListKHR` structure is defined as:

```c
// Provided by VK_KHR_object_refresh
typedef struct VkRefreshObjectListKHR {
    VkStructureType sType;
    const void* pNext;
    uint32_t objectCount;
    const VkRefreshObjectKHR* pObjects;
} VkRefreshObjectListKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `objectCount` is the number of objects to refresh.
- `pObjects` is a pointer to an array of `VkRefreshObjectKHR` structures, defining the objects to refresh.

**Valid Usage (Implicit)**

- VUID-VkRefreshObjectListKHR-sType-sType `sType` must be `VK_STRUCTURE_TYPE_REFRESH_OBJECT_LIST_KHR`
- VUID-VkRefreshObjectListKHR-pNext-pNext `pNext` must be `NULL`
- VUID-VkRefreshObjectListKHR-pObjects-parameter `pObjects` must be a valid pointer to an array of `objectCount` valid `VkRefreshObjectKHR` structures
- VUID-VkRefreshObjectListKHR-objectCount-arraylength `objectCount` must be greater than `0`

The `VkRefreshObjectKHR` structure is defined as:

```c
// Provided by VK_KHR_object_refresh
typedef struct VkRefreshObjectKHR {
    VkObjectType objectType;
    uint64_t objectHandle;
    VkRefreshObjectFlagsKHR flags;
} VkRefreshObjectKHR;
```
• **objectType** is a `VkObjectType` specifying the type of the object to refresh.

• **objectHandle** is the object to refresh.

• **flags** is a bitmask of `VkRefreshObjectFlagsKHR`.

### Valid Usage

- **VUID-VkRefreshObjectKHR-objectHandle-05069**
  
  `objectHandle must` be a valid Vulkan handle of the type associated with `objectType` as defined in the `VkObjectType and Vulkan Handle Relationship` table.

- **VUID-VkRefreshObjectKHR-objectType-05070**
  
  `objectType must` not be `VK_OBJECT_TYPE_UNKNOWN`.

### Valid Usage (Implicit)

- **VUID-VkRefreshObjectKHR-objectType-parameter**
  
  `objectType must` be a valid `VkObjectType` value.

- **VUID-VkRefreshObjectKHR-flags-zerobitmask**
  
  `flags must` be 0.

### Host Synchronization

- Host access to `objectHandle` must be externally synchronized.

---

```c
// Provided by VK_KHR_object_refresh
typedef enum VkRefreshObjectFlagBitsKHR {
} VkRefreshObjectFlagBitsKHR;
```

```c
// Provided by VK_KHR_object_refresh
typedef VkFlags VkRefreshObjectFlagsKHR;
```

`VkRefreshObjectFlagsKHR` is a bitmask type for setting a mask, but is currently reserved for future use.
Chapter 20. Drawing Commands

*Drawing commands* (commands with *Draw* in the name) provoke work in a graphics pipeline. Drawing commands are recorded into a command buffer and when executed by a queue, will produce work which executes according to the bound graphics pipeline. A graphics pipeline must be bound to a command buffer before any drawing commands are recorded in that command buffer.

Each draw is made up of zero or more vertices and zero or more instances, which are processed by the device and result in the assembly of primitives. Primitives are assembled according to the *pInputAssemblyState* member of the `VkGraphicsPipelineCreateInfo` structure, which is of type `VkPipelineInputAssemblyStateCreateInfo`:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineInputAssemblyStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineInputAssemblyStateCreateFlags flags;
    VkPrimitiveTopology topology;
    VkBool32 primitiveRestartEnable;
} VkPipelineInputAssemblyStateCreateInfo;
```

- *sType* is the type of this structure.
- *pNext* is NULL or a pointer to a structure extending this structure.
- *flags* is reserved for future use.
- *topology* is a `VkPrimitiveTopology` defining the primitive topology, as described below.
- *primitiveRestartEnable* controls whether a special vertex index value is treated as restarting the assembly of primitives. This enable only applies to indexed draws (*vkCmdDrawIndexed*, and *vkCmdDrawIndexedIndirect*), and the special index value is either 0xFFFFFFFF when the *indexType* parameter of *vkCmdBindIndexBuffer* is equal to `VK_INDEX_TYPE_UINT32`, 0xFF when *indexType* is equal to `VK_INDEX_TYPE_UINT8_EXT`, or 0xFFFF when *indexType* is equal to `VK_INDEX_TYPE_UINT16`. Primitive restart is not allowed for “list” topologies.

Restarting the assembly of primitives discards the most recent index values if those elements formed an incomplete primitive, and restarts the primitive assembly using the subsequent indices, but only assembling the immediately following element through the end of the originally specified elements. The primitive restart index value comparison is performed before adding the *vertexOffset* value to the index value.

**Valid Usage**

- **VUID-VkPipelineInputAssemblyStateCreateInfo-topology-00428**
  If *topology* is `VK_PRIMITIVE_TOPOLOGY_POINT_LIST`, `VK_PRIMITIVE_TOPOLOGY_LINE_LIST`, `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`, `VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY`, `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY` or `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`,
primitiveRestartEnable must be VK_FALSE

- VUID-VkPipelineInputAssemblyStateCreateInfo-topology-00429
  If the geometry shaders feature is not enabled, topology must not be any of VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY, VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY or VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY

- VUID-VkPipelineInputAssemblyStateCreateInfo-topology-00430
  If the tessellation shaders feature is not enabled, topology must not be VK_PRIMITIVE_TOPOLOGY_PATCH_LIST

Valid Usage (Implicit)

- VUID-VkPipelineInputAssemblyStateCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO

- VUID-VkPipelineInputAssemblyStateCreateInfo-pNext-pNext
  pNext must be NULL

- VUID-VkPipelineInputAssemblyStateCreateInfo-flags-zerobitmask
  flags must be 0

- VUID-VkPipelineInputAssemblyStateCreateInfo-topology-parameter
  topology must be a valid VkPrimitiveTopology value

// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineInputAssemblyStateCreateFlags;

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

To dynamically control whether a special vertex index value is treated as restarting the assembly of primitives, call:

// Provided by VK_EXT_extended_dynamic_state2
void vkCmdSetPrimitiveRestartEnableEXT(
  VkCommandBuffer commandBuffer,
  VkBool32 primitiveRestartEnable);

- commandBuffer is the command buffer into which the command will be recorded.
- primitiveRestartEnable controls whether a special vertex index value is treated as restarting the assembly of primitives. It behaves in the same way as VkPipelineInputAssemblyStateCreateInfo::primitiveRestartEnable

This command sets the primitive restart enable for subsequent drawing commands when the
graphics pipeline is created with `VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineInputAssemblyStateCreateInfo::primitiveRestartEnable` value used to create the currently active pipeline.

### Valid Usage

- VUID-vkCmdSetPrimitiveRestartEnableEXT-None-04866
  The `extendedDynamicState2` feature must be enabled

### Valid Usage (Implicit)

- VUID-vkCmdSetPrimitiveRestartEnableEXT-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdSetPrimitiveRestartEnableEXT-commandBuffer-recording
  `commandBuffer` must be in the recording state
- VUID-vkCmdSetPrimitiveRestartEnableEXT-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## 20.1. Primitive Topologies

*Primitive topology* determines how consecutive vertices are organized into primitives, and determines the type of primitive that is used at the beginning of the graphics pipeline. The effective topology for later stages of the pipeline is altered by tessellation or geometry shading (if either is in use) and depends on the execution modes of those shaders.

The primitive topologies defined by `VkPrimitiveTopology` are:
typedef enum VkPrimitiveTopology {
  VK_PRIMITIVE_TOPOLOGY_POINT_LIST = 0,
  VK_PRIMITIVE_TOPOLOGY_LINE_LIST = 1,
  VK_PRIMITIVE_TOPOLOGY_LINE_STRIP = 2,
  VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST = 3,
  VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP = 4,
  VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN = 5,
  VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY = 6,
  VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY = 7,
  VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY = 8,
  VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY = 9,
  VK_PRIMITIVE_TOPOLOGY_PATCH_LIST = 10,
} VkPrimitiveTopology;

- **VK_PRIMITIVE_TOPOLOGY_POINT_LIST** specifies a series of separate point primitives.
- **VK_PRIMITIVE_TOPOLOGY_LINE_LIST** specifies a series of separate line primitives.
- **VK_PRIMITIVE_TOPOLOGY_LINE_STRIP** specifies a series of connected line primitives with consecutive lines sharing a vertex.
- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST** specifies a series of separate triangle primitives.
- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP** specifies a series of connected triangle primitives with consecutive triangles sharing an edge.
- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN** specifies a series of connected triangle primitives with all triangles sharing a common vertex.
- **VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY** specifies a series of separate line primitives with adjacency.
- **VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY** specifies a series of connected line primitives with adjacency, with consecutive primitives sharing three vertices.
- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY** specifies a series of separate triangle primitives with adjacency.
- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY** specifies connected triangle primitives with adjacency, with consecutive triangles sharing an edge.
- **VK_PRIMITIVE_TOPOLOGY_PATCH_LIST** specifies separate patch primitives.

Each primitive topology, and its construction from a list of vertices, is described in detail below with a supporting diagram, according to the following key:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>⚫️</td>
<td>Vertex</td>
</tr>
<tr>
<td>⚫️ 5</td>
<td>Vertex Number</td>
</tr>
<tr>
<td>❗️ 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><strong>Primitive Edge</strong></td>
<td>An edge connecting the points of a main primitive.</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td><strong>Adjacency Edge</strong></td>
<td>Points connected by these lines do not contribute to a main primitive, and are only accessible in a <em>geometry shader</em>.</td>
</tr>
<tr>
<td><strong>Winding Order</strong></td>
<td>The relative order in which vertices are defined within a primitive, used in the <em>facing determination</em>. This ordering has no specific start or endpoint.</td>
</tr>
</tbody>
</table>

The diagrams are supported with mathematical definitions where the vertices \( v \) and primitives \( p \) are numbered starting from 0; \( v_0 \) is the first vertex in the provided data and \( p_0 \) is the first primitive in the set of primitives defined by the vertices and topology.

To *dynamically set* primitive topology, call:

```c
// Provided by VK_EXT_extended_dynamic_state
void vkCmdSetPrimitiveTopologyEXT(
    VkCommandBuffer commandBuffer,
    VkPrimitiveTopology primitiveTopology);
```

- *commandBuffer* is the command buffer into which the command will be recorded.
- *primitiveTopology* specifies the primitive topology to use for drawing.

This command sets the primitive topology for subsequent drawing commands when the graphics pipeline is created with *VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT* set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineInputAssemblyStateCreateInfo::topology` value used to create the currently active pipeline.

### Valid Usage

- VUID-vkCmdSetPrimitiveTopologyEXT-None-03347
  The *extendedDynamicState* feature *must* be enabled

### Valid Usage (Implicit)

- VUID-vkCmdSetPrimitiveTopologyEXT-commandBuffer-parameter
  *commandBuffer* *must* be a valid *VkCommandBuffer* handle

- VUID-vkCmdSetPrimitiveTopologyEXT-primitiveTopology-parameter
  *primitiveTopology* *must* be a valid *VkPrimitiveTopology* value

- VUID-vkCmdSetPrimitiveTopologyEXT-commandBuffer-recording
  *commandBuffer* *must* be in the *recording state*

- VUID-vkCmdSetPrimitiveTopologyEXT-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

20.1.1. Topology Class

The primitive topologies are grouped into the following topology classes:

Table 24. Topology classes

<table>
<thead>
<tr>
<th>Topology Class</th>
<th>Primitive Topology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point</td>
<td>VK_PRIMITIVE_TOPOLOGY_POINT_LIST</td>
</tr>
<tr>
<td>Line</td>
<td>VK_PRIMITIVE_TOPOLOGY_LINE_LIST,</td>
</tr>
<tr>
<td></td>
<td>VK_PRIMITIVE_TOPOLOGY_LINE_STRIP,</td>
</tr>
<tr>
<td></td>
<td>VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY,</td>
</tr>
<tr>
<td></td>
<td>VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY</td>
</tr>
<tr>
<td>Triangle</td>
<td>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST,</td>
</tr>
<tr>
<td></td>
<td>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP,</td>
</tr>
<tr>
<td></td>
<td>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN,</td>
</tr>
<tr>
<td></td>
<td>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY,</td>
</tr>
<tr>
<td></td>
<td>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY</td>
</tr>
<tr>
<td>Patch</td>
<td>VK_PRIMITIVE_TOPOLOGY_PATCH_LIST</td>
</tr>
</tbody>
</table>

20.1.2. Point Lists

When the topology is `VK_PRIMITIVE_TOPOLOGY_POINT_LIST`, each consecutive vertex defines a single point primitive, according to the equation:

\[ p_i = \{v_i\} \]

As there is only one vertex, that vertex is the provoking vertex. The number of primitives generated is equal to `vertexCount`. 
20.1.3. Line Lists

When the primitive topology is `VK_PRIMITIVE_TOPOLOGY_LINE_LIST`, each consecutive pair of vertices defines a single line primitive, according to the equation:

\[ p_i = \{v_{2i}, v_{2i+1}\} \]

The number of primitives generated is equal to \( \lfloor \text{vertexCount}/2 \rfloor \).

The provoking vertex for \( p_i \) is \( v_{2i} \).

20.1.4. Line Strips

When the primitive topology is `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP`, one line primitive is defined by each vertex and the following vertex, according to the equation:

\[ p_i = \{v_i, v_{i+1}\} \]

The number of primitives generated is equal to \( \max(0, \text{vertexCount}-1) \).

The provoking vertex for \( p_i \) is \( v_i \).

20.1.5. Triangle Lists

When the primitive topology is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`, each consecutive set of three vertices defines a single triangle primitive, according to the equation:
\( p_i = \{v_{3i}, v_{3i+1}, v_{3i+2}\} \)

The number of primitives generated is equal to \( \lfloor \text{vertexCount}/3 \rfloor \).

The provoking vertex for \( p_i \) is \( v_{3i} \).

### 20.1.6. Triangle Strips

When the primitive topology is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP`, one triangle primitive is defined by each vertex and the two vertices that follow it, according to the equation:

\[ p_i = \{v_i, v_{i+(1-i^2/2)}, v_{i+(2-i^2/2)}\} \]

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.

### 20.1.7. Triangle Fans

When the primitive topology is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN`, triangle primitives are defined around a shared common vertex, according to the equation:

\[ p_i = \{v_{i+1}, v_{i+2}, v_0\} \]

The number of primitives generated is equal to \( \max(0, \text{vertexCount}-2) \).

The provoking vertex for \( p_i \) is \( v_{i+1} \).
20.1.8. Line Lists With Adjacency

When the primitive topology is `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 \( v_{4i+1} \).

20.1.9. 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 \( \max(0, \text{vertexCount}-3) \).

The provoking vertex for \( p_i \) is \( v_{i+1} \).
20.1.10. Triangle Lists With Adjacency

When the primitive topology is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY`, each consecutive set of six vertices defines a single triangle primitive with adjacency, according to the equations:

\[ p_i = \{v_{6i}, v_{6i+1}, v_{6i+2}, v_{6i+3}, v_{6i+4}, v_{6i+5}\} \]

A triangle primitive is described by the first, third, and fifth vertices of the total primitive, with the remaining three vertices only accessible in a geometry shader.

The number of primitives generated is equal to \( \lfloor \frac{\text{vertexCount}}{6} \rfloor \).

The provoking vertex for \( p_i \) is \( v_{6i} \).

20.1.11. Triangle Strips With Adjacency

When the primitive topology is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY`, one triangle primitive with adjacency is defined by each vertex and the following 5 vertices.

The number of primitives generated, \( n \), is equal to \( \lfloor \max(0, \text{vertexCount} - 4)/2 \rfloor \).

If \( n=1 \), the primitive is defined as:

\[ p = \{v_0, v_1, v_2, v_5, v_4, v_3\} \]

If \( n>1 \), the total primitive consists of different vertices according to where it is in the strip:

\[ p_i = \{v_{2i}, v_{2i+1}, v_{2i+2}, v_{2i+3}, v_{2i+4}, v_{2i+5}\} \text{ when } i=0 \]

\[ p_i = \{v_{2i}, v_{2i+3}, v_{2i+4}, v_{2i+5}, v_{2i+2}, v_{2i+1}\} \text{ when } i>0, i<n-1, \text{ and } i\%2=1 \]

\[ p_i = \{v_{2i}, v_{2i+2}, v_{2i+3}, v_{2i+4}, v_{2i+5}\} \text{ when } i>0, i<n-1, \text{ and } i\%2=0 \]
\[ p_i = \{v_{2i}, v_{2i+3}, v_{2i+4}, v_{2i+5}, v_{2i+2}, v_{2i+1}\} \text{ when } i=n-1 \text{ and } i\%2=1 \]

\[ p_i = \{v_{2i}, v_{2i+2}, v_{2i+3}, v_{2i+4}, v_{2i+5}\} \text{ when } i=n-1 \text{ and } i\%2=0 \]

A triangle primitive is described by the first, third, and fifth vertices of the total primitive in all cases, with the remaining three vertices only accessible in a geometry shader.

**Note**

The ordering of the vertices in each successive triangle is altered so that the winding order is consistent throughout the strip.

The provoking vertex for \( p_i \) is always \( v_{2i} \).

---

**20.1.12. Patch Lists**

When the primitive topology is `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`, each consecutive set of \( m \) vertices defines a single patch primitive, according to the equation:

\[ p_i = \{v_{mi}, v_{mi+1}, \ldots, v_{mi+(m-2)}, v_{mi+(m-1)}\} \]
where \( m \) is equal to \( \text{VkPipelineTessellationStateCreateInfo::patchControlPoints} \).

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 \( \text{VkDrawIndirectCommand} \) 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 \( \text{vertexIndex} \) to a higher numbered \( \text{vertexIndex} \).
   - 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 \( \text{VK_POLYGON_MODE_FILL} \), by an implementation-dependent ordering of the new primitives generated within the original primitive.

Primitive order is later used to define **rasterization order**, which determines the order in which fragments output results to a framebuffer.

### 20.3. Programmable Primitive Shading

Once primitives are assembled, they proceed to the vertex shading stage of the pipeline. If the draw includes multiple instances, then the set of primitives is sent to the vertex shading stage multiple times, once for each instance.

It is implementation-dependent whether vertex shading occurs on vertices that are discarded as part of incomplete primitives, but if it does occur then it operates as if they were vertices in complete primitives and such invocations can have side effects.
Vertex shading receives two per-vertex inputs from the primitive assembly stage - the `vertexIndex` and the `instanceIndex`. How these values are generated is defined below, with each command.

**Drawing commands** fall roughly into two categories:

- **Non-indexed** drawing commands present a sequential `vertexIndex` to the vertex shader. The sequential index is generated automatically by the device (see Fixed-Function Vertex Processing for details on both specifying the vertex attributes indexed by `vertexIndex`, as well as binding vertex buffers containing those attributes to a command buffer). These commands are:
  - `vkCmdDraw`
  - `vkCmdDrawIndirect`
  - `vkCmdDrawIndirectCount`

- **Indexed** drawing commands read index values from an *index buffer* and use this to compute the `vertexIndex` value for the vertex shader. These commands are:
  - `vkCmdDrawIndexed`
  - `vkCmdDrawIndexedIndirect`
  - `vkCmdDrawIndexedIndirectCount`

To bind an index buffer to a command buffer, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdBindIndexBuffer(
    VkCommandBuffer commandBuffer,
    VkBuffer buffer,
    VkDeviceSize offset,
    VkIndexType indexType);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `buffer` is the buffer being bound.
- `offset` is the starting offset in bytes within `buffer` used in index buffer address calculations.
- `indexType` is a `VkIndexType` value specifying the size of the indices.

**Valid Usage**

- VUID-vkCmdBindIndexBuffer-offset-00431
  `offset` must be less than the size of `buffer`

- VUID-vkCmdBindIndexBuffer-offset-00432
  The sum of `offset` and the address of the range of `VkDeviceMemory` object that is backing `buffer`, must be a multiple of the type indicated by `indexType`

- VUID-vkCmdBindIndexBuffer-buffer-00433
  `buffer` must have been created with the `VK_BUFFER_USAGE_INDEX_BUFFER_BIT` flag

- VUID-vkCmdBindIndexBuffer-buffer-00434
If buffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

- VUID-vkCmdBindIndexBuffer-indexType-02765
  If indexType is VK_INDEX_TYPE_UINT8_EXT, the indexTypeUint8 feature must be enabled

### 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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</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,
};
```
• **`VK_INDEX_TYPE_UINT16`** specifies that indices are 16-bit unsigned integer values.
• **`VK_INDEX_TYPE_UINT32`** specifies that indices are 32-bit unsigned integer values.
• **`VK_INDEX_TYPE_UINT8_EXT`** specifies that indices are 8-bit unsigned integer values.

The parameters for each drawing command are specified directly in the command or read from buffer memory, depending on the command. Drawing commands that source their parameters from buffer memory are known as *indirect* drawing commands.

All drawing commands interact with the **Robust Buffer Access** feature.

To record a non-indexed draw, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdDraw(
    VkCommandBuffer commandBuffer,
    uint32_t vertexCount,
    uint32_t instanceCount,
    uint32_t firstVertex,
    uint32_t firstInstance);
```

• **commandBuffer** is the command buffer into which the command is recorded.
• **vertexCount** is the number of vertices to draw.
• **instanceCount** is the number of instances to draw.
• **firstVertex** is the index of the first vertex to draw.
• **firstInstance** is the instance ID of the first instance to draw.

When the command is executed, primitives are assembled using the current primitive topology and **vertexCount** consecutive vertex indices with the first **vertexIndex** value equal to **firstVertex**. The primitives are drawn **instanceCount** times with **instanceIndex** starting with **firstInstance** and increasing sequentially for each instance. The assembled primitives execute the bound graphics pipeline.

### Valid Usage

• VUID-vkCmdDraw-magFilter-04553
  If a **VkSampler** created with **magFilter** or **minFilter** equal to **VK_FILTER_LINEAR** and **compareEnable** equal to **VK_FALSE** is used to sample a **VkImageView** as a result of this command, then the image view's **format features** must contain **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT**

• VUID-vkCmdDraw-mipmapMode-04770
  If a **VkSampler** created with **mipmapMode** equal to **VK_SAMPLER_MIPMAP_MODE_LINEAR** and
**compareEnable** equal to **VK_FALSE** is used to sample a **VkImageView** as a result of this command, then the image view’s **format features** **must** contain **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT**

- **VUID-vkCmdDraw-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-02692**
  If a **VkImageView** is sampled with **VK_FILTER_CUBIC_EXT** as a result of this command, then the image view’s **format features** **must** contain **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT**

- **VUID-vkCmdDraw-filterCubic-02694**
  Any **VkImageView** being sampled with **VK_FILTER_CUBIC_EXT** as a result of this command **must** have a **VkImageViewType** and format that supports cubic filtering, as specified by **VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic** returned by **vkGetPhysicalDeviceImageFormatProperties2**

- **VUID-vkCmdDraw-filterCubicMinmax-02695**
  Any **VkImageView** being sampled with **VK_FILTER_CUBIC_EXT** with a reduction mode of either **VK_SAMPLER_REDUCTION_MODE_MIN** or **VK_SAMPLER_REDUCTION_MODE_MAX** as a result of this command **must** have a **VkImageViewType** and format that supports cubic filtering together with minmax filtering, as specified by **VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax** returned by **vkGetPhysicalDeviceImageFormatProperties2**

- **VUID-vkCmdDraw-None-02697**
  For each set **n** that is statically used by the **VkPipeline** bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to **n** at the same pipeline bind point, with a **VkPipelineLayout** that is compatible for set **n**, with the **VkPipelineLayout** used to create the current **VkPipeline**, as described in **Pipeline Layout Compatibility**

- **VUID-vkCmdDraw-None-02698**
  For each push constant that is statically used by the **VkPipeline** bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a **VkPipelineLayout** that is compatible for push constants, with the **VkPipelineLayout** used to create the current **VkPipeline**, as described in **Pipeline Layout Compatibility**

- **VUID-vkCmdDraw-None-02699**
  Descriptors in each bound descriptor set, specified via **vkCmdBindDescriptorSets**, **must** be valid if they are statically used by the **VkPipeline** bound to the pipeline bind point used by this command

- **VUID-vkCmdDraw-None-02700**
  A valid pipeline **must** be bound to the pipeline bind point used by this command

- **VUID-vkCmdDraw-commandBuffer-02701**
  If the ** VkPipeline** object bound to the pipeline bind point used by this command requires
any dynamic state, that state **must** have been set or inherited (if the
VK_NV_inherited_viewport_scissor extension is enabled) for commandBuffer, and done so
after any previously bound pipeline with the corresponding state not specified as
dynamic

• VUID-vkCmdDraw-None-02859
  There **must** not have been any calls to dynamic state setting commands for any state not
  specified as dynamic in the VkPipeline object bound to the pipeline bind point used by this
  command, since that pipeline was bound

• VUID-vkCmdDraw-None-02702
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a
  VkSampler object that uses unnormalized coordinates, that sampler **must** not be used to
  sample from any VkImage with a VkImageView of the type VK_IMAGE_VIEW_TYPE_3D,
  VK_IMAGE_VIEW_TYPE_CUBE, VK_IMAGE_VIEW_TYPE_1D_ARRAY, VK_IMAGE_VIEW_TYPE_2D_ARRAY or
  VK_IMAGE_VIEW_TYPE_CUBE_ARRAY, in any shader stage

• VUID-vkCmdDraw-None-02703
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a
  VkSampler object that uses unnormalized coordinates, that sampler **must** not be used with
  any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions with ImplicitLod,
  Dref or Proj in their name, in any shader stage

• VUID-vkCmdDraw-None-02704
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a
  VkSampler object that uses unnormalized coordinates, that sampler **must** not be used with
  any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions that includes a LOD
  bias or any offset values, in any shader stage

• VUID-vkCmdDraw-None-02705
  If the robust buffer access feature is not enabled, and if the VkPipeline object bound to the
  pipeline bind point used by this command accesses a uniform buffer, it **must** not access
  values outside of the range of the buffer as specified in the descriptor set bound to the
  same pipeline bind point

• VUID-vkCmdDraw-None-02706
  If the robust buffer access feature is not enabled, and if the VkPipeline object bound to the
  pipeline bind point used by this command accesses a storage buffer, it **must** not access
  values outside of the range of the buffer as specified in the descriptor set bound to the
  same pipeline bind point

• VUID-vkCmdDraw-commandBuffer-02707
  If commandBuffer is an unprotected command buffer and protectedNoFault is not supported,
  any resource accessed by the VkPipeline object bound to the pipeline bind point used by
  this command **must** not be a protected resource

• VUID-vkCmdDraw-None-04115
  If a VkImageView is accessed using OpImageWrite as a result of this command, then the
  Type of the Texel operand of that instruction **must** have at least as many components as
  the image view’s format

• VUID-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-SampledType-04470
  If a VkImageView with a VkFormat that has a 64-bit component width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64

- VUID-vkCmdDraw-SampledType-04471
  If a VkImageView with a VkFormat that has a component width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32

- VUID-vkCmdDraw-SampledType-04472
  If a VkBufferView with a VkFormat that has a 64-bit component width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64

- VUID-vkCmdDraw-SampledType-04473
  If a VkBufferView with a VkFormat that has a component width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32

- VUID-vkCmdDraw-sparseImageInt64Atomics-04474
  If the sparseImageInt64Atomics feature is not enabled, VkImage objects created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command

- VUID-vkCmdDraw-sparseImageInt64Atomics-04475
  If the sparseImageInt64Atomics feature is not enabled, VkBuffer objects created with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command

- 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-02686
  Every input attachment used by the current subpass must be bound to the pipeline via a descriptor set

- VUID-vkCmdDraw-None-04584
  Image subresources used as attachments in the current render pass must not be accessed in any way other than as an attachment by this command, except for cases involving read-only access to depth/stencil attachments as described in the Render Pass chapter

- VUID-vkCmdDraw-maxMultiviewInstanceIndex-02688
  If the draw is recorded in a render pass instance with multiview enabled, the maximum
instance index **must** be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`

- **VUID-vkCmdDraw-sampleLocationsEnable-02689**
  If the bound graphics pipeline was created with `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` set to `VK_TRUE` and the current subpass has a depth/stencil attachment, then that attachment **must** have been created with the `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` bit set.

- **VUID-vkCmdDraw-viewportCount-03417**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` dynamic state enabled, then `vkCmdSetViewportWithCountEXT` **must** have been called in the current command buffer prior to this drawing command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` **must** match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline.

- **VUID-vkCmdDraw-scissorCount-03418**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, then `vkCmdSetScissorWithCountEXT` **must** have been called in the current command buffer prior to this drawing command, and the `scissorCount` parameter of `vkCmdSetScissorWithCountEXT` **must** match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline.

- **VUID-vkCmdDraw-viewportCount-03419**
  If the bound graphics pipeline state was created with both the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic states enabled, then both `vkCmdSetViewportWithCountEXT` and `vkCmdSetScissorWithCountEXT` **must** have been called in the current command buffer prior to this drawing command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` **must** match the `scissorCount` parameter of `vkCmdSetScissorWithCountEXT`.

- **VUID-vkCmdDraw-None-04876**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT` dynamic state enabled then `vkCmdSetRasterizerDiscardEnableEXT` **must** have been called in the current command buffer prior to this drawing command.

- **VUID-vkCmdDraw-None-04877**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT` dynamic state enabled then `vkCmdSetDepthBiasEnableEXT` **must** have been called in the current command buffer prior to this drawing command.

- **VUID-vkCmdDraw-logicOp-04878**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_LOGIC_OP_EXT` dynamic state enabled then `vkCmdSetLogicOpEXT` **must** have been called in the current command buffer prior to this drawing command and the `logicOp` **must** be a valid...
If the `primitiveFragmentShadingRateWithMultipleViewports` limit is not supported, the bound graphics pipeline was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, and any of the shader stages of the bound graphics pipeline write to the `PrimitiveShadingRateKHR` built-in, then `vkCmdSetViewportWithCountEXT` must have been called in the current command buffer prior to this drawing command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` must be 1.

If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view’s format features do not contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT`, then the `blendEnable` member of the corresponding element of the `pAttachments` member of `pColorBlendState` must be `VK_FALSE`.

If rasterization is not disabled in the bound graphics pipeline, and neither the `VK_AMD_mixed_attachment_samples` nor the `VK_NV_framebuffer_mixed_samples` extensions are enabled, then the `VkPipelineMultisampleStateCreateInfo::rasterizationSamples` must be the same as the current subpass color and/or depth/stencil attachments.

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.

If any of the shader stages of the `VkPipeline` bound to the pipeline bind point used by this command uses the `RayQueryKHR` capability, then commandBuffer must not be a protected command buffer.

All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must have either valid or `VK_NULL_HANDLE` buffers bound.

If the `nullDescriptor` feature is not enabled, all vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must not be `VK_NULL_HANDLE`.

For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in `Vertex Input Description`
If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` dynamic state enabled then `vkCmdSetPrimitiveTopologyEXT` must have been called in the current command buffer prior to this drawing command, and the `primitiveTopology` parameter of `vkCmdSetPrimitiveTopologyEXT` must be of the same topology class as the pipeline `VkPipelineInputAssemblyStateCreateInfo::topology` state.

If the bound graphics pipeline was created with both the `VK_DYNAMIC_STATE_VERTEX_INPUT_EXT` and `VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT` dynamic states enabled, then `vkCmdSetVertexInputEXT` must have been called in the current command buffer prior to this draw command.

If the bound graphics pipeline was created with the `VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VERTEX_INPUT_EXT` dynamic state enabled, then `vkCmdBindVertexBuffers2EXT` must have been called in the current command buffer prior to this draw command, and the `pStrides` parameter of `vkCmdBindVertexBuffers2EXT` must not be NULL.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VERTEX_INPUT_EXT` dynamic state enabled, then `vkCmdSetVertexInputEXT` must have been called in the current command buffer prior to this draw command.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT` dynamic state enabled then `vkCmdSetPatchControlPointsEXT` 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_PRIMITIVE_RESTART_ENABLE_EXT` dynamic state enabled then `vkCmdSetPrimitiveRestartEnableEXT` must have been called in the current command buffer prior to this drawing command.

Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `commandBuffer` must be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations
VUID-vkCmdDraw-renderpass
This command **must** only be called inside of a render pass instance

**Host Synchronization**

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

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To record an indexed draw, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdDrawIndexed(
    VkCommandBuffer commandBuffer,
    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_UINT8_EXT or VK_INDEX_TYPE_UINT16) and have vertexOffset added to them, before being supplied as the vertexIndex value.

The primitives are drawn instanceCount times with instanceIndex starting with firstInstance and increasing sequentially for each instance. The assembled primitives execute the bound graphics pipeline.

### Valid Usage

- **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-02692**
  If a VkImageView is sampled with VK_FILTER_CUBIC_EXT as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT

- **VUID-vkCmdDrawIndexed-filterCubic-02694**
  Any VkImageView being sampled with VK_FILTER_CUBIC_EXT as a result of this command must have a VkImageViewType and format that supports cubic filtering, as specified by VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic returned by vkGetPhysicalDeviceImageFormatProperties2

- **VUID-vkCmdDrawIndexed-filterCubicMinmax-02695**
  Any VkImageView being sampled with VK_FILTER_CUBIC_EXT with a reduction mode of either VK_SAMPLER_REDUCTION_MODE_MIN or VK_SAMPLER_REDUCTION_MODE_MAX as a result of this command must have a VkImageViewType and format that supports cubic filtering together with minmax filtering, as specified by VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax returned by vkGetPhysicalDeviceImageFormatProperties2

- **VUID-vkCmdDrawIndexed-None-02697**
  For each set n that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a descriptor set must have been bound to n at the same pipeline bind point, with a VkPipelineLayout that is compatible for set n, with the VkPipelineLayout
used to create the current VkPipeline, as described in Pipeline Layout Compatibility

- VUID-vkCmdDrawIndexed-None-02698
  For each push constant that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a VkPipelineLayout that is compatible for push constants, with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout Compatibility

- VUID-vkCmdDrawIndexed-None-02699
  Descriptors in each bound descriptor set, specified via vkCmdBindDescriptorSets, must be valid if they are statically used by the VkPipeline bound to the pipeline bind point used by this command

- VUID-vkCmdDrawIndexed-None-02700
  A valid pipeline must be bound to the pipeline bind point used by this command

- VUID-vkCmdDrawIndexed-commandBuffer-02701
  If the VkPipeline object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited (if the VK_NV_inherited_viewport_scissor extension is enabled) for commandBuffer, and done so after any previously bound pipeline with the corresponding state not specified as dynamic

- VUID-vkCmdDrawIndexed-None-02859
  There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the VkPipeline object bound to the pipeline bind point used by this command, since that pipeline was bound

- VUID-vkCmdDrawIndexed-None-02702
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used to sample from any VkImage with a VkImageView of the type VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, VK_IMAGE_VIEW_TYPE_1D_ARRAY, VK_IMAGE_VIEW_TYPE_2D_ARRAY or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY, in any shader stage

- VUID-vkCmdDrawIndexed-None-02703
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions with ImplicitLod, Dref or Proj in their name, in any shader stage

- VUID-vkCmdDrawIndexed-None-02704
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions that includes a LOD bias or any offset values, in any shader stage

- VUID-vkCmdDrawIndexed-None-02705
  If the robust buffer access feature is not enabled, and if the VkPipeline object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
If the robust buffer access feature is not enabled, and if the VkPipeline object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, any resource accessed by the VkPipeline object bound to the pipeline bind point used by this command must not be a protected resource.

If 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.

If a VkImageView with a VkFormat that has a 64-bit component width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

If a VkImageView with a VkFormat that has a component width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

If a VkBufferView with a VkFormat that has a 64-bit component width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

If a VkBufferView with a VkFormat that has a component width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

If the sparseImageInt64Atomics feature is not enabled, VkImage objects created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

If the sparseImageInt64Atomics feature is not enabled, VkBuffer objects created with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

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-02686
  Every input attachment used by the current subpass must be bound to the pipeline via a descriptor set

• VUID-vkCmdDrawIndexed-None-04584
  Image subresources used as attachments in the current render pass must not be accessed in any way other than as an attachment by this command, except for cases involving read-only access to depth/stencil attachments as described in the Render Pass chapter

• VUID-vkCmdDrawIndexed-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-vkCmdDrawIndexed-sampleLocationsEnable-02689
  If the bound graphics pipeline was created with VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable set to VK_TRUE and the current subpass has a depth/stencil attachment, then that attachment must have been created with the VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT bit set

• VUID-vkCmdDrawIndexed-viewportCount-03417
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT dynamic state enabled, then vkCmdSetViewportWithCountEXT must have been called in the current command buffer prior to this drawing command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must match the VkPipelineViewportStateCreateInfo::viewportCount of the pipeline

• VUID-vkCmdDrawIndexed-scissorCount-03418
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, then vkCmdSetScissorWithCountEXT must have been called in the current command buffer prior to this drawing command, and the scissorCount parameter of vkCmdSetScissorWithCountEXT must match the VkPipelineViewportStateCreateInfo::viewportCount of the pipeline

• VUID-vkCmdDrawIndexed-viewportCount-03419
  If the bound graphics pipeline state was created with both the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT and VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic states enabled then both vkCmdSetViewportWithCountEXT and vkCmdSetScissorWithCountEXT must have been called in the current command buffer prior to this drawing command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must match the scissorCount parameter of vkCmdSetScissorWithCountEXT
If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT` dynamic state enabled then `vkCmdSetRasterizerDiscardEnableEXT` must have been called in the current command buffer prior to this drawing command.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT` dynamic state enabled then `vkCmdSetDepthBiasEnableEXT` 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_LOGIC_OP_EXT` dynamic state enabled then `vkCmdSetLogicOpEXT` must have been called in the current command buffer prior to this drawing command and the `logicOp` must be a valid `VkLogicOp` value.

If the `primitiveFragmentShadingRateWithMultipleViewports` limit is not supported, the bound graphics pipeline was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, and any of the shader stages of the bound graphics pipeline write to the `PrimitiveShadingRateKHR` built-in, then `vkCmdSetViewportWithCountEXT` must have been called in the current command buffer prior to this drawing command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` must be 1.

If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view's `format features` do not contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT`, then the `blendEnable` member of the corresponding element of the `pAttachments` member of `pColorBlendState` must be `VK_FALSE`.

If rasterization is not disabled in the bound graphics pipeline, and neither the `VK_AMD_mixed_attachment_samples` nor the `VK_NV_framebuffer_mixed_samples` extensions are enabled, then `VkPipelineMultisampleStateCreateInfo::rasterizationSamples` must be the same as the current subpass color and/or depth/stencil attachments.

If `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.

If any of the shader stages of the `VkPipeline` bound to the pipeline bind point used by this
command uses the `RayQueryKHR` capability, then `commandBuffer` must not be a protected command buffer

- **VUID-vkCmdDrawIndexed-None-04007**
  All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface must have either valid or `VK_NULL_HANDLE` buffers bound

- **VUID-vkCmdDrawIndexed-None-04008**
  If the `nullDescriptor` feature is not enabled, all vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface must not be `VK_NULL_HANDLE`

- **VUID-vkCmdDrawIndexed-None-02721**
  For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in `Vertex Input Description`

- **VUID-vkCmdDrawIndexed-primitiveTopology-03420**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` dynamic state enabled then `vkCmdSetPrimitiveTopologyEXT` must have been called in the current command buffer prior to this drawing command, and the `primitiveTopology` parameter of `vkCmdSetPrimitiveTopologyEXT` must be of the same topology class as the pipeline `VkPipelineInputAssemblyStateCreateInfo::topology` state

- **VUID-vkCmdDrawIndexed-None-04912**
  If the bound graphics pipeline was created with both the `VK_DYNAMIC_STATE_VERTEX_INPUT_EXT` and `VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT` dynamic states enabled, then `vkCmdSetVertexInputEXT` must have been called in the current command buffer prior to this draw command

- **VUID-vkCmdDrawIndexed-pStrides-04913**
  If the bound graphics pipeline was created with the `VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VERTEX_INPUT_EXT` dynamic state enabled, then `vkCmdBindVertexBuffers2EXT` must have been called in the current command buffer prior to this draw command, and the `pStrides` parameter of `vkCmdBindVertexBuffers2EXT` must not be `NULL`

- **VUID-vkCmdDrawIndexed-None-04914**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VERTEX_INPUT_EXT` dynamic state enabled, then `vkCmdSetVertexInputEXT` must have been called in the current command buffer prior to this draw command

- **VUID-vkCmdDrawIndexed-None-04875**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT` dynamic state enabled then `vkCmdSetPatchControlPointsEXT` must have been called in the current command buffer prior to this drawing command

- **VUID-vkCmdDrawIndexed-None-04879**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT` dynamic state enabled then
vkCmdSetPrimitiveRestartEnableEXT must have been called in the current command buffer prior to this drawing command

- VUID-vkCmdDrawIndexed-firstIndex-04932 (indexSize × (firstIndex + indexCount) + offset) must be less than or equal to the size of the bound index buffer, with indexSize being based on the type specified by indexType, where the index buffer, indexType, and offset are specified via vkCmdBindIndexBuffer

Valid Usage (Implicit)

- VUID-vkCmdDrawIndexed-commandBuffer-parameter commandBuffer must be a valid VkCommandBuffer handle
- VUID-vkCmdDrawIndexed-commandBuffer-recording commandBuffer must be in the recording state
- VUID-vkCmdDrawIndexed-commandBuffer-cmdpool The VkCommandPool that commandBuffer was allocated from must support graphics operations
- VUID-vkCmdDrawIndexed-renderpass This command must only be called inside of a render pass instance

Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To record a non-indexed indirect drawing command, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdDrawIndirect(
    VkCommandBuffer commandBuffer, 
    VkBuffer buffer, 
    VkDeviceSize offset, 
    uint32_t drawCount, 
    uint32_t stride);
```
• `commandBuffer` is the command buffer into which the command is recorded.
• `buffer` is the buffer containing draw parameters.
• `offset` is the byte offset into `buffer` where parameters begin.
• `drawCount` is the number of draws to execute, and can be zero.
• `stride` is the byte stride between successive sets of draw parameters.

`vkCmdDrawIndirect` behaves similarly to `vkCmdDraw` except that the parameters are read by the device from a buffer during execution. `drawCount` draws are executed by the command, with parameters taken from `buffer` starting at `offset` and increasing by `stride` bytes for each successive draw. The parameters of each draw are encoded in an array of `VkDrawIndirectCommand` structures. If `drawCount` is less than or equal to one, `stride` is ignored.

### Valid Usage

- **VUID-vkCmdDrawIndirect-magFilter-04553**
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.

- **VUID-vkCmdDrawIndirect-mipmapMode-04770**
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.

- **VUID-vkCmdDrawIndirect-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-02692**
  If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`.

- **VUID-vkCmdDrawIndirect-filterCubic-02694**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`.

- **VUID-vkCmdDrawIndirect-filterCubicMinmax-02695**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by
VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax returned by vkGetPhysicalDeviceImageFormatProperties2

- **VUID-vkCmdDrawIndirect-None-02697**
  For each set \( n \) that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a descriptor set must have been bound to \( n \) at the same pipeline bind point, with a VkPipelineLayout that is compatible for set \( n \), with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout Compatibility.

- **VUID-vkCmdDrawIndirect-None-02698**
  For each push constant that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout Compatibility.

- **VUID-vkCmdDrawIndirect-None-02699**
  Descriptors in each bound descriptor set, specified via vkCmdBindDescriptorSets, must be valid if they are statically used by the VkPipeline bound to the pipeline bind point used by this command.

- **VUID-vkCmdDrawIndirect-None-02700**
  A valid pipeline must be bound to the pipeline bind point used by this command.

- **VUID-vkCmdDrawIndirect-commandBuffer-02701**
  If the VkPipeline object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited (if the VK_NV_inherited_viewport_scissor extension is enabled) for commandBuffer, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

- **VUID-vkCmdDrawIndirect-None-02859**
  There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the VkPipeline object bound to the pipeline bind point used by this command, since that pipeline was bound.

- **VUID-vkCmdDrawIndirect-None-02702**
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used to sample from any VkImage with a VkImageView of the type VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, VK_IMAGE_VIEW_TYPE_1D_ARRAY, VK_IMAGE_VIEW_TYPE_2D_ARRAY or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY, in any shader stage.

- **VUID-vkCmdDrawIndirect-None-02703**
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions with ImplicitLod, Dref or Proj in their name, in any shader stage.

- **VUID-vkCmdDrawIndirect-None-02704**
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions that includes a LOD.
bias or any offset values, in any shader stage

- **VUID-vkCmdDrawIndirect-None-02705**
  If the robust buffer access feature is not enabled, and if the VkPipeline object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- **VUID-vkCmdDrawIndirect-None-02706**
  If the robust buffer access feature is not enabled, and if the VkPipeline object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- **VUID-vkCmdDrawIndirect-commandBuffer-02707**
  If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, any resource accessed by the VkPipeline object bound to the pipeline bind point used by this command **must** not be a protected resource.

- **VUID-vkCmdDrawIndirect-None-04115**
  If a VkImageView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction **must** have at least as many components as the image view's format.

- **VUID-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-SampledType-04470**
  If a VkImageView with a VkFormat that has a 64-bit component width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction **must** have a Width of 64.

- **VUID-vkCmdDrawIndirect-SampledType-04471**
  If a VkImageView with a VkFormat that has a component width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction **must** have a Width of 32.

- **VUID-vkCmdDrawIndirect-SampledType-04472**
  If a VkBufferView with a VkFormat that has a 64-bit component width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction **must** have a Width of 64.

- **VUID-vkCmdDrawIndirect-SampledType-04473**
  If a VkBufferView with a VkFormat that has a component width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction **must** have a Width of 32.

- **VUID-vkCmdDrawIndirect-sparseImageInt64Atomics-04474**
  If the sparseImageInt64Atomics feature is not enabled, VkImage objects created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag **must** not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.
If the `sparseImageInt64Atomics` feature is not enabled, `VkBuffer` objects created with the `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` flag **must** not be accessed by atomic instructions through an `OpTypeImage` with a `SampledType` with a `Width` of 64 by this command.

The current render pass **must** be compatible with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.

The subpass index of the current render pass **must** be equal to the `subpass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.

Every input attachment used by the current subpass **must** be bound to the pipeline via a descriptor set.

Image subresources used as attachments in the current render pass **must** not be accessed in any way other than as an attachment by this command, except for cases involving read-only access to depth/stencil attachments as described in the `Render Pass` chapter.

If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`.

If the bound graphics pipeline was created with `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` set to `VK_TRUE` and the current subpass has a depth/stencil attachment, then that attachment **must** have been created with the `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` bit set.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` dynamic state enabled, then `vkCmdSetViewportWithCountEXT` **must** have been called in the current command buffer prior to this drawing command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` **must** match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, then `vkCmdSetScissorWithCountEXT` **must** have been called in the current command buffer prior to this drawing command, and the `scissorCount` parameter of `vkCmdSetScissorWithCountEXT` **must** match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline.
If the bound graphics pipeline state was created with both the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic states enabled then both `vkCmdSetViewportWithCountEXT` and `vkCmdSetScissorWithCountEXT` must have been called in the current command buffer prior to this drawing command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` must match the `scissorCount` parameter of `vkCmdSetScissorWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT` dynamic state enabled then `vkCmdSetRasterizerDiscardEnableEXT` must have been called in the current command buffer prior to this drawing command.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT` dynamic state enabled then `vkCmdSetDepthBiasEnableEXT` 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_LOGIC_OP_EXT` dynamic state enabled then `vkCmdSetLogicOpEXT` must have been called in the current command buffer prior to this drawing command and the `logicOp` must be a valid `VkLogicOp` value.

If the `primitiveFragmentShadingRateWithMultipleViewports` limit is not supported, the bound graphics pipeline was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, and any of the shader stages of the bound graphics pipeline write to the `PrimitiveShadingRateKHR` built-in, then `vkCmdSetViewportWithCountEXT` must have been called in the current command buffer prior to this drawing command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` must be 1.

If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view's format features do not contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT`, then the `blendEnable` member of the corresponding element of the `pAttachments` member of `pColorBlendState` must be `VK_FALSE`.

If rasterization is not disabled in the bound graphics pipeline, and neither the `VK_AMD_mixed_attachment_samples` nor the `VK_NV_framebuffer_mixed_samples` extensions are enabled, then `VkPipelineMultisampleStateCreateInfo::rasterizationSamples` must be the same as the current subpass color and/or depth/stencil attachments.

All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface must have either valid or `VK_NULL_HANDLE` buffers bound.
If the `nullDescriptor` feature is not enabled, all vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must not be `VK_NULL_HANDLE`.

For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in Vertex Input Description.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` dynamic state enabled then `vkCmdSetPrimitiveTopologyEXT` must have been called in the current command buffer prior to this drawing command, and the `primitiveTopology` parameter of `vkCmdSetPrimitiveTopologyEXT` must be of the same topology class as the pipeline `VkPipelineInputAssemblyStateCreateInfo::topology` state.

If the bound graphics pipeline was created with both the `VK_DYNAMIC_STATE_VERTEX_INPUT_EXT` and `VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT` dynamic states enabled, then `vkCmdSetVertexInputEXT` must have been called in the current command buffer prior to this draw command.

If the bound graphics pipeline was created with the `VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VERTEX_INPUT_EXT` dynamic state enabled, then `vkCmdBindVertexBuffers2EXT` must have been called in the current command buffer prior to this draw command, and the `pStrides` parameter of `vkCmdBindVertexBuffers2EXT` must not be `NULL`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VERTEX_INPUT_EXT` dynamic state enabled, then `vkCmdSetVertexInputEXT` must have been called in the current command buffer prior to this draw command.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT` dynamic state enabled then `vkCmdSetPatchControlPointsEXT` 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_PRIMITIVE_RESTART_ENABLE_EXT` dynamic state enabled then `vkCmdSetPrimitiveRestartEnableEXT` must have been called in the current command buffer prior to this drawing command.

If `buffer` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.

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

- 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 multi-draw indirect 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-firstInstance-00478
  If the drawIndirectFirstInstance feature is not enabled, all the firstInstance members of the VkDrawIndirectCommand structures accessed by this command must be 0

- 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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkDrawIndirectCommand` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDrawIndirectCommand {
    uint32_t vertexCount;
    uint32_t instanceCount;
    uint32_t firstVertex;
    uint32_t firstInstance;
} VkDrawIndirectCommand;
```

- `vertexCount` is the number of vertices to draw.
- `instanceCount` is the number of instances to draw.
- `firstVertex` is the index of the first vertex to draw.
- `firstInstance` is the instance ID of the first instance to draw.

The members of `VkDrawIndirectCommand` have the same meaning as the similarly named parameters of `vkCmdDraw`.

Valid Usage

- VUID-VkDrawIndirectCommand-None-00500
  For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in Vertex Input Description
- VUID-VkDrawIndirectCommand-firstInstance-00501
  If the `drawIndirectFirstInstance` feature is not enabled, `firstInstance` must be 0

To record a non-indexed draw call with a draw call count sourced from a buffer, call:

```c
// Provided by VK_VERSION_1_2
```
void vkCmdDrawIndirectCount(
    VkCommandBuffer commandBuffer,
    VkBuffer buffer,
   VkDeviceSize offset,
    VkBuffer countBuffer,
    VkDeviceSize countBufferOffset,
    uint32_t maxDrawCount,
    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.

**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-02692**
  If a **VkImageView** is sampled with **VK_FILTER_CUBIC_EXT** as a result of this command, then the image view’s **format features** must contain **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT**.
Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`.

Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`.

For each set `n` that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set must have been bound to `n` at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set `n`, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility.

For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with the `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility.

Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

A valid pipeline must be bound to the pipeline bind point used by this command.

If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited (if the `VK_NV_inherited_viewport_scissor` extension is enabled) for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

940
If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions with ImplicitLod, Dref or Proj in their name, in any shader stage.

If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions that includes a LOD bias or any offset values, in any shader stage.

If the robust buffer access feature is not enabled, and if the VkPipeline object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

If the robust buffer access feature is not enabled, and if the VkPipeline object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, any resource accessed by the VkPipeline object bound to the pipeline bind point used by this command must not be a protected resource.

If 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.

If a VkImageView with a VkFormat that has a 64-bit component width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

If a VkImageView with a VkFormat that has a component width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

If a VkBufferView with a VkFormat that has a 64-bit component width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.
If a `VkBufferView` with a `VkFormat` that has a component width less than 64-bit is accessed as a result of this command, the `SampledType` of the `OpTypeImage` operand of that instruction must have a `Width` of 32.

If the `sparseImageInt64Atomics` feature is not enabled, `VkImage` objects created with the `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` flag must not be accessed by atomic instructions through an `OpTypeImage` with a `SampledType` with a `Width` of 64 by this command.

If the `sparseImageInt64Atomics` feature is not enabled, `VkBuffer` objects created with the `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` flag must not be accessed by atomic instructions through an `OpTypeImage` with a `SampledType` with a `Width` of 64 by this command.

The current render pass must be compatible with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.

The subpass index of the current render pass must be equal to the `subpass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.

Every input attachment used by the current subpass must be bound to the pipeline via a descriptor set.

Image subresources used as attachments in the current render pass must not be accessed in any way other than as an attachment by this command, except for cases involving read-only access to depth/stencil attachments as described in the Render Pass chapter.

If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index must be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`.

If the bound graphics pipeline was created with `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` set to `VK_TRUE` and the current subpass has a depth/stencil attachment, then that attachment must have been created with the `VK_IMAGE_CREATESAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` bit set.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` dynamic state enabled, then `vkCmdSetViewportWithCountEXT` must have been called in the current command buffer prior to this drawing command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` must match the `VkPipelineViewportStateCreateInfo::scissorCount` of the pipeline.
If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, then `vkCmdSetScissorWithCountEXT` must have been called in the current command buffer prior to this drawing command, and the `scissorCount` parameter of `vkCmdSetScissorWithCountEXT` must match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline.

If the bound graphics pipeline state was created with both the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic states enabled then both `vkCmdSetViewportWithCountEXT` and `vkCmdSetScissorWithCountEXT` must have been called in the current command buffer prior to this drawing command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` must match the `scissorCount` parameter of `vkCmdSetScissorWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT` dynamic state enabled then `vkCmdSetRasterizerDiscardEnableEXT` must have been called in the current command buffer prior to this drawing command.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT` dynamic state enabled then `vkCmdSetDepthBiasEnableEXT` 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_LOGIC_OP_EXT` dynamic state enabled then `vkCmdSetLogicOpEXT` must have been called in the current command buffer prior to this drawing command and the `logicOp` must be a valid `VkLogicOp` value.

If the `primitiveFragmentShadingRateWithMultipleViewports` limit is not supported, the bound graphics pipeline was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, and any of the shader stages of the bound graphics pipeline write to the `PrimitiveShadingRateKHR` built-in, then `vkCmdSetViewportWithCountEXT` must have been called in the current command buffer prior to this drawing command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` must be 1.

If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view's `format features` do not contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT`, then the `blendEnable` member of the corresponding element of the `pAttachments` member of `pColorBlendState` must be `VK_FALSE`.
If rasterization is not disabled in the bound graphics pipeline, and neither the
VK_AMD_mixed_attachment_samples nor the VK_NV_framebuffer_mixed_samples extensions are
enabled, then VkPipelineMultisampleStateCreateInfo::rasterizationSamples must be the
same as the current subpass color and/or depth/stencil attachments

All vertex input bindings accessed via vertex input variables declared in the vertex
shader entry point’s interface must have either valid or VK_NULL_HANDLE buffers bound

If the nullDescriptor feature is not enabled, all vertex input bindings accessed via vertex
input variables declared in the vertex shader entry point’s interface must not be
VK_NULL_HANDLE

For a given vertex buffer binding, any attribute data fetched must be entirely contained
within the corresponding vertex buffer binding, as described in Vertex Input Description

If the bound graphics pipeline state was created with the
VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT dynamic state enabled then
vkCmdSetPrimitiveTopologyEXT must have been called in the current command buffer prior
to this drawing command, and the primitiveTopology parameter of
vkCmdSetPrimitiveTopologyEXT must be of the same topology class as the pipeline
VkPipelineInputAssemblyStateCreateInfo::topology state

If the bound graphics pipeline was created with both the
VK_DYNAMIC_STATE_VERTEX_INPUT_EXT and VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT
dynamic states enabled, then vkCmdSetVertexInputEXT must have been called in the current command buffer prior to this draw command

If the bound graphics pipeline was created with the
VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT dynamic state enabled, but not the
VK_DYNAMIC_STATE_VERTEX_INPUT_EXT dynamic state enabled, then
vkCmdBindVertexBuffers2EXT must have been called in the current command buffer prior to this draw command, and the pStrides parameter of
vkCmdBindVertexBuffers2EXT must not be NULL

If the bound graphics pipeline state was created with the
VK_DYNAMIC_STATE_VERTEX_INPUT_EXT dynamic state enabled, then
vkCmdSetVertexInputEXT must have been called in the current command buffer prior to this draw command

If the bound graphics pipeline state was created with the
VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT dynamic state enabled then
vkCmdSetPatchControlPointsEXT 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_PRIMITIVE_RESTART_ENABLE_EXT` dynamic state enabled then `vkCmdSetPrimitiveRestartEnableEXT` must have been called in the current command buffer prior to this drawing command.

If `buffer` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.

`buffer` must have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set.

`offset` must be a multiple of 4.

`commandBuffer` must not be a protected command buffer.

If `countBuffer` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.

`countBuffer` must have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set.

`countBufferOffset` must be a multiple of 4.

The count stored in `countBuffer` must be less than or equal to `VkPhysicalDeviceLimits::maxDrawIndirectCount`.

`(countBufferOffset + sizeof(uint32_t))` must be less than or equal to the size of `countBuffer`.

If `drawIndirectCount` is not enabled this function must not be used.

`stride` must be a multiple of 4 and must be greater than or equal to `sizeof(VkDrawIndirectCommand)`.

If `maxDrawCount` is greater than or equal to 1, `(stride × (maxDrawCount - 1) + offset + sizeof(VkDrawIndirectCommand))` must be less than or equal to the size of `buffer`.

If the count stored in `countBuffer` is equal to 1, `(offset + sizeof(VkDrawIndirectCommand))` must be less than or equal to the size of `buffer`.

If the count stored in `countBuffer` is greater than 1, `(stride × (drawCount - 1) + offset + sizeof(VkDrawIndirectCommand))` must be less than or equal to the size of `buffer`.
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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To record an indexed indirect drawing command, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdDrawIndexedIndirect(
    VkCommandBuffer commandBuffer,
    VkBuffer buffer,
    VkDeviceSize offset,
    uint32_t drawCount,
    uint32_t stride);
```
• `commandBuffer` is the command buffer into which the command is recorded.
• `buffer` is the buffer containing draw parameters.
• `offset` is the byte offset into `buffer` where parameters begin.
• `drawCount` is the number of draws to execute, and can be zero.
• `stride` is the byte stride between successive sets of draw parameters.

`vkCmdDrawIndexedIndirect` behaves similarly to `vkCmdDrawIndexed` except that the parameters are read by the device from a buffer during execution. `drawCount` draws are executed by the command, with parameters taken from `buffer` starting at `offset` and increasing by `stride` bytes for each successive draw. The parameters of each draw are encoded in an array of `VkDrawIndexedIndirectCommand` structures. If `drawCount` is less than or equal to one, `stride` is ignored.

### Valid Usage

- **VUID-vkCmdDrawIndexedIndirect-magFilter-04553**
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.

- **VUID-vkCmdDrawIndexedIndirect-mipmapMode-04770**
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.

- **VUID-vkCmdDrawIndexedIndirect-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-02692**
  If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`.

- **VUID-vkCmdDrawIndexedIndirect-filterCubic-02694**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`.

- **VUID-vkCmdDrawIndexedIndirect-filterCubicMinmax-02695**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering.
together with minmax filtering, as specified by
\texttt{VkFilterCubicImageViewImageFormatPropertiesEXT} returned by
\texttt{vkGetPhysicalDeviceImageFormatProperties2}

- **VUID-vkCmdDrawIndexedIndirect-None-02697**
  For each set \( n \) that is statically used by the \texttt{VkPipeline} bound to the pipeline bind point used by this command, a descriptor set \texttt{must} have been bound to \( n \) at the same pipeline bind point, with a \texttt{VkPipelineLayout} that is compatible for set \( n \), with the \texttt{VkPipelineLayout} used to create the current \texttt{VkPipeline}, as described in \textit{Pipeline Layout Compatibility}.

- **VUID-vkCmdDrawIndexedIndirect-None-02698**
  For each push constant that is statically used by the \texttt{VkPipeline} bound to the pipeline bind point used by this command, a push constant value \texttt{must} have been set for the same pipeline bind point, with a \texttt{VkPipelineLayout} that is compatible for push constants, with the \texttt{VkPipelineLayout} used to create the current \texttt{VkPipeline}, as described in \textit{Pipeline Layout Compatibility}.

- **VUID-vkCmdDrawIndexedIndirect-None-02699**
  Descriptors in each bound descriptor set, specified via \texttt{vkCmdBindDescriptorSets}, \texttt{must} be valid if they are statically used by the \texttt{VkPipeline} bound to the pipeline bind point used by this command.

- **VUID-vkCmdDrawIndexedIndirect-None-02700**
  A valid pipeline \texttt{must} be bound to the pipeline bind point used by this command.

- **VUID-vkCmdDrawIndexedIndirect-commandBuffer-02701**
  If the \texttt{VkPipeline} object bound to the pipeline bind point used by this command requires any dynamic state, that state \texttt{must} have been set or inherited (if the \texttt{VK_NV_inherited_viewport_scissor} extension is enabled) for \texttt{commandBuffer}, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

- **VUID-vkCmdDrawIndexedIndirect-None-02859**
  There \texttt{must} not have been any calls to dynamic state setting commands for any state not specified as dynamic in the \texttt{VkPipeline} object bound to the pipeline bind point used by this command, since that pipeline was bound.

- **VUID-vkCmdDrawIndexedIndirect-None-02702**
  If the \texttt{VkPipeline} object bound to the pipeline bind point used by this command accesses a \texttt{VkSampler} object that uses unnormalized coordinates, that sampler \texttt{must} not be used to sample from any \texttt{VkImage} with a \texttt{VkImageView} of the type \texttt{VK_IMAGE_VIEW_TYPE_3D}, \texttt{VK_IMAGE_VIEW_TYPE_CUBE}, \texttt{VK_IMAGE_VIEW_TYPE_1D_ARRAY}, \texttt{VK_IMAGE_VIEW_TYPE_2D_ARRAY} or \texttt{VK_IMAGE_VIEW_TYPE_CUBE_ARRAY}, in any shader stage.

- **VUID-vkCmdDrawIndexedIndirect-None-02703**
  If the \texttt{VkPipeline} object bound to the pipeline bind point used by this command accesses a \texttt{VkSampler} object that uses unnormalized coordinates, that sampler \texttt{must} not be used with any of the \texttt{SPIR-V OpImageSample*} or \texttt{OpImageSparseSample*} instructions with \texttt{ImplicitLod}, \texttt{Dref} or \texttt{Proj} in their name, in any shader stage.

- **VUID-vkCmdDrawIndexedIndirect-None-02704**
  If the \texttt{VkPipeline} object bound to the pipeline bind point used by this command accesses a \texttt{VkSampler} object that uses unnormalized coordinates, that sampler \texttt{must} not be used with...
any of the SPIR-V `OpImageSample` or `OpImageSparseSample` instructions that includes a LOD bias or any offset values, in any shader stage

- **VUID-vkCmdDrawIndexedIndirect-None-02705**
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point

- **VUID-vkCmdDrawIndexedIndirect-None-02706**
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point

- **VUID-vkCmdDrawIndexedIndirect-commandBuffer-02707**
  If `commandBuffer` is an unprotected command buffer and protectedNoFault is not supported, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command must not be a protected resource

- **VUID-vkCmdDrawIndexedIndirect-None-04115**
  If a `VkImageView` is accessed using `OpImageWrite` as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view’s format

- **VUID-vkCmdDrawIndexedIndirect-OpImageWrite-04469**
  If a `VkBufferView` is accessed using `OpImageWrite` as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the buffer view’s format

- **VUID-vkCmdDrawIndexedIndirect-SampledType-04470**
  If a `VkImageView` with a `VkFormat` that has a 64-bit component width is accessed as a result of this command, the SampledType of the `OpTypeImage` operand of that instruction must have a Width of 64

- **VUID-vkCmdDrawIndexedIndirect-SampledType-04471**
  If a `VkImageView` with a `VkFormat` that has a component width less than 64-bit is accessed as a result of this command, the SampledType of the `OpTypeImage` operand of that instruction must have a Width of 32

- **VUID-vkCmdDrawIndexedIndirect-SampledType-04472**
  If a `VkBufferView` with a `VkFormat` that has a 64-bit component width is accessed as a result of this command, the SampledType of the `OpTypeImage` operand of that instruction must have a Width of 64

- **VUID-vkCmdDrawIndexedIndirect-SampledType-04473**
  If a `VkBufferView` with a `VkFormat` that has a component width less than 64-bit is accessed as a result of this command, the SampledType of the `OpTypeImage` operand of that instruction must have a Width of 32

- **VUID-vkCmdDrawIndexedIndirect-sparseImageInt64Atomics-04474**
  If the sparseImageInt64Atomics feature is not enabled, `VkImage` objects created with the `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` flag must not be accessed by atomic instructions through an `OpTypeImage` with a SampledType with a Width of 64 by this command
• VUID-vkCmdDrawIndexedIndirect-sparseImageInt64Atomics-04475
  If the sparseImageInt64Atomics feature is not enabled, VkBuffer objects created with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command

• 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-02686
  Every input attachment used by the current subpass must be bound to the pipeline via a descriptor set

• VUID-vkCmdDrawIndexedIndirect-None-04584
  Image subresources used as attachments in the current render pass must not be accessed in any way other than as an attachment by this command, except for cases involving read-only access to depth/stencil attachments as described in the Render Pass chapter

• VUID-vkCmdDrawIndexedIndirect-maxMultiviewInstanceIndex-02688
  If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index must be less than or equal to VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex

• VUID-vkCmdDrawIndexedIndirect-sampleLocationsEnable-02689
  If the bound graphics pipeline was created with VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable set to VK_TRUE and the current subpass has a depth/stencil attachment, then that attachment must have been created with the VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT bit set

• VUID-vkCmdDrawIndexedIndirect-viewportCount-03417
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT dynamic state enabled, then vkCmdSetViewportWithCountEXT must have been called in the current command buffer prior to this drawing command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must match the VkPipelineViewportStateCreateInfo::viewportCount of the pipeline

• VUID-vkCmdDrawIndexedIndirect-scissorCount-03418
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, then vkCmdSetScissorWithCountEXT must have been called in the current command buffer prior to this drawing command, and the scissorCount parameter of vkCmdSetScissorWithCountEXT must match the VkPipelineViewportStateCreateInfo::viewportCount of the pipeline
If the bound graphics pipeline state was created with both the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic states enabled then both `vkCmdSetViewportWithCountEXT` and `vkCmdSetScissorWithCountEXT` must have been called in the current command buffer prior to this drawing command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` must match the `scissorCount` parameter of `vkCmdSetScissorWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT` dynamic state enabled then `vkCmdSetRasterizerDiscardEnableEXT` must have been called in the current command buffer prior to this drawing command.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT` dynamic state enabled then `vkCmdSetDepthBiasEnableEXT` 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_LOGIC_OP_EXT` dynamic state enabled then `vkCmdSetLogicOpEXT` must have been called in the current command buffer prior to this drawing command and the `logicOp` must be a valid `VkLogicOp` value.

If the primitive fragment shading rate with multiple viewports limit is not supported, the bound graphics pipeline was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, and any of the shader stages of the bound graphics pipeline write to the `PrimitiveShadingRateKHR` built-in, then `vkCmdSetViewportWithCountEXT` must have been called in the current command buffer prior to this drawing command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` must be 1.

If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view’s format features do not contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT`, then the `blendEnable` member of the corresponding element of the `pAttachments` member of `pColorBlendState` must be `VK_FALSE`.

If rasterization is not disabled in the bound graphics pipeline, and neither the `VK_AMD_mixed_attachment_samples` nor the `VK_NV_framebuffer_mixed_samples` extensions are enabled, then `VkPipelineMultisampleStateCreateInfo::rasterizationSamples` must be the same as the current subpass color and/or depth/stencil attachments.

All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must have either valid or `VK_NULL_HANDLE` buffers.
bound

- **VUID-vkCmdDrawIndexedIndirect-None-04008**
  If the `nullDescriptor` feature is not enabled, all vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must not be `VK_NULL_HANDLE`

- **VUID-vkCmdDrawIndexedIndirect-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-vkCmdDrawIndexedIndirect-primitiveTopology-03420**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` dynamic state enabled then `vkCmdSetPrimitiveTopologyEXT` must have been called in the current command buffer prior to this drawing command, and the `primitiveTopology` parameter of `vkCmdSetPrimitiveTopologyEXT` must be of the same topology class as the pipeline `VkPipelineInputAssemblyStateCreateInfo::topology` state

- **VUID-vkCmdDrawIndexedIndirect-None-04912**
  If the bound graphics pipeline was created with both the `VK_DYNAMIC_STATE_VERTEX_INPUT_EXT` and `VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT` dynamic states enabled, then `vkCmdSetVertexInputEXT` must have been called in the current command buffer prior to this draw command

- **VUID-vkCmdDrawIndexedIndirect-pStrides-04913**
  If the bound graphics pipeline was created with the `VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VERTEX_INPUT_EXT` dynamic state enabled, then `vkCmdBindVertexBuffers2EXT` must have been called in the current command buffer prior to this draw command, and the `pStrides` parameter of `vkCmdBindVertexBuffers2EXT` must not be `NULL`

- **VUID-vkCmdDrawIndexedIndirect-None-04914**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VERTEX_INPUT_EXT` dynamic state enabled, then `vkCmdSetVertexInputEXT` must have been called in the current command buffer prior to this draw command

- **VUID-vkCmdDrawIndexedIndirect-None-04875**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT` dynamic state enabled then `vkCmdSetPatchControlPointsEXT` must have been called in the current command buffer prior to this drawing command

- **VUID-vkCmdDrawIndexedIndirect-None-04879**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT` dynamic state enabled then `vkCmdSetPrimitiveRestartEnableEXT` must have been called in the current command buffer prior to this drawing command

- **VUID-vkCmdDrawIndexedIndirect-buffer-02708**
  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 multi-draw indirect 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-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-firstInstance-00530**
  If the drawIndirectFirstInstance feature is not enabled, all the firstInstance members of the `VkDrawIndexedIndirectCommand` structures accessed by this command must be 0

• **VUID-vkCmdDrawIndexedIndirect-drawCount-00539**
  If drawCount is equal to 1, `(offset + sizeof(VkDrawIndexedIndirectCommand))` must be less than or equal to the size of buffer

• **VUID-vkCmdDrawIndexedIndirect-drawCount-00540**
  If drawCount is greater than 1, 
  ```
  (stride × (drawCount - 1) + offset + sizeof(VkDrawIndexedIndirectCommand))
  ```
  must be less than or equal to the size of buffer

---

**Valid Usage (Implicit)**

• **VUID-vkCmdDrawIndexedIndirect-commandBuffer-parameter**
  commandBuffer must be a valid `VkCommandBuffer` handle

• **VUID-vkCmdDrawIndexedIndirect-buffer-parameter**
  buffer must be a valid `VkBuffer` handle

• **VUID-vkCmdDrawIndexedIndirect-commandBuffer-recording**
  commandBuffer must be in the recording state

• **VUID-vkCmdDrawIndexedIndirect-commandBuffer-cmdpool**
  The `VkCommandPool` that commandBuffer was allocated from must support graphics operations

• **VUID-vkCmdDrawIndexedIndirect-renderpass**
  This command must only be called inside of a render pass instance

• **VUID-vkCmdDrawIndexedIndirect-commonparent**
  Both of buffer, and commandBuffer must have been created, allocated, or retrieved from the same `VkDevice`
Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkDrawIndexedIndirectCommand` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDrawIndexedIndirectCommand {
    uint32_t indexCount;
    uint32_t instanceCount;
    uint32_t firstIndex;
    int32_t  vertexOffset;
    uint32_t firstInstance;
} VkDrawIndexedIndirectCommand;
```

- `indexCount` is the number of vertices to draw.
- `instanceCount` is the number of instances to draw.
- `firstIndex` is the base index within the index buffer.
- `vertexOffset` is the value added to the vertex index before indexing into the vertex buffer.
- `firstInstance` is the instance ID of the first instance to draw.

The members of `VkDrawIndexedIndirectCommand` have the same meaning as the similarly named parameters of `vkCmdDrawIndexed`.

Valid Usage

- VUID-VkDrawIndexedIndirectCommand-None-00552
  For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in `Vertex Input Description`.

- VUID-VkDrawIndexedIndirectCommand-indexSize-00553
  \((\text{indexSize} \times (\text{firstIndex} + \text{indexCount}) + \text{offset})\) 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`.
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`.

- **VUID-vkCmdDrawIndexedIndirectCount-mipmapMode-04770**
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view’s format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.

- **VUID-vkCmdDrawIndexedIndirectCount-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`. 
If a VkImageView is accessed using atomic operations as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT.

If a VkImageView is sampled with VK_FILTER_CUBIC_EXT as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT.

Any VkImageView being sampled with VK_FILTER_CUBIC_EXT as a result of this command must have a VkImageViewType and format that supports cubic filtering, as specified by VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic returned by vkGetPhysicalDeviceImageFormatProperties2.

Any VkImageView being sampled with VK_FILTER_CUBIC_EXT with a reduction mode of either VK_SAMPLER_REDUCTION_MODE_MIN or VK_SAMPLER_REDUCTION_MODE_MAX as a result of this command must have a VkImageViewType and format that supports cubic filtering together with minmax filtering, as specified by VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax returned by vkGetPhysicalDeviceImageFormatProperties2.

For each set n that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a descriptor set must have been bound to n at the same pipeline bind point, with a VkPipelineLayout that is compatible for set n, with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout Compatibility.

For each push constant that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a VkPipelineLayout that is compatible for push constants, with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout Compatibility.

Descriptors in each bound descriptor set, specified via vkCmdBindDescriptorSets, must be valid if they are statically used by the VkPipeline bound to the pipeline bind point used by this command.

A valid pipeline must be bound to the pipeline bind point used by this command.

If the VkPipeline object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited (if the VK_NV_inherited_viewport_scissor extension is enabled) for commandBuffer, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

There must not have been any calls to dynamic state setting commands for any state not
specified as dynamic in the VkPipeline object bound to the pipeline bind point used by this command, since that pipeline was bound

- **VUID-vkCmdDrawIndexedIndirectCount-None-02702**
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used to sample from any VkImage with a VkImageView of the type VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, VK_IMAGE_VIEW_TYPE_1D_ARRAY, VK_IMAGE_VIEW_TYPE_2D_ARRAY or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY, in any shader stage

- **VUID-vkCmdDrawIndexedIndirectCount-None-02703**
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions with ImplicitLod, Dref or Proj in their name, in any shader stage

- **VUID-vkCmdDrawIndexedIndirectCount-None-02704**
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions that includes a LOD bias or any offset values, in any shader stage

- **VUID-vkCmdDrawIndexedIndirectCount-None-02705**
  If the robust buffer access feature is not enabled, and if the VkPipeline object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point

- **VUID-vkCmdDrawIndexedIndirectCount-None-02706**
  If the robust buffer access feature is not enabled, and if the VkPipeline object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point

- **VUID-vkCmdDrawIndexedIndirectCount-commandBuffer-02707**
  If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, any resource accessed by the VkPipeline object bound to the pipeline bind point used by this command must not be a protected resource

- **VUID-vkCmdDrawIndexedIndirectCount-None-04115**
  If a VkImageView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view's format

- **VUID-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-SampledType-04470**
  If a VkImageView with a VkFormat that has a 64-bit component width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64
If a `VkImageView` with a `VkFormat` that has a component width less than 64-bit is accessed as a result of this command, the `SampledType` of the `OpTypeImage` operand of that instruction must have a *Width* of 32.

If a `VkBufferView` with a `VkFormat` that has a 64-bit component width is accessed as a result of this command, the `SampledType` of the `OpTypeImage` operand of that instruction must have a *Width* of 64.

If a `VkBufferView` with a `VkFormat` that has a component width less than 64-bit is accessed as a result of this command, the `SampledType` of the `OpTypeImage` operand of that instruction must have a *Width* of 32.

If the `sparseImageInt64Atomics` feature is not enabled, `VkImage` objects created with the `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` flag must not be accessed by atomic instructions through an `OpTypeImage` with a `SampledType` with a *Width* of 64 by this command.

If the `sparseImageInt64Atomics` feature is not enabled, `VkBuffer` objects created with the `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` flag must not be accessed by atomic instructions through an `OpTypeImage` with a `SampledType` with a *Width* of 64 by this command.

The current render pass must be compatible with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.

The subpass index of the current render pass must be equal to the `subpass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.

Every input attachment used by the current subpass must be bound to the pipeline via a descriptor set.

Image subresources used as attachments in the current render pass must not be accessed in any way other than as an attachment by this command, except for cases involving read-only access to depth/stencil attachments as described in the `Render Pass` chapter.

If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index must be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`.

If the bound graphics pipeline was created with `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` set to `VK_TRUE` and the current subpass has a depth/stencil attachment, then that attachment must have been created with the `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` bit set.
If the bound graphics pipeline state was created with the 
`VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the 
`VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` dynamic state enabled, then 
`vkCmdSetViewportWithCountEXT` must have been called in the current command buffer 
 prior to this drawing command, and the `viewportCount` parameter of 
`vkCmdSetViewportWithCountEXT` must match the `VkPipelineViewportStateCreateInfo` ::`viewportCount` of the pipeline

If the bound graphics pipeline state was created with the 
`VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` dynamic state enabled, but not the 
`VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, then 
`vkCmdSetScissorWithCountEXT` must have been called in the current command buffer 
 prior to this drawing command, and the `scissorCount` parameter of 
`vkCmdSetScissorWithCountEXT` must match the `VkPipelineViewportStateCreateInfo` ::`viewportCount` of the pipeline

If the bound graphics pipeline state was created with both the 
`VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` and 
`VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic states enabled then both 
`vkCmdSetViewportWithCountEXT` and 
`vkCmdSetScissorWithCountEXT` must have been called in the current command buffer 
 prior to this drawing command, and the `viewportCount` parameter of 
`vkCmdSetViewportWithCountEXT` must match the `scissorCount` parameter of 
`vkCmdSetScissorWithCountEXT`

If the bound graphics pipeline state was created with the 
`VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT` dynamic state enabled then 
`vkCmdSetRasterizerDiscardEnableEXT` must have been called in the current command buffer 
 prior to this drawing command

If the bound graphics pipeline state was created with the 
`VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT` dynamic state enabled then 
`vkCmdSetDepthBiasEnableEXT` 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_LOGIC_OP_EXT` dynamic state enabled then 
`vkCmdSetLogicOpEXT` must have been called in the current command buffer 
 prior to this drawing command and the `logicOp` must be a valid 
`VkLogicOp` value

If the `primitiveFragmentShadingRateWithMultipleViewports` limit is not supported, the 
bound graphics pipeline was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` 
dynamic state enabled, and any of the shader stages of the bound graphics pipeline write 
to the `PrimitiveShadingRateKHR` built-in, then `vkCmdSetViewportWithCountEXT` must have
been called in the current command buffer prior to this drawing command, and the viewportCount parameter of \texttt{vkCmdSetViewportWithCountEXT} must be 1.

- **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 \texttt{VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT}, then the blendEnable member of the corresponding element of the \texttt{pAttachments} member of \texttt{pColorBlendState} must be \texttt{VK_FALSE}.

- **VUID-vkCmdDrawIndexedIndirectCount-rasterizationSamples-04740**
  If rasterization is not disabled in the bound graphics pipeline, and neither the \texttt{VK_AMD_mixed_attachment_samples} nor \texttt{VK_NV_framebuffer_mixed_samples} extensions are enabled, then \texttt{VkPipelineMultisampleStateCreateInfo::rasterizationSamples} must be the same as the current subpass color and/or depth/stencil attachments.

- **VUID-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 \texttt{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 \texttt{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 \texttt{Vertex Input Description}.

- **VUID-vkCmdDrawIndexedIndirectCount-primitiveTopology-03420**
  If the bound graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT} dynamic state enabled then \texttt{vkCmdSetPrimitiveTopologyEXT} must have been called in the current command buffer prior to this drawing command, and the primitiveTopology parameter of \texttt{vkCmdSetPrimitiveTopologyEXT} must be of the same topology class as the pipeline \texttt{VkPipelineInputAssemblyStateCreateInfo::topology} state.

- **VUID-vkCmdDrawIndexedIndirectCount-None-04912**
  If the bound graphics pipeline was created with both the \texttt{VK_DYNAMIC_STATE_VERTEX_INPUT_EXT} and \texttt{VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT} dynamic states enabled, then \texttt{vkCmdSetVertexInputEXT} must have been called in the current command buffer prior to this draw command.

- **VUID-vkCmdDrawIndexedIndirectCount-pStrides-04913**
  If the bound graphics pipeline was created with both the \texttt{VK_DYNAMIC_STATE_VERTEX_INPUT_EXT} and \texttt{VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT} dynamic states enabled, but not the \texttt{VK_DYNAMIC_STATE_VERTEX_INPUT_EXT} dynamic state enabled, then \texttt{vkCmdBindVertexBuffers2EXT} must have been called in the current command buffer prior to this draw command, and the pStrides parameter of \texttt{vkCmdBindVertexBuffers2EXT} must not be \texttt{NULL}.

- **VUID-vkCmdDrawIndexedIndirectCount-None-04914**
If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VERTEX_INPUT_EXT` dynamic state enabled, then `vkCmdSetVertexInputEXT` must have been called in the current command buffer prior to this draw command.

- **VUID-vkCmdDrawIndexedIndirectCount-None-04875**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT` dynamic state enabled then `vkCmdSetPatchControlPointsEXT` must have been called in the current command buffer prior to this drawing command.

- **VUID-vkCmdDrawIndexedIndirectCount-None-04879**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT` dynamic state enabled then `vkCmdSetPrimitiveRestartEnableEXT` must have been called in the current command buffer prior to this drawing command.

- **VUID-vkCmdDrawIndexedIndirectCount-buffer-02708**
  If `buffer` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.

- **VUID-vkCmdDrawIndexedIndirectCount-buffer-02709**
  `buffer` must have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set.

- **VUID-vkCmdDrawIndexedIndirectCount-offset-02710**
  `offset` must be a multiple of 4.

- **VUID-vkCmdDrawIndexedIndirectCount-commandBuffer-02711**
  `commandBuffer` must not be a protected command buffer.

- **VUID-vkCmdDrawIndexedIndirectCount-countBuffer-02714**
  If `countBuffer` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.

- **VUID-vkCmdDrawIndexedIndirectCount-countBuffer-02715**
  `countBuffer` must have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set.

- **VUID-vkCmdDrawIndexedIndirectCount-countBufferOffset-02716**
  `countBufferOffset` must be a multiple of 4.

- **VUID-vkCmdDrawIndexedIndirectCount-countBuffer-02717**
  The count stored in `countBuffer` must be less than or equal to `VkPhysicalDeviceLimits::maxDrawIndirectCount`.

- **VUID-vkCmdDrawIndexedIndirectCount-countBufferOffset-04129**
  `(countBufferOffset + sizeof(uint32_t))` must be less than or equal to the size of `countBuffer`.

- **VUID-vkCmdDrawIndexedIndirectCount-None-04445**
  If `drawIndirectCount` is not enabled this function must not be used.

- **VUID-vkCmdDrawIndexedIndirectCount-stride-03142**
  `stride` must be a multiple of 4 and `maxDrawCount` 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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</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

```spir-v
...%1 = OpExtInstImport "GLSL.std.450"
...
OpName %9 "variableName"
OpName %15 "variableNameArray"
OpDecorate %18 BuiltIn VertexIndex
```
21.1.1. Attribute Location and Component Assignment

Vertex shaders allow Location and Component decorations on input variable declarations. The Location decoration specifies which vertex input attribute is used to read and interpret the data that a variable will consume. The Component decoration allows the location to be more finely specified for scalars and vectors, down to the individual components within a location that are consumed. The components within a location are 0, 1, 2, and 3. A variable starting at component N will consume components N, N+1, N+2, … up through its size. For single precision types, it is invalid if the sequence of components gets larger than 3.

When a vertex shader input variable declared using a 16- or 32-bit scalar or vector data type is assigned a location, its value(s) are taken from the components of the input attribute specified with the corresponding VkVertexInputAttributeDescription::location. The components used depend on the type of variable and the Component decoration specified in the variable declaration, as identified in 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.

Table 25. 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>16-bit or 32-bit data type</td>
<td>Component decoration</td>
<td>Components consumed</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------------------</td>
<td>---------------------</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>
<tr>
<td>three-component vector</td>
<td>0 or unspecified</td>
<td>(x, y, z, o)</td>
</tr>
<tr>
<td>three-component vector</td>
<td>1</td>
<td>(o, y, z, w)</td>
</tr>
<tr>
<td>four-component vector</td>
<td>0 or unspecified</td>
<td>(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 32-bit floating point matrix type is assigned a location \(i\), its values are taken from consecutive input attributes starting with the corresponding `VkVertexInputAttributeDescription::location`. Such matrices are treated as an array of column vectors with values taken from the input attributes identified in **Input attributes accessed by 32-bit input matrix variables**. The `VkVertexInputAttributeDescription::format` must be specified with a `VkFormat` that corresponds to the appropriate type of column vector. The **Component decoration** must not be used with matrix types.

**Table 26. 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 (if present), or the default value.
When a vertex shader input variable declared using a scalar or vector 64-bit data type is assigned a location \(i\), its values are taken from consecutive input attributes starting with the corresponding `VkVertexInputAttributeDescription::location`. The locations and components used depend on the type of variable and the `Component` decoration specified in the variable declaration, as identified in Input attribute locations and components accessed by 64-bit input variables. For 64-bit data types, no default attribute values are provided. Input variables must not use more components than provided by the attribute. Input attributes which have one- or two-component 64-bit formats will consume a single location. Input attributes which have three- or four-component 64-bit formats will consume two consecutive locations. A 64-bit scalar data type will consume two components, and a 64-bit two-component vector data type will consume all four components available within a location. A three- or four-component 64-bit data type must not specify a component. A three-component 64-bit data type will consume all four components of the first location and components 0 and 1 of the second location. This leaves components 2 and 3 available for other component-qualified declarations. A four-component 64-bit data type will consume all four components of the first location and all four components of the second location. It is invalid for a scalar or two-component 64-bit data type to specify a component of 1 or 3.

Table 27. Input attribute locations and components accessed by 64-bit input variables

<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 s consumed</th>
</tr>
</thead>
<tbody>
<tr>
<td>R64</td>
<td>i</td>
<td>scalar</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, -, -)</td>
</tr>
<tr>
<td>R64G64</td>
<td>i</td>
<td>scalar</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, o, o)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar</td>
<td>i</td>
<td>2</td>
<td>(o, o, z, w)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>two-component vector</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, z, w)</td>
</tr>
<tr>
<td>R64G64B64</td>
<td>i, i+1</td>
<td>scalar</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, o, o), (o, o, -, -)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar</td>
<td>i</td>
<td>2</td>
<td>(o, o, z, w), (o, o, -, -)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar</td>
<td>i+1</td>
<td>0 or unspecified</td>
<td>(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>Input format</td>
<td>Locations consumed</td>
<td>64-bit data type</td>
<td>Location decoration</td>
<td>Component decoration</td>
<td>32-bit components consumed</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------</td>
<td>------------------------</td>
<td>---------------------</td>
<td>----------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>R64G64B64A64</td>
<td>i, i+1</td>
<td>scalar</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, o, o), (o, o, o, o)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar</td>
<td>i+1</td>
<td>0 or unspecified</td>
<td>(o, o, o, o), (x, y, o, o)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar</td>
<td>i+1</td>
<td>2</td>
<td>(o, o, o, o), (o, o, z, w)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>two-component vector</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, z, w), (o, o, o, o)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>two-component vector</td>
<td>i+1</td>
<td>0 or unspecified</td>
<td>(o, o, o, o), (x, y, z, w)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>three-component vector</td>
<td>i</td>
<td>unspecified</td>
<td>(x, y, z, w), (x, y, o, o)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>four-component vector</td>
<td>i</td>
<td>unspecified</td>
<td>(x, y, z, w), (x, y, z, w)</td>
</tr>
</tbody>
</table>

Components indicated by “o” are available for use by other input variables which are sourced from the same attribute. Components indicated by “-” are not available for input variables as there are no default values provided for 64-bit data types, and there is no data provided by the input format.

When a vertex shader input variable declared using a 64-bit floating-point matrix type is assigned a location \( i \), its values are taken from consecutive input attribute locations. Such matrices are treated as an array of column vectors with values taken from the input attributes as shown in Input attribute locations and components accessed by 64-bit input variables. Each column vector starts at the location immediately following the last location of the previous column vector. The number of attributes and components assigned to each matrix is determined by the matrix dimensions and ranges from two to eight locations.

When a vertex shader input variable declared using an array type is assigned a location, its values are taken from consecutive input attributes starting with the corresponding `VkVertexInputAttributeDescription::location`. The number of attributes and components assigned to each element are determined according to the data type of the array elements and Component decoration (if any) specified in the declaration of the array, as described above. Each element of the array, in order, is assigned to consecutive locations, but all at the same specified component within each location.

Only input variables declared with the data types and component decorations as specified above are supported. Location aliasing is causing two variables to have the same location number. Component aliasing is assigning the same (or overlapping) component number for two location aliases. Location aliasing is allowed only if it does not cause component aliasing. Further, when
location aliasing, the aliases sharing the location must all have the same SPIR-V floating-point component type or all have the same width integer-type components.

# 21.2. Vertex Input Description

Applications specify vertex input attribute and vertex input binding descriptions as part of graphics pipeline creation by setting the `VkGraphicsPipelineCreateInfo::pVertexInputState` pointer to a `VkPipelineVertexInputStateCreateInfo` structure. Alternatively, if the graphics pipeline is created with the `VK_DYNAMIC_STATE_VERTEX_INPUT_EXT` dynamic state enabled, then the vertex input attribute and vertex input binding descriptions are specified dynamically with `vkCmdSetVertexInputEXT`, and the `VkGraphicsPipelineCreateInfo::pVertexInputState` pointer is ignored.

The `VkPipelineVertexInputStateCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineVertexInputStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineVertexInputStateCreateFlags(flags);
    uint32_t vertexBindingDescriptionCount;
    const VkVertexInputBindingDescription* pVertexBindingDescriptions;
    uint32_t vertexAttributeDescriptionCount;
    const VkVertexInputAttributeDescription* pVertexAttributeDescriptions;
} VkPipelineVertexInputStateCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **vertexBindingDescriptionCount** is the number of vertex binding descriptions provided in **pVertexBindingDescriptions**.
- **pVertexBindingDescriptions** is a pointer to an array of `VkVertexInputBindingDescription` structures.
- **vertexAttributeDescriptionCount** is the number of vertex attribute descriptions provided in **pVertexAttributeDescriptions**.
- **pVertexAttributeDescriptions** is a pointer to an array of `VkVertexInputAttributeDescription` structures.

## Valid Usage

- **VUID-VkPipelineVertexInputStateCreateInfo-vertexBindingDescriptionCount-00613**
  - `vertexBindingDescriptionCount` must be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputBindings`

- **VUID-VkPipelineVertexInputStateCreateInfo-vertexAttributeDescriptionCount-00614**
  - `vertexAttributeDescriptionCount` must be less than or equal to `VkPhysicalDeviceLimits`
::maxVertexInputAttributes

- VUID-VkPipelineVertexInputStateCreateInfo-binding-00615
  For every binding specified by each element of `pVertexAttributeDescriptions`, a `VkVertexInputBindingDescription` must exist in `pVertexBindingDescriptions` with the same value of binding.

- VUID-VkPipelineVertexInputStateCreateInfo-pVertexBindingDescriptions-00616
  All elements of `pVertexBindingDescriptions` must describe distinct binding numbers.

- VUID-VkPipelineVertexInputStateCreateInfo-pVertexAttributeDescriptions-00617
  All elements of `pVertexAttributeDescriptions` must describe distinct attribute locations.

### Valid Usage (Implicit)

- VUID-VkPipelineVertexInputStateCreateInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO`.

- VUID-VkPipelineVertexInputStateCreateInfo-pNext-pNext
  `pNext` must be `NULL` or a pointer to a valid instance of `VkPipelineVertexInputDivisorStateCreateInfoEXT`.

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

- VUID-VkPipelineVertexInputStateCreateInfo-flags-zerobitmask
  `flags` must be `0`.

- VUID-VkPipelineVertexInputStateCreateInfo-pVertexBindingDescriptions-parameter
  If `vertexBindingDescriptionCount` is not `0`, `pVertexBindingDescriptions` must be a valid pointer to an array of `vertexBindingDescriptionCount` valid `VkVertexInputBindingDescription` structures.

- VUID-VkPipelineVertexInputStateCreateInfo-pVertexAttributeDescriptions-parameter
  If `vertexAttributeDescriptionCount` is not `0`, `pVertexAttributeDescriptions` must be a valid pointer to an array of `vertexAttributeDescriptionCount` valid `VkVertexInputAttributeDescription` structures.

// Provided by VK_VERSION_1_0

typedef VkFlags VkPipelineVertexInputStateCreateFlags;

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

Each vertex input binding is specified by the `VkVertexInputBindingDescription` structure, defined as:

// Provided by VK_VERSION_1_0

typedef struct VkVertexInputBindingDescription {
    uint32_t binding;
    uint32_t stride;
}
VkVertexInputRate inputRate;
}

VkVertexInputBindingDescription;

- binding is the binding number that this structure describes.
- stride is the byte stride between consecutive elements within the buffer.
- inputRate is a VkVertexInputRate value specifying whether vertex attribute addressing is a function of the vertex index or of the instance index.

## Valid Usage

- VUID-VkVertexInputBindingDescription-binding-00618
  binding must be less than VkPhysicalDeviceLimits::maxVertexInputBindings
- VUID-VkVertexInputBindingDescription-stride-00619
  stride must be less than or equal to VkPhysicalDeviceLimits::maxVertexInputBindingStride

## Valid Usage (Implicit)

- VUID-VkVertexInputBindingDescription-inputRate-parameter
  inputRate must be a valid VkVertexInputRate value

Possible values of VkVertexInputBindingDescription::inputRate, specifying the rate at which vertex attributes are pulled from buffers, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkVertexInputRate {
    VK_VERTEX_INPUT_RATE_VERTEX = 0,
    VK_VERTEX_INPUT_RATE_INSTANCE = 1,
} VkVertexInputRate;
```

- VK_VERTEX_INPUT_RATE_VERTEX specifies that vertex attribute addressing is a function of the vertex index.
- VK_VERTEX_INPUT_RATE_INSTANCE specifies that vertex attribute addressing is a function of the instance index.

Each vertex input attribute is specified by the VkVertexInputAttributeDescription structure, defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkVertexInputAttributeDescription {
    uint32_t location;
    uint32_t binding;
    VkFormat format;
    uint32_t offset;
} VkVertexInputAttributeDescription;
```
• **location** is the shader input location number for this attribute.
• **binding** is the binding number which this attribute takes its data from.
• **format** is the size and type of the vertex attribute data.
• **offset** is a byte offset of this attribute relative to the start of an element in the vertex input 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**
  - `format` must be allowed as a vertex buffer format, as specified by the `VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT` flag in `VkFormatProperties::bufferFeatures` returned by `vkGetPhysicalDeviceFormatProperties`

### Valid Usage ( Implicit)

- **VUID-VkVertexInputAttributeDescription-format-parameter**
  - `format` must be a valid `VkFormat` value

To dynamically set the vertex input attribute and vertex input binding descriptions, call:

```c
// Provided by VK_EXT_vertex_input_dynamic_state
void vkCmdSetVertexInputEXT(VkCommandBuffer commandBuffer, uint32_t vertexBindingDescriptionCount, const VkVertexInputBindingDescription2EXT* pVertexBindingDescriptions, uint32_t vertexAttributeDescriptionCount, const VkVertexInputAttributeDescription2EXT* pVertexAttributeDescriptions);
```

- **commandBuffer** is the command buffer into which the command will be recorded.
- **vertexBindingDescriptionCount** is the number of vertex binding descriptions provided in `pVertexBindingDescriptions`.
- **pVertexBindingDescriptions** is a pointer to an array of `VkVertexInputBindingDescription2EXT` structures.
• **vertexAttributeDescriptionCount** is the number of vertex attribute descriptions provided in `pVertexAttributeDescriptions`.

• **pVertexAttributeDescriptions** is a pointer to an array of `VkVertexInputAttributeDescription2EXT` structures.

This command sets the vertex input attribute and vertex input binding descriptions state for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_VERTEX_INPUT_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkGraphicsPipelineCreateInfo::pVertexInputState` values used to create the currently active pipeline.

If the bound pipeline state object was also created with the `VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT` dynamic state enabled, then `vkCmdBindVertexBuffers2EXT` can be used instead of `vkCmdSetVertexInputEXT` to dynamically set the stride.

### Valid Usage

- **VUID-vkCmdSetVertexInputEXT-None-04790**
  The `vertexInputDynamicState` feature must be enabled

- **VUID-vkCmdSetVertexInputEXT-vertexBindingDescriptionCount-04791**
  `vertexBindingDescriptionCount` must be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputBindings`

- **VUID-vkCmdSetVertexInputEXT-vertexAttributeDescriptionCount-04792**
  `vertexAttributeDescriptionCount` must be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputAttributes`

- **VUID-vkCmdSetVertexInputEXT-binding-04793**
  For every binding specified by each element of `pVertexAttributeDescriptions`, a `VkVertexInputBindingDescription2EXT` must exist in `pVertexBindingDescriptions` with the same value of binding

- **VUID-vkCmdSetVertexInputEXT-pVertexBindingDescriptions-04794**
  All elements of `pVertexBindingDescriptions` must describe distinct binding numbers

- **VUID-vkCmdSetVertexInputEXT-pVertexAttributeDescriptions-04795**
  All elements of `pVertexAttributeDescriptions` must describe distinct attribute locations

### Valid Usage (Implicit)

- **VUID-vkCmdSetVertexInputEXT-commandBuffer-parameter**
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdSetVertexInputEXT-pVertexBindingDescriptions-parameter**
  If `vertexBindingDescriptionCount` is not 0, `pVertexBindingDescriptions` must be a valid pointer to an array of `vertexBindingDescriptionCount` valid `VkVertexInputBindingDescription2EXT` structures

- **VUID-vkCmdSetVertexInputEXT-pVertexAttributeDescriptions-parameter**
If `vertexAttributeDescriptionCount` is not 0, `pVertexAttributeDescriptions` must be a valid pointer to an array of `vertexAttributeDescriptionCount` valid `VkVertexInputAttributeDescription2EXT` structures.

- VUID-vkCmdSetVertexInputEXT-commandBuffer-recording `commandBuffer` must be in the recording state.
- VUID-vkCmdSetVertexInputEXT-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkVertexInputBindingDescription2EXT` structure is defined as:

```c
// Provided by VK_EXT_vertex_input_dynamic_state
typedef struct VkVertexInputBindingDescription2EXT {
    VkStructureType    sType;
    void*               pNext;
    uint32_t            binding;
    uint32_t            stride;
    VkVertexInputRate   inputRate;
    uint32_t            divisor;
} VkVertexInputBindingDescription2EXT;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `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.
- `divisor` is the number of successive instances that will use the same value of the vertex attribute when instanced rendering is enabled. This member can be set to a value other than 1 if the
The **vertexAttributeInstanceRateDivisor** feature is enabled. For example, if the divisor is N, the same vertex attribute will be applied to N successive instances before moving on to the next vertex attribute. The maximum value of **divisor** is implementation-dependent and can be queried using **VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT::maxVertexAttribDivisor**. A value of 0 can be used for the divisor if the **vertexAttributeInstanceRateZeroDivisor** feature is enabled. In this case, the same vertex attribute will be applied to all instances.

### Valid Usage

- **VUID-VkVertexInputBindingDescription2EXT-binding-04796**
  - binding **must** be less than **VkPhysicalDeviceLimits::maxVertexInputBindings**
- **VUID-VkVertexInputBindingDescription2EXT-stride-04797**
  - stride **must** be less than or equal to **VkPhysicalDeviceLimits::maxVertexInputBindingStride**
- **VUID-VkVertexInputBindingDescription2EXT-divisor-04798**
  - If the **vertexAttributeInstanceRateZeroDivisor** feature is not enabled, **divisor** **must** not be 0
- **VUID-VkVertexInputBindingDescription2EXT-divisor-04799**
  - If the **vertexAttributeInstanceRateDivisor** feature is not enabled, **divisor** **must** be 1
- **VUID-VkVertexInputBindingDescription2EXT-divisor-06226**
  - divisor **must** be a value between 0 and **VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT::maxVertexAttribDivisor**, inclusive
- **VUID-VkVertexInputBindingDescription2EXT-divisor-06227**
  - If divisor is not 1 then inputRate **must** be of type **VK_VERTEX_INPUT_RATE_INSTANCE**

### Valid Usage (Implicit)

- **VUID-VkVertexInputBindingDescription2EXT-sType-sType**
  - sType **must** be **VK_STRUCTURE_TYPE_VERTEX_INPUT_BINDING_DESCRIPTION_2_EXT**
- **VUID-VkVertexInputBindingDescription2EXT-inputRate-parameter**
  - inputRate **must** be a valid **VkVertexInputRate** value

The **VkVertexInputAttributeDescription2EXT** structure is defined as:

```c
// Provided by VK_EXT_vertex_input_dynamic_state
typedef struct VkVertexInputAttributeDescription2EXT {
    VkStructureType sType;
    void* pNext;
    uint32_t location;
    uint32_t binding;
    VkFormat format;
    uint32_t offset;
} VkVertexInputAttributeDescription2EXT;
```
• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **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-VkVertexInputAttributeDescription2EXT-location-06228**
  - *location must be less than** `VkPhysicalDeviceLimits::maxVertexInputAttributes`

- **VUID-VkVertexInputAttributeDescription2EXT-binding-06229**
  - *binding must be less than** `VkPhysicalDeviceLimits::maxVertexInputBindings`

- **VUID-VkVertexInputAttributeDescription2EXT-offset-06230**
  - *offset must be less than or equal to** `VkPhysicalDeviceLimits::maxVertexInputAttributeOffset`

- **VUID-VkVertexInputAttributeDescription2EXT-format-04805**
  - *format must be allowed as a vertex buffer format, as specified by the** `VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT` flag in `VkFormatProperties::bufferFeatures` returned by `vkGetPhysicalDeviceFormatProperties`

### Valid Usage (Implicit)

- **VUID-VkVertexInputAttributeDescription2EXT-sType-sType**
  - *sType must be** `VK_STRUCTURE_TYPE_VERTEX_INPUT_ATTRIBUTE_DESCRIPTION_2_EXT`

- **VUID-VkVertexInputAttributeDescription2EXT-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,  // commandBuffer is the command buffer into which the command is recorded.
    uint32_t firstBinding,         // firstBinding is the index of the first vertex input binding whose state is updated by the
    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. If the `nullDescriptor` feature is enabled, elements of `pBuffers` can be `VK_NULL_HANDLE`, and can be used by the vertex shader. If a vertex input attribute is bound to a vertex input binding that is `VK_NULL_HANDLE`, the values taken from memory are considered to be zero, and missing G, B, or A components are filled with `(0,0,1)`.

### 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`*

- **VUID-vkCmdBindVertexBuffers-pBuffers-04002**
  *If an element of `pBuffers` is `VK_NULL_HANDLE`, then the corresponding element of `pOffsets` must be zero*

### 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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Alternatively, to bind vertex buffers, along with their sizes and strides, to a command buffer for use in subsequent drawing commands, call:

```c
// Provided by VK_EXT_extended_dynamic_state
daVoid void vkCmdBindVertexBuffers2EXT(   
    VkCommandBuffer commandBuffer,   
    uint32_t firstBinding,   
    uint32_t bindingCount,   
    const VkBuffer* pBuffers,   
    const VkDeviceSize* pOffsets,   
    const VkDeviceSize* pSizes,   
    const VkDeviceSize* pStrides);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `firstBinding` is the index of the first vertex input binding whose state is updated by the
command.

- **bindingCount** is the number of vertex input bindings whose state is updated by the command.
- **pBuffers** is a pointer to an array of buffer handles.
- **pOffsets** is a pointer to an array of buffer offsets.
- **pSizes** is **NULL** or a pointer to an array of the size in bytes of vertex data bound from **pBuffers**.
- **pStrides** is **NULL** or a pointer to an array of buffer strides.

The values taken from elements i of **pBuffers** and **pOffsets** replace the current state for the vertex input binding **firstBinding** + i, for i in [0, bindingCount). The vertex input binding is updated to start at the offset indicated by **pOffsets[i]** from the start of the buffer **pBuffers[i]**. If **pSizes** is not **NULL** then **pSizes[i]** specifies the bound size of the vertex buffer starting from the corresponding elements of **pBuffers[i]** plus **pOffsets[i]**. All vertex input attributes that use each of these bindings will use these updated addresses in their address calculations for subsequent drawing commands. If the **nullDescriptor** feature is enabled, elements of **pBuffers** can be **VK_NULL_HANDLE**, and can be used by the vertex shader. If a vertex input attribute is bound to a vertex input binding that is **VK_NULL_HANDLE**, the values taken from memory are considered to be zero, and missing G, B, or A components are filled with (0,0,1).

This command also sets the byte strides between consecutive elements within buffer **pBuffers[i]** to the corresponding **pStrides[i]** value when the graphics pipeline is created with **VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT** set in **VkPipelineDynamicStateCreateInfo::pDynamicStates**. Otherwise, strides are specified by the **VkVertexInputBindingDescription::stride** values used to create the currently active pipeline.

If the bound pipeline state object was also created with the **VK_DYNAMIC_STATE_VERTEX_INPUT_EXT** dynamic state enabled then **vkCmdSetVertexInputEXT** can be used instead of **vkCmdBindVertexBuffers2EXT** to set the stride.

### Valid Usage

- **VUID-vkCmdBindVertexBuffers2EXT-firstBinding-03355**
  firstBinding must be less than **VkPhysicalDeviceLimits::maxVertexInputBindings**

- **VUID-vkCmdBindVertexBuffers2EXT-firstBinding-03356**
  The sum of firstBinding and bindingCount must be less than or equal to **VkPhysicalDeviceLimits::maxVertexInputBindings**

- **VUID-vkCmdBindVertexBuffers2EXT-pOffsets-03357**
  All elements of pOffsets must be less than the size of the corresponding element in pBuffers

- **VUID-vkCmdBindVertexBuffers2EXT-pSizes-03358**
  If pSizes is not **NULL**, all elements of pOffsets plus pSizes must be less than or equal to the size of the corresponding element in pBuffers

- **VUID-vkCmdBindVertexBuffers2EXT-pBuffers-03359**
  All elements of pBuffers must have been created with the **VK_BUFFER_USAGE_VERTEX_BUFFER_BIT** flag
• VUID-vkCmdBindVertexBuffers2EXT-pBuffers-03360
  Each element of \textit{pBuffers} that is non-sparse **must** be bound completely and contiguously to a single \texttt{VkDeviceMemory} object

• VUID-vkCmdBindVertexBuffers2EXT-pBuffers-04111
  If the \texttt{nullDescriptor} feature is not enabled, all elements of \textit{pBuffers} **must** not be \texttt{VK_NULL_HANDLE}

• VUID-vkCmdBindVertexBuffers2EXT-pBuffers-04112
  If an element of \textit{pBuffers} is \texttt{VK_NULL_HANDLE}, then the corresponding element of \textit{pOffsets} **must** be zero

• VUID-vkCmdBindVertexBuffers2EXT-pStrides-03362
  If \textit{pStrides} is not \texttt{NULL} each element of \textit{pStrides} **must** be less than or equal to \texttt{VkPhysicalDeviceLimits::maxVertexInputBindingStride}

• VUID-vkCmdBindVertexBuffers2EXT-pStrides-06209
  If \textit{pStrides} is not \texttt{NULL} each element of \textit{pStrides} **must** be either 0 or greater than or equal to the maximum extent of all vertex input attributes fetched from the corresponding binding, where the extent is calculated as the \texttt{VkVertexInputAttributeDescription::offset} plus \texttt{VkVertexInputAttributeDescription::format} size

---

**Valid Usage (Implicit)**

• VUID-vkCmdBindVertexBuffers2EXT-commandBuffer-parameter
  \texttt{commandBuffer} **must** be a valid \texttt{VkCommandBuffer} handle

• VUID-vkCmdBindVertexBuffers2EXT-pBuffers-parameter
  \textit{pBuffers} **must** be a valid pointer to an array of \texttt{bindingCount} valid or \texttt{VK_NULL_HANDLE} \texttt{VkBuffer} handles

• VUID-vkCmdBindVertexBuffers2EXT-pOffsets-parameter
  \textit{pOffsets} **must** be a valid pointer to an array of \texttt{bindingCount} \texttt{VkDeviceSize} values

• VUID-vkCmdBindVertexBuffers2EXT-pSizes-parameter
  If \textit{pSizes} is not \texttt{NULL}, \textit{pSizes} **must** be a valid pointer to an array of \texttt{bindingCount} \texttt{VkDeviceSize} values

• VUID-vkCmdBindVertexBuffers2EXT-pStrides-parameter
  If \textit{pStrides} is not \texttt{NULL}, \textit{pStrides} **must** be a valid pointer to an array of \texttt{bindingCount} \texttt{VkDeviceSize} values

• VUID-vkCmdBindVertexBuffers2EXT-commandBuffer-recording
  \texttt{commandBuffer} **must** be in the \texttt{recording} state

• VUID-vkCmdBindVertexBuffers2EXT-commandBuffer-cmdpool
  The \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from **must** support graphics operations

• VUID-vkCmdBindVertexBuffers2EXT-bindingCount-arraylength
  If any of \textit{pSizes}, or \textit{pStrides} are not \texttt{NULL}, \texttt{bindingCount} **must** be greater than \texttt{0}

• VUID-vkCmdBindVertexBuffers2EXT-commonparent
  Both of \texttt{commandBuffer}, and the elements of \textit{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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

21.3. Vertex Attribute Divisor in Instanced Rendering

If vertexAttributeInstanceRateDivisor feature is enabled and the pNext chain of VkPipelineVertexInputStateCreateInfo includes a VkPipelineVertexInputDivisorStateCreateInfoEXT structure, then that structure controls how vertex attributes are assigned to an instance when instanced rendering is enabled.

The VkPipelineVertexInputDivisorStateCreateInfoEXT structure is defined as:

```c
// Provided by VK_EXT_vertex_attribute_divisor
typedef struct VkPipelineVertexInputDivisorStateCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    uint32_t vertexBindingDivisorCount;
    const VkVertexInputBindingDivisorDescriptionEXT* pVertexBindingDivisors;
} VkPipelineVertexInputDivisorStateCreateInfoEXT;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- vertexBindingDivisorCount is the number of elements in the pVertexBindingDivisors array.
- pVertexBindingDivisors is a pointer to an array of VkVertexInputBindingDivisorDescriptionEXT structures specifying the divisor value for each binding.

Valid Usage (Implicit)

- VUID-VkPipelineVertexInputDivisorStateCreateInfoEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_DIVISOR_STATE_CREATE_INFO_EXT
The individual divisor values per binding are specified using the VkVertexInputBindingDivisorDescriptionEXT structure which is defined as:

```c
// Provided by VK_EXT_vertex_attribute_divisor
typedef struct VkVertexInputBindingDivisorDescriptionEXT {
    uint32_t binding;
    uint32_t divisor;
} VkVertexInputBindingDivisorDescriptionEXT;
```

- **binding** is the binding number for which the divisor is specified.
- **divisor** is the number of successive instances that will use the same value of the vertex attribute when instanced rendering is enabled. For example, if the divisor is N, the same vertex attribute will be applied to N successive instances before moving on to the next vertex attribute. The maximum value of **divisor** is implementation-dependent and can be queried using VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT::maxVertexAttribDivisor. A value of 0 can be used for the divisor if the vertexAttributeInstanceRateZeroDivisor feature is enabled. In this case, the same vertex attribute will be applied to all instances.

If this structure is not used to define a divisor value for an attribute, then the divisor has a logical default value of 1.

**Valid Usage**

- **VUID-VkVertexInputBindingDivisorDescriptionEXT-binding-01869**
  
  **binding** must be less than VkPhysicalDeviceLimits::maxVertexInputBindings

- **VUID-VkVertexInputBindingDivisorDescriptionEXT-vertexAttributeInstanceRateZeroDivisor-02228**
  
  If the vertexAttributeInstanceRateZeroDivisor feature is not enabled, **divisor** must not be 0

- **VUID-VkVertexInputBindingDivisorDescriptionEXT-vertexAttributeInstanceRateDivisor-02229**
  
  If the vertexAttributeInstanceRateDivisor feature is not enabled, **divisor** must be 1

- **VUID-VkVertexInputBindingDivisorDescriptionEXT-divisor-01870**
  
  **divisor** must be a value between 0 and VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT::maxVertexAttribDivisor, inclusive

- **VUID-VkVertexInputBindingDivisorDescriptionEXT-inputRate-01871**
21.4. 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 for vkCmdDrawIndexed), and let instanceIndex be the instance number of the draw (a value between firstInstance and firstInstance+instanceCount).
- Let divisor be the member of VkPipelineVertexInputDivisorStateCreateInfoEXT::pVertexBindingDivisors with VkVertexInputBindingDivisorDescriptionEXT::binding equal to attribDesc.binding.

```plaintext
bufferBindingAddress = buffer[binding].baseAddress + offset[binding];

if (bindingDesc.inputRate == VK_VERTEX_INPUT_RATE_VERTEX)
    vertexOffset = vertexIndex * bindingDesc.stride;
else
    if (divisor == 0)
        vertexOffset = firstInstance * bindingDesc.stride;
    else
        vertexOffset = (firstInstance + ((instanceIndex - firstInstance) / divisor)) * bindingDesc.stride;

attribAddress = bufferBindingAddress + vertexOffset + attribDesc.offset;
```

21.4.1. Vertex Input Extraction

For each attribute, raw data is extracted starting at attribAddress and is converted from the VkVertexInputAttributeDescription's format to either floating-point, unsigned integer, or signed integer based on the base type of the format; the base type of the format must match the base type of the input variable in the shader. The input variable in the shader must be declared as a 64-bit data type if and only if format is a 64-bit data type. If format is a packed format, attribAddress must be a multiple of the size in bytes of the whole attribute data type as described in Packed Formats. Otherwise, attribAddress must be a multiple of the size in bytes of the component type indicated by format (see Formats). For attributes that are not 64-bit data types, each component is converted to the format of the input variable based on its type and size (as defined in the Format Definition...
section for each \texttt{VkFormat}, using the appropriate equations in \textit{16-Bit Floating-Point Numbers, Unsigned 11-Bit Floating-Point Numbers, Unsigned 10-Bit Floating-Point Numbers, Fixed-Point Data Conversion}, and \textit{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 \textit{conversion to RGBA}. The number of components in the vertex shader input variable need not exactly match the number of components in the format. If the vertex shader has fewer components, the extra components are discarded.
Chapter 22. Tessellation

Tessellation involves three pipeline stages. First, a tessellation control shader transforms control points of a patch and can produce per-patch data. Second, a fixed-function tessellator generates multiple primitives corresponding to a tessellation of the patch in (u,v) or (u,v,w) parameter space. Third, a tessellation evaluation shader transforms the vertices of the tessellated patch, for example to compute their positions and attributes as part of the tessellated surface. The tessellator is enabled when the pipeline contains both a tessellation control shader and a tessellation evaluation shader.

22.1. Tessellator

If a pipeline includes both tessellation shaders (control and evaluation), the tessellator consumes each input patch (after vertex shading) and produces a new set of independent primitives (points, lines, or triangles). These primitives are logically produced by subdividing a geometric primitive (rectangle or triangle) according to the per-patch outer and inner tessellation levels written by the tessellation control shader. These levels are specified using the built-in variables TessLevelOuter and TessLevelInner, respectively. This subdivision is performed in an implementation-dependent manner. If no tessellation shaders are present in the pipeline, the tessellator is disabled and incoming primitives are passed through without modification.

The type of subdivision performed by the tessellator is specified by an OpExecutionMode instruction in the tessellation evaluation or tessellation control shader using one of execution modes Triangles, Quads, and IsoLines. Other tessellation-related execution modes can also be specified in either the tessellation control or tessellation evaluation shaders, and if they are specified in both then the modes must be the same.

Tessellation execution modes include:

- **Triangles, Quads, and IsoLines.** These control the type of subdivision and topology of the output primitives. One mode must be set in at least one of the tessellation shader stages.

- **VertexOrderCw** and **VertexOrderCcw.** These control the orientation of triangles generated by the tessellator. One mode must be set in at least one of the tessellation shader stages.

- **PointMode.** Controls generation of points rather than triangles or lines. This functionality defaults to disabled, and is enabled if either shader stage includes the execution mode.

- **SpacingEqual, SpacingFractionalEven, and SpacingFractionalOdd.** Controls the spacing of segments on the edges of tessellated primitives. One mode must be set in at least one of the tessellation shader stages.

- **OutputVertices.** Controls the size of the output patch of the tessellation control shader. One value must be set in at least one of the tessellation shader stages.

For triangles, the tessellator subdivides a triangle primitive into smaller triangles. For quads, the tessellator subdivides a rectangle primitive into smaller triangles. For isolines, the tessellator subdivides a rectangle primitive into a collection of line segments arranged in strips stretching across the rectangle in the u dimension (i.e. the coordinates in TessCoord are of the form (0,x) through (1,x) for all tessellation evaluation shader invocations that share a line).
Each vertex produced by the tessellator has an associated \((u,v,w)\) or \((u,v)\) position in a normalized parameter space, with parameter values in the range \([0,1]\), as illustrated in figures Domain parameterization for tessellation primitive modes (upper-left origin) and Domain parameterization for tessellation primitive modes (lower-left origin). The domain space can have either an upper-left or lower-left origin, selected by the \texttt{domainOrigin} member of \texttt{VkPipelineTessellationDomainOriginStateCreateInfo}.

**Figure 11. Domain parameterization for tessellation primitive modes (upper-left origin)**
In the domain parameterization diagrams, the coordinates illustrate the value of \texttt{TessCoord} at the corners of the domain. The labels on the edges indicate the inner (IL0 and IL1) and outer (OL0 through OL3) tessellation level values used to control the number of subdivisions along each edge of the domain.

For triangles, the vertex's position is a barycentric coordinate \((u,v,w)\), where \(u + v + w = 1.0\), and indicates the relative influence of the three vertices of the triangle on the position of the vertex. For quads and isolines, the position is a \((u,v)\) coordinate indicating the relative horizontal and vertical position of the vertex relative to the subdivided rectangle. The subdivision process is explained in more detail in subsequent sections.

### 22.2. Tessellator Patch Discard

A patch is discarded by the tessellator if any relevant outer tessellation level is less than or equal to zero.

Patches will also be discarded if any relevant outer tessellation level corresponds to a floating-point
NaN (not a number) in implementations supporting NaN.

No new primitives are generated and the tessellation evaluation shader is not executed for patches that are discarded. For Quads, all four outer levels are relevant. For Triangles and IsoLines, only the first three or two outer levels, respectively, are relevant. Negative inner levels will not cause a patch to be discarded; they will be clamped as described below.

### 22.3. Tessellator Spacing

Each of the tessellation levels is used to determine the number and spacing of segments used to subdivide a corresponding edge. The method used to derive the number and spacing of segments is specified by an OpExecutionMode in the tessellation control or tessellation evaluation shader using one of the identifiers SpacingEqual, SpacingFractionalEven, or SpacingFractionalOdd.

If SpacingEqual is used, the floating-point tessellation level is first clamped to \([1, \text{maxLevel}]\), where maxLevel is the implementation-dependent maximum tessellation level (VkPhysicalDeviceLimits::maxTessellationGenerationLevel). The result is rounded up to the nearest integer \(n\), and the corresponding edge is divided into \(n\) segments of equal length in (u,v) space.

If SpacingFractionalEven is used, the tessellation level is first clamped to \([2, \text{maxLevel}]\) and then rounded up to the nearest even integer \(n\). If SpacingFractionalOdd is used, the tessellation level is clamped to \([1, \text{maxLevel} - 1]\) and then rounded up to the nearest odd integer \(n\). If \(n\) is one, the edge will not be subdivided. Otherwise, the corresponding edge will be divided into \(n - 2\) segments of equal length, and two additional segments of equal length that are typically shorter than the other segments. The length of the two additional segments relative to the others will decrease monotonically with \(n - f\), where \(f\) is the clamped floating-point tessellation level. When \(n - f\) is zero, the additional segments will have equal length to the other segments. As \(n - f\) approaches 2.0, the relative length of the additional segments approaches zero. The two additional segments must be placed symmetrically on opposite sides of the subdivided edge. The relative location of these two segments is implementation-dependent, but must be identical for any pair of subdivided edges with identical values of \(f\).

When tessellating triangles or quads using point mode with fractional odd spacing, the tessellator may produce interior vertices that are positioned on the edge of the patch if an inner tessellation level is less than or equal to one. Such vertices are considered distinct from vertices produced by subdividing the outer edge of the patch, even if there are pairs of vertices with identical coordinates.

### 22.4. Tessellation Primitive Ordering

Few guarantees are provided for the relative ordering of primitives produced by tessellation, as they pertain to primitive order.

- The output primitives generated from each input primitive are passed to subsequent pipeline stages in an implementation-dependent order.
- All output primitives generated from a given input primitive are passed to subsequent pipeline stages before any output primitives generated from subsequent input primitives.
22.5. Tessellator Vertex Winding Order

When the tessellator produces triangles (in the Triangles or Quads modes), the orientation of all triangles is specified with an OpExecutionMode of VertexOrderCw or VertexOrderCcw in the tessellation control or tessellation evaluation shaders. If the order is VertexOrderCw, the vertices of all generated triangles will have clockwise ordering in (u,v) or (u,v,w) space. If the order is VertexOrderCcw, the vertices will have counter-clockwise ordering in that space.

If the tessellation domain has an upper-left origin, the vertices of a triangle have counter-clockwise ordering if

\[ a = u_0 v_1 - u_1 v_0 + u_1 v_2 - u_2 v_1 + u_2 v_0 - u_0 v_2 \]

is negative, and clockwise ordering if a is positive. \( u_i \) and \( v_i \) are the \( u \) and \( v \) coordinates in normalized parameter space of the \( i \)th vertex of the triangle. If the tessellation domain has a lower-left origin, the vertices of a triangle have counter-clockwise ordering if \( a \) is positive, and clockwise ordering if \( a \) is negative.

**Note**

The value \( a \) is proportional (with a positive factor) to the signed area of the triangle.

In Triangles mode, even though the vertex coordinates have a \( w \) value, it does not participate directly in the computation of \( a \), being an affine combination of \( u \) and \( v \).

22.6. Triangle Tessellation

If the tessellation primitive mode is Triangles, an equilateral triangle is subdivided into a collection of triangles covering the area of the original triangle. First, the original triangle is subdivided into a collection of concentric equilateral triangles. The edges of each of these triangles are subdivided, and the area between each triangle pair is filled by triangles produced by joining the vertices on the subdivided edges. The number of concentric triangles and the number of subdivisions along each triangle except the outermost is derived from the first inner tessellation level. The edges of the outermost triangle are subdivided independently, using the first, second, and third outer tessellation levels to control the number of subdivisions of the \( u = 0 \) (left), \( v = 0 \) (bottom), and \( w = 0 \) (right) edges, respectively. The second inner tessellation level and the fourth outer tessellation level have no effect in this mode.

If the first inner tessellation level and all three outer tessellation levels are exactly one after clamping and rounding, only a single triangle with \((u,v,w)\) coordinates of \((0,0,1)\), \((1,0,0)\), and \((0,1,0)\) is generated. If the inner tessellation level is one and any of the outer tessellation levels is greater than one, the inner tessellation level is treated as though it were originally specified as \(1 + \varepsilon\) and will result in a two- or three-segment subdivision depending on the tessellation spacing. When used with fractional odd spacing, the three-segment subdivision may produce inner vertices positioned on the edge of the triangle.
If any tessellation level is greater than one, tessellation begins by producing a set of concentric inner triangles and subdividing their edges. First, the three outer edges are temporarily subdivided using the clamped and rounded first inner tessellation level and the specified tessellation spacing, generating \( n \) segments. For the outermost inner triangle, the inner triangle is degenerate — a single point at the center of the triangle — if \( n \) is two. Otherwise, for each corner of the outer triangle, an inner triangle corner is produced at the intersection of two lines extended perpendicular to the corner’s two adjacent edges running through the vertex of the subdivided outer edge nearest that corner. If \( n \) is three, the edges of the inner triangle are not subdivided and it is the final triangle in the set of concentric triangles. Otherwise, each edge of the inner triangle is divided into \( n - 2 \) segments, with the \( n - 1 \) vertices of this subdivision produced by intersecting the inner edge with lines perpendicular to the edge running through the \( n - 1 \) innermost vertices of the subdivision of the outer edge. Once the outermost inner triangle is subdivided, the previous subdivision process repeats itself, using the generated triangle as an outer triangle. This subdivision process is illustrated in Inner Triangle Tessellation.

\[\text{Figure 13. Inner Triangle Tessellation}\]

\[\text{Caption}\]

In the Inner Triangle Tessellation diagram, inner tessellation levels of (a) five and (b) four are shown (not to scale). Solid black circles depict vertices along the edges of the concentric triangles. The edges of inner triangles are subdivided by intersecting the edge with segments perpendicular to the edge running through the \( n - 1 \) innermost vertices of the subdivision of the outer edge. Dotted lines depict edges connecting corresponding vertices on the inner and outer triangle edges.

Once all the concentric triangles are produced and their edges are subdivided, the area between each pair of adjacent inner triangles is filled completely with a set of non-overlapping triangles. In this subdivision, two of the three vertices of each triangle are taken from adjacent vertices on a subdivided edge of one triangle; the third is one of the vertices on the corresponding edge of the other triangle. If the innermost triangle is degenerate (i.e., a point), the triangle containing it is subdivided into six triangles by connecting each of the six vertices on that triangle with the center point. If the innermost triangle is not degenerate, that triangle is added to the set of generated triangles as-is.
After the area corresponding to any inner triangles is filled, the tessellator generates triangles to cover the area between the outermost triangle and the outermost inner triangle. To do this, the temporary subdivision of the outer triangle edge above is discarded. Instead, the $u = 0$, $v = 0$, and $w = 0$ edges are subdivided according to the first, second, and third outer tessellation levels, respectively, and the tessellation spacing. The original subdivision of the first inner triangle is retained. The area between the outer and first inner triangles is completely filled by non-overlapping triangles as described above. If the first (and only) inner triangle is degenerate, a set of triangles is produced by connecting each vertex on the outer triangle edges with the center point. After all triangles are generated, each vertex in the subdivided triangle is assigned a barycentric $(u,v,w)$ coordinate based on its location relative to the three vertices of the outer triangle.

The algorithm used to subdivide the triangular domain in $(u,v,w)$ space into individual triangles is implementation-dependent. However, the set of triangles produced will completely cover the domain, and no portion of the domain will be covered by multiple triangles.

Output triangles are generated with a topology similar to triangle lists, except that the order in which each triangle is generated, and the order in which the vertices are generated for each triangle, are implementation-dependent. However, the order of vertices in each triangle is consistent across the domain as described in Tessellator Vertex Winding Order.

22.7. Quad Tessellation

If the tessellation primitive mode is Quads, a rectangle is subdivided into a collection of triangles covering the area of the original rectangle. First, the original rectangle is subdivided into a regular mesh of rectangles, where the number of rectangles along the $u = 0$ and $u = 1$ (vertical) and $v = 0$ and $v = 1$ (horizontal) edges are derived from the first and second inner tessellation levels, respectively. All rectangles, except those adjacent to one of the outer rectangle edges, are decomposed into triangle pairs. The outermost rectangle edges are subdivided independently, using the first, second, third, and fourth outer tessellation levels to control the number of subdivisions of the $u = 0$ (left), $v = 0$ (bottom), $u = 1$ (right), and $v = 1$ (top) edges, respectively. The area between the inner rectangles of the mesh and the outer rectangle edges are filled by triangles produced by joining the vertices on the subdivided outer edges to the vertices on the edge of the inner rectangle mesh.

If both clamped inner tessellation levels and all four clamped outer tessellation levels are exactly one, only a single triangle pair covering the outer rectangle is generated. Otherwise, if either clamped inner tessellation level is one, that tessellation level is treated as though it was originally specified as $1 + \varepsilon$ and will result in a two- or three-segment subdivision depending on the tessellation spacing. When used with fractional odd spacing, the three-segment subdivision may produce inner vertices positioned on the edge of the rectangle.

If any tessellation level is greater than one, tessellation begins by subdividing the $u = 0$ and $u = 1$ edges of the outer rectangle into $m$ segments using the clamped and rounded first inner tessellation level and the tessellation spacing. The $v = 0$ and $v = 1$ edges are subdivided into $n$ segments using the second inner tessellation level. Each vertex on the $u = 0$ and $v = 0$ edges are joined with the corresponding vertex on the $u = 1$ and $v = 1$ edges to produce a set of vertical and horizontal lines that divide the rectangle into a grid of smaller rectangles. The primitive generator emits a pair of
non-overlapping triangles covering each such rectangle not adjacent to an edge of the outer rectangle. The boundary of the region covered by these triangles forms an inner rectangle, the edges of which are subdivided by the grid vertices that lie on the edge. If either \( m \) or \( n \) is two, the inner rectangle is degenerate, and one or both of the rectangle's edges consist of a single point. This subdivision is illustrated in Figure Inner Quad Tessellation.

![Inner Quad Tessellation](image)

**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 {
    VkStructureType
    const void* pNext;

    // 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 the type of this structure.
• `pNext` is NULL or a pointer to a structure extending this structure.

• `domainOrigin` is a `VkTessellationDomainOrigin` value controlling the origin of the tessellation domain space.

If the `VkPipelineTessellationDomainOriginStateCreateInfo` structure is included in the `pNext` chain of `VkPipelineTessellationStateCreateInfo`, it controls the origin of the tessellation domain. If this structure is not present, it is as if `domainOrigin` was `VK_TESSELLATION_DOMAIN_ORIGIN_UPPER_LEFT`.

### Valid Usage (Implicit)

- `VUID-VkPipelineTessellationDomainOriginStateCreateInfo-sType-sType`  
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_DOMAIN_ORIGIN_STATE_CREATE_INFO`

- `VUID-VkPipelineTessellationDomainOriginStateCreateInfo-domainOrigin-parameter`  
  `domainOrigin` must be a valid `VkTessellationDomainOrigin` value.

The possible tessellation domain origins are specified by the `VkTessellationDomainOrigin` enumeration:

```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, OutputLineStrips, or OutputTriangleStrips execution mode.

If tessellation shading is active but geometry shading is not, the provoking vertex may be any of the vertices in each primitive.

24.2. Primitive Clipping

Primitives are culled against the cull volume and then clipped to the clip volume. In clip coordinates, the view volume is defined by:

\[-W_c \leq x_c \leq W_c,\]
\[-W_c \leq y_c \leq W_c,\]
\[-Z_m \leq z_c \leq W_c\]

where \(Z_m\) is equal to zero.

This view volume can be further restricted by as many as VkPhysicalDeviceLimits::maxClipDistances client-defined half-spaces.
The cull volume is the intersection of up to `VkPhysicalDeviceLimits::maxCullDistances` client-defined half-spaces (if no client-defined cull half-spaces are enabled, culling against the cull volume is skipped).

A shader **must** write a single cull distance for each enabled cull half-space to elements of the `CullDistance` array. If the cull distance for any enabled cull half-space is negative for all of the vertices of the primitive under consideration, the primitive is discarded. Otherwise the primitive is clipped against the clip volume as defined below.

The clip volume is the intersection of up to `VkPhysicalDeviceLimits::maxClipDistances` client-defined half-spaces with the view volume (if no client-defined clip half-spaces are enabled, the clip volume is the view volume).

A shader **must** write a single clip distance for each enabled clip half-space to elements of the `ClipDistance` array. Clip half-space $i$ is then given by the set of points satisfying the inequality

$$c_i(P) \geq 0$$

where $c_i(P)$ is the clip distance $i$ at point $P$. For point primitives, $c_i(P)$ is simply the clip distance for the vertex in question. For line and triangle primitives, per-vertex clip distances are interpolated using a weighted mean, with weights derived according to the algorithms described in sections *Basic Line Segment Rasterization* and *Basic Polygon Rasterization*, using the perspective interpolation equations.

The number of client-defined clip and cull half-spaces that are enabled is determined by the explicit size of the built-in arrays `ClipDistance` and `CullDistance`, respectively, declared as an output in the interface of the entry point of the final shader stage before clipping.

If `VkPipelineRasterizationDepthClipStateCreateInfoEXT` is present in the graphics pipeline state then depth clipping is disabled if `VkPipelineRasterizationDepthClipStateCreateInfoEXT ::depthClipEnable` is `VK_FALSE`. Otherwise, if `VkPipelineRasterizationDepthClipStateCreateInfoEXT` is not present, depth clipping is disabled when `VkPipelineRasterizationStateCreateInfo ::depthClampEnable` is `VK_TRUE`. When depth clipping is disabled, the plane equation

$$z_m \leq z_c \leq w_c$$

(see the clip volume definition above) is ignored by view volume clipping (effectively, there is no near or far plane clipping).

If the primitive under consideration is a point or line segment, then clipping passes it unchanged if its vertices lie entirely within the clip volume.

Possible values of `VkPhysicalDevicePointClippingProperties::pointClippingBehavior`, specifying clipping behavior of a point primitive whose vertex lies outside the clip volume, are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkPointClippingBehavior {
    VK_POINT_CLIPPING_BEHAVIOR_ALL_CLIP_PLANES = 0,
    ...
};
```
VK_POINT_CLIPPING_BEHAVIOR_USER_CLIP_PLANES_ONLY = 1,
} VkPointClippingBehavior;

- **VK_POINT_CLIPPING_BEHAVIOR_ALL_CLIP_PLANES** specifies that the primitive is discarded if the vertex lies outside any clip plane, including the planes bounding the view volume.
- **VK_POINT_CLIPPING_BEHAVIOR_USER_CLIP_PLANES_ONLY** specifies that the primitive is discarded only if the vertex lies outside any user clip plane.

If either of a line segment's vertices lie outside of the clip volume, the line segment may be clipped, with new vertex coordinates computed for each vertex that lies outside the clip volume. A clipped line segment endpoint lies on both the original line segment and the boundary of the clip volume.

This clipping produces a value, $0 \leq t \leq 1$, for each clipped vertex. If the coordinates of a clipped vertex are $P$ and the unclipped line segment's vertex coordinates are $P_1$ and $P_2$, then $t$ satisfies the following equation

$$ P = t P_1 + (1-t) P_2. $$

$t$ is used to clip vertex output attributes as described in Clipping Shader Outputs.

If the primitive is a polygon, it passes unchanged if every one of its edges lies entirely inside the clip volume, and is either clipped or discarded otherwise. If the edges of the polygon intersect the boundary of the clip volume, the intersecting edges are reconnected by new edges that lie along the boundary of the clip volume - in some cases requiring the introduction of new vertices into a polygon.

If a polygon intersects an edge of the clip volume's boundary, the clipped polygon must include a point on this boundary edge.

Primitives rendered with user-defined half-spaces must satisfy a complementarity criterion. Suppose a series of primitives is drawn where each vertex $i$ has a single specified clip distance $d_i$ (or a number of similarly specified clip distances, if multiple half-spaces are enabled). Next, suppose that the same series of primitives are drawn again with each such clip distance replaced by $-d_i$ (and the graphics pipeline is otherwise the same). In this case, primitives must not be missing any pixels, and pixels must not be drawn twice in regions where those primitives are cut by the clip planes.

### 24.3. Clipping Shader Outputs

Next, vertex output attributes are clipped. The output values associated with a vertex that lies within the clip volume are unaffected by clipping. If a primitive is clipped, however, the output values assigned to vertices produced by clipping are clipped.

Let the output values assigned to the two vertices $P_1$ and $P_2$ of an unclipped edge be $c_1$ and $c_2$. The value of $t$ (see Primitive Clipping) for a clipped point $P$ is used to obtain the output value associated with $P$ as
\[
\mathbf{c} = t \mathbf{c}_1 + (1-t) \mathbf{c}_2.
\]

(Multiplying an output value by a scalar means multiplying each of \(x, y, z,\) and \(w\) by the scalar.)

Since this computation is performed in clip space before division by \(w,\) 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 \texttt{must} always be flat shaded. Flat shaded attributes are constant over the primitive being rasterized (see \textit{Basic Line Segment Rasterization} and \textit{Basic Polygon Rasterization}), and no interpolation is performed. The output value \(\mathbf{c}\) 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 \textit{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 \( \text{min} \) and \( \text{max} \) determining a depth range scale value \( p_z \) and a depth range bias value \( o_z \) (defined below). The vertex's framebuffer coordinates \((x_f, y_f, z_f)\) are given by

\[
x_f = \left(\frac{p_x}{2}\right) x_d + o_x
\]

\[
y_f = \left(\frac{p_y}{2}\right) y_d + o_y
\]

\[
z_f = p_z \times z_d + o_z
\]

Multiple viewports are available, numbered zero up to \( \text{VkPhysicalDeviceLimits::maxViewports} \) minus one. The number of viewports used by a pipeline is controlled by the \( \text{viewportCount} \) member of the \( \text{VkPipelineViewportStateCreateInfo} \) structure used in pipeline creation.

\( x_f \) and \( y_f \) have limited precision, where the number of fractional bits retained is specified by \( \text{VkPhysicalDeviceLimits::subPixelPrecisionBits} \). When rasterizing line segments, the number of fractional bits is specified by \( \text{VkPhysicalDeviceLineRasterizationPropertiesEXT::lineSubPixelPrecisionBits} \).

The \( \text{VkPipelineViewportStateCreateInfo} \) structure is defined as:

```cpp
// Provided by VK_VERSION_1_0
typedef struct VkPipelineViewportStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineViewportStateCreateFlags flags;
    uint32_t viewportCount;
    const VkViewport* pViewports;
    uint32_t scissorCount;
    const VkRect2D* pScissors;
} VkPipelineViewportStateCreateInfo;
```

- \( \text{sType} \) is the type of this structure.
- \( \text{pNext} \) is NULL or a pointer to a structure extending this structure.
- \( \text{flags} \) is reserved for future use.
- \( \text{viewportCount} \) is the number of viewports used by the pipeline.
- \( \text{pViewports} \) is a pointer to an array of \( \text{VkViewport} \) structures, defining the viewport transforms. If the viewport state is dynamic, this member is ignored.
- \( \text{scissorCount} \) is the number of \text{scissors} and \text{must} match the number of viewports.
- \( \text{pScissors} \) is a pointer to an array of \( \text{VkRect2D} \) structures defining the rectangular bounds of the
If the multiple viewports feature is not enabled, **viewportCount** must not be greater than 1

If the multiple viewports feature is not enabled, **scissorCount** must not be greater than 1

**viewportCount** must be less than or equal to **VkPhysicalDeviceLimits::maxViewports**

**scissorCount** must be less than or equal to **VkPhysicalDeviceLimits::maxViewports**

The x and y members of **offset** member of any element of **pScissors** must be greater than or equal to 0

Evaluation of (offset.x + extent.width) must not cause a signed integer addition overflow for any element of **pScissors**

Evaluation of (offset.y + extent.height) must not cause a signed integer addition overflow for any element of **pScissors**

If the graphics pipeline is being created without **VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT** and **VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT** set then **scissorCount** and **viewportCount** must be identical

If the graphics pipeline is being created with **VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT** set then **viewportCount** must be 0, otherwise it must be greater than 0

If the graphics pipeline is being created with **VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT** set then **scissorCount** must be 0, otherwise it must be greater than 0

**sType** must be **VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO**

**pNext** must be NULL

**flags** must be 0
To **dynamically set** the viewport count and viewports, call:

```c
// Provided by VK_EXT_extended_dynamic_state
void vkCmdSetViewportWithCountEXT(
    VkCommandBuffer commandBuffer,
    uint32_t viewportCount,
    const VkViewport* pViewports);
```

- **commandBuffer** is the command buffer into which the command will be recorded.
- **viewportCount** specifies the viewport count.
- **pViewports** specifies the viewports to use for drawing.

This command sets the viewport count and viewports state for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the corresponding `VkPipelineViewportStateCreateInfo::viewportCount` and `pViewports` values used to create the currently active pipeline.

### Valid Usage

- **VUID-vkCmdSetViewportWithCountEXT-None-03393**
  The `extendedDynamicState` feature must be enabled

- **VUID-vkCmdSetViewportWithCountEXT-viewportCount-03394**
  The `viewportCount` must be between 1 and `VkPhysicalDeviceLimits::maxViewports`, inclusive

- **VUID-vkCmdSetViewportWithCountEXT-viewportCount-03395**
  If the `multipleViewports` feature is not enabled, `viewportCount` must be 1

### Valid Usage (Implicit)

- **VUID-vkCmdSetViewportWithCountEXT-commandBuffer-parameter**
  The `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdSetViewportWithCountEXT-pViewports-parameter**
  The `pViewports` must be a valid pointer to an array of `viewportCount` valid `VkViewport` structures

- **VUID-vkCmdSetViewportWithCountEXT-commandBuffer-recording**
  The `commandBuffer` must be in the recording state

- **VUID-vkCmdSetViewportWithCountEXT-commandBuffer-cmdpool**
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

- **VUID-vkCmdSetViewportWithCountEXT-viewportCount-arraylength**
  The `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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To dynamically set the scissor count and scissor rectangular bounds, call:

```c
void vkCmdSetScissorWithCountEXT(
    VkCommandBuffer commandBuffer,
    uint32_t scissorCount,
    const VkRect2D* pScissors);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `scissorCount` specifies the scissor count.
- `pScissors` specifies the scissors to use for drawing.

This command sets the scissor count and scissor rectangular bounds state for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the corresponding `VkPipelineViewportStateCreateInfo::scissorCount` and `pScissors` values used to create the currently active pipeline.

Valid Usage

- VUID-vkCmdSetScissorWithCountEXT-None-03396
  The `extendedDynamicState` feature must be enabled
- VUID-vkCmdSetScissorWithCountEXT-scissorCount-03397
  `scissorCount` must be between 1 and `VkPhysicalDeviceLimits::maxViewports`, inclusive
- VUID-vkCmdSetScissorWithCountEXT-scissorCount-03398
  If the `multiple viewports` feature is not enabled, `scissorCount` must be 1
- VUID-vkCmdSetScissorWithCountEXT-x-03399
  The x and y members of `offset` member of any element of `pScissors` must be greater than or equal to 0
Evaluation of \((\text{offset.x} + \text{extent.width})\) must not cause a signed integer addition overflow for any element of \(\text{pScissors}\).

Evaluation of \((\text{offset.y} + \text{extent.height})\) must not cause a signed integer addition overflow for any element of \(\text{pScissors}\).

**Valid Usage (Implicit)**

- VUID-vkCmdSetScissorWithCountEXT-commandBuffer-parameter
  \(\text{commandBuffer}\) must be a valid \(\text{VkCommandBuffer}\) handle

- VUID-vkCmdSetScissorWithCountEXT-pScissors-parameter
  \(\text{pScissors}\) must be a valid pointer to an array of \(\text{scissorCount}\) \(\text{VkRect2D}\) structures

- VUID-vkCmdSetScissorWithCountEXT-commandBuffer-recording
  \(\text{commandBuffer}\) must be in the recording state

- VUID-vkCmdSetScissorWithCountEXT-commandBuffer-cmdpool
  The \(\text{VkCommandPool}\) that \(\text{commandBuffer}\) was allocated from must support graphics operations

- VUID-vkCmdSetScissorWithCountEXT-scissorCount-arraylength
  \(\text{scissorCount}\) must be greater than 0

**Host Synchronization**

- Host access to \(\text{commandBuffer}\) must be externally synchronized

- Host access to the \(\text{VkCommandPool}\) that \(\text{commandBuffer}\) was allocated from must be externally synchronized

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\(\text{VkPipelineViewportStateCreateFlags}\) is a bitmask type for setting a mask, but is currently reserved for future use.

A *pre-rasterization shader stage* can direct each primitive to one of several viewports. The
destination viewport for a primitive is selected by the last active pre-rasterization shader stage that has an output variable decorated with ViewportIndex. The viewport transform uses the viewport corresponding to the value assigned to ViewportIndex, and taken from an implementation-dependent vertex of each primitive. If ViewportIndex is outside the range zero to viewportCount minus one for a primitive, or if the last active pre-rasterization shader stage did not assign a value to ViewportIndex for all vertices of a primitive due to flow control, the values resulting from the viewport transformation of the vertices of such primitives are undefined. If the last pre-rasterization shader stage does not have an output decorated with ViewportIndex, the viewport numbered zero is used by the viewport transformation.

A single vertex can be used in more than one individual primitive, in primitives such as VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP. In this case, the viewport transformation is applied separately for each primitive.

To dynamically set the viewport transformation parameters, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetViewport(
    VkCommandBuffer commandBuffer, 
    uint32_t firstViewport, 
    uint32_t viewportCount, 
    const VkViewport* pViewports);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `firstViewport` is the index of the first viewport whose parameters are updated by the command.
- `viewportCount` is the number of viewports whose parameters are updated by the command.
- `pViewports` is a pointer to an array of VkViewport structures specifying viewport parameters.

This command sets the viewport transformation parameters state for subsequent drawing commands when the graphics pipeline is created with VK_DYNAMIC_STATE_VIEWPORT set in VkPipelineDynamicStateCreateInfo::pDynamicStates. Otherwise, this state is specified by the VkPipelineViewportStateCreateInfo::pViewports values used to create the currently active pipeline.

The viewport parameters taken from element i of pViewports replace the current state for the viewport index firstViewport + i, for i in [0, viewportCount).

### Valid Usage

- **VUID-vkCmdSetViewport-firstViewport-01223**
  The sum of `firstViewport` and `viewportCount` must be between 1 and VkPhysicalDeviceLimits::maxViewports, inclusive

- **VUID-vkCmdSetViewport-firstViewport-01224**
  If the multiple viewports feature is not enabled, `firstViewport` must be 0

- **VUID-vkCmdSetViewport-viewportCount-01225**
  If the multiple viewports feature is not enabled, `viewportCount` must be 1
Valid Usage (Implicit)

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

Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</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:

```
// Provided by VK_VERSION_1_0
typedef struct VkViewport {
    float x;
    float y;
    float width;
    float height;
    float minDepth;
    float maxDepth;
} VkViewport;
```
• $x$ and $y$ are the viewport's upper left corner $(x,y)$.
• width and height are the viewport's width and height, respectively.
• minDepth and maxDepth are the depth range for the viewport.

**Note**
Despite their names, minDepth can be less than, equal to, or greater than maxDepth.

The framebuffer depth coordinate $z$ may be represented using either a fixed-point or floating-point representation. However, a floating-point representation must be used if the depth/stencil attachment has a floating-point depth component. If an m-bit fixed-point representation is used, we assume that it represents each value $\frac{k}{2^m-1}$, where $k \in \{ 0, 1, ... , 2^m-1 \}$, as $k$ (e.g. 1.0 is represented in binary as a string of all ones).

The viewport parameters shown in the above equations are found from these values as

\[
\begin{align*}
o_x &= x + \text{width} / 2 \\
o_y &= y + \text{height} / 2 \\
o_z &= \text{minDepth} \\
p_x &= \text{width} \\
p_y &= \text{height} \\
p_z &= \text{maxDepth} - \text{minDepth}.
\end{align*}
\]

The application can specify a negative term for height, which has the effect of negating the y coordinate in clip space before performing the transform. When using a negative height, the application should also adjust the y value to point to the lower left corner of the viewport instead of the upper left corner. Using the negative height allows the application to avoid having to negate the y component of the Position output from the last pre-rasterization shader stage.

The width and height of the implementation-dependent maximum viewport dimensions must be greater than or equal to the width and height of the largest image which can be created and attached to a framebuffer.

The floating-point viewport bounds are represented with an implementation-dependent precision.

**Valid Usage**
• VUID-VkViewport-width-01770
  width **must** be greater than 0.0

• VUID-VkViewport-width-01771
  width **must** be less than or equal to VkPhysicalDeviceLimits::maxViewportDimensions[0]

• VUID-VkViewport-height-01773
  The absolute value of height **must** be less than or equal to VkPhysicalDeviceLimits::maxViewportDimensions[1]

• VUID-VkViewport-x-01774
  x **must** be greater than or equal to viewportBoundsRange[0]

• VUID-VkViewport-x-01232
  (x + width) **must** be less than or equal to viewportBoundsRange[1]

• VUID-VkViewport-y-01775
  y **must** be greater than or equal to viewportBoundsRange[0]

• VUID-VkViewport-y-01776
  y **must** be less than or equal to viewportBoundsRange[1]

• VUID-VkViewport-y-01777
  (y + height) **must** be greater than or equal to viewportBoundsRange[0]

• VUID-VkViewport-y-01233
  (y + height) **must** be less than or equal to viewportBoundsRange[1]

• VUID-VkViewport-minDepth-01234
  Unless VK_EXT_depth_range_unrestricted extension is enabled minDepth **must** be between 0.0 and 1.0, inclusive

• VUID-VkViewport-maxDepth-01235
  Unless VK_EXT_depth_range_unrestricted extension is enabled 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 depthBiasConstantFactor;
    float depthBiasClamp;
    float depthBiasSlopeFactor;
    float lineWidth;
} VkPipelineRasterizationStateCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
• **depthClampEnable** controls whether to clamp the fragment’s depth values as described in Depth Test. If the pipeline is not created with `VkPipelineRasterizationDepthClipStateCreateInfoEXT` present then enabling depth clamp will also disable clipping primitives to the z planes of the frustrum as described in Primitive Clipping. Otherwise depth clipping is controlled by the state set in `VkPipelineRasterizationDepthClipStateCreateInfoEXT`.

• **rasterizerDiscardEnable** controls whether primitives are discarded immediately before the rasterization stage.

• **polygonMode** is the triangle rendering mode. See `VkPolygonMode`.

• **cullMode** is the triangle facing direction used for primitive culling. See `VkCullModeFlagBits`.

• **frontFace** is a `VkFrontFace` value specifying the front-facing triangle orientation to be used for culling.

• **depthBiasEnable** controls whether to bias fragment depth values.

• **depthBiasConstantFactor** is a scalar factor controlling the constant depth value added to each fragment.

• **depthBiasClamp** is the maximum (or minimum) depth bias of a fragment.

• **depthBiasSlopeFactor** is a scalar factor applied to a fragment’s slope in depth bias calculations.

• **lineWidth** is the width of rasterized line segments.

---

### Valid Usage

- **VUID-VkPipelineRasterizationStateCreateInfo-depthClampEnable-00782**
  If the depth clamping feature is not enabled, `depthClampEnable` must be `VK_FALSE`.

- **VUID-VkPipelineRasterizationStateCreateInfo-polygonMode-01413**
  If the non-solid fill modes 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**
  Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkPipelineRasterizationConservativeStateCreateInfoEXT`, `VkPipelineRasterizationDepthClipStateCreateInfoEXT`, or `VkPipelineRasterizationLineStateCreateInfoEXT`.

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

- **VUID-VkPipelineRasterizationStateCreateInfo-flags-zerobitmask**
  `flags` must be `0`.

- **VUID-VkPipelineRasterizationStateCreateInfo-polygonMode-parameter**
  1011
**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

```c
// 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.

If the `pNext` chain of `VkPipelineRasterizationStateCreateInfo` includes a `VkPipelineRasterizationDepthClipStateCreateInfoEXT` structure, then that structure controls whether depth clipping is enabled or disabled.

The `VkPipelineRasterizationDepthClipStateCreateInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_depth_clip_enable
typedef struct VkPipelineRasterizationDepthClipStateCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkPipelineRasterizationDepthClipStateCreateFlagsEXT flags;
    VkBool32 depthClipEnable;
} VkPipelineRasterizationDepthClipStateCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `depthClipEnable` controls whether depth clipping is enabled as described in [Primitive Clipping](#).

**Valid Usage (Implicit)**

- VUID-VkPipelineRasterizationDepthClipStateCreateInfoEXT-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_DEPTH_CLIP_STATE_CREATE_INFO_EXT`
- VUID-VkPipelineRasterizationDepthClipStateCreateInfoEXT-flags-zerobitmask
  `flags` must be 0

```c
// Provided by VK_EXT_depth_clip_enable
typedef VkFlags VkPipelineRasterizationDepthClipStateCreateFlagsEXT;
```
VkPipelineRasterizationDepthClipStateCreateFlagsEXT is a bitmask type for setting a mask, but is currently reserved for future use.

The VkPipelineMultisampleStateCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineMultisampleStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineMultisampleStateCreateFlags flags;
    VkSampleCountFlagBits rasterizationSamples;
    VkBool32 sampleShadingEnable;
    float minSampleShading;
    const VkSampleMask* pSampleMask;
    VkBool32 alphaToCoverageEnable;
    VkBool32 alphaToOneEnable;
} VkPipelineMultisampleStateCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **rasterizationSamples** is a VkSampleCountFlagBits value specifying the number of samples used in rasterization.
- **sampleShadingEnable** can be used to enable Sample Shading.
- **minSampleShading** specifies a minimum fraction of sample shading if sampleShadingEnable is set to VK_TRUE.
- **pSampleMask** is a pointer to an array of VkSampleMask values used in the sample mask test.
- **alphaToCoverageEnable** controls whether a temporary coverage value is generated based on the alpha component of the fragment’s first color output as specified in the Multisample Coverage section.
- **alphaToOneEnable** controls whether the alpha component of the fragment’s first color output is replaced with one as described in Multisample Coverage.

Each bit in the sample mask is associated with a unique sample index as defined for the coverage mask. Each bit b for mask word w in the sample mask corresponds to sample index i, where i = 32 × 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 sample rate shading feature is not enabled, sampleShadingEnable must be VK_FALSE

- **VUID-VkPipelineMultisampleStateCreateInfo-alphaToOneEnable-00785**
If the alpha to one 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` or a pointer to a valid instance of
  `VkPipelineSampleLocationsStateCreateInfoEXT`

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

- VUID-VkPipelineMultisampleStateCreateInfo-flags-zerobitmask
  `flags` must be `0`

- VUID-VkPipelineMultisampleStateCreateInfo-rasterizationSamples-parameter
  `rasterizationSamples` must be a valid `VkSampleCountFlagBits` value

- VUID-VkPipelineMultisampleStateCreateInfo-pSampleMask-parameter
  If `pSampleMask` is not `NULL`, `pSampleMask` must be a valid pointer to an array of
  `VkSampleMask` values

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

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

The elements of the sample mask array are of type `VkSampleMask`, each representing 32 bits of coverage information:

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

Rasterization only generates fragments which cover one or more pixels inside the framebuffer. Pixels outside the framebuffer are never considered covered in the fragment. Fragments which would be produced by application of any of the primitive rasterization rules described below but which lie outside the framebuffer are not produced, nor are they processed by any later stage of the pipeline, including any of the fragment operations.

Surviving fragments are processed by fragment shaders. Fragment shaders determine associated data for fragments, and can also modify or replace their assigned depth values.
25.1. Discarding Primitives Before Rasterization

Primitives are discarded before rasterization if the `rasterizerDiscardEnable` member of `VkPipelineRasterizationStateCreateInfo` is enabled. When enabled, primitives are discarded after they are processed by the last active shader stage in the pipeline before rasterization.

To **dynamically enable** whether primitives are discarded before the rasterization stage, call:

```c
// Provided by VK_EXT_extended_dynamic_state2
void vkCmdSetRasterizerDiscardEnableEXT(
    VkCommandBuffer commandBuffer,
    VkBool32 rasterizerDiscardEnable);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `rasterizerDiscardEnable` controls whether primitives are discarded immediately before the rasterization stage.

This command sets the discard enable for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineRasterizationStateCreateInfo::rasterizerDiscardEnable` value used to create the currently active pipeline.

**Valid Usage**

- VUID-vkCmdSetRasterizerDiscardEnableEXT-None-04871
  The `extendedDynamicState2` feature **must** be enabled

**Valid Usage (Implicit)**

- VUID-vkCmdSetRasterizerDiscardEnableEXT-commandBuffer-parameter
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- VUID-vkCmdSetRasterizerDiscardEnableEXT-commandBuffer-recording
  `commandBuffer` **must** be in the **recording state**
- VUID-vkCmdSetRasterizerDiscardEnableEXT-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

**Host Synchronization**

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

Within a subpass of a render pass instance, for a given \((x,y,layer,sample)\) sample location, the following operations are guaranteed to execute in rasterization order, for each separate primitive that includes that sample location:

1. Fragment operations, in the order defined
2. Blending, logic operations, and color writes

Execution of these operations for each primitive in a subpass occurs in primitive order.

25.3. Multisampling

Multisampling is a mechanism to antialias all Vulkan primitives: points, lines, and polygons. The technique is to sample all primitives multiple times at each pixel. Each sample in each framebuffer attachment has storage for a color, depth, and/or stencil value, such that per-primitive 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. An implementation may choose to assign the same associated data to more than one sample. The location for evaluating such associated data may be anywhere within the fragment area including the fragment’s center location \((x_f, y_f)\) or any of the sample locations. When `rasterizationSamples` is `VK_SAMPLE_COUNT_1_BIT`, the fragment’s center location must be used. The different associated data values need not all be evaluated at the same location.

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 \texttt{Sample} and \texttt{Input} need not be evaluated independently for each sample.

A \textit{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. Multi-pixel fragments defined by setting the \texttt{fragment shading rate} have one set of samples per pixel. Each set of samples has a number of samples determined by \texttt{VkPipelineMultisampleStateCreateInfo::rasterizationSamples}. Each sample in a set is assigned a unique \textit{sample index} \(i\) in the range \([0, \text{rasterizationSamples})\).

Each sample in a fragment is also assigned a unique \textit{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 \textit{coverage index} is always equal to the \textit{sample index}.

If the \texttt{fragment shading rate} is set, the coverage index \(j\) is determined as a function of the \textit{pixel index} \(p\), the \textit{sample index} \(i\), and the number of rasterization samples \(r\) as:

\[ j = i + r \times ((f_w \times f_h) - 1 - p) \]

where the pixel index \(p\) is determined as a function of the pixel’s framebuffer location \((x,y)\) and the fragment size \((f_w,f_h)\):

\[ p_x = x \mod f_w \]

\[ p_y = y \mod f_h \]

\[ p = p_x + (p_y \times f_w) \]

The table below illustrates the pixel index for multi-pixel fragments:

\textit{Table 28. Pixel indices - 1 wide}
The coverage mask includes $B$ bits packed into $W$ words, defined as:

$$B = n \times \text{rasterizationSamples}$$
\[ W = \lceil \frac{B}{32} \rceil \]

Bit \( b \) in coverage mask word \( w \) is 1 if the sample with coverage index \( j = 32w + b \) is covered, and 0 otherwise.

If the `standardSampleLocations` member of `VkPhysicalDeviceLimits` is `VK_TRUE`, then the sample counts `VK_SAMPLE_COUNT_1_BIT`, `VK_SAMPLE_COUNT_2_BIT`, `VK_SAMPLE_COUNT_4_BIT`, `VK_SAMPLE_COUNT_8_BIT`, and `VK_SAMPLE_COUNT_16_BIT` have sample locations as listed in the following table, with the \( i \)th entry in the table corresponding to sample index \( i \). `VK_SAMPLE_COUNT_32_BIT` and `VK_SAMPLE_COUNT_64_BIT` do not have standard sample locations. Locations are defined relative to an origin in the upper left corner of the fragment.
<table>
<thead>
<tr>
<th>Sample count</th>
<th>Sample Locations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VK_SAMPLE_COUNT_1_BIT</strong></td>
<td>(0.5,0.5)</td>
</tr>
<tr>
<td><strong>VK_SAMPLE_COUNT_2_BIT</strong></td>
<td>(0.75,0.75)</td>
</tr>
<tr>
<td></td>
<td>(0.25,0.25)</td>
</tr>
<tr>
<td><strong>VK_SAMPLE_COUNT_4_BIT</strong></td>
<td>(0.375, 0.125)</td>
</tr>
<tr>
<td></td>
<td>(0.875, 0.375)</td>
</tr>
<tr>
<td></td>
<td>(0.125, 0.625)</td>
</tr>
<tr>
<td></td>
<td>(0.625, 0.875)</td>
</tr>
<tr>
<td><strong>VK_SAMPLE_COUNT_8_BIT</strong></td>
<td>(0.5625, 0.3125)</td>
</tr>
<tr>
<td></td>
<td>(0.4375, 0.6875)</td>
</tr>
<tr>
<td></td>
<td>(0.8125, 0.5625)</td>
</tr>
<tr>
<td></td>
<td>(0.3125, 0.1875)</td>
</tr>
<tr>
<td></td>
<td>(0.1875, 0.8125)</td>
</tr>
<tr>
<td></td>
<td>(0.0625, 0.4375)</td>
</tr>
<tr>
<td></td>
<td>(0.6875, 0.9375)</td>
</tr>
<tr>
<td></td>
<td>(0.9375, 0.0625)</td>
</tr>
<tr>
<td><strong>VK_SAMPLE_COUNT_16_BIT</strong></td>
<td>(0.5625, 0.5625)</td>
</tr>
<tr>
<td></td>
<td>(0.4375, 0.3125)</td>
</tr>
<tr>
<td></td>
<td>(0.3125, 0.625)</td>
</tr>
<tr>
<td></td>
<td>(0.75, 0.4375)</td>
</tr>
<tr>
<td></td>
<td>(0.1875, 0.375)</td>
</tr>
<tr>
<td></td>
<td>(0.625, 0.8125)</td>
</tr>
<tr>
<td></td>
<td>(0.8125, 0.6875)</td>
</tr>
<tr>
<td></td>
<td>(0.6875, 0.1875)</td>
</tr>
<tr>
<td></td>
<td>(0.375, 0.875)</td>
</tr>
<tr>
<td></td>
<td>(0.5, 0.0625)</td>
</tr>
<tr>
<td></td>
<td>(0.25, 0.125)</td>
</tr>
<tr>
<td></td>
<td>(0.125, 0.75)</td>
</tr>
<tr>
<td></td>
<td>(0.0, 0.5)</td>
</tr>
<tr>
<td></td>
<td>(0.9375, 0.25)</td>
</tr>
<tr>
<td></td>
<td>(0.875, 0.9375)</td>
</tr>
<tr>
<td></td>
<td>(0.0625, 0.0)</td>
</tr>
</tbody>
</table>
25.4. Custom Sample Locations

Applications can also control the sample locations used for rasterization.

If the pNext chain of the VkPipelineMultisampleStateCreateInfo structure specified at pipeline creation time includes a VkPipelineSampleLocationsStateCreateInfoEXT structure, then that structure controls the sample locations used when rasterizing primitives with the pipeline.

The VkPipelineSampleLocationsStateCreateInfoEXT structure is defined as:

```c
// Provided by VK_EXT_sample_locations
typedef struct VkPipelineSampleLocationsStateCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkBool32 sampleLocationsEnable;
    VkSampleLocationsInfoEXT sampleLocationsInfo;
} VkPipelineSampleLocationsStateCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `sampleLocationsEnable` controls whether custom sample locations are used. If `sampleLocationsEnable` is VK_FALSE, the default sample locations are used and the values specified in `sampleLocationsInfo` are ignored.
- `sampleLocationsInfo` is the sample locations to use during rasterization if `sampleLocationsEnable` is VK_TRUE and the graphics pipeline is not created with VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT.

Valid Usage (Implicit)

- `VUID-VkPipelineSampleLocationsStateCreateInfoEXT-sType-sType` 
  `sType` must be VK_STRUCTURE_TYPE_PIPELINE_SAMPLE_LOCATIONS_STATE_CREATE_INFO_EXT
- `VUID-VkPipelineSampleLocationsStateCreateInfoEXT-sampleLocationsInfo-parameter` 
  `sampleLocationsInfo` must be a valid `VkSampleLocationsInfoEXT` structure

The VkSampleLocationsInfoEXT structure is defined as:

```c
// Provided by VK_EXT_sample_locations
typedef struct VkSampleLocationsInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkSampleCountFlagBits sampleLocationsPerPixel;
    VkExtent2D sampleLocationGridSize;
    uint32_t sampleLocationsCount;
    const VkSampleLocationEXT* pSampleLocations;
} VkSampleLocationsInfoEXT;
```
• **sType** is the type of this structure.

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

• **sampleLocationsPerPixel** is a `VkSampleCountFlagBits` value specifying the number of sample locations per pixel.

• **sampleLocationGridSize** is the size of the sample location grid to select custom sample locations for.

• **sampleLocationsCount** is the number of sample locations in **pSampleLocations**.

• **pSampleLocations** is a pointer to an array of **sampleLocationsCount** `VkSampleLocationEXT` structures.

This structure **can** be used either to specify the sample locations to be used for rendering or to specify the set of sample locations an image subresource has been last rendered with for the purposes of layout transitions of depth/stencil images created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT`.

The sample locations in **pSampleLocations** specify **sampleLocationsPerPixel** number of sample locations for each pixel in the grid of the size specified in **sampleLocationGridSize**. The sample location for sample i at the pixel grid location (x,y) is taken from `pSampleLocations[(x + y \times sampleLocationGridSize.width) \times sampleLocationsPerPixel + i]`

### Valid Usage

- **VUID-VkSampleLocationsInfoEXT-sampleLocationsPerPixel-01526**  
  **sampleLocationsPerPixel** must be a bit value that is set in `VkPhysicalDeviceSampleLocationsPropertiesEXT::sampleLocationSampleCounts`

- **VUID-VkSampleLocationsInfoEXT-sampleLocationsCount-01527**  
  **sampleLocationsCount** must equal **sampleLocationsPerPixel** \times **sampleLocationGridSize.width** \times **sampleLocationGridSize.height**

### Valid Usage (Implicit)

- **VUID-VkSampleLocationsInfoEXT-sType-sType**  
  **sType** must be `VK_STRUCTURE_TYPE_SAMPLE_LOCATIONS_INFO_EXT`

- **VUID-VkSampleLocationsInfoEXT-pSampleLocations-parameter**  
  If **sampleLocationsCount** is not 0, **pSampleLocations** must be a valid pointer to an array of **sampleLocationsCount** `VkSampleLocationEXT` structures

The `VkSampleLocationEXT` structure is defined as:

```c
// Provided by VK_EXT_sample_locations
typedef struct VkSampleLocationEXT {
    float x;
    float y;
} VkSampleLocationEXT;
```
• $x$ is the horizontal coordinate of the sample’s location.
• $y$ is the vertical coordinate of the sample’s location.

The domain space of the sample location coordinates has an upper-left origin within the pixel in framebuffer space.

The values specified in a `VkSampleLocationEXT` structure are always clamped to the implementation-dependent sample location coordinate range $[\text{sampleLocationCoordinateRange}[0], \text{sampleLocationCoordinateRange}[1]]$ that can be queried using `VkPhysicalDeviceSampleLocationsPropertiesEXT`.

To dynamically set the sample locations used for rasterization, call:

```c
// Provided by VK_EXT_sample_locations
void vkCmdSetSampleLocationsEXT(
    VkCommandBuffer commandBuffer,
    const VkSampleLocationsInfoEXT* pSampleLocationsInfo);
```

• `commandBuffer` is the command buffer into which the command will be recorded.
• `pSampleLocationsInfo` is the sample locations state to set.

This command sets the custom sample locations for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`, and when the `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` property of the bound graphics pipeline is `VK_TRUE`. Otherwise, this state is specified by the `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsInfo` values used to create the currently active pipeline.

### Valid Usage

- **VUID-vkCmdSetSampleLocationsEXT-sampleLocationsPerPixel-01529**
  The `sampleLocationsPerPixel` member of `pSampleLocationsInfo` must equal the `rasterizationSamples` member of the `VkPipelineMultisampleStateCreateInfo` structure the bound graphics pipeline has been created with.

- **VUID-vkCmdSetSampleLocationsEXT-variableSampleLocations-01530**
  If `VkPhysicalDeviceSampleLocationsPropertiesEXT::variableSampleLocations` is `VK_FALSE` then the current render pass must have been begun by specifying a `VkRenderPassSampleLocationsBeginInfoEXT` structure whose `pPostSubpassSampleLocations` member contains an element with a `subpassIndex` matching the current subpass index and the `sampleLocationsInfo` member of that element must match the sample locations state pointed to by `pSampleLocationsInfo`
Valid Usage (Implicit)

- VUID-vkCmdSetSampleLocationsEXT-commandBuffer-parameter commandBuffer must be a valid VkCommandBuffer handle
- VUID-vkCmdSetSampleLocationsEXT-pSampleLocationsInfo-parameter pSampleLocationsInfo must be a valid pointer to a valid VkSampleLocationsInfoEXT structure
- VUID-vkCmdSetSampleLocationsEXT-commandBuffer-recording commandBuffer must be in the recording state
- VUID-vkCmdSetSampleLocationsEXT-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

25.5. Fragment Shading Rates

The features advertised by VkPhysicalDeviceFragmentShadingRateFeaturesKHR allow an application to control the shading rate of a given fragment shader invocation.

The fragment shading rate strongly interacts with Multisampling, and the set of available rates for an implementation may be restricted by sample rate.

To query available shading rates, call:

```c
// Provided by VK_KHR_fragment_shading_rate
VkResult vkGetPhysicalDeviceFragmentShadingRatesKHR(    
VkPhysicalDevice physicalDevice,                      
uint32_t* pFragmentShadingRateCount,                  
VkPhysicalDeviceFragmentShadingRateKHR* pFragmentShadingRates);
```
• physicalDevice is the handle to the physical device whose properties will be queried.

• pFragmentShadingRateCount is a pointer to an integer related to the number of fragment shading rates available or queried, as described below.

• pFragmentShadingRates is either NULL or a pointer to an array of VkPhysicalDeviceFragmentShadingRateKHR structures.

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

The returned array of fragment shading rates must be ordered from largest fragmentSize.width value to smallest, and each set of fragment shading rates with the same fragmentSize.width value must be ordered from largest fragmentSize.height to smallest. Any two entries in the array must not have the same fragmentSize values.

For any entry in the array, the following rules also apply:

• The value of fragmentSize.width must be less than or equal to maxFragmentSize.width.

• The value of fragmentSize.width must be greater than or equal to 1.

• The value of fragmentSize.width must be a power-of-two.

• The value of fragmentSize.height must be less than or equal to maxFragmentSize.height.

• The value of fragmentSize.height must be greater than or equal to 1.

• The value of fragmentSize.height must be a power-of-two.

• The highest sample count in sampleCounts must be less than or equal to maxFragmentShadingRateRasterizationSamples.

• The product of fragmentSize.width, fragmentSize.height, and the highest sample count in sampleCounts must be less than or equal to maxFragmentShadingRateCoverageSamples.

Implementations must support at least the following shading rates:

<table>
<thead>
<tr>
<th>sampleCounts</th>
<th>fragmentSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT</td>
</tr>
<tr>
<td>VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT</td>
</tr>
<tr>
<td>~0</td>
<td>{1,1}</td>
</tr>
</tbody>
</table>

If framebufferColorSampleCounts, includes VK_SAMPLE_COUNT_2_BIT, the required rates must also include VK_SAMPLE_COUNT_2_BIT.

Note
Including the {1,1} fragment size is done for completeness; it has no actual effect.
on the support of rendering without setting the fragment size. All sample counts are supported for this rate.

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

**Valid Usage (Implicit)**

- VUID-vkGetPhysicalDeviceFragmentShadingRatesKHR-physicalDevice-parameter `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceFragmentShadingRatesKHR-pFragmentShadingRateCount-parameter `pFragmentShadingRateCount` must be a valid pointer to a `uint32_t` value
- VUID-vkGetPhysicalDeviceFragmentShadingRatesKHR-pFragmentShadingRates-parameter If the value referenced by `pFragmentShadingRateCount` is not 0, and `pFragmentShadingRates` is not NULL, `pFragmentShadingRates` must be a valid pointer to an array of `pFragmentShadingRateCount` `VkPhysicalDeviceFragmentShadingRateKHR` structures

**Return Codes**

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

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkPhysicalDeviceFragmentShadingRateKHR` structure is defined as

```c
// Provided by VK_KHR_fragment_shading_rate
typedef struct VkPhysicalDeviceFragmentShadingRateKHR {
    VkStructureType sType;
    void* pNext;
    VkSampleCountFlags sampleCounts;
    VkExtent2D fragmentSize;
} VkPhysicalDeviceFragmentShadingRateKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `sampleCounts` is a bitmask of sample counts for which the shading rate described by `fragmentSize` is supported.
- `fragmentSize` is a `VkExtent2D` describing the width and height of a supported shading rate.
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceFragmentShadingRateKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICALDEVICE_FRAGMENTSHADINGRATETER_KHR
- VUID-VkPhysicalDeviceFragmentShadingRateKHR-pNext-pNext
  pNext must be NULL

Fragment shading rates can be set at three points, with the three rates combined to determine the final shading rate.

### 25.5.1. Pipeline Fragment Shading Rate

The pipeline fragment shading rate can be set on a per-draw basis by either setting the rate in a graphics pipeline, or dynamically via vkCmdSetFragmentShadingRateKHR.

The VkPipelineFragmentShadingRateStateCreateInfoKHR structure is defined as:

```c
// Provided by VK_KHR_fragment_shading_rate
typedef struct VkPipelineFragmentShadingRateStateCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkExtent2D fragmentSize;
    VkFragmentShadingRateCombinerOpKHR combinerOps[2];
} VkPipelineFragmentShadingRateStateCreateInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **fragmentSize** specifies a VkExtent2D structure containing the fragment size used to define the pipeline fragment shading rate for drawing commands using this pipeline.
- **combinerOps** specifies a VkFragmentShadingRateCombinerOpKHR value determining how the pipeline, primitive, and attachment shading rates are combined for fragments generated by drawing commands using the created pipeline.

If the pNext chain of VkGraphicsPipelineCreateInfo includes a VkPipelineFragmentShadingRateStateCreateInfoKHR structure, then that structure includes parameters controlling the pipeline fragment shading rate.

If this structure is not present, **fragmentSize** is considered to be equal to (1,1), and both elements of **combinerOps** are considered to be equal to VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR.

Valid Usage (Implicit)

- VUID-VkPipelineFragmentShadingRateStateCreateInfoKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_PIPELINE_FRAGMENT_SHADING_RATE_STATE_CREATE_INFO_KHR
- VUID-VkPipelineFragmentShadingRateStateCreateInfoKHR-combinerOps-parameter
Any given element of `combinerOps` must be a valid `VkFragmentShadingRateCombinerOpKHR` value.

To **dynamically set** the pipeline fragment shading rate and combiner operation, call:

```c
// Provided by VK_KHR_fragment_shading_rate
void vkCmdSetFragmentShadingRateKHR(
    VkCommandBuffer commandBuffer,       // Provided by VK_KHR_fragment_shading_rate
    const VkExtent2D* pFragmentSize,     // Specified in VkPipelineDynamicStateCreateInfo
    const VkFragmentShadingRateCombinerOpKHR combinerOps[2]);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pFragmentSize` specifies the pipeline fragment shading rate for subsequent drawing commands.
- `combinerOps` specifies a `VkFragmentShadingRateCombinerOpKHR` determining how the pipeline, primitive, and attachment shading rates are combined for fragments generated by subsequent drawing commands.

This command sets the pipeline fragment shading rate and combiner operation for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineFragmentShadingRateStateCreateInfoKHR` values used to create the currently active pipeline.

**Valid Usage**

- **VUID-vkCmdSetFragmentShadingRateKHR-pipelineFragmentShadingRate-04507**
  If `pipelineFragmentShadingRate` is not enabled, `pFragmentSize->width` must be 1.

- **VUID-vkCmdSetFragmentShadingRateKHR-pipelineFragmentShadingRate-04508**
  If `pipelineFragmentShadingRate` is not enabled, `pFragmentSize->height` must be 1.

- **VUID-vkCmdSetFragmentShadingRateKHR-pipelineFragmentShadingRate-04509**
  One of `pipelineFragmentShadingRate`, `primitiveFragmentShadingRate`, or `attachmentFragmentShadingRate` must be enabled.

- **VUID-vkCmdSetFragmentShadingRateKHR-primitiveFragmentShadingRate-04510**
  If the `primitiveFragmentShadingRate` feature is not enabled, `combinerOps[0]` must be `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR`.

- **VUID-vkCmdSetFragmentShadingRateKHR-attachmentFragmentShadingRate-04511**
  If the `attachmentFragmentShadingRate` feature is not enabled, `combinerOps[1]` must be `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR`.

- **VUID-vkCmdSetFragmentShadingRateKHR-fragmentSizeNonTrivialCombinerOps-04512**
  If the `fragmentSizeNonTrivialCombinerOps` limit is not supported, elements of `combinerOps` must be either `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR` or `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_REPLACE_KHR`. 

1028
• VUID-vkCmdSetFragmentShadingRateKHR-pFragmentSize-04513
  \texttt{pFragmentSize->width} must be greater than or equal to 1

• VUID-vkCmdSetFragmentShadingRateKHR-pFragmentSize-04514
  \texttt{pFragmentSize->height} must be greater than or equal to 1

• VUID-vkCmdSetFragmentShadingRateKHR-pFragmentSize-04515
  \texttt{pFragmentSize->width} must be a power-of-two value

• VUID-vkCmdSetFragmentShadingRateKHR-pFragmentSize-04516
  \texttt{pFragmentSize->height} must be a power-of-two value

• VUID-vkCmdSetFragmentShadingRateKHR-pFragmentSize-04517
  \texttt{pFragmentSize->width} must be less than or equal to 4

• VUID-vkCmdSetFragmentShadingRateKHR-pFragmentSize-04518
  \texttt{pFragmentSize->height} must be less than or equal to 4

### Valid Usage (Implicit)

• VUID-vkCmdSetFragmentShadingRateKHR-commandBuffer-parameter
  \texttt{commandBuffer} must be a valid \texttt{VkCommandBuffer} handle

• VUID-vkCmdSetFragmentShadingRateKHR-pFragmentSize-parameter
  \texttt{pFragmentSize} must be a valid pointer to a valid \texttt{VkExtent2D} structure

• VUID-vkCmdSetFragmentShadingRateKHR-combinerOps-parameter
  Any given element of \texttt{combinerOps} must be a valid \texttt{VkFragmentShadingRateCombinerOpKHR} value

• VUID-vkCmdSetFragmentShadingRateKHR-commandBuffer-recording
  \texttt{commandBuffer} must be in the \texttt{recording state}

• VUID-vkCmdSetFragmentShadingRateKHR-commandBuffer-cmdpool
  The \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from must support graphics operations

### Host Synchronization

• Host access to \texttt{commandBuffer} must be externally synchronized

• Host access to the \texttt{VkCommandPool} that \texttt{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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
25.5.2. Primitive Fragment Shading Rate

The primitive fragment shading rate can be set via the `PrimitiveShadingRateKHR` built-in in the last active pre-rasterization shader stage. The rate associated with a given primitive is sourced from the value written to `PrimitiveShadingRateKHR` by that primitive's provoking vertex.

25.5.3. Attachment Fragment Shading Rate

The attachment shading rate can be set by including `VkFragmentShadingRateAttachmentInfoKHR` in a subpass to define a fragment shading rate attachment. Each pixel in the framebuffer is assigned an attachment fragment shading rate by the corresponding texel in the fragment shading rate attachment, according to:

\[
    x' = \text{floor}(x / \text{region}_x) \\
    y' = \text{floor}(y / \text{region}_y)
\]

where \(x'\) and \(y'\) are the coordinates of a texel in the fragment shading rate attachment, \(x\) and \(y\) are the coordinates of the pixel in the framebuffer, and \(\text{region}_x\) and \(\text{region}_y\) are the size of the region each texel corresponds to, as defined by the `shadingRateAttachmentTexelSize` member of `VkFragmentShadingRateAttachmentInfoKHR`.

If multiview is enabled and the shading rate attachment has multiple layers, the shading rate attachment texel is selected from the layer determined by the `ViewIndex` built-in. If multiview is disabled, and both the shading rate attachment and the framebuffer have multiple layers, the shading rate attachment texel is selected from the layer determined by the `Layer` built-in. Otherwise, the texel is unconditionally selected from the first layer of the attachment.

The fragment size is encoded into the first component of the identified texel as follows:

\[
    \text{size}_w = 2^{((\text{texel} / 4) \& 3)} \\
    \text{size}_h = 2^{(\text{texel} \& 3)}
\]

where `texel` is the value in the first component of the identified texel, and `\text{size}_w` and `\text{size}_h` are the width and height of the fragment size, decoded from the texel.

If no fragment shading rate attachment is specified, this size is calculated as \(\text{size}_w = \text{size}_h = 1\). Applications must not specify a width or height greater than 4 by this method.

The Fragment Shading Rate enumeration in SPIR-V adheres to the above encoding.

25.5.4. Combining the Fragment Shading Rates

The final rate \((C'_{xy})\) used for fragment shading must be one of the rates returned by `vkGetPhysicalDeviceFragmentShadingRatesKHR` for the sample count used by rasterization.
If any of the following conditions are met, $C_{xy}$ must be set to {1,1}:

- If Sample Shading is enabled.
- The fragmentShadingRateWithSampleMask limit is not supported, and VkPipelineMultisampleStateCreateInfo::pSampleMask contains a zero value in any bit used by fragment operations.
- The fragmentShadingRateWithShaderSampleMask is not supported, and the fragment shader has SampleMask in the input or output interface.
- The fragmentShadingRateWithShaderDepthStencilWrites limit is not supported, and the fragment shader declares the FragDepth or FragStencilRefEXT built-in.
- The fragmentShadingRateWithConservativeRasterization limit is not supported, and VkPipelineRasterizationConservativeStateCreateInfoEXT::conservativeRasterizationMode is not VK_CONSERVATIVE_RASTERIZATION_MODE_DISABLED_EXT.
- The fragmentShadingRateWithFragmentShaderInterlock limit is not supported, and the fragment shader declares any of the fragment shader interlock execution modes.
- The fragmentShadingRateWithCustomSampleLocations limit is not supported, and VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable is VK_TRUE.

Otherwise, each of the specified shading rates are combined and then used to derive the value of $C_{xy}$. As there are three ways to specify shading rates, two combiner operations are specified - between the pipeline and primitive shading rates, and between the result of that and the attachment shading rate.

The equation used for each combiner operation is defined by VkFragmentShadingRateCombinerOpKHR:

```c
// Provided by VK_KHR_fragment_shading_rate
typedef enum VkFragmentShadingRateCombinerOpKHR {
    VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR = 0,
    VK_FRAGMENT_SHADING_RATE_COMBINER_OP_REPLACE_KHR = 1,
    VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MIN_KHR = 2,
    VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MAX_KHR = 3,
    VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MUL_KHR = 4,
} VkFragmentShadingRateCombinerOpKHR;
```

- **VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR** specifies a combiner operation of $\text{combine}(A_{xy},B_{xy}) = A_{xy}$.
- **VK_FRAGMENT_SHADING_RATE_COMBINER_OP_REPLACE_KHR** specifies a combiner operation of $\text{combine}(A_{xy},B_{xy}) = B_{xy}$.
- **VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MIN_KHR** specifies a combiner operation of $\text{combine}(A_{xy},B_{xy}) = \text{min}(A_{xy},B_{xy})$.
- **VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MAX_KHR** specifies a combiner operation of $\text{combine}(A_{xy},B_{xy}) = \text{max}(A_{xy},B_{xy})$.
- **VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MUL_KHR** specifies a combiner operation of $\text{combine}(A_{xy},B_{xy}) = A_{xy} \times B_{xy}$. 

where $\text{combine}(A_{xy}, B_{xy})$ is the combine operation, and $A_{xy}$ and $B_{xy}$ are the inputs to the operation.

If $\text{fragmentShadingRateStrictMultiplyCombiner}$ is $\text{VK_FALSE}$, using $\text{VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MUL_KHR}$ with values of 1 for both $A$ and $B$ in the same dimension results in the value 2 being produced for that dimension. See the definition of $\text{fragmentShadingRateStrictMultiplyCombiner}$ for more information.

These operations are performed in a component-wise fashion.

This is used to generate a combined fragment area using the equation:

$$C_{xy} = \text{combine}(A_{xy}, B_{xy})$$

where $C_{xy}$ is the combined fragment area result, and $A_{xy}$ and $B_{xy}$ are the fragment areas of the fragment shading rates being combined.

Two combine operations are performed, first with $A_{xy}$ equal to the pipeline fragment shading rate and $B_{xy}$ equal to the primitive fragment shading rate, with the combine() operation selected by combinerOps[0]. A second combination is then performed, with $A_{xy}$ equal to the result of the first combination and $B_{xy}$ equal to the attachment fragment shading rate, with the combine() operation selected by combinerOps[1]. The result of the second combination is used as the final fragment shading rate, reported via the $\text{ShadingRateKHR}$ built-in.

Implementations may clamp the $C_{xy}$ result of each combiner operation separately, or only after the second combiner operation.

If the final combined rate is one of the rates returned by $\text{vkGetPhysicalDeviceFragmentShadingRatesKHR}$ for the sample count used by rasterization, $C_{xy'} = C_{xy}$. Otherwise, $C_{xy'}$ is selected from the rates returned by $\text{vkGetPhysicalDeviceFragmentShadingRatesKHR}$ for the sample count used by rasterization. From this list of supported rates, the following steps are applied in order, to select a single value:

1. Keep only rates where $C_{x'} \leq C_x$ and $C_{y'} \leq C_y$.
   
   ◦ Implementations may also keep rates where $C_{x'} \leq C_y$ and $C_{y'} \leq C_x$.

2. Keep only rates with the highest area ($C_{x'} \times C_{y'}$).

3. Keep only rates with the lowest aspect ratio ($C_{x'} + C_{y'}$).

4. In cases where a wide (e.g. 4x1) and tall (e.g. 1x4) rate remain, the implementation may choose either rate. However, it must choose this rate consistently for the same shading rates, and combiner operations for the lifetime of the $\text{VkDevice}$.

### 25.6. Sample Shading

Sample shading can be used to specify a minimum number of unique samples to process for each fragment. If sample shading is enabled an implementation must provide a minimum of $\max(\lceil \text{minSampleShadingFactor} \times \text{totalSamples} \rceil, 1)$ unique associated data for each fragment, where $\text{minSampleShadingFactor}$ is the minimum fraction of sample shading, $\text{totalSamples}$ is the value of $\text{VkPipelineMultisampleStateCreateInfo.rasterizationSamples}$ specified at pipeline creation time.
These are associated with the samples in an implementation-dependent manner. When \( \text{minSampleShadingFactor} \) is 1.0, a separate set of associated data are evaluated for each sample, and each set of values is evaluated at the sample location.

Sample shading is enabled for a graphics pipeline:

- If the interface of the fragment shader entry point of the graphics pipeline includes an input variable decorated with \text{SampleId} or \text{SamplePosition}. In this case \( \text{minSampleShadingFactor} \) takes the value 1.0.

- Else if the \text{sampleShadingEnable} member of the \text{VkPipelineMultisampleStateCreateInfo} structure specified when creating the graphics pipeline is set to \text{VK_TRUE}. In this case \( \text{minSampleShadingFactor} \) takes the value of \text{VkPipelineMultisampleStateCreateInfo}::\text{minSampleShading}.

Otherwise, sample shading is considered disabled.

### 25.7. 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.

#### 25.7.1. 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 \text{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 \(t\):

\[
\begin{align*}
    s &= \frac{x_f - x_0}{x_1 - x_0} \\
    t &= \frac{y_f - y_0}{y_1 - y_0}
\end{align*}
\]
where size is the point's size; \((x_p, y_p)\) is the location at which the point sprite coordinates are evaluated - this \textbf{may} be the framebuffer coordinates of the fragment center, or the location of a sample; and \((x_f, y_f)\) is the exact, unrounded framebuffer coordinate of the vertex for the point.

### 25.8. Line Segments

Line segment rasterization options are controlled by the `VkPipelineRasterizationLineStateCreateInfoEXT` structure.

The `VkPipelineRasterizationLineStateCreateInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_line_rasterization
typedef struct VkPipelineRasterizationLineStateCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkLineRasterizationModeEXT lineRasterizationMode;
    VkBool32 stippledLineEnable;
    uint32_t lineStippleFactor;
    uint16_t lineStipplePattern;
} VkPipelineRasterizationLineStateCreateInfoEXT;
```

- \textbf{sType} is the type of this structure.
- \textbf{pNext} is \textbf{NULL} or a pointer to a structure extending this structure.
- \textbf{lineRasterizationMode} is a `VkLineRasterizationModeEXT` value selecting the style of line rasterization.
- \textbf{stippledLineEnable} enables \textbf{stippled line rasterization}.
- \textbf{lineStippleFactor} is the repeat factor used in stippled line rasterization.
- \textbf{lineStipplePattern} is the bit pattern used in stippled line rasterization.

If \textbf{stippledLineEnable} is \textbf{VK_FALSE}, the values of \textbf{lineStippleFactor} and \textbf{lineStipplePattern} are ignored.

### Valid Usage

- \textbf{VUID-VkPipelineRasterizationLineStateCreateInfoEXT-lineRasterizationMode-02768}
  If \textbf{lineRasterizationMode} is \textbf{VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT}, then the \textbf{rectangularLines} feature \textbf{must} be enabled

- \textbf{VUID-VkPipelineRasterizationLineStateCreateInfoEXT-lineRasterizationMode-02769}
  If \textbf{lineRasterizationMode} is \textbf{VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT}, then the
**bresenhamLines** feature **must** be enabled

- **VUID-VkPipelineRasterizationLineStateCreateInfoEXT-lineRasterizationMode-02770**
  If `lineRasterizationMode` is **VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT**, then the **smoothLines** feature **must** be enabled

- **VUID-VkPipelineRasterizationLineStateCreateInfoEXT-stippledLineEnable-02771**
  If `stippledLineEnable` is **VK_TRUE** and `lineRasterizationMode` is **VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT**, then the **stippledRectangularLines** feature **must** be enabled

- **VUID-VkPipelineRasterizationLineStateCreateInfoEXT-stippledLineEnable-02772**
  If `stippledLineEnable` is **VK_TRUE** and `lineRasterizationMode` is **VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT**, then the **stippledBresenhamLines** feature **must** be enabled

- **VUID-VkPipelineRasterizationLineStateCreateInfoEXT-stippledLineEnable-02773**
  If `stippledLineEnable` is **VK_TRUE** and `lineRasterizationMode` is **VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT**, then the **stippledSmoothLines** feature **must** be enabled

- **VUID-VkPipelineRasterizationLineStateCreateInfoEXT-stippledLineEnable-02774**
  If `stippledLineEnable` is **VK_TRUE** and `lineRasterizationMode` is **VK_LINE_RASTERIZATION_MODE_DEFAULT_EXT**, then the **stippledRectangularLines** feature **must** be enabled and **VkPhysicalDeviceLimits::strictLines** **must** be **VK_TRUE**

---

**Valid Usage (Implicit)**

- **VUID-VkPipelineRasterizationLineStateCreateInfoEXT-sType-02775**
  `sType` **must** be **VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_LINE_STATE_CREATE_INFO_EXT**

- **VUID-VkPipelineRasterizationLineStateCreateInfoEXT-lineRasterizationMode-02776**
  `lineRasterizationMode` **must** be a valid **VkLineRasterizationMode** value

Possible values of **VkPipelineRasterizationLineStateCreateInfoEXT::lineRasterizationMode** are:

```c
// Provided by VK_EXT_line_rasterization
typedef enum VkLineRasterizationModeEXT {
    VK_LINE_RASTERIZATION_MODE_DEFAULT_EXT = 0,
    VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT = 1,
    VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT = 2,
    VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT = 3,
} VkLineRasterizationModeEXT;
```

- **VK_LINE_RASTERIZATION_MODE_DEFAULT_EXT** is equivalent to **VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT** if **VkPhysicalDeviceLimits::strictLines** is **VK_TRUE**, otherwise lines are drawn as non-**strictLines** parallelograms. Both of these modes are defined in **Basic Line Segment Rasterization**.

- **VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT** specifies lines drawn as if they were rectangles
extruded from the line

- **VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT** specifies lines drawn by determining which pixel diamonds the line intersects and exits, as defined in Bresenham Line Segment Rasterization.

- **VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT** specifies lines drawn if they were rectangles extruded from the line, with alpha falloff, as defined in Smooth Lines.

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 wide lines 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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</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.

**25.8.1. Basic Line Segment Rasterization**

If the `lineRasterizationMode` member of `VkPipelineRasterizationLineStateCreateInfoEXT` is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT`, rasterized line segments produce fragments which intersect a rectangle centered on the line segment. Two of the edges are parallel to the specified line segment; each is at a distance of one-half the current width from that segment in directions perpendicular to the direction of the line. The other two edges pass through the line endpoints and are perpendicular to the direction of the specified line segment. Coverage bits that correspond to sample points that intersect the rectangle are 1, other coverage bits are 0.

Next we specify how the data associated with each rasterized fragment are obtained. Let $p_r = (x_d, y_d)$ 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.)

The value of an associated datum $f$ for the fragment, whether it be a shader output or the clip $w$ coordinate, **must** be determined using **perspective interpolation**:

$$f = \frac{(1 - t)f_a / w_a + tf_b / w_b}{(1 - t) / w_a + t / w_b}$$

where $f_a$ and $f_b$ are the data associated with the starting and ending endpoints of the segment, respectively; $w_a$ and $w_b$ are the clip $w$ coordinates of the starting and ending endpoints of the segment, respectively.

Depth values for lines **must** be determined using **linear interpolation**:  

1037
\[ z = (1 - t) z_a + t z_b \]

where \( z_a \) and \( z_b \) are the depth values of the starting and ending endpoints of the segment, respectively.

The **NoPerspective** and **Flat** interpolation decorations can be used with fragment shader inputs to declare how they are interpolated. When neither decoration is applied, perspective interpolation is performed as described above. When the **NoPerspective** decoration is used, linear interpolation is performed in the same fashion as for depth values, as described above. When the **Flat** decoration is used, no interpolation is performed, and outputs are taken from the corresponding input value of the *provoking vertex* corresponding to that primitive.

The above description documents the preferred method of line rasterization, and *must* be used when the implementation advertises the **strictLines** limit in **VkPhysicalDeviceLimits** as **VK_TRUE**.

When **strictLines** is **VK_FALSE**, the edges of the lines are generated as a parallelogram surrounding the original line. The major axis is chosen by noting the axis in which there is the greatest distance between the line start and end points. If the difference is equal in both directions then the X axis is chosen as the major axis. Edges 2 and 3 are aligned to the minor axis and are centered on the endpoints of the line as in **Non strict lines**, and each is *lineWidth* long. Edges 0 and 1 are parallel to the line and connect the endpoints of edges 2 and 3. Coverage bits that correspond to sample points that intersect the parallelogram are 1, other coverage bits are 0.

Samples that fall exactly on the edge of the parallelogram follow the polygon rasterization rules.

Interpolation occurs as if the parallelogram was decomposed into two triangles where each pair of vertices at each end of the line has identical attributes.

![Figure 15. Non strict lines](image-url)

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.

25.8.2. Bresenham Line Segment Rasterization

If lineRasterizationMode is VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT, then the following rules replace the line rasterization rules defined in Basic Line Segment Rasterization.

Non-strict lines may also follow these rasterization rules for non-antialiased lines.

Line segment rasterization begins by characterizing the segment as either x-major or y-major. x-major line segments have slope in the closed interval [-1,1]; all other line segments are y-major (slope is determined by the segment’s endpoints). We specify rasterization only for x-major segments except in cases where the modifications for y-major segments are not self-evident.

Ideally, Vulkan uses a diamond-exit rule to determine those fragments that are produced by rasterizing a line segment. For each fragment $f$ with center at framebuffer coordinates $x_f$ and $y_f$, define a diamond-shaped region that is the intersection of four half planes:

$$R_f = \{(x, y) | |x - x_f| + |y - y_f| < \frac{1}{2}\}$$

Essentially, a line segment starting at $p_a$ and ending at $p_b$ produces those fragments $f$ for which the segment intersects $R_f$, except if $p_b$ is contained in $R_f$.

Figure 16. Visualization of Bresenham’s algorithm
To avoid difficulties when an endpoint lies on a boundary of $R_f$ we (in principle) perturb the supplied endpoints by a tiny amount. Let $p_a$ and $p_b$ have framebuffer coordinates $(x_a, y_a)$ and $(x_b, y_b)$, respectively. Obtain the perturbed endpoints $p_a'$ given by $(x_a, y_a) - (\epsilon, \epsilon^2)$ 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 no more than one.
- For an $x$-major line, two fragments that lie in the same framebuffer-coordinate column must not be produced (for a $y$-major line, two fragments that lie in the same framebuffer-coordinate row must not be produced).
- If two line segments share a common endpoint, and both segments are either $x$-major (both left-to-right or both right-to-left) or $y$-major (both bottom-to-top or both top-to-bottom), then rasterizing both segments must not produce duplicate fragments. Fragments also must not be omitted so as to interrupt continuity of the connected segments.

The actual width $w$ of Bresenham lines is determined by rounding the line width to the nearest integer, clamping it to the implementation-dependent `lineWidthRange` (with both values rounded to the nearest integer), then clamping it to be no less than 1.

Bresenham line segments of width other than one are rasterized by offsetting them in the minor direction (for an $x$-major line, the minor direction is $y$, and for a $y$-major line, the minor direction is $x$) and producing a row or column of fragments in the minor direction. If the line segment has endpoints given by $(x_0, y_0)$ and $(x_1, y_1)$ in framebuffer coordinates, the segment with endpoints $(x_0, y_0 - \frac{w - 1}{2})$ and $(x_1, y_1 - \frac{w - 1}{2})$ is rasterized, but instead of a single fragment, a column of fragments of height $w$ (a row of fragments of length $w$ for a $y$-major segment) is produced at each $x$ ($y$ for $y$-major) location. The lowest fragment of this column is the fragment that would be produced by rasterizing the segment of width 1 with the modified coordinates.

The preferred method of attribute interpolation for a wide line is to generate the same attribute values for all fragments in the row or column described above, as if the adjusted line was used for interpolation and those values replicated to the other fragments, except for `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).

**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).

### 25.8.3. Line Stipple

If the stippledLineEnable member of VkPipelineRasterizationLineStateCreateInfoEXT is VK_TRUE, then lines are rasterized with a line stipple determined by lineStippleFactor and lineStipplePattern. lineStipplePattern is an unsigned 16-bit integer that determines which fragments are to be drawn or discarded when the line is rasterized. lineStippleFactor is a count that is used to modify the effective line stipple by causing each bit in lineStipplePattern to be used lineStippleFactor times.

Line stippling discards certain fragments that are produced by rasterization. The masking is achieved using three parameters: the 16-bit line stipple pattern $p$, the line stipple factor $r$, and an integer stipple counter $s$. Let

$$b = \left\lfloor \frac{s}{r} \right\rfloor \mod 16$$

Then a fragment is produced if the $b$'th bit of $p$ is 1, and discarded otherwise. The bits of $p$ are numbered with 0 being the least significant and 15 being the most significant.

The initial value of $s$ is zero. For VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT lines, $s$ is incremented after production of each fragment of a line segment (fragments are produced in order, beginning at the starting point and working towards the ending point). For VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT and VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT lines, the rectangular region is subdivided into adjacent unit-length rectangles, and $s$ is incremented once for each rectangle. Rectangles with a value of $s$ such that the $b$'th bit of $p$ is zero are discarded. If the last rectangle in a line segment is shorter than unit-length, then the remainder may carry over to the next line segment in the line strip using the same value of $s$ (this is the preferred behavior, for the stipple pattern to appear more consistent through the strip).

$s$ is reset to 0 at the start of each strip (for line strips), and before every line segment in a group of independent segments.

If the line segment has been clipped, then the value of $s$ at the beginning of the line segment is implementation-dependent.

To dynamically set the line stipple state, call:
void vkCmdSetLineStippleEXT(
    VkCommandBuffer commandBuffer,
    uint32_t lineStippleFactor,
    uint16_t lineStipplePattern);

- commandBuffer is the command buffer into which the command will be recorded.
- lineStippleFactor is the repeat factor used in stippled line rasterization.
- lineStipplePattern is the bit pattern used in stippled line rasterization.

This command sets the line stipple state for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_LINE_STIPPLE_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineRasterizationLineStateCreateInfoEXT::lineStippleFactor` and `VkPipelineRasterizationLineStateCreateInfoEXT::lineStipplePattern` values used to create the currently active pipeline.

### Valid Usage

- VUID-vkCmdSetLineStippleEXT-lineStippleFactor-02776
  
  lineStippleFactor must be in the range [1,256]

### Valid Usage (Implicit)

- VUID-vkCmdSetLineStippleEXT-commandBuffer-parameter
  
  commandBuffer must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetLineStippleEXT-commandBuffer-recording
  
  commandBuffer must be in the recording state

- VUID-vkCmdSetLineStippleEXT-commandBuffer-cmdpool
  
  The `VkCommandPool` that commandBuffer was allocated from must support graphics operations

### Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the `VkCommandPool` that commandBuffer was allocated from must be externally synchronized
25.8.4. Smooth Lines

If the `lineRasterizationMode` member of `VkPipelineRasterizationLineStateCreateInfoEXT` is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT`, then lines are considered to be rectangles using the same geometry as for `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT` lines. The rules for determining which pixels are covered are implementation-dependent, and may include nearby pixels where no sample locations are covered or where the rectangle does not intersect the pixel at all. For each pixel that is considered covered, the fragment computes a coverage value that approximates the area of the intersection of the rectangle with the pixel square, and this coverage value is multiplied into the color location 0's alpha value after fragment shading, as described in Multisample Coverage.

Note

The details of the rasterization rules and area calculation are left intentionally vague, to allow implementations to generate coverage and values that are aesthetically pleasing.

25.9. Polygons

A polygon results from the decomposition of a triangle strip, triangle fan or a series of independent triangles. Like points and line segments, polygon rasterization is controlled by several variables in the `VkPipelineRasterizationStateCreateInfo` structure.

25.9.1. 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 \(i \not\equiv 1\) is \((i + 1)\ mod\ n\).

The interpretation of the sign of \(a\) is determined by the `VkPipelineRasterizationStateCreateInfo` ::`frontFace` property of the currently active pipeline. Possible values are:
typedef enum VkFrontFace {
    VK_FRONT_FACE_COUNTER_CLOCKWISE = 0,
    VK_FRONT_FACE_CLOCKWISE = 1,
} VkFrontFace;

- **VK_FRONT_FACE_COUNTER_CLOCKWISE** specifies that a triangle with positive area is considered front-facing.
- **VK_FRONT_FACE_CLOCKWISE** specifies that a triangle with negative area is considered front-facing.

Any triangle which is not front-facing is back-facing, including zero-area triangles.

To **dynamically set** the front face orientation, call:

```c
void vkCmdSetFrontFaceEXT(
    VkCommandBuffer commandBuffer,
    VkFrontFace frontFace);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `frontFace` is a `VkFrontFace` value specifying the front-facing triangle orientation to be used for culling.

This command sets the front face orientation for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_FRONT_FACE_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineRasterizationStateCreateInfo::frontFace` value used to create the currently active pipeline.

### Valid Usage

- VUID-vkCmdSetFrontFaceEXT-None-03383
  The `extendedDynamicState` feature **must** be enabled

### Valid Usage (Implicit)

- VUID-vkCmdSetFrontFaceEXT-commandBuffer-parameter
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- VUID-vkCmdSetFrontFaceEXT-frontFace-parameter
  `frontFace` **must** be a valid `VkFrontFace` value
- VUID-vkCmdSetFrontFaceEXT-commandBuffer-recording
  `commandBuffer` **must** be in the `recording state`
- VUID-vkCmdSetFrontFaceEXT-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Once the orientation of triangles is determined, they are culled according to the `VkPipelineRasterizationStateCreateInfo`::`cullMode` property of the currently active pipeline. Possible values are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkCullModeFlagBits {
    VK_CULL_MODE_NONE = 0,
    VK_CULL_MODE_FRONT_BIT = 0x00000001,
    VK_CULL_MODE_BACK_BIT = 0x00000002,
    VK_CULL_MODE_FRONT_AND_BACK = 0x00000003,
} VkCullModeFlagBits;
```

- `VK_CULL_MODE_NONE` specifies that no triangles are discarded
- `VK_CULL_MODE_FRONT_BIT` specifies that front-facing triangles are discarded
- `VK_CULL_MODE_BACK_BIT` specifies that back-facing triangles are discarded
- `VK_CULL_MODE_FRONT_AND_BACK` specifies that all triangles are discarded.

Following culling, fragments are produced for any triangles which have not been discarded.

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

`VkCullModeFlags` is a bitmask type for setting a mask of zero or more `VkCullModeFlagBits`.

To dynamically set the cull mode, call:
// Provided by VK_EXT_extended_dynamic_state
void vkCmdSetCullModeEXT(
    VkCommandBuffer commandBuffer,  
    VkCullModeFlags cullMode);

- `commandBuffer` is the command buffer into which the command will be recorded.
- `cullMode` specifies the cull mode property to use for drawing.

This command sets the cull mode for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_CULL_MODE_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineRasterizationStateCreateInfo::cullMode` value used to create the currently active pipeline.

### Valid Usage
- VUID-vkCmdSetCullModeEXT-None-03384
  The `extendedDynamicState` feature **must** be enabled

### Valid Usage (Implicit)
- VUID-vkCmdSetCullModeEXT-commandBuffer-parameter
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- VUID-vkCmdSetCullModeEXT-cullMode-parameter
  `cullMode` **must** be a valid combination of `VkCullModeFlagBits` values
- VUID-vkCmdSetCullModeEXT-commandBuffer-recording
  `commandBuffer` **must** be in the recording state
- VUID-vkCmdSetCullModeEXT-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:

\[
a = \frac{A(p \, p_b \, p_c)}{A(p_a \, p_b \, p_c)}, \quad b = \frac{A(p \, p_a \, p_c)}{A(p_a \, p_b \, p_c)}, \quad c = \frac{A(p \, p_a \, p_b)}{A(p_a \, p_b \, p_c)},
\]

where \(A(lmn)\) denotes the area in framebuffer coordinates of the triangle with vertices \(l\), \(m\), and \(n\).

Denote an associated datum at \(p_a\), \(p_b\), or \(p_c\) as \(f_a\), \(f_b\), or \(f_c\), respectively.

The value of an associated datum \(f\) for a fragment produced by rasterizing a triangle, whether it be a shader output or the clip \(w\) coordinate, **must** be determined using perspective interpolation:

\[
f = \frac{a \, f_a / w_a + b \, f_b / w_b + c \, f_c / w_c}{a / w_a + b / w_b + c / w_c}
\]

where \(w_a\), \(w_b\), and \(w_c\) are the clip \(w\) coordinates of \(p_a\), \(p_b\), and \(p_c\) respectively. a, b, and c are the barycentric coordinates of the location at which the data are produced - this **must** be the location of the fragment center or the location of a sample. When `rasterizationSamples` is `VK_SAMPLE_COUNT_1_BIT`, the fragment center **must** be used.

Depth values for triangles **must** be determined using linear interpolation:

\[
z = a \, z_a + b \, z_b + c \, z_c
\]

where \(z_a\), \(z_b\), and \(z_c\) are the depth values of \(p_a\), \(p_b\), and \(p_c\) respectively.

The *NoPerspective* and *Flat* interpolation decorations **can** be used with fragment shader inputs to declare how they are interpolated. When neither decoration is applied, perspective interpolation is performed as described above. When the *NoPerspective* decoration is used, linear interpolation is performed in the same fashion as for depth values, as described above. When the *Flat* decoration is used, no interpolation is performed, and outputs are taken from the corresponding input value of
the provoking vertex corresponding to that primitive.

For a polygon with more than three edges, such as are produced by clipping a triangle, a convex combination of the values of the datum at the polygon’s vertices must be used to obtain the value assigned to each fragment produced by the rasterization algorithm. That is, it must be the case that at every fragment

\[ f = \sum_{i=1}^{n} a_i f_i \]

where \( n \) is the number of vertices in the polygon and \( f_i \) is the value of \( f \) at vertex \( i \). For each \( i \), \( 0 \leq a_i \leq 1 \) and \( \sum_{i=1}^{n} a_i = 1 \). The values of \( a_i \) may differ from fragment to fragment, but at vertex \( i \), \( a_i = 1 \) and \( a_j = 0 \) for \( j \neq i \).

Note

One algorithm that achieves the required behavior is to triangulate a polygon (without adding any vertices) and then treat each triangle individually as already discussed. A scan-line rasterizer that linearly interpolates data along each edge and then linearly interpolates data across each horizontal span from edge to edge also satisfies the restrictions (in this case the numerator and denominator of perspective interpolation are iterated independently, and a division is performed for each fragment).

25.9.2. 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.

25.9.3. 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 \( b \) that is computed for that polygon.
**Depth Bias Enable**

The depth bias computation is enabled by the `depthBiasEnable` set with `vkCmdSetDepthBiasEnableEXT`, or the corresponding `VkPipelineRasterizationStateCreateInfo`::`depthBiasEnable` value used to create the currently active pipeline. If the depth bias enable is `VK_FALSE`, no bias is applied and the fragment’s depth values are unchanged.

To **dynamically enable** whether to bias fragment depth values, call:

```c
// Provided by VK_EXT_extended_dynamic_state2
void vkCmdSetDepthBiasEnableEXT(
    VkCommandBuffer commandBuffer,       // commandBuffer is the command buffer into which the command will be recorded.
    VkBool32 depthBiasEnable);           // depthBiasEnable controls whether to bias fragment depth values.
```

This command sets the depth bias enable for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT` set in `VkPipelineDynamicStateCreateInfo`::`pDynamicStates`. Otherwise, this state is specified by the `VkPipelineRasterizationStateCreateInfo`::`depthBiasEnable` value used to create the currently active pipeline.

**Valid Usage**

- VUID-vkCmdSetDepthBiasEnableEXT-None-04872
  The extendedDynamicState2 feature **must** be enabled

**Valid Usage (Implicit)**

- VUID-vkCmdSetDepthBiasEnableEXT-commandBuffer-parameter
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetDepthBiasEnableEXT-commandBuffer-recording
  `commandBuffer` **must** be in the recording state

- VUID-vkCmdSetDepthBiasEnableEXT-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Depth Bias Computation**

The depth bias depends on three parameters:

- `depthBiasSlopeFactor` scales the maximum depth slope $m$ of the polygon
- `depthBiasConstantFactor` scales the minimum resolvable difference $r$ of the depth attachment
- the scaled terms are summed to produce a value which is then clamped to a minimum or maximum value specified by `depthBiasClamp`

`depthBiasSlopeFactor`, `depthBiasConstantFactor`, and `depthBiasClamp` can each be positive, negative, or zero. These parameters are set as described for `vkCmdSetDepthBias` below.

The maximum depth slope $m$ of a triangle is

$$m = \sqrt{\left(\frac{\partial z_f}{\partial x_f}\right)^2 + \left(\frac{\partial z_f}{\partial y_f}\right)^2}$$

where $(x_0, y_0, z_0)$ is a point on the triangle. $m$ may be approximated as

$$m = \max\left(|\frac{\partial z_f}{\partial x_f}|, |\frac{\partial z_f}{\partial y_f}|\right).$$

The minimum resolvable difference $r$ is a parameter that depends on the depth attachment representation. It is the smallest difference in framebuffer coordinate $z$ values that is guaranteed to remain distinct throughout polygon rasterization and in the depth attachment. All pairs of fragments generated by the rasterization of two polygons with otherwise identical vertices, but $z_f$ values that differ by $r$, will have distinct depth values.

For fixed-point depth attachment representations, $r$ is constant throughout the range of the entire depth attachment. Its value is implementation-dependent but **must** be at most

$$r = 2 \times 2^n$$

for an $n$-bit buffer. For floating-point depth attachment, there is no single minimum resolvable difference. In this case, the minimum resolvable difference for a given polygon is dependent on the maximum exponent, $e$, in the range of $z$ values spanned by the primitive. If $n$ is the number of bits in the floating-point mantissa, the minimum resolvable difference, $r$, for the given primitive is defined as...
If no depth attachment is present, \( r \) is undefined.

The bias value \( o \) for a polygon is

\[
o = \begin{cases} 
  m \times \text{depthBiasSlopeFactor} + r \times \text{depthBiasConstantFactor} & \text{depthBiasClamp} = 0 \text{ or } \text{NaN} \\
  \min(x, \text{depthBiasClamp}) & \text{depthBiasClamp} > 0 \\
  \max(x, \text{depthBiasClamp}) & \text{depthBiasClamp} < 0 
\end{cases}
\]

\( m \) is computed as described above. If the depth 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.

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);
```

- \textit{commandBuffer} is the command buffer into which the command will be recorded.
- \textit{depthBiasConstantFactor} is a scalar factor controlling the constant depth value added to each fragment.
- \textit{depthBiasClamp} is the maximum (or minimum) depth bias of a fragment.
- \textit{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 \texttt{VK_DYNAMIC_STATE_DEPTH_BIAS} set in \texttt{VkPipelineDynamicStateCreateInfo::pDynamicStates}. Otherwise, this state is specified by the corresponding \texttt{VkPipelineInputAssemblyStateCreateInfo::depthBiasConstantFactor}, \texttt{depthBiasClamp}, and \texttt{depthBiasSlopeFactor} values used to create the currently active pipeline.

### Valid Usage

- \textbf{VUID-vkCmdSetDepthBias-depthBiasClamp-00790}
  If the **depth bias clamping** feature is not enabled, \texttt{depthBiasClamp must} be \( 0.0 \)

### Valid Usage (Implicit)

- \textbf{VUID-vkCmdSetDepthBias-commandBuffer-parameter}
  \texttt{commandBuffer must} be a valid \texttt{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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

25.9.4. Conservative Rasterization

Polygon rasterization can be made conservative by setting `conservativeRasterizationMode` to `VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT` or `VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT` in `VkPipelineRasterizationConservativeStateCreateInfoEXT`. The state is set by adding this structure to the `pNext` chain of a `VkPipelineRasterizationStateCreateInfo` structure when creating the graphics pipeline. Enabling these modes also affects line and point rasterization if the implementation sets `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::conservativePointAndLineRasterization` to `VK_TRUE`.

`VkPipelineRasterizationConservativeStateCreateInfoEXT` is defined as:

```c
// Provided by VK_EXT_conservative_rasterization
typedef struct VkPipelineRasterizationConservativeStateCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkPipelineRasterizationConservativeStateCreateFlagsEXT flags;
    VkConservativeRasterizationModeEXT conservativeRasterizationMode;
    float extraPrimitiveOverestimationSize;
} VkPipelineRasterizationConservativeStateCreateInfoEXT;
```
• `sType` is the type of this structure.
• `pNext` is `NULL` or a pointer to a structure extending this structure.
• `flags` is reserved for future use.
• `conservativeRasterizationMode` is the conservative rasterization mode to use.
• `extraPrimitiveOverestimationSize` is the extra size in pixels to increase the generating primitive during conservative rasterization at each of its edges in X and Y equally in screen space beyond the base overestimation specified in `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::primitiveOverestimationSize`.

### Valid Usage

- `VUID-VkPipelineRasterizationConservativeStateCreateInfoEXT-extraPrimitiveOverestimationSize-01769`
  
  `extraPrimitiveOverestimationSize` must be in the range of 0.0 to ` VkPhysicalDeviceConservativeRasterizationPropertiesEXT::maxExtraPrimitiveOverestimationSize` inclusive.

### Valid Usage (Implicit)

- `VUID-VkPipelineRasterizationConservativeStateCreateInfoEXT-sType-sType`
  
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_CONSERVATIVE_STATE_CREATE_INFO_EXT`.

- `VUID-VkPipelineRasterizationConservativeStateCreateInfoEXT-flags-zerobitmask`
  
  `flags` must be 0.

- `VUID-VkPipelineRasterizationConservativeStateCreateInfoEXT-conservativeRasterizationMode-parameter`
  
  `conservativeRasterizationMode` must be a valid `VkConservativeRasterizationModeEXT` value.

// Provided by VK_EXT_conservative_rasterization

typedef `VkFlags` `VkPipelineRasterizationConservativeStateCreateFlagsEXT`;

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

### Possible values of `VkPipelineRasterizationConservativeStateCreateInfoEXT::conservativeRasterizationMode`, specifying the conservative rasterization mode are:

// Provided by VK_EXT_conservative_rasterization

typedef enum `VkConservativeRasterizationModeEXT` {
  `VK_CONSERVATIVE_RASTERIZATION_MODE_DISABLED_EXT` = 0,
  `VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT` = 1,
};
• **VK_CONSERVATIVE_RASTERIZATION_MODE_DISABLED_EXT** specifies that conservative rasterization is disabled and rasterization proceeds as normal.

• **VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT** specifies that conservative rasterization is enabled in overestimation mode.

• **VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT** specifies that conservative rasterization is enabled in underestimation mode.

When overestimate conservative rasterization is enabled, rather than evaluating coverage at individual sample locations, a determination is made of whether any portion of the pixel (including its edges and corners) is covered by the primitive. If any portion of the pixel is covered, then all bits of the **coverage mask** for the fragment corresponding to that pixel are enabled.

For the purposes of evaluating which pixels are covered by the primitive, implementations can increase the size of the primitive by up to **VkPhysicalDeviceConservativeRasterizationPropertiesEXT::primitiveOverestimationSize** pixels at each of the primitive edges. This may increase the number of fragments generated by this primitive and represents an overestimation of the pixel coverage.

This overestimation size can be increased further by setting the **extraPrimitiveOverestimationSize** value above 0.0 in steps of **VkPhysicalDeviceConservativeRasterizationPropertiesEXT::extraPrimitiveOverestimationSizeGranularity** up to and including **VkPhysicalDeviceConservativeRasterizationPropertiesEXT::extraPrimitiveOverestimationSize**. This will: further increase the number of fragments generated by this primitive.

The actual precision of the overestimation size used for conservative rasterization may vary between implementations and produce results that only approximate the **primitiveOverestimationSize** and **extraPrimitiveOverestimationSizeGranularity** properties.

For triangles if **VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT** is enabled, fragments will be generated if the primitive area covers any portion of any pixel inside the fragment area, including their edges or corners. The tie-breaking rule described in **Basic Polygon Rasterization** does not apply during conservative rasterization and coverage is set for all fragments generated from shared edges of polygons. Degenerate triangles that evaluate to zero area after rasterization, even for pixels containing a vertex or edge of the zero-area polygon, will be culled if **VkPhysicalDeviceConservativeRasterizationPropertiesEXT::degenerateTrianglesRasterized** is VK_FALSE or will generate fragments if **degenerateTrianglesRasterized** is VK_TRUE. The fragment input values for these degenerate triangles take their attribute and depth values from the provoking vertex. Degenerate triangles are considered backfacing and the application can enable backface culling if desired. Triangles that are zero area before rasterization may be culled regardless.

For lines if **VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT** is enabled, and the implementation sets **VkPhysicalDeviceConservativeRasterizationPropertiesEXT::conservativePointAndLineRasterization** to VK_TRUE, fragments will be generated if the line covers any portion of any pixel inside the fragment area, including their edges or corners. Degenerate lines that evaluate to zero length after rasterization will be culled if **VkPhysicalDeviceConservativeRasterizationPropertiesEXT::degenerateLinesRasterized** is VK_FALSE or
will generate fragments if `degenerateLinesRasterized` is `VK_TRUE`. The fragments input values for these degenerate lines take their attribute and depth values from the provoking vertex. Lines that are zero length before rasterization may be culled regardless.

For points if `VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT` is enabled, and the implementation sets `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::conservativePointAndLineRasterization` to `VK_TRUE`, fragments will be generated if the point square covers any portion of any pixel inside the fragment area, including their edges or corners.

When underestimate conservative rasterization is enabled, rather than evaluating coverage at individual sample locations, a determination is made of whether all of the pixel (including its edges and corners) is covered by the primitive. If the entire pixel is covered, then a fragment is generated with all bits of its `coverage mask` corresponding to the pixel enabled, otherwise the pixel is not considered covered even if some portion of the pixel is covered. The fragment is discarded if no pixels inside the fragment area are considered covered.

For triangles, if `VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT` is enabled, fragments will only be generated if any pixel inside the fragment area is fully covered by the generating primitive, including its edges and corners.

For lines, if `VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT` is enabled, fragments will be generated if any pixel inside the fragment area, including its edges and corners, are entirely covered by the line.

For points, if `VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT` is enabled, fragments will only be generated if the point square covers the entirety of any pixel square inside the fragment area, including its edges or corners.

For both overestimate and underestimate conservative rasterization modes a fragment has all of its pixel squares fully covered by the generating primitive must set `FullyCoveredEXT` to `VK_TRUE` if the implementation enables the `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::fullyCoveredFragmentShaderInputVariable` feature.

When setting the fragment shading rate results in fragments covering multiple pixels, coverage for conservative rasterization is still evaluated on a per-pixel basis and may result in fragments with partial coverage. For fragment shader inputs decorated with `FullyCoveredEXT`, a fragment is considered fully covered if and only if all pixels in the fragment are fully covered by the generating primitive.
Chapter 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. Discard rectangles test
2. Scissor test
3. Sample mask test
4. Certain Fragment shading operations:
   - Sample Mask Accesses
   - Depth Replacement
   - Stencil Reference Replacement
   - Interlocked Operations
5. Multisample coverage
6. Depth bounds test
7. Stencil test
8. Depth test
9. Sample counting
10. Coverage reduction

The coverage mask generated by rasterization describes the initial coverage of each sample covered by the fragment. Fragment operations will update the coverage mask to add or subtract coverage where appropriate. If a fragment operation results in all bits of the coverage mask being 0, the fragment is discarded, and no further operations are performed. Fragments can also be programmatically discarded in a fragment shader by executing one of

- OpTerminateInvocation
- OpDemoteToHelperInvocationEXT
- OpKill.

When one of the fragment operations in this chapter is described as “replacing” a fragment shader output, that output is replaced unconditionally, even if no fragment shader previously wrote to that output.

If the fragment shader declares the PostDepthCoverage execution mode, the sample mask test is instead performed after the depth test.

If the fragment shader declares the EarlyFragmentTests execution mode, fragment shading and multisample coverage operations are instead performed after sample counting.
Once all fragment operations have completed, fragment shader outputs for covered color attachment samples pass through framebuffer operations.

## 26.1. Discard Rectangles Test

The discard rectangle test compares the framebuffer coordinates \((x_f, y_f)\) of each sample covered by a fragment against a set of discard rectangles.

Each discard rectangle is defined by a `VkRect2D`. These values are either set by the `VkPipelineDiscardRectangleStateCreateInfoEXT` structure during pipeline creation, or dynamically by the `vkCmdSetDiscardRectangleEXT` command.

A given sample is considered inside a discard rectangle if the \(x_f\) is in the range \([VkRect2D::offset.x, VkRect2D::offset.x + VkRect2D::extent.x)\), and \(y_f\) is in the range \([VkRect2D::offset.y, VkRect2D::offset.y + VkRect2D::extent.y)\). If the test is set to be inclusive, samples that are not inside any of the discard rectangles will have their coverage set to 0. If the test is set to be exclusive, samples that are inside any of the discard rectangles will have their coverage set to 0.

If no discard rectangles are specified, the coverage mask is unmodified by this operation.

### 26.1.1. `VkPipelineDiscardRectangleStateCreateInfoEXT` Structure

The `VkPipelineDiscardRectangleStateCreateInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_discard_rectangles
typedef struct VkPipelineDiscardRectangleStateCreateInfoEXT {
    VkStructureType     sType;
    const void*         pNext;
    VkPipelineDiscardRectangleStateCreateFlagsEXT flags;
    VkDiscardRectangleModeEXT discardRectangleMode;
    uint32_t            discardRectangleCount;
    const VkRect2D*     pDiscardRectangles;
} VkPipelineDiscardRectangleStateCreateInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **discardRectangleMode** is a `VkDiscardRectangleModeEXT` value determining whether the discard rectangle test is inclusive or exclusive.
- **discardRectangleCount** is the number of discard rectangles to use.
- **pDiscardRectangles** is a pointer to an array of `VkRect2D` structures defining discard rectangles.

If the `VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT` dynamic state is enabled for a pipeline, the `pDiscardRectangles` member is ignored.

When this structure is included in the `pNext` chain of `VkGraphicsPipelineCreateInfo`, it defines parameters of the discard rectangle test. If this structure is not included in the `pNext` chain, it is equivalent to specifying this structure with a `discardRectangleCount` of 0.
Valid Usage

- **VUID-VkPipelineDiscardRectangleStateCreateInfoEXT-discardRectangleCount-00582**
  - `discardRectangleCount` must be less than or equal to `VkPhysicalDeviceDiscardRectanglePropertiesEXT::maxDiscardRectangles`

Valid Usage (Implicit)

- **VUID-VkPipelineDiscardRectangleStateCreateInfoEXT-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_DISCARD_RECTANGLE_STATE_CREATE_INFO_EXT`

- **VUID-VkPipelineDiscardRectangleStateCreateInfoEXT-flags-zero bitmask**
  - `flags` must be `0`

- **VUID-VkPipelineDiscardRectangleStateCreateInfoEXT-discardRectangleMode-parameter**
  - `discardRectangleMode` must be a valid `VkDiscardRectangleModeEXT` value

```c
// Provided by VK_EXT_discard_rectangles
typedef VkFlags VkPipelineDiscardRectangleStateCreateFlagsEXT;
```

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

`VkDiscardRectangleModeEXT` values are:

```c
// Provided by VK_EXT_discard_rectangles
typedef enum VkDiscardRectangleModeEXT {
    VK_DISCARD_RECTANGLE_MODE_INCLUSIVE_EXT = 0,
    VK_DISCARD_RECTANGLE_MODE_EXCLUSIVE_EXT = 1,
} VkDiscardRectangleModeEXT;
```

- `VK_DISCARD_RECTANGLE_MODE_INCLUSIVE_EXT` specifies that the discard rectangle test is inclusive.
- `VK_DISCARD_RECTANGLE_MODE_EXCLUSIVE_EXT` specifies that the discard rectangle test is exclusive.

To dynamically set the discard rectangles, call:

```c
// Provided by VK_EXT_discard_rectangles
void vkCmdSetDiscardRectangleEXT(
    VkCommandBuffer commandBuffer,         commandBuffer,
    uint32_t firstDiscardRectangle,        firstDiscardRectangle,
    uint32_t discardRectangleCount,        discardRectangleCount,
    const VkRect2D* pDiscardRectangles);  pDiscardRectangles);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
• **firstDiscardRectangle** is the index of the first discard rectangle whose state is updated by the command.

• **discardRectangleCount** is the number of discard rectangles whose state are updated by the command.

• **pDiscardRectangles** is a pointer to an array of **VkRect2D** structures specifying discard rectangles.

The discard rectangle taken from element $i$ of **pDiscardRectangles** replace the current state for the discard rectangle at index $firstDiscardRectangle + i$, for $i$ in $[0, discardRectangleCount)$.

This command sets the discard rectangles for subsequent drawing commands when the graphics pipeline is created with **VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT** set in **VkPipelineDynamicStateCreateInfo::pDynamicStates**. Otherwise, this state is specified by the **VkPipelineDiscardRectangleStateCreateInfoEXT::pDiscardRectangles** values used to create the currently active pipeline.

### Valid Usage

- **VUID-vkCmdSetDiscardRectangleEXT-firstDiscardRectangle-00585**
  The sum of **firstDiscardRectangle** and **discardRectangleCount** must be less than or equal to **VkPhysicalDeviceDiscardRectanglePropertiesEXT::maxDiscardRectangles**

- **VUID-vkCmdSetDiscardRectangleEXT-x-00587**
  The $x$ and $y$ member of **offset** in each **VkRect2D** element of **pDiscardRectangles** must be greater than or equal to $0$

- **VUID-vkCmdSetDiscardRectangleEXT-offset-00588**
  Evaluation of ($offset.x + extent.width$) in each **VkRect2D** element of **pDiscardRectangles** must not cause a signed integer addition overflow

- **VUID-vkCmdSetDiscardRectangleEXT-offset-00589**
  Evaluation of ($offset.y + extent.height$) in each **VkRect2D** element of **pDiscardRectangles** must not cause a signed integer addition overflow

### Valid Usage (Implicit)

- **VUID-vkCmdSetDiscardRectangleEXT-commandBuffer-parameter**
  **commandBuffer** must be a valid **VkCommandBuffer** handle

- **VUID-vkCmdSetDiscardRectangleEXT-pDiscardRectangles-parameter**
  **pDiscardRectangles** must be a valid pointer to an array of **discardRectangleCount** **VkRect2D** structures

- **VUID-vkCmdSetDiscardRectangleEXT-commandBuffer-recording**
  **commandBuffer** must be in the recording state

- **VUID-vkCmdSetDiscardRectangleEXT-commandBuffer-cmdpool**
  The **VkCommandPool** that **commandBuffer** was allocated from must support graphics operations

- **VUID-vkCmdSetDiscardRectangleEXT-discardRectangleCount-arraylength**
26.2. Scissor Test

The scissor test compares the framebuffer coordinates \((x_f, y_f)\) of each sample covered by a fragment against a scissor rectangle at the index equal to the fragment’s ViewportIndex.

Each scissor rectangle is defined by a VkRect2D. These values are either set by the VkPipelineViewportStateCreateInfo structure during pipeline creation, or dynamically by the vkCmdSetScissor command.

A given sample is considered inside a scissor rectangle if \(x_f\) is in the range \([\text{VkRect2D}::\text{offset}.x, \text{VkRect2D}::\text{offset}.x + \text{VkRect2D}::\text{extent}.x)\), and \(y_f\) is in the range \([\text{VkRect2D}::\text{offset}.y, \text{VkRect2D}::\text{offset}.y + \text{VkRect2D}::\text{extent}.y)\). 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:

```
// Provided by VK_VERSION_1_0
void vkCmdSetScissor(
    VkCommandBuffer commandBuffer,
    uint32_t firstScissor,
    uint32_t scissorCount,
    const VkRect2D* pScissors);
```

- \text{commandBuffer} is the command buffer into which the command will be recorded.
- \text{firstScissor} is the index of the first scissor whose state is updated by the command.
- \text{scissorCount} is the number of scissors whose rectangles are updated by the command.
- \text{pScissors} is a pointer to an array of \text{VkRect2D} structures defining scissor rectangles.
The scissor rectangles taken from element \(i\) of \(\text{pScissors}\) replace the current state for the scissor index \(\text{firstScissor} + i\), for \(i\) in \([0, \text{scissorCount})\).

This command sets the scissor rectangles for subsequent drawing commands when the graphics pipeline is created with \(\text{VK_DYNAMIC_STATE_SCISSOR}\) set in \(\text{VkPipelineDynamicStateCreateInfo}\) ::\(\text{pDynamicStates}\). Otherwise, this state is specified by the \(\text{VkPipelineViewportStateCreateInfo}\) ::\(\text{pScissors}\) values used to create the currently active pipeline.

### Valid Usage

- **VUID-vkCmdSetScissor-firstScissor-00592**
  The sum of \(\text{firstScissor}\) and \(\text{scissorCount}\) must be between 1 and \(\text{VkPhysicalDeviceLimits::maxViewports}\), inclusive.

- **VUID-vkCmdSetScissor-firstScissor-00593**
  If the multiple viewports feature is not enabled, \(\text{firstScissor}\) must be 0.

- **VUID-vkCmdSetScissor-scissorCount-00594**
  If the multiple viewports feature is not enabled, \(\text{scissorCount}\) must be 1.

- **VUID-vkCmdSetScissor-x-00595**
  The \(x\) and \(y\) members of \(\text{offset}\) member of any element of \(\text{pScissors}\) must be greater than or equal to 0.

- **VUID-vkCmdSetScissor-offset-00596**
  Evaluation of \((\text{offset.x} + \text{extent.width})\) must not cause a signed integer addition overflow for any element of \(\text{pScissors}\).

- **VUID-vkCmdSetScissor-offset-00597**
  Evaluation of \((\text{offset.y} + \text{extent.height})\) must not cause a signed integer addition overflow for any element of \(\text{pScissors}\).

### Valid Usage (Implicit)

- **VUID-vkCmdSetScissor-commandBuffer-parameter**
  \(\text{commandBuffer}\) must be a valid \(\text{VkCommandBuffer}\) handle.

- **VUID-vkCmdSetScissor-pScissors-parameter**
  \(\text{pScissors}\) must be a valid pointer to an array of \(\text{scissorCount}\) \(\text{VkRect2D}\) structures.

- **VUID-vkCmdSetScissor-commandBuffer-recording**
  \(\text{commandBuffer}\) must be in the recording state.

- **VUID-vkCmdSetScissor-commandBuffer-cmdpool**
  The \(\text{VkCommandPool}\) that \(\text{commandBuffer}\) was allocated from must support graphics operations.

- **VUID-vkCmdSetScissor-scissorCount-arraylength**
  \(\text{scissorCount}\) must be greater than 0.
Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

26.3. 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.

26.4. 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.
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.

26.4.1. 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.

26.4.2. 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 \(z\).

26.4.3. Stencil Reference Replacement

Writing to the FragStencilRefEXT built-in will replace the fragment’s stencil reference value for each sample in the input SampleMask. Stencil testing performed after the fragment shader for this fragment will use this new value as \(s\).

26.4.4. Interlocked Operations

OpBeginInvocationInterlockEXT and OpEndInvocationInterlockEXT define a section of a fragment shader which imposes additional ordering constraints on operations performed within them. These operations are defined as interlocked operations. How interlocked operations are ordered against other fragment shader invocations depends on the specified execution modes.

If the ShadingRateInterlockOrderedEXT execution mode is specified, any interlocked operations in a fragment shader must happen before interlocked operations in fragment shader invocations that execute later in rasterization order and cover at least one sample in the same fragment area, and must happen after interlocked operations in a fragment shader that executes earlier in rasterization order and cover at least one sample in the same fragment area.
If the `ShadingRateInterlockUnorderedEXT` execution mode is specified, any interlocked operations in a fragment shader **must** happen before or after interlocked operations in fragment shader invocations that execute earlier or later in rasterization order and cover at least one sample in the same fragment area.

If the `PixelInterlockOrderedEXT` execution mode is specified, any interlocked operations in a fragment shader **must** happen before interlocked operations in fragment shader invocations that execute later in rasterization order and cover at least one sample in the same pixel, and **must** happen after interlocked operations in a fragment shader that executes earlier in rasterization order and cover at least one sample in the same pixel.

If the `PixelInterlockUnorderedEXT` execution mode is specified, any interlocked operations in a fragment shader **must** happen before or after interlocked operations in fragment shader invocations that execute earlier or later in rasterization order and cover at least one sample in the same pixel.

If the `SampleInterlockOrderedEXT` execution mode is specified, any interlocked operations in a fragment shader **must** happen before interlocked operations in fragment shader invocations that execute later in rasterization order and cover at least one of the same samples, and **must** happen after interlocked operations in a fragment shader that executes earlier in rasterization order and cover at least one of the same samples.

If the `SampleInterlockUnorderedEXT` execution mode is specified, any interlocked operations in a fragment shader **must** happen before or after interlocked operations in fragment shader invocations that execute earlier or later in rasterization order and cover at least one of the same samples.

### 26.5. 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 line coverage factor if the `lineRasterizationMode` member of the `VkPipelineRasterizationStateCreateInfo` structure is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT`, and the `alphaToCoverageEnable` and `alphaToOneEnable` members of the `VkPipelineMultisampleStateCreateInfo` structure.

All alpha values in this section refer only to the alpha component of the fragment shader output that has a `Location` and `Index` decoration of zero (see the Fragment Output Interface section). If that shader output has an integer or unsigned integer type, then these operations are skipped.

If the `lineRasterizationMode` member of the `VkPipelineRasterizationStateCreateInfo` structure is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT` and the fragment came from a line segment, then the alpha value is replaced by multiplying it by the coverage factor for the fragment computed during smooth line rasterization.
If `alphaToCoverageEnable` is enabled, a temporary coverage 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]

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using different algorithms at different framebuffer coordinates may help to avoid artifacts caused by regular coverage sample locations.</td>
</tr>
</tbody>
</table>

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.

### 26.6. 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 the type of 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 the comparison operator used in the depth test.
- `depthBoundsTestEnable` controls whether **depth bounds testing** is enabled.
- `stencilTestEnable` controls whether **stencil testing** is enabled.
- `front` and `back` control the parameters of the **stencil test**.
- `minDepthBounds` 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 **depth bounds testing** 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

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

### 26.7. Depth Bounds Test

The depth bounds test compares the depth value $z_a$ in the depth/stencil attachment at each sample’s framebuffer coordinates $(x,f)_i$ and **sample index** $i$ against a set of **depth bounds**.

The depth bounds are determined by two floating point values defining a minimum (`minDepthBounds`) and maximum (`maxDepthBounds`) depth value. These values are either set by the `VkPipelineDepthStencilStateCreateInfo` structure during pipeline creation, or dynamically by `vkCmdSetDepthBoundsTestEnableEXT` and `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 enable or disable** the depth bounds test, call:

```c
// Provided by VK_EXT_extended_dynamic_state
define vkCmdSetDepthBoundsTestEnableEXT(  
    VkCommandBuffer commandBuffer,  
    VkBool32 depthBoundsTestEnable);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `depthBoundsTestEnable` specifies if the depth bounds test is enabled.

This command sets the depth bounds enable for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_DEPTH_BOUNDS_TEST_ENABLE_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineDepthStencilStateCreateInfo::depthBoundsTestEnable` value used to create the currently active pipeline.

## Valid Usage

- **VUID-vkCmdSetDepthBoundsTestEnableEXT-None-03349**
  The `extendedDynamicState` feature **must** be enabled

## Valid Usage (Implicit)

- **VUID-vkCmdSetDepthBoundsTestEnableEXT-commandBuffer-parameter**
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle

- **VUID-vkCmdSetDepthBoundsTestEnableEXT-commandBuffer-recording**
  `commandBuffer` **must** be in the **recording state**

- **VUID-vkCmdSetDepthBoundsTestEnableEXT-commandBuffer-cmdpool**
  The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized
To **dynamically set** the 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**
  Unless the `VK_EXT_depth_range_unrestricted` extension is enabled, `minDepthBounds` must be between 0.0 and 1.0, inclusive

- **VUID-vkCmdSetDepthBounds-maxDepthBounds-00601**
  Unless the `VK_EXT_depth_range_unrestricted` extension is enabled, `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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

26.8. 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 `vkCmdSetStencilTestEnableEXT` or `VkPipelineDepthStencilStateCreateInfo::stencilTestEnable`, or if there is no stencil attachment, the coverage mask is unmodified by this operation.

The stencil test is controlled by one of two sets of stencil-related state, the front stencil state and the back stencil state. Stencil tests and writes use the back stencil state when processing fragments generated by back-facing polygons, and the front stencil state when processing fragments generated by front-facing polygons or any other primitives.

The comparison performed is based on the `VkCompareOp`, compare mask $s_c$, and stencil reference value $s_r$ of the relevant state set. The compare mask and stencil reference value are set by either the `VkPipelineDepthStencilStateCreateInfo` structure during pipeline creation, or by the `vkCmdSetStencilCompareMask` and `vkCmdSetStencilReference` commands respectively. The compare operation is set by `VkStencilOpState::compareOp` during pipeline creation.

The stencil reference and attachment values $s_r$ and $s_a$ are each independently combined with the compare mask $s_c$ using a logical AND operation to create masked reference and attachment values $s'_r$ and $s'_a$. $s'_r$ and $s'_a$ are used as A and B, respectively, in the operation specified by `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 `vkCmdSetStencilOpEXT` or `VkPipelineDepthStencilStateCreateInfo`. If the stencil test fails, `failOp` defines the stencil operation used. If the stencil test passes however, the stencil op used is based on the depth test - if it passes, `VkPipelineDepthStencilStateCreateInfo::passOp` is used, otherwise `VkPipelineDepthStencilStateCreateInfo::depthFailOp` is used.

The stencil attachment value $s_a$ is then updated with the generated stencil value $s_g$ according to the
write mask \( s_w \) defined by \( \text{VkPipelineDepthStencilStateCreateInfo}::\text{writeMask} \) as:

\[ s_a = (s_a \& \neg s_w) \mid (s_a \& s_w) \]

If there is no stencil attachment, no value is written.

To **dynamically enable or disable** the stencil test, call:

```c
// Provided by VK_EXT_extended_dynamic_state
void vkCmdSetStencilTestEnableEXT(
    VkCommandBuffer commandBuffer,  // Provided by VK_EXT_extended_dynamic_state
    VkBool32 stencilTestEnable);    // Provided by VK_EXT_extended_dynamic_state
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `stencilTestEnable` specifies if the stencil test is enabled.

This command sets the stencil test enable for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_STENCIL_TEST_ENABLE_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineDepthStencilStateCreateInfo::stencilTestEnable` value used to create the currently active pipeline.

### Valid Usage

- VUID-vkCmdSetStencilTestEnableEXT-None-03350
  The `extendedDynamicState` feature **must** be enabled

### Valid Usage (Implicit)

- VUID-vkCmdSetStencilTestEnableEXT-commandBuffer-parameter
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- VUID-vkCmdSetStencilTestEnableEXT-commandBuffer-recording
  `commandBuffer` **must** be in the `recording state`
- VUID-vkCmdSetStencilTestEnableEXT-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

### Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized
To **dynamically set** the stencil operation, call:

```c
// Provided by VK_EXT_extended_dynamic_state
void vkCmdSetStencilOpEXT(
    VkCommandBuffer commandBuffer,  // commandBuffer is the command buffer into which the command will be recorded.
    VkStencilFaceFlags faceMask,    // faceMask is a bitmask of VkStencilFaceFlagBits specifying the set of stencil state for which to update the stencil operation.
    VkStencilOp failOp,             // failOp is a VkStencilOp value specifying the action performed on samples that fail the stencil test.
    VkStencilOp passOp,             // passOp is a VkStencilOp value specifying the action performed on samples that pass both the depth and stencil tests.
    VkStencilOp depthFailOp,        // depthFailOp is a VkStencilOp value specifying the action performed on samples that pass the stencil test and fail the depth test.
    VkCompareOp compareOp)          // compareOp is a VkCompareOp value specifying the comparison operator used in the stencil test.
)
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `faceMask` is a bitmask of `VkStencilFaceFlagBits` specifying the set of stencil state for which to update the stencil operation.
- `failOp` is a `VkStencilOp` value specifying the action performed on samples that fail the stencil test.
- `passOp` is a `VkStencilOp` value specifying the action performed on samples that pass both the depth and stencil tests.
- `depthFailOp` is a `VkStencilOp` value specifying the action performed on samples that pass the stencil test and fail the depth test.
- `compareOp` is a `VkCompareOp` value specifying the comparison operator used in the stencil test.

This command sets the stencil operation for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_STENCIL_OP_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the corresponding `VkPipelineDepthStencilStateCreateInfo::failOp, passOp, depthFailOp, and compareOp` values used to create the currently active pipeline, for both front and back faces.

### Valid Usage

- **VUID-vkCmdSetStencilOpEXT-None-03351**
  The `extendedDynamicState` feature **must** be enabled

### Valid Usage (Implicit)

- **VUID-vkCmdSetStencilOpEXT-commandBuffer-parameter**
**commandBuffer** **must** be a valid **VkCommandBuffer** handle

- **VUID-vkCmdSetStencilOpEXT-faceMask-parameter**
  **faceMask** **must** be a valid combination of **VkStencilFaceFlagBits** values

- **VUID-vkCmdSetStencilOpEXT-faceMask-required bitmask**
  **faceMask** **must** not be 0

- **VUID-vkCmdSetStencilOpEXT-failOp-parameter**
  **failOp** **must** be a valid **VkStencilOp** value

- **VUID-vkCmdSetStencilOpEXT-passOp-parameter**
  **passOp** **must** be a valid **VkStencilOp** value

- **VUID-vkCmdSetStencilOpEXT-depthFailOp-parameter**
  **depthFailOp** **must** be a valid **VkStencilOp** value

- **VUID-vkCmdSetStencilOpEXT-compareOp-parameter**
  **compareOp** **must** be a valid **VkCompareOp** value

- **VUID-vkCmdSetStencilOpEXT-commandBuffer-recording**
  **commandBuffer** **must** be in the **recording** state

- **VUID-vkCmdSetStencilOpEXT-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The **VkStencilOpState** structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkStencilOpState {
    VkStencilOp failOp;
    VkStencilOp passOp;
    VkStencilOp depthFailOp;
    VkCompareOp compareOp;
    uint32_t compareMask;
    uint32_t writeMask;
};
```
• **failOp** is a `VkStencilOp` value specifying the action performed on samples that fail the stencil test.

• **passOp** is a `VkStencilOp` value specifying the action performed on samples that pass both the depth and stencil tests.

• **depthFailOp** is a `VkStencilOp` value specifying the action performed on samples that pass the stencil test and fail the depth test.

• **compareOp** is a `VkCompareOp` value specifying the comparison operator used in the stencil test.

• **compareMask** selects the bits of the unsigned integer stencil values participating in the stencil test.

• **writeMask** selects the bits of the unsigned integer stencil values updated by the stencil test in the stencil framebuffer attachment.

• **reference** is an integer reference value that is used in the unsigned stencil comparison.

---

**Valid Usage (Implicit)**

- 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_STENCILCOMPARE_MASK` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the
VkPipelineDepthStencilStateCreateInfo::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-required bitmask 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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</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.

// 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:

// Provided by VK_VERSION_1_0
void vkCmdSetStencilWriteMask(
    VkCommandBuffer commandBuffer,
    VkStencilFaceFlags faceMask,
    uint32_t writeMask);

- commandBuffer is the command buffer into which the command will be recorded.
- faceMask is a bitmask of VkStencilFaceFlagBits specifying the set of stencil state for which to update the write mask, as described above for vkCmdSetStencilCompareMask.
- writeMask is the new value to use as the stencil write mask.

This command sets the stencil write mask for subsequent drawing commands when the graphics pipeline is created with VK_DYNAMIC_STATE_STENCIL_WRITE_MASK set in VkPipelineDynamicStateCreateInfo::pDynamicStates. Otherwise, this state is specified by the VkPipelineDepthStencilStateCreateInfo::writeMask value used to create the currently active pipeline, for both front and 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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Possible values of `VkStencilOpState::compareOp`, specifying the stencil comparison function, 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 test evaluates to false.
- `VK_COMPARE_OP_LESS` specifies that the test evaluates $A < B$.
- `VK_COMPARE_OP_EQUAL` specifies that the test evaluates $A = B$.
- `VK_COMPARE_OP_LESS_OR_EQUAL` specifies that the test evaluates $A \leq B$.
- `VK_COMPARE_OP_GREATER` specifies that the test evaluates $A > B$.
- `VK_COMPARE_OP_NOT_EQUAL` specifies that the test evaluates $A \neq B$.
- `VK_COMPARE_OP_GREATER_OR_EQUAL` specifies that the test evaluates $A \geq B$.
- `VK_COMPARE_OP_ALWAYS` specifies that the test evaluates to true.

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_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.

### 26.9. 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.

#### 26.9.1. Depth Clamping and Range Adjustment

If **VkPipelineRasterizationStateCreateInfo::depthClampEnable** is enabled, before the sample’s $z_f$ is compared to $z_a$, $z_f$ is clamped to $[\min(n,f), \max(n,f)]$, where $n$ and $f$ are the minDepth and maxDepth depth range values of the viewport used by this fragment, respectively.

If depth clamping is not enabled and $z_f$ is not in the range $[0, 1]$ and either **VK_EXT_depth_range_unrestricted** is not enabled, or the depth attachment has a fixed-point format, then $z_i$ is undefined following this step.

#### 26.9.2. Depth Comparison

If the depth test is not enabled, as specified by **vkCmdSetDepthTestEnableEXT** or **VkPipelineDepthStencilStateCreateInfo::depthTestEnable**, then this step is skipped.
The comparison performed is based on the `VkCompareOp`, set by `vkCmdSetDepthCompareOpEXT` or `VkPipelineDepthStencilStateCreateInfo::depthCompareOp` during pipeline creation. \( z_f \) and \( z_a \) are used as A and B, respectively, in the operation specified by the `VkCompareOp`.

If the comparison evaluates to false, the coverage for the sample is set to 0.

### 26.9.3. Depth Attachment Writes

If depth writes are enabled, as specified by `vkCmdSetDepthWriteEnableEXT` or `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.

To dynamically enable or disable the depth test, call:

```c
// Provided by VK_EXT_extended_dynamic_state
define void vkCmdSetDepthTestEnableEXT(
    VkCommandBuffer commandBuffer,
    VkBool32 depthTestEnable);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `depthTestEnable` specifies if the depth test is enabled.

This command sets the depth test enable for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_DEPTH_TEST_ENABLE_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineDepthStencilStateCreateInfo::depthTestEnable` value used to create the currently active pipeline.

#### Valid Usage

- VUID-vkCmdSetDepthTestEnableEXT-None-03352
  The `extendedDynamicState` feature **must** be enabled

#### Valid Usage (Implicit)

- VUID-vkCmdSetDepthTestEnableEXT-commandBuffer-parameter
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetDepthTestEnableEXT-commandBuffer-recording
  `commandBuffer` **must** be in the `recording` state

- VUID-vkCmdSetDepthTestEnableEXT-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To dynamically set the depth compare operator, call:

```c
// Provided by VK_EXT_extended_dynamic_state
void vkCmdSetDepthCompareOpEXT(
    VkCommandBuffer commandBuffer,
    VkCompareOp depthCompareOp);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `depthCompareOp` specifies the depth comparison operator.

This command sets the depth comparison operator for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_DEPTH_COMPARE_OP_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineDepthStencilStateCreateInfo::depthCompareOp` value used to create the currently active pipeline.

Valid Usage

- VUID-vkCmdSetDepthCompareOpEXT-None-03353 The `extendedDynamicState` feature must be enabled

Valid Usage (Implicit)

- VUID-vkCmdSetDepthCompareOpEXT-commandBuffer-parameter `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdSetDepthCompareOpEXT-depthCompareOp-parameter `depthCompareOp` must be a valid `VkCompareOp` value
- VUID-vkCmdSetDepthCompareOpEXT-commandBuffer-recording `commandBuffer` must be in the recording state
The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations.

### Host Synchronization

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To dynamically set the depth write enable, call:

```c
// Provided by VK_EXT_extended_dynamic_state
void vkCmdSetDepthWriteEnableEXT(
    VkCommandBuffer commandBuffer,
    VkBool32 depthWriteEnable);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `depthWriteEnable` specifies if depth writes are enabled.

This command sets the depth write enable for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_DEPTH_WRITE_ENABLE_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineDepthStencilStateCreateInfo::depthWriteEnable` value used to create the currently active pipeline.

### Valid Usage

- **VUID-vkCmdSetDepthWriteEnableEXT-None-03354**
  The `extendedDynamicState` feature must be enabled.

### Valid Usage (Implicit)

- **VUID-vkCmdSetDepthWriteEnableEXT-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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

26.10. 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.

26.11. 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.

26.11.1. 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.

If the `fragment shading rate` is set, and the fragment covers multiple pixels, each pixel’s coverage consists of the coverage samples with a `pixel index` matching that pixel, and each sample retains its unique `sample index i`.

1082
26.11.2. Color Sample Coverage

Once pixel coverage is determined, coverage for each individual color sample corresponding to that pixel is determined.

The number of \textit{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 \textit{sample index} \( i \) is covered.
Chapter 27. The Framebuffer

27.1. Blending

Blending combines the incoming source fragment’s R, G, B, and A values with the destination R, G, B, and A values of each sample stored in the framebuffer at the fragment’s \((x_f, y_f)\) location. Blending is performed for each color sample covered by the fragment, rather than just once for each fragment.

Source and destination values are combined according to the blend operation, quadruplets of source and destination weighting factors determined by the blend factors, and a blend constant, to obtain a new set of R, G, B, and A values, as described below.

Blending is computed and applied separately to each color attachment used by the subpass, with separate controls for each attachment.

Prior to performing the blend operation, signed and unsigned normalized fixed-point color components undergo an implied conversion to floating-point as specified by Conversion from Normalized Fixed-Point to Floating-Point. Blending computations are treated as if carried out in floating-point, and basic blend operations are performed with a precision and dynamic range no lower than that used to represent destination components. Advanced blending operations are performed with a precision and dynamic range no lower than the smaller of that used to represent destination components or that used to represent 16-bit floating-point values.

\[\text{Note}\]

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 the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **flags** is reserved for future use.
• **logicOpEnable** controls whether to apply Logical Operations.
• **logicOp** selects which logical operation to apply.
• **attachmentCount** is the number of **VkPipelineColorBlendAttachmentState** elements in **pAttachments**.
• **pAttachments** is a pointer to an array of per target attachment states.
• **blendConstants** is a pointer to an array of four values used as the R, G, B, and A components of the blend constant that are used in blending, depending on the blend factor.

Each element of the **pAttachments** array is a **VkPipelineColorBlendAttachmentState** structure specifying per-target blending state for each individual color attachment. If the independent blending feature is not enabled on the device, all **VkPipelineColorBlendAttachmentState** elements in the **pAttachments** array must be identical.

The value of **attachmentCount** must be greater than the index of all color attachments that are not **VK_ATTACHMENT_UNUSED** in **VkSubpassDescription::pColorAttachments** or **VkSubpassDescription2::pColorAttachments** for the subpass in which this pipeline is used.

### Valid Usage

- VUID-VkPipelineColorBlendStateCreateInfo-pAttachments-00605
  If the independent blending feature is not enabled, all elements of **pAttachments** must be identical
- VUID-VkPipelineColorBlendStateCreateInfo-logicOpEnable-00606
  If the logic operations 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

### Valid Usage (Implicit)

- VUID-VkPipelineColorBlendStateCreateInfo-sType-sType
  **sType** must be **VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO**
- VUID-VkPipelineColorBlendStateCreateInfo-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 **VkPipelineColorBlendAdvancedStateCreateInfoEXT** or **VkPipelineColorWriteCreateInfoEXT**
- VUID-VkPipelineColorBlendStateCreateInfo-sType-unique
  The **sType** value of each struct in the **pNext** chain must be unique
- VUID-VkPipelineColorBlendStateCreateInfo-flags-zerobitmask
  **flags** must be **0**
- VUID-VkPipelineColorBlendStateCreateInfo-pAttachments-parameter
  If **attachmentCount** is not **0**, **pAttachments** must be a valid pointer to an array of
attachmentCount valid VkPipelineColorBlendAttachmentState structures

// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineColorBlendStateCreateFlags;

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

The VkPipelineColorBlendAttachmentState structure is defined as:

// 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 use 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 dual source blending feature is not enabled, srcColorBlendFactor must not be
  VK_BLEND_FACTOR_SRC1_COLOR, VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR,
  VK_BLEND_FACTOR_SRC1_ALPHA, or VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA
• VUID-VkPipelineColorBlendAttachmentState-dstColorBlendFactor-00609
  If the dual source blending feature is not enabled, dstColorBlendFactor must not be
  VK_BLEND_FACTOR_SRC1_COLOR, VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR,
  VK_BLEND_FACTOR_SRC1_ALPHA, or VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA

• VUID-VkPipelineColorBlendAttachmentState-srcAlphaBlendFactor-00610
  If the dual source blending feature is not enabled, srcAlphaBlendFactor must not be
  VK_BLEND_FACTOR_SRC1_COLOR, VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR,
  VK_BLEND_FACTOR_SRC1_ALPHA, or VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA

• VUID-VkPipelineColorBlendAttachmentState-dstAlphaBlendFactor-00611
  If the dual source blending feature is not enabled, dstAlphaBlendFactor must not be
  VK_BLEND_FACTOR_SRC1_COLOR, VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR,
  VK_BLEND_FACTOR_SRC1_ALPHA, or VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA

• VUID-VkPipelineColorBlendAttachmentState-colorBlendOp-01406
  If either of colorBlendOp or alphaBlendOp is an advanced blend operation, then
  colorBlendOp must equal alphaBlendOp

• VUID-VkPipelineColorBlendAttachmentState-advancedBlendIndependentBlend-01407
  If VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT::advancedBlendIndependentBlend is VK_FALSE and colorBlendOp is an advanced blend operation, then colorBlendOp must be the same for all attachments

• VUID-VkPipelineColorBlendAttachmentState-advancedBlendIndependentBlend-01408
  If VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT::advancedBlendIndependentBlend is VK_FALSE and alphaBlendOp is an advanced blend operation, then alphaBlendOp must be the same for all attachments

• VUID-VkPipelineColorBlendAttachmentState-advancedBlendAllOperations-01409
  If VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT::advancedBlendAllOperations is VK_FALSE, then colorBlendOp must not be
  VK_BLEND_OP_ZERO_EXT, VK_BLEND_OP_SRC_EXT,
  VK_BLEND_OP_DST_EXT, VK_BLEND_OP_SRC_OVER_EXT, VK_BLEND_OP_DST_OVER_EXT,
  VK_BLEND_OP_SRC_IN_EXT, VK_BLEND_OP_DST_IN_EXT, VK_BLEND_OP_SRC_ATOP_EXT, VK_BLEND_OP_DST_ATOP_EXT,
  VK_BLEND_OP_XOR_EXT, VK_BLEND_OP_INVERT_EXT, VK_BLEND_OP_INVERT_RGB_EXT,
  VK_BLEND_OP_LINEARDODGE_EXT, VK_BLEND_OP_LINEARBURN_EXT, VK_BLEND_OP_VIVIDLIGHT_EXT,
  VK_BLEND_OP_LINEARLIGHT_EXT, VK_BLEND_OP_PINLIGHT_EXT, VK_BLEND_OP_HARDMIX_EXT,
  VK_BLEND_OP_PLUS_EXT, VK_BLEND_OP_PLUS_CLAMPED_EXT, VK_BLEND_OP_PLUS_CLAMPED_ALPHA_EXT,
  VK_BLEND_OP_PLUS_DARKER_EXT, VK_BLEND_OP_MINUS_EXT, VK_BLEND_OP_MINUS_CLAMPED_EXT,
  VK_BLEND_OP_CONTRAST_EXT, VK_BLEND_OP_INVERT_OVG_EXT, VK_BLEND_OP_RED_EXT,
  VK_BLEND_OP_GREEN_EXT, or VK_BLEND_OP_BLUE_EXT

• VUID-VkPipelineColorBlendAttachmentState-colorBlendOp-01410
  If colorBlendOp or alphaBlendOp is an advanced blend operation, then colorAttachmentCount of the subpass this pipeline is compiled against must be less than or equal to VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT::advancedBlendMaxColorAttachments
Valid Usage (Implicit)

- VUID-VkPipelineColorBlendAttachmentState-srcColorBlendFactor-parameter
  `srcColorBlendFactor` must be a valid `VkBlendFactor` value
- VUID-VkPipelineColorBlendAttachmentState-dstColorBlendFactor-parameter
  `dstColorBlendFactor` must be a valid `VkBlendFactor` value
- VUID-VkPipelineColorBlendAttachmentState-colorBlendOp-parameter
  `colorBlendOp` must be a valid `VkBlendOp` value
- VUID-VkPipelineColorBlendAttachmentState-srcAlphaBlendFactor-parameter
  `srcAlphaBlendFactor` must be a valid `VkBlendFactor` value
- VUID-VkPipelineColorBlendAttachmentState-dstAlphaBlendFactor-parameter
  `dstAlphaBlendFactor` must be a valid `VkBlendFactor` value
- VUID-VkPipelineColorBlendAttachmentState-alphaBlendOp-parameter
  `alphaBlendOp` must be a valid `VkBlendOp` value
- VUID-VkPipelineColorBlendAttachmentState-colorWriteMask-parameter
  `colorWriteMask` must be a valid combination of `VkColorComponentFlagBits` values

27.1.1. Blend Factors

The source and destination color and alpha blending factors are selected from the enum:

```c
// Provided by VK_VERSION_1_0
typedef enum VkBlendFactor {
    VK_BLEND_FACTOR_ZERO = 0,
    VK_BLEND_FACTOR_ONE = 1,
    VK_BLEND_FACTOR_SRC_COLOR = 2,
    VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR = 3,
    VK_BLEND_FACTOR_DST_COLOR = 4,
    VK_BLEND_FACTOR_ONE_MINUS_DST_COLOR = 5,
    VK_BLEND_FACTOR_SRC_ALPHA = 6,
    VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA = 7,
    VK_BLEND_FACTOR_DST_ALPHA = 8,
    VK_BLEND_FACTOR_ONE_MINUS_DST_ALPHA = 9,
    VK_BLEND_FACTOR_CONSTANT_COLOR = 10,
    VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_COLOR = 11,
    VK_BLEND_FACTOR_CONSTANT_ALPHA = 12,
    VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_ALPHA = 13,
    VK_BLEND_FACTOR_SRC_ALPHA_SATURATE = 14,
    VK_BLEND_FACTOR_SRC1_COLOR = 15,
    VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR = 16,
    VK_BLEND_FACTOR_SRC1_ALPHA = 17,
    VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA = 18,
} VkBlendFactor;
```

The semantics of the enum values are described in the table below:
### Blend Factors

<table>
<thead>
<tr>
<th>VkBlendFactor</th>
<th>RGB Blend Factors (S&lt;sub&gt;r&lt;/sub&gt;,S&lt;sub&gt;g&lt;/sub&gt;,S&lt;sub&gt;b&lt;/sub&gt;) or (D&lt;sub&gt;r&lt;/sub&gt;,D&lt;sub&gt;g&lt;/sub&gt;,D&lt;sub&gt;b&lt;/sub&gt;)</th>
<th>Alpha Blend Factor (S&lt;sub&gt;a&lt;/sub&gt; or D&lt;sub&gt;a&lt;/sub&gt;)</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&lt;sub&gt;s0&lt;/sub&gt;,G&lt;sub&gt;s0&lt;/sub&gt;,B&lt;sub&gt;s0&lt;/sub&gt;)</td>
<td>A&lt;sub&gt;s0&lt;/sub&gt;</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR</td>
<td>(1-R&lt;sub&gt;s0&lt;/sub&gt;,1-G&lt;sub&gt;s0&lt;/sub&gt;,1-B&lt;sub&gt;s0&lt;/sub&gt;)</td>
<td>1-A&lt;sub&gt;s0&lt;/sub&gt;</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_DST_COLOR</td>
<td>(R&lt;sub&gt;d&lt;/sub&gt;,G&lt;sub&gt;d&lt;/sub&gt;,B&lt;sub&gt;d&lt;/sub&gt;)</td>
<td>A&lt;sub&gt;d&lt;/sub&gt;</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_DST_COLOR</td>
<td>(1-R&lt;sub&gt;d&lt;/sub&gt;,1-G&lt;sub&gt;d&lt;/sub&gt;,1-B&lt;sub&gt;d&lt;/sub&gt;)</td>
<td>1-A&lt;sub&gt;d&lt;/sub&gt;</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_SRC_ALPHA</td>
<td>(A&lt;sub&gt;s0&lt;/sub&gt;,A&lt;sub&gt;s0&lt;/sub&gt;,A&lt;sub&gt;s0&lt;/sub&gt;)</td>
<td>A&lt;sub&gt;d&lt;/sub&gt;</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA</td>
<td>(1-A&lt;sub&gt;s0&lt;/sub&gt;,1-A&lt;sub&gt;s0&lt;/sub&gt;,1-A&lt;sub&gt;s0&lt;/sub&gt;)</td>
<td>1-A&lt;sub&gt;d&lt;/sub&gt;</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_DST_ALPHA</td>
<td>(A&lt;sub&gt;d&lt;/sub&gt;,A&lt;sub&gt;d&lt;/sub&gt;,A&lt;sub&gt;d&lt;/sub&gt;)</td>
<td>A&lt;sub&gt;d&lt;/sub&gt;</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_DST_ALPHA</td>
<td>(1-A&lt;sub&gt;d&lt;/sub&gt;,1-A&lt;sub&gt;d&lt;/sub&gt;,1-A&lt;sub&gt;d&lt;/sub&gt;)</td>
<td>1-A&lt;sub&gt;d&lt;/sub&gt;</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_CONSTANT_COLOR</td>
<td>(R&lt;sub&gt;c&lt;/sub&gt;,G&lt;sub&gt;c&lt;/sub&gt;,B&lt;sub&gt;c&lt;/sub&gt;)</td>
<td>A&lt;sub&gt;c&lt;/sub&gt;</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_COLOR</td>
<td>(1-R&lt;sub&gt;c&lt;/sub&gt;,1-G&lt;sub&gt;c&lt;/sub&gt;,1-B&lt;sub&gt;c&lt;/sub&gt;)</td>
<td>1-A&lt;sub&gt;c&lt;/sub&gt;</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_CONSTANT_ALPHA</td>
<td>(A&lt;sub&gt;c&lt;/sub&gt;,A&lt;sub&gt;c&lt;/sub&gt;,A&lt;sub&gt;c&lt;/sub&gt;)</td>
<td>A&lt;sub&gt;c&lt;/sub&gt;</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_ALPHA</td>
<td>(1-A&lt;sub&gt;c&lt;/sub&gt;,1-A&lt;sub&gt;c&lt;/sub&gt;,1-A&lt;sub&gt;c&lt;/sub&gt;)</td>
<td>1-A&lt;sub&gt;c&lt;/sub&gt;</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_SRC_ALPHA_SATURATE</td>
<td>(f,f,f); f = min(A&lt;sub&gt;s0&lt;/sub&gt;,1-A&lt;sub&gt;d&lt;/sub&gt;)</td>
<td>1</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_SRC1_COLOR</td>
<td>(R&lt;sub&gt;s1&lt;/sub&gt;,G&lt;sub&gt;s1&lt;/sub&gt;,B&lt;sub&gt;s1&lt;/sub&gt;)</td>
<td>A&lt;sub&gt;s1&lt;/sub&gt;</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR</td>
<td>(1-R&lt;sub&gt;s1&lt;/sub&gt;,1-G&lt;sub&gt;s1&lt;/sub&gt;,1-B&lt;sub&gt;s1&lt;/sub&gt;)</td>
<td>1-A&lt;sub&gt;s1&lt;/sub&gt;</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_SRC1_ALPHA</td>
<td>(A&lt;sub&gt;s1&lt;/sub&gt;,A&lt;sub&gt;s1&lt;/sub&gt;,A&lt;sub&gt;s1&lt;/sub&gt;)</td>
<td>A&lt;sub&gt;s1&lt;/sub&gt;</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA</td>
<td>(1-A&lt;sub&gt;s1&lt;/sub&gt;,1-A&lt;sub&gt;s1&lt;/sub&gt;,1-A&lt;sub&gt;s1&lt;/sub&gt;)</td>
<td>1-A&lt;sub&gt;s1&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

In this table, the following conventions are used:

- R<sub>s0</sub>,G<sub>s0</sub>,B<sub>s0</sub> and A<sub>s0</sub> 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<sub>s1</sub>,G<sub>s1</sub>,B<sub>s1</sub> and A<sub>s1</sub> represent the second source color R, G, B, and A components, respectively, used in dual source blending modes, for the fragment output location corresponding to the color attachment being blended.
- R<sub>d</sub>,G<sub>d</sub>,B<sub>d</sub> and A<sub>d</sub> represent the R, G, B, and A components of the destination color. That is, the color currently in the corresponding color attachment for this fragment/sample.
- R<sub>c</sub>,G<sub>c</sub>,B<sub>c</sub> and A<sub>c</sub> represent the blend constant R, G, B, and A components, respectively.

To **dynamically set and change** the blend constants, call:

```
// Provided by VK_VERSION_1_0
```
```c
void vkCmdSetBlendConstants(
    VkCommandBuffer commandBuffer,  
    const float blendConstants[4]);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `blendConstants` is a pointer to an array of four values specifying the \( R_c \), \( G_c \), \( B_c \), and \( A_c \) components of the blend constant color used in blending, depending on the blend factor.

This command sets blend constants for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_BLEND_CONSTANTS` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineColorBlendStateCreateInfo::blendConstants` values used to create the currently active pipeline.

### Valid Usage (Implicit)
- VUID-vkCmdSetBlendConstants-commandBuffer-parameter
  - `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdSetBlendConstants-commandBuffer-recording
  - `commandBuffer` must be in the recording state
- VUID-vkCmdSetBlendConstants-commandBuffer-cmdpool
  - The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 27.1.2. Dual-Source Blending

Blend factors that use the secondary color input \((R_{s1}, G_{s1}, B_{s1}, A_{s1})\) (`VK_BLEND_FACTOR_SRC1_COLOR`, `VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_FACTOR_SRC1_ALPHA`, and `VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA`) may consume implementation resources that could otherwise be used for rendering to multiple color attachments. Therefore, the number of color attachments that can be used in a framebuffer may be lower when using dual-source blending.
Dual-source blending is only supported if the `dualSrcBlend` feature is enabled.

The maximum number of color attachments that can be used in a subpass when using dual-source blending functions is implementation-dependent and is reported as the `maxFragmentDualSrcAttachments` member of `VkPhysicalDeviceLimits`.

When using a fragment shader with dual-source blending functions, the color outputs are bound to the first and second inputs of the blender using the `Index` decoration, as described in Fragment Output Interface. If the second color input to the blender is not written in the shader, or if no output is bound to the second input of a blender, the result of the blending operation is not defined.

### 27.1.3. Blend Operations

Once the source and destination blend factors have been selected, they along with the source and destination components are passed to the blending operations. RGB and alpha components can use different operations. Possible values of `VkBlendOp`, specifying the operations, are:

```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,
    // Provided by VK_EXT_blend_operation_advanced
    VK_BLEND_OP_ZERO_EXT = 1000148000,
    // Provided by VK_EXT_blend_operation_advanced
    VK_BLEND_OP_SRC_EXT = 1000148001,
    // Provided by VK_EXT_blend_operation_advanced
    VK_BLEND_OP_DST_EXT = 1000148002,
    // Provided by VK_EXT_blend_operation_advanced
    VK_BLEND_OP_SRC_OVER_EXT = 1000148003,
    // Provided by VK_EXT_blend_operation_advanced
    VK_BLEND_OP_DST_OVER_EXT = 1000148004,
    // Provided by VK_EXT_blend_operation_advanced
    VK_BLEND_OP_SRC_IN_EXT = 1000148005,
    // Provided by VK_EXT_blend_operation_advanced
    VK_BLEND_OP_DST_IN_EXT = 1000148006,
    // Provided by VK_EXT_blend_operation_advanced
    VK_BLEND_OP_SRC_ATOP_EXT = 1000148009,
    // Provided by VK_EXT_blend_operation_advanced
    VK_BLEND_OP_DST_ATOP_EXT = 1000148010,
    // Provided by VK_EXT_blend_operation_advanced
    VK_BLEND_OP_XOR_EXT = 1000148011,
    // Provided by VK_EXT_blend_operation_advanced
    VK_BLEND_OP_MULTIPLY_EXT = 1000148012,
}
```
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_SCREEN_EXT = 1000148013,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_OVERLAY_EXT = 1000148014,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_DARKEN_EXT = 1000148015,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_LIGHTEN_EXT = 1000148016,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_COLORDOODGE_EXT = 1000148017,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_COLORBURN_EXT = 1000148018,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_HARDLIGHT_EXT = 1000148019,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_SOFTLIGHT_EXT = 1000148020,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_DIFFERENCE_EXT = 1000148021,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_EXCLUSION_EXT = 1000148022,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_INVERT_EXT = 1000148023,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_INVERT_RGB_EXT = 1000148024,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_LINEARDODGE_EXT = 1000148025,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_LINEARBURN_EXT = 1000148026,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_VIVIDLIGHT_EXT = 1000148027,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_LINEARLIGHT_EXT = 1000148028,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_PINLIGHT_EXT = 1000148029,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_HSL_HUE_EXT = 1000148031,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_HSL_SATURATION_EXT = 1000148032,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_HSL_COLOR_EXT = 1000148033,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_HSL_LUMINOSITY_EXT = 1000148034,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_PLUS_EXT = 1000148035,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_PLUS_CLAMPED_EXT = 1000148036,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_PLUS_CLAMPED_ALPHA_EXT = 1000148037,
VK_BLEND_OP_PLUS_DARKER_EXT = 1000148038,
  // Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_MINUS_EXT = 1000148039,
  // Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_MINUS_CLAMPED_EXT = 1000148040,
  // Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_CONTRAST_EXT = 1000148041,
  // Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_INVERT_OVG_EXT = 1000148042,
  // Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_RED_EXT = 1000148043,
  // Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_GREEN_EXT = 1000148044,
  // Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_BLUE_EXT = 1000148045,
} VkBlendOp;
The semantics of the basic blend operations are described in the table below:

**Table 33. 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
If the numeric format of a framebuffer color attachment is not sRGB encoded then the resulting c values for R, G and B are unmodified. The value of A is never sRGB encoded. That is, the alpha component is always stored in memory as linear.

If the framebuffer color attachment is VK_ATTACHMENT_UNUSED, no writes are performed through that attachment. Writes are not performed to framebuffer color attachments greater than or equal to the VkSubpassDescription::colorAttachmentCount or VkSubpassDescription2::colorAttachmentCount value.

### 27.1.4. Advanced Blend Operations

The advanced blend operations are those listed in tables f/X/Y/Z Advanced Blend Operations, Hue-Saturation-Luminosity Advanced Blend Operations, and Additional RGB Blend Operations.

If the pNext chain of VkPipelineColorBlendStateCreateInfo includes a VkPipelineColorBlendAdvancedStateCreateInfoEXT structure, then that structure includes parameters that affect advanced blend operations.

The VkPipelineColorBlendAdvancedStateCreateInfoEXT structure is defined as:

```c
// Provided by VK_EXT_blend_operation_advanced
typedef struct VkPipelineColorBlendAdvancedStateCreateInfoEXT {
    VkStructureType      sType;
    const void*          pNext;
    VkBool32             srcPremultiplied;
    VkBool32             dstPremultiplied;
    VkBlendOverlapEXT    blendOverlap;
} VkPipelineColorBlendAdvancedStateCreateInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **srcPremultiplied** specifies whether the source color of the blend operation is treated as premultiplied.
- **dstPremultiplied** specifies whether the destination color of the blend operation is treated as premultiplied.
- **blendOverlap** is a VkBlendOverlapEXT value specifying how the source and destination sample's coverage is correlated.

If this structure is not present, **srcPremultiplied** and **dstPremultiplied** are both considered to be VK_TRUE, and **blendOverlap** is considered to be VK_BLEND_OVERLAP_UNCORRELATED_EXT.

#### Valid Usage

- VUID-VkPipelineColorBlendAdvancedStateCreateInfoEXT-srcPremultiplied-01424

If the non-premultiplied source color property is not supported, **srcPremultiplied** must be
VK_TRUE

- VUID-VkPipelineColorBlendAdvancedStateCreateInfoEXT-dstPremultiplied-01425
  If the non-premultiplied destination color property is not supported, `dstPremultiplied` must be `VK_TRUE`.

- VUID-VkPipelineColorBlendAdvancedStateCreateInfoEXT-blendOverlap-01426
  If the correlated overlap property is not supported, `blendOverlap` must be `VK_BLEND_OVERLAP_UNCORRELATED_EXT`.

Valid Usage (Implicit)

- VUID-VkPipelineColorBlendAdvancedStateCreateInfoEXT-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_ADVANCED_STATE_CREATE_INFO_EXT`.

- VUID-VkPipelineColorBlendAdvancedStateCreateInfoEXT-blendOverlap-parameter
  `blendOverlap` must be a valid `VkBlendOverlapEXT` value.

When using one of the operations in table **f/X/Y/Z Advanced Blend Operations** or **Hue-Saturation-Luminosity Advanced Blend Operations**, blending is performed according to the following equations:

\[
\begin{align*}
R &= f(R_s', R_d') \times p_0(A_s, A_d) + Y \times R_s' \times p_1(A_s, A_d) + Z \times R_d' \times p_2(A_s, A_d) \\
G &= f(G_s', G_d') \times p_0(A_s, A_d) + Y \times G_s' \times p_1(A_s, A_d) + Z \times G_d' \times p_2(A_s, A_d) \\
B &= f(B_s', B_d') \times p_0(A_s, A_d) + Y \times B_s' \times p_1(A_s, A_d) + Z \times B_d' \times p_2(A_s, A_d) \\
A &= X \times p_0(A_s, A_d) + Y \times p_1(A_s, A_d) + Z \times p_2(A_s, A_d)
\end{align*}
\]

where the function `f` and terms `X`, `Y`, and `Z` are specified in the table. The R, G, and B components of the source color used for blending are derived according to `srcPremultiplied`. If `srcPremultiplied` is set to `VK_TRUE`, the fragment color components are considered to have been premultiplied by the A component prior to blending. The base source color \((R_s', G_s', B_s')\) is obtained by dividing through by the A component:

\[
(R_s', G_s', B_s') = \begin{cases} 
(0, 0, 0) & A_s = 0 \\
\left( \frac{R_s}{A_s}, \frac{G_s}{A_s}, \frac{B_s}{A_s} \right) & \text{otherwise}
\end{cases}
\]

If `srcPremultiplied` is `VK_FALSE`, the fragment color components are used as the base color:

\[
(R_s', G_s', B_s') = (R_s, G_s, B_s)
\]

The R, G, and B components of the destination color used for blending are derived according to `dstPremultiplied`. If `dstPremultiplied` is set to `VK_TRUE`, the destination components are considered to have been premultiplied by the A component prior to blending. The base destination color \((R_d, G_d, B_d)\) is obtained by dividing through by the A component:
If `dstPremultiplied` is `VK_FALSE`, the destination color components are used as the base color:

\[
(R_d', G_d', B_d') = \begin{cases} 
(0, 0, 0) & A_d = 0 \\
\left(\frac{R_d}{A_d}, \frac{G_d}{A_d}, \frac{B_d}{A_d} \right) & \text{otherwise}
\end{cases}
\]

When blending using advanced blend operations, we expect that the R, G, and B components of premultiplied source and destination color inputs be stored as the product of non-premultiplied R, G, and B component values and the A component of the color. If any R, G, or B component of a premultiplied input color is non-zero and the A component is zero, the color is considered ill-formed, and the corresponding component of the blend result is undefined.

All of the advanced blend operation formulas in this chapter compute the result as a premultiplied color. If `dstPremultiplied` is `VK_FALSE`, that result color’s R, G, and B components are divided by the A component before being written to the framebuffer. If any R, G, or B component of the color is non-zero and the A component is zero, the result is considered ill-formed, and the corresponding component of the blend result is undefined. If all components are zero, that value is unchanged.

If the A component of any input or result color is less than zero, the color is considered ill-formed, and all components of the blend result are undefined.

The weighting functions \(p_0\), \(p_1\), and \(p_2\) are defined in table `Advanced Blend Overlap Modes`. In these functions, the A components of the source and destination colors are taken to indicate the portion of the pixel covered by the fragment (source) and the fragments previously accumulated in the pixel (destination). The functions \(p_0\), \(p_1\), and \(p_2\) approximate the relative portion of the pixel covered by the intersection of the source and destination, covered only by the source, and covered only by the destination, respectively.

Possible values of `VkPipelineColorBlendAdvancedStateCreateInfoEXT::blendOverlap`, specifying the blend overlap functions, are:

```c
// Provided by VK_EXT_blend_operation_advanced
typedef enum VkBlendOverlapEXT {
    VK_BLEND_OVERLAP_UNCORRELATED_EXT = 0,
    VK_BLEND_OVERLAP_CONJOINT_EXT = 2,
} VkBlendOverlapEXT;
```

- `VK_BLEND_OVERLAP_UNCORRELATED_EXT` specifies that there is no correlation between the source and destination coverage.
- `VK_BLEND_OVERLAP_CONJOINT_EXT` specifies that the source and destination coverage are considered to have maximal overlap.
- `VK_BLEND_OVERLAP_DISJOINT_EXT` specifies that the source and destination coverage are considered to have minimal overlap.
### Table 34. Advanced Blend Overlap Modes

<table>
<thead>
<tr>
<th>Overlap Mode</th>
<th>Weighting Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_BLEND_OVERLAP_UNCORRELATED_EXT</td>
<td>$p_0(A_s, A_d) = A_s A_d$</td>
</tr>
<tr>
<td></td>
<td>$p_1(A_s, A_d) = A_s (1 - A_d)$</td>
</tr>
<tr>
<td></td>
<td>$p_2(A_s, A_d) = A_d (1 - A_s)$</td>
</tr>
<tr>
<td>VK_BLEND_OVERLAP_CONJOINT_EXT</td>
<td>$p_0(A_s, A_d) = \min(A_s, A_d)$</td>
</tr>
<tr>
<td></td>
<td>$p_1(A_s, A_d) = \max(A_s - A_d, 0)$</td>
</tr>
<tr>
<td></td>
<td>$p_2(A_s, A_d) = \max(A_d - A_s, 0)$</td>
</tr>
<tr>
<td>VK_BLEND_OVERLAP_DISJOINT_EXT</td>
<td>$p_0(A_s, A_d) = \max(A_s + A_d - 1, 0)$</td>
</tr>
<tr>
<td></td>
<td>$p_1(A_s, A_d) = \min(A_s, 1 - A_d)$</td>
</tr>
<tr>
<td></td>
<td>$p_2(A_s, A_d) = \min(A_d, 1 - A_s)$</td>
</tr>
</tbody>
</table>

### Table 35. f/X/Y/Z Advanced Blend Operations

<table>
<thead>
<tr>
<th>Mode</th>
<th>Blend Coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_BLEND_OP_ZERO_EXT</td>
<td>$(X, Y, Z) = (0, 0, 0)$</td>
</tr>
<tr>
<td></td>
<td>$f(C_s, C_d) = 0$</td>
</tr>
<tr>
<td>VK_BLEND_OP_SRC_EXT</td>
<td>$(X, Y, Z) = (1, 1, 0)$</td>
</tr>
<tr>
<td></td>
<td>$f(C_s, C_d) = C_s$</td>
</tr>
<tr>
<td>VK_BLEND_OP_DST_EXT</td>
<td>$(X, Y, Z) = (1, 0, 1)$</td>
</tr>
<tr>
<td></td>
<td>$f(C_s, C_d) = C_d$</td>
</tr>
<tr>
<td>VK_BLEND_OP_SRC_OVER_EXT</td>
<td>$(X, Y, Z) = (1, 1, 1)$</td>
</tr>
<tr>
<td></td>
<td>$f(C_s, C_d) = C_s$</td>
</tr>
<tr>
<td>VK_BLEND_OP_DST_OVER_EXT</td>
<td>$(X, Y, Z) = (1, 1, 1)$</td>
</tr>
<tr>
<td></td>
<td>$f(C_s, C_d) = C_d$</td>
</tr>
<tr>
<td>VK_BLEND_OP_SRC_IN_EXT</td>
<td>$(X, Y, Z) = (1, 0, 0)$</td>
</tr>
<tr>
<td></td>
<td>$f(C_s, C_d) = C_s$</td>
</tr>
<tr>
<td>VK_BLEND_OP_DST_IN_EXT</td>
<td>$(X, Y, Z) = (1, 0, 0)$</td>
</tr>
<tr>
<td></td>
<td>$f(C_s, C_d) = C_d$</td>
</tr>
<tr>
<td>VK_BLEND_OP_SRC_ATOP_EXT</td>
<td>$(X, Y, Z) = (0, 1, 0)$</td>
</tr>
<tr>
<td></td>
<td>$f(C_s, C_d) = 0$</td>
</tr>
<tr>
<td>VK_BLEND_OP_DST_ATOP_EXT</td>
<td>$(X, Y, Z) = (0, 0, 1)$</td>
</tr>
<tr>
<td></td>
<td>$f(C_s, C_d) = 0$</td>
</tr>
<tr>
<td>VK_BLEND_OP_SRC_ATOP_EXT</td>
<td>$(X, Y, Z) = (1, 0, 1)$</td>
</tr>
<tr>
<td></td>
<td>$f(C_s, C_d) = C_s$</td>
</tr>
<tr>
<td>VK_BLEND_OP_DST_ATOP_EXT</td>
<td>$(X, Y, Z) = (1, 1, 0)$</td>
</tr>
<tr>
<td></td>
<td>$f(C_s, C_d) = C_d$</td>
</tr>
<tr>
<td>Mode</td>
<td>Blend Coefficients</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VK_BLEND_OP_XOR_EXT</td>
<td>((X, Y, Z) = (0, 1, 1) )</td>
</tr>
<tr>
<td></td>
<td>(f(C_s, C_d) = 0)</td>
</tr>
<tr>
<td>VK_BLEND_OP_MULTIPLY_EXT</td>
<td>((X, Y, Z) = (1, 1, 1) )</td>
</tr>
<tr>
<td></td>
<td>(f(C_s, C_d) = C_sC_d)</td>
</tr>
<tr>
<td>VK_BLEND_OP_SCREEN_EXT</td>
<td>((X, Y, Z) = (1, 1, 1) )</td>
</tr>
<tr>
<td></td>
<td>(f(C_s, C_d) = C_s + C_d - C_sC_d)</td>
</tr>
<tr>
<td>VK_BLEND_OP_OVERLAY_EXT</td>
<td>((X, Y, Z) = (1, 1, 1) )</td>
</tr>
</tbody>
</table>
|                                          | \(f(C_s, C_d) = \begin{cases} 2C_sC_d & C_d \leq 0.5 \\ 
                                  & \ 1 - 2(1 - C_s)(1 - C_d) & \text{otherwise} \end{cases}\) |
| VK_BLEND_OP_DARKEN_EXT                   | \((X, Y, Z) = (1, 1, 1) \)                                                       |
|                                          | \(f(C_s, C_d) = min(C_s, C_d)\)                                                   |
| VK_BLEND_OP_LIGHTEN_EXT                  | \((X, Y, Z) = (1, 1, 1) \)                                                       |
|                                          | \(f(C_s, C_d) = max(C_s, C_d)\)                                                   |
| VK_BLEND_OP_COLORDOGE_EXT                | \((X, Y, Z) = (1, 1, 1) \)                                                       |
|                                          | \(f(C_s, C_d) = \begin{cases} 1 & C_d \geq 0 \\ 
                                  & \ 0 & \text{otherwise} \end{cases}\) |
| VK_BLEND_OP_COLORBURN_EXT                | \((X, Y, Z) = (1, 1, 1) \)                                                       |
|                                          | \(f(C_s, C_d) = \begin{cases} 1 & C_d \leq 0 \\ 
                                  & \ 1 - min(1, \frac{C_d}{1 - C_s}) & \text{otherwise} \end{cases}\) |
| VK_BLEND_OP_HARDLIGHT_EXT                | \((X, Y, Z) = (1, 1, 1) \)                                                       |
|                                          | \(f(C_s, C_d) = \begin{cases} 2C_sC_d & C_s \leq 0.5 \\ 
                                  & \ 1 - 2(1 - C_s)(1 - C_d) & \text{otherwise} \end{cases}\) |
| VK_BLEND_OP_SOFTLIGHT_EXT                | \((X, Y, Z) = (1, 1, 1) \)                                                       |
|                                          | \(f(C_s, C_d) = \begin{cases} 1 - 2C_s(3C_d - C_d) & C_s \leq 0.5 \\ 
                                  & \ (3C_s - 2C_s^2 - 2C_d + C_d) & \text{otherwise} \end{cases}\) |
| VK_BLEND_OP_DIFFERENCE_EXT               | \((X, Y, Z) = (1, 1, 1) \)                                                       |
|                                          | \(f(C_s, C_d) = \mid C_d - C_s \mid\)                                             |
| VK_BLEND_OP_EXCLUSION_EXT                | \((X, Y, Z) = (1, 1, 1) \)                                                       |
|                                          | \(f(C_s, C_d) = C_s + C_d - 2C_sC_d\)                                             |
| VK_BLEND_OP_INVERT_EXT                   | \((X, Y, Z) = (1, 0, 1) \)                                                       |
|                                          | \(f(C_s, C_d) = 1 - C_d\)                                                         |
| VK_BLEND_OP_INVERT_RGB_EXT               | \((X, Y, Z) = (1, 0, 1) \)                                                       |
|                                          | \(f(C_s, C_d) = C_s(1 - C_d)\)                                                    |
| VK_BLEND_OP_LINEARDODGE_EXT              | \((X, Y, Z) = (1, 1, 1) \)                                                       |
|                                          | \(f(C_s, C_d) = \begin{cases} C_s + C_d & C_s + C_d \leq 1 \\ 
                                  & \ 1 & \text{otherwise} \end{cases}\) |

1099
<table>
<thead>
<tr>
<th>Mode</th>
<th>Blend Coefficients</th>
</tr>
</thead>
</table>
| VK_BLEND_OP_LINEARBURN_EXT               | \((X, Y, Z) = (1, 1, 1)\)  
\[ f(C_s, C_d) = \begin{cases}  
C_s + C_d - 1 & C_s + C_d > 1 \\
0 & \text{otherwise} 
\end{cases} \]  |
| VK_BLEND_OP_VIVIDLIGHT_EXT               | \((X, Y, Z) = (1, 1, 1)\)  
\[ f(C_s, C_d) = \begin{cases}  
1 - \min (1, \frac{1 - C_d}{2C_s}) & 0 < C_s < 0.5 \\
0 & C_s \leq 0 \\
\min (1, \frac{C_d}{2(1 - C_s)}) & 0.5 \leq C_s < 1 \\
1 & C_s \geq 1 
\end{cases} \]  |
| VK_BLEND_OP_LINEARLIGHT_EXT              | \((X, Y, Z) = (1, 1, 1)\)  
\[ f(C_s, C_d) = \begin{cases}  
1 & 2C_s + C_d > 2 \\
2C_s - C_d - 1 & 1 < 2C_s + C_d \leq 2 \\
0 & 2C_s + C_d \leq 1 
\end{cases} \]  |
| VK_BLEND_OP_PINLIGHT_EXT                 | \((X, Y, Z) = (1, 1, 1)\)  
\[ f(C_s, C_d) = \begin{cases}  
0 & 2C_s - 1 > C_d \text{ and } C_s < 0.5 \\
2C_s - 1 & 2C_s - 1 > C_d \text{ and } C_s \geq 0.5 \\
C_d & 2C_s - 1 \leq C_d \text{ and } C_s < 0.5C_d \\
2C_s - 1 - C_d & 2C_s - 1 \leq C_d \text{ and } C_s \geq 0.5C_d 
\end{cases} \]  |
| VK_BLEND_OP_HARDMIX_EXT                  | \((X, Y, Z) = (1, 1, 1)\)  
\[ f(C_s, C_d) = \begin{cases}  
0 & C_s + C_d < 1 \\
1 & \text{otherwise} 
\end{cases} \]  |

When using one of the HSL blend operations in table **Hue-Saturation-Luminosity Advanced Blend Operations** as the blend operation, the RGB color components produced by the function \( f \) are effectively obtained by converting both the non-premultiplied source and destination colors to the HSL (hue, saturation, luminosity) color space, generating a new HSL color by selecting \( H \), \( S \), and \( L \) components from the source or destination according to the blend operation, and then converting the result back to RGB. In the equations below, a blended RGB color is produced according to the following pseudocode:

```cpp
float minv3(vec3 c) {
    return min(min(c.r, c.g), c.b);
}
float maxv3(vec3 c) {
    return max(max(c.r, c.g), c.b);
}
float lumv3(vec3 c) {
    return dot(c, vec3(0.30, 0.59, 0.11));
}
float satv3(vec3 c) {
    return maxv3(c) - minv3(c);
}

// If any color components are outside [0,1], adjust the color to
// get the components in range.
vec3 ClipColor(vec3 color) {
    float lum = lumv3(color);
    float mincol = minv3(color);
    ...}
```
float maxcol = maxv3(color);
if (mincol < 0.0) {
  color = lum + ((color-lum)*lum) / (lum-mincol);
}
if (maxcol > 1.0) {
  color = lum + ((color-lum)*(1-lum)) / (maxcol-lum);
}
return color;

// Take the base RGB color <cbase> and override its luminosity
// with that of the RGB color <clum>.
vec3 SetLum(vec3 cbase, vec3 clum) {
  float lbase = lumv3(cbase);
  float llum = lumv3(clum);
  float ldiff = llum - lbase;
  vec3 color = cbase + vec3(ldiff);
  return ClipColor(color);
}

// Take the base RGB color <cbase> and override its saturation with
// that of the RGB color <csat>. The override the luminosity of the
// result with that of the RGB color <clum>.
vec3 SetLumSat(vec3 cbase, vec3 csat, vec3 clum) {
  float minbase = minv3(cbase);
  float sbase = satv3(cbase);
  float ssat = satv3(csat);
  vec3 color;
  if (sbase > 0) {
    // Equivalent (modulo rounding errors) to setting the
    // smallest (R,G,B) component to 0, the largest to <ssat>,
    // and interpolating the "middle" component based on its
    // original value relative to the smallest/largest.
    color = (cbase - minbase) * ssat / sbase;
  } else {
    color = vec3(0.0);
  }
  return SetLum(color, clum);
}

<table>
<thead>
<tr>
<th>Mode</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_BLEND_OP_HSL_HUE_EXT</td>
<td>(X, Y, Z) = (1, 1, 1)</td>
</tr>
<tr>
<td></td>
<td>$f(C_s, C_d) = SetLumSat(C_s, C_d, C_d)$</td>
</tr>
<tr>
<td>VK_BLEND_OP_HSL_SATURATION_EXT</td>
<td>(X, Y, Z) = (1, 1, 1)</td>
</tr>
<tr>
<td></td>
<td>$f(C_s, C_d) = SetLumSat(C_d, C_s, C_d)$</td>
</tr>
</tbody>
</table>
When using one of the operations in table Additional RGB Blend Operations as the blend operation, the source and destination colors used by these blending operations are interpreted according to srcPremultiplied and dstPremultiplied. The blending operations below are evaluated where the RGB source and destination color components are both considered to have been premultiplied by the corresponding A component.

\[
(R_s', G_s', B_s') = \begin{cases} 
(R_s, G_s, B_s) & \text{if srcPremultiplied is VK_TRUE} \\
(R_s A_s, G_s A_s, B_s A_s) & \text{if srcPremultiplied is VK_FALSE}
\end{cases}
\]

\[
(R_d', G_d', B_d') = \begin{cases} 
(R_d, G_d, B_d) & \text{if dstPremultiplied is VK_TRUE} \\
(R_d A_d, G_d A_d, B_d A_d) & \text{if dstPremultiplied is VK_FALSE}
\end{cases}
\]

### Additional RGB Blend Operations

<table>
<thead>
<tr>
<th>Mode</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_BLEND_OP_HSL_COLOR_EXT</td>
<td>((X, Y, Z) = (1, 1, 1) ) (f(C_s, C_d) = SetLum(C_s, C_d))</td>
</tr>
<tr>
<td>VK_BLEND_OP_HSL_LUMINOSITY_EXT</td>
<td>((X, Y, Z) = (1, 1, 1) ) (f(C_s, C_d) = SetLum(C_d, C_s))</td>
</tr>
</tbody>
</table>

Table 37. Additional RGB Blend Operations

<table>
<thead>
<tr>
<th>Mode</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_BLEND_OP_PLUS_EXT</td>
<td>((R, G, B, A) = (R_s' + R_d', ) (G_s' + G_d', ) (B_s' + B_d', ) (A_s + A_d))</td>
</tr>
<tr>
<td>VK_BLEND_OP_PLUS_CLAMPED_EXT</td>
<td>((R, G, B, A) = (\min(1, R_s' + R_d'), ) (\min(1, G_s' + G_d'), ) (\min(1, B_s' + B_d'), ) (\min(1, A_s + A_d)) )</td>
</tr>
<tr>
<td>VK_BLEND_OP_PLUS_CLAMPED_ALPHA_EXT</td>
<td>((R, G, B, A) = (\min(\min(1, A_s + A_d), R_s' + R_d'), ) (\min(\min(1, A_s + A_d), G_s' + G_d'), ) (\min(\min(1, A_s + A_d), B_s' + B_d'), ) (\min(1, A_s + A_d)) )</td>
</tr>
<tr>
<td>VK_BLEND_OP_PLUS_DARKER_EXT</td>
<td>((R, G, B, A) = (\max(0, \min(1, A_s + A_d) - (A_s - R_s' - (A_s - B_s')), ) (\max(0, \min(1, A_s + A_d) - (A_s - G_s' - (A_s - B_s')), ) (\max(0, \min(1, A_s + A_d) - (A_s - B_s' - (A_s - B_s')), ) (\min(1, A_s + A_d)) )</td>
</tr>
<tr>
<td>VK_BLEND_OP_MINUS_EXT</td>
<td>((R, G, B, A) = (R_d' - R_s', ) (G_d' - G_s', ) (B_d' - B_s', ) (A_d - A_s) )</td>
</tr>
<tr>
<td>VK_BLEND_OP_MINUS_CLAMPED_EXT</td>
<td>((R, G, B, A) = (\max(0, R_d' - R_s'), ) (\max(0, G_d' - G_s'), ) (\max(0, B_d' - B_s'), ) (\max(0, A_d - A_s)) )</td>
</tr>
<tr>
<td>Mode</td>
<td>Result</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VK_BLEND_OP_CONTRAST_EXT</td>
<td>( (R, G, B, A) = \left( \frac{A_d}{2} + 2(R_d - \frac{A_d}{2})(R_s - \frac{A_s}{2}), \right. )</td>
</tr>
<tr>
<td></td>
<td>( \frac{A_d}{2} + 2(G_d - \frac{A_d}{2})(G_s - \frac{A_s}{2}), )</td>
</tr>
<tr>
<td></td>
<td>( \frac{A_d}{2} + 2(B_d - \frac{A_d}{2})(B_s - \frac{A_s}{2}), )</td>
</tr>
<tr>
<td></td>
<td>( \frac{A_d}{2} )</td>
</tr>
<tr>
<td>VK_BLEND_OP_INVERT_OVG_EXT</td>
<td>( (R, G, B, A) = \left( A_s(1 - R_d) + (1 - A_s)R_d, \right. )</td>
</tr>
<tr>
<td></td>
<td>( A_s(1 - G_d) + (1 - A_s)G_d, )</td>
</tr>
<tr>
<td></td>
<td>( A_s(1 - B_d) + (1 - A_s)B_d, )</td>
</tr>
<tr>
<td></td>
<td>( A_s + A_d - A_sA_d )</td>
</tr>
<tr>
<td>VK_BLEND_OP_RED_EXT</td>
<td>( (R, G, B, A) = (R_d, G_d, B_d, A_d) )</td>
</tr>
<tr>
<td>VK_BLEND_OP_GREEN_EXT</td>
<td>( (R, G, B, A) = (R_d, G_d, B_d, A_d) )</td>
</tr>
<tr>
<td>VK_BLEND_OP_BLUE_EXT</td>
<td>( (R, G, B, A) = (R_d, G_d, B_d, A_d) )</td>
</tr>
</tbody>
</table>

### 27.2. Logical Operations

The application can enable a logical operation between the fragment’s color values and the existing value in the framebuffer attachment. This logical operation is applied prior to updating the framebuffer attachment. Logical operations are applied only for signed and unsigned integer and normalized integer framebuffers. Logical operations are not applied to floating-point or sRGB format color attachments.

Logical operations are controlled by the logicOpEnable and logicOp members of VkPipelineColorBlendStateCreateInfo. It can also be controlled by vkCmdSetLogicOpEXT if graphics pipeline is created with VK_DYNAMIC_STATE_LOGIC_OP_EXT set in VkPipelineDynamicStateCreateInfo::pDynamicStates. If logicOpEnable is VK_TRUE, then a logical operation selected by logicOp is applied between each color attachment and the fragment’s corresponding output value, and blending of all attachments is treated as if it were disabled. Any attachments using color formats for which logical operations are not supported simply pass through the color values unmodified. The logical operation is applied independently for each of the red, green, blue, and alpha components. The logicOp is selected from the following operations:

```c
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 \( R_{so}, G_{so}, B_{so}, \) or \( A_{so} \) 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 38. Logical Operations**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_LOGIC_OP_CLEAR</td>
<td>0</td>
</tr>
<tr>
<td>VK_LOGIC_OP_AND</td>
<td>( s &amp; d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_AND_REVERSE</td>
<td>( s &amp; \neg d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_COPY</td>
<td>( s )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_AND_INVERTED</td>
<td>( \neg s &amp; d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_NO_OP</td>
<td>( d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_XOR</td>
<td>( s \oplus d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_OR</td>
<td>( s \lor d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_NOR</td>
<td>( \neg (s \lor d) )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_EQUIVALENT</td>
<td>( \neg (s \oplus d) )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_INVERT</td>
<td>( \neg d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_OR_REVERSE</td>
<td>( s \lor \neg d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_COPY_INVERTED</td>
<td>( \neg s )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_OR_INVERTED</td>
<td>( \neg s \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.

To **dynamically set** the logical operation to apply for blend state, call:

```c
// Provided by VK_EXT_extended_dynamic_state2
void vkCmdSetLogicOpEXT(
    VkCommandBuffer commandBuffer,
    VkLogicOp logicOp);
```
• **commandBuffer** is the command buffer into which the command will be recorded.
• **logicOp** specifies the logical operation to apply for blend state.

This command sets the logical operation for blend state for subsequent drawing commands when the graphics pipeline is created with `VK_DYNAMIC_STATE_LOGIC_OP_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`. Otherwise, this state is specified by the `VkPipelineColorBlendStateCreateInfo::logicOp` value used to create the currently active pipeline.

### Valid Usage

- VUID-vkCmdSetLogicOpEXT-None-04867
  The `extendedDynamicState2LogicOp` feature must be enabled

### Valid Usage (Implicit)

- VUID-vkCmdSetLogicOpEXT-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdSetLogicOpEXT-logicOp-parameter
  `logicOp` must be a valid `VkLogicOp` value
- VUID-vkCmdSetLogicOpEXT-commandBuffer-recording
  `commandBuffer` must be in the recording state
- VUID-vkCmdSetLogicOpEXT-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 27.3. Color Write Mask

Bits which **can** be set in `VkPipelineColorBlendAttachmentState::colorWriteMask` to determine
whether the final color values R, G, B and A are written to the framebuffer attachment are:

```
// 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.

The color write mask operation is applied only if **Color Write Enable** is enabled for the respective attachment. Otherwise the color write mask is ignored and writes to all components of the attachment are disabled.

```
// Provided by VK_VERSION_1_0
typedef VkFlags VkColorComponentFlags;
```

**VkColorComponentFlags** is a bitmask type for setting a mask of zero or more **VkColorComponentFlagBits**.

### 27.4. Color Write Enable

The **VkPipelineColorWriteCreateInfoEXT** structure is defined as:

```
// Provided by VK_EXT_color_write_enable
typedef struct VkPipelineColorWriteCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    uint32_t attachmentCount;
    const VkBool32* pColorWriteEnables;
} VkPipelineColorWriteCreateInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
attachmentCount is the number of VkBool32 elements in pColorWriteEnables.
pColorWriteEnables is a pointer to an array of per target attachment boolean values specifying whether color writes are enabled for the given attachment.

When this structure is included in the pNext chain of VkPipelineColorBlendStateCreateInfo, it defines per-attachment color write state. If this structure is not included in the pNext chain, it is equivalent to specifying this structure with attachmentCount equal to the attachmentCount member of VkPipelineColorBlendStateCreateInfo, and pColorWriteEnables pointing to an array of as many VK_TRUE values.

If the colorWriteEnable feature is not enabled on the device, all VkBool32 elements in the pColorWriteEnables array must be VK_TRUE.

Color Write Enable interacts with the Color Write Mask as follows:

• If colorWriteEnable is VK_TRUE, writes to the attachment are determined by the colorWriteMask.
• If colorWriteEnable is VK_FALSE, the colorWriteMask is ignored and writes to all components of the attachment are disabled. This is equivalent to specifying a colorWriteMask of 0.

Valid Usage

• VUID-VkPipelineColorWriteCreateInfoEXT-pAttachments-04801
  If the colorWriteEnable feature is not enabled, all elements of pColorWriteEnables must be VK_TRUE

• VUID-VkPipelineColorWriteCreateInfoEXT-attachmentCount-04802
  attachmentCount must be equal to the attachmentCount member of the VkPipelineColorBlendStateCreateInfo structure specified during pipeline creation

Valid Usage (Implicit)

• VUID-VkPipelineColorWriteCreateInfoEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_PIPELINE_COLOR_WRITE_CREATE_INFO_EXT

• VUID-VkPipelineColorWriteCreateInfoEXT-pColorWriteEnables-parameter
  If attachmentCount is not 0, pColorWriteEnables must be a valid pointer to an array of attachmentCount VkBool32 values

To dynamically enable or disable writes to a color attachment, call:

```c
// Provided by VK_EXT_color_write_enable
void vkCmdSetColorWriteEnableEXT(VkCommandBuffer commandBuffer, uint32_t attachmentCount, const VkBool32* pColorWriteEnables);
```
- **commandBuffer** is the command buffer into which the command will be recorded.
- **attachmentCount** is the number of VkBool32 elements in pColorWriteEnables.
- **pColorWriteEnables** is a pointer to an array of per target attachment boolean values specifying whether color writes are enabled for the given attachment.

This command sets the color write enables for subsequent drawing commands when the graphics pipeline is created with VK_DYNAMIC_STATE_COLOR_WRITE_ENABLE_EXT set in VkPipelineDynamicStateCreateInfo::pDynamicStates. Otherwise, this state is specified by the VkPipelineColorWriteCreateInfoEXT::pColorWriteEnables values used to create the currently active pipeline.

### Valid Usage

- **VUID-vkCmdSetColorWriteEnableEXT-None-04803**
  The colorWriteEnable feature must be enabled

- **VUID-vkCmdSetColorWriteEnableEXT-attachmentCount-04804**
  attachmentCount must be equal to the attachmentCount member of the VkPipelineColorBlendStateCreateInfo structure specified during pipeline creation

### Valid Usage (Implicit)

- **VUID-vkCmdSetColorWriteEnableEXT-commandBuffer-parameter**
  commandBuffer must be a valid VkCommandBuffer handle

- **VUID-vkCmdSetColorWriteEnableEXT-pColorWriteEnables-parameter**
  pColorWriteEnables must be a valid pointer to an array of attachmentCount VkBool32 values

- **VUID-vkCmdSetColorWriteEnableEXT-commandBuffer-recording**
  commandBuffer must be in the recording state

- **VUID-vkCmdSetColorWriteEnableEXT-commandBuffer-cmdpool**
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

- **VUID-vkCmdSetColorWriteEnableEXT-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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 28. Dispatching Commands

Dispatching commands (commands with `Dispatch` in the name) provoke work in a compute pipeline. Dispatching commands are recorded into a command buffer and when executed by a queue, will produce work which executes according to the bound compute pipeline. A compute pipeline must be bound to a command buffer before any dispatching commands are recorded in that command buffer.

To record a dispatch, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdDispatch(
    VkCommandBuffer commandBuffer,
    uint32_t groupCountX,
    uint32_t groupCountY,
    uint32_t groupCountZ);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `groupCountX` is the number of local workgroups to dispatch in the X dimension.
- `groupCountY` is the number of local workgroups to dispatch in the Y dimension.
- `groupCountZ` is the number of local workgroups to dispatch in the Z dimension.

When the command is executed, a global workgroup consisting of `groupCountX × groupCountY × groupCountZ` local workgroups is assembled.

### Valid Usage

- **VUID-vkCmdDispatch-magFilter-04553**
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDispatch-mipmapMode-04770**
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDispatch-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 `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`.

- **VUID-vkCmdDispatch-filterCubic-02694**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`.

- **VUID-vkCmdDispatch-filterCubicMinmax-02695**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`.

- **VUID-vkCmdDispatch-None-02697**
  For each set \( n \) that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set must have been bound to \( n \) at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set \( n \), with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility.

- **VUID-vkCmdDispatch-None-02698**
  For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility.

- **VUID-vkCmdDispatch-None-02699**
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

- **VUID-vkCmdDispatch-None-02700**
  A valid pipeline must be bound to the pipeline bind point used by this command.

- **VUID-vkCmdDispatch-commandBuffer-02701**
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited (if the `VK_NV_inherited_viewport_scissor` extension is enabled) for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

- **VUID-vkCmdDispatch-None-02859**
  There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

- **VUID-vkCmdDispatch-None-02702**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a
VkSampler object that uses unnormalized coordinates, that sampler must not be used to sample from any VkImage with a VkImageView of the type VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, VK_IMAGE_VIEW_TYPE_1D_ARRAY, VK_IMAGE_VIEW_TYPE_2D_ARRAY or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY, in any shader stage

- VUID-vkCmdDispatch-None-02703
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions with ImplicitLod, Dref or Proj in their name, in any shader stage

- VUID-vkCmdDispatch-None-02704
  If the VkPipeline object bound to the pipeline bind point used by this command accesses a VkSampler object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V OpImageSample* or OpImageSparseSample* instructions that includes a LOD bias or any offset values, in any shader stage

- VUID-vkCmdDispatch-None-02705
  If the robust buffer access feature is not enabled, and if the VkPipeline object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point

- VUID-vkCmdDispatch-None-02706
  If the robust buffer access feature is not enabled, and if the VkPipeline object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point

- VUID-vkCmdDispatch-commandBuffer-02707
  If commandBuffer is an unprotected command buffer and protectedNoFault is not supported, any resource accessed by the VkPipeline object bound to the pipeline bind point used by this command must not be a protected resource

- VUID-vkCmdDispatch-None-04115
  If a VkImageView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view's format

- VUID-vkCmdDispatch-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-vkCmdDispatch-SampledType-04470
  If a VkImageView with a VkFormat that has a 64-bit component width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64

- VUID-vkCmdDispatch-SampledType-04471
  If a VkImageView with a VkFormat that has a component width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32
• VUID-vkCmdDispatch-SampledType-04472
If a VkBufferView with a VkFormat that has a 64-bit component width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64

• VUID-vkCmdDispatch-SampledType-04473
If a VkBufferView with a VkFormat that has a component width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32

• VUID-vkCmdDispatch-sparseImageInt64Atomics-04474
If the sparseImageInt64Atomics feature is not enabled, VkImage objects created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command

• VUID-vkCmdDispatch-sparseImageInt64Atomics-04475
If the sparseImageInt64Atomics feature is not enabled, VkBuffer objects created with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command

• VUID-vkCmdDispatch-commandBuffer-02712
If commandBuffer is a protected command buffer and protectedNoFault is not supported, any resource written to by the VkPipeline object bound to the pipeline bind point used by this command must not be an unprotected resource

• VUID-vkCmdDispatch-commandBuffer-02713
If commandBuffer is a protected command buffer and protectedNoFault is not supported, pipeline stages other than the framebuffer-space and compute stages in the VkPipeline object bound to the pipeline bind point used by this command must not write to any resource

• VUID-vkCmdDispatch-commandBuffer-04617
If any of the shader stages of the VkPipeline bound to the pipeline bind point used by this command uses the RayQueryKHR capability, then commandBuffer must not be a protected command buffer

• VUID-vkCmdDispatch-groupCountX-00386
groupCountX must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0]

• VUID-vkCmdDispatch-groupCountY-00387
groupCountY must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1]

• VUID-vkCmdDispatch-groupCountZ-00388
groupCountZ must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2]

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>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Compute</td>
</tr>
<tr>
<td>Secondary</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
If a \( \text{VkSampler} \) created with \text{mipmapMode} equal to \text{VK_SAMPLER_MIPMAP_MODE_LINEAR} and \text{compareEnable} equal to \text{VK_FALSE} is used to sample a \( \text{VkImageView} \) as a result of this command, then the image view's format features \textbf{must} contain \text{VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT}.

If a \( \text{VkImageView} \) is sampled with \text{depth comparison}, the image view \textbf{must} have been created with an \text{aspectMask} that contains \text{VK_IMAGE_ASPECT_DEPTH_BIT}.

If a \( \text{VkImageView} \) is accessed using atomic operations as a result of this command, then the image view's format features \textbf{must} contain \text{VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT}.

If a \( \text{VkImageView} \) is sampled with \text{VK_FILTER_CUBIC_EXT} as a result of this command, then the image view's format features \textbf{must} contain \text{VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT}.

Any \( \text{VkImageView} \) being sampled with \text{VK_FILTER_CUBIC_EXT} as a result of this command \textbf{must} have a \( \text{VkImageViewType} \) and format that supports cubic filtering, as specified by \text{VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic} returned by \text{vkGetPhysicalDeviceImageFormatProperties2}.

Any \( \text{VkImageView} \) being sampled with \text{VK_FILTER_CUBIC_EXT} with a reduction mode of either \text{VK_SAMPLER_REDUCTION_MODE_MIN} or \text{VK_SAMPLER_REDUCTION_MODE_MAX} as a result of this command \textbf{must} have a \( \text{VkImageViewType} \) and format that supports cubic filtering together with minmax filtering, as specified by \text{VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax} returned by \text{vkGetPhysicalDeviceImageFormatProperties2}.

For each set \( n \) that is statically used by the \( \text{VkPipeline} \) bound to the pipeline bind point used by this command, a descriptor set \textbf{must} have been bound to \( n \) at the same pipeline bind point, with a \( \text{VkPipelineLayout} \) that is compatible for set \( n \), with the \( \text{VkPipelineLayout} \) used to create the current \( \text{VkPipeline} \), as described in \text{Pipeline Layout Compatibility}.

For each push constant that is statically used by the \( \text{VkPipeline} \) bound to the pipeline bind point used by this command, a push constant value \textbf{must} have been set for the same pipeline bind point, with a \( \text{VkPipelineLayout} \) that is compatible for push constants, with the \( \text{VkPipelineLayout} \) used to create the current \( \text{VkPipeline} \), as described in \text{Pipeline Layout Compatibility}.

Descriptors in each bound descriptor set, specified via \text{vkCmdBindDescriptorSets}, \textbf{must} be valid if they are statically used by the \( \text{VkPipeline} \) bound to the pipeline bind point used by this command.

1116
A valid pipeline must be bound to the pipeline bind point used by this command

- **VUID-vkCmdDispatchIndirect-commandBuffer-02701**
  
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited (if the `VK_NV_inherited_viewport_scissor` extension is enabled) for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

- **VUID-vkCmdDispatchIndirect-None-02859**
  
  There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

- **VUID-vkCmdDispatchIndirect-None-02702**
  
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

- **VUID-vkCmdDispatchIndirect-None-02703**
  
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

- **VUID-vkCmdDispatchIndirect-None-02704**
  
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

- **VUID-vkCmdDispatchIndirect-None-02705**
  
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- **VUID-vkCmdDispatchIndirect-None-02706**
  
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- **VUID-vkCmdDispatchIndirect-commandBuffer-02707**
  
  If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command must not be a protected resource.

- **VUID-vkCmdDispatchIndirect-None-04115**
  
  If a `VkImageView` is accessed using `OpImageWrite` as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as
If a \texttt{VkBufferView} is accessed using \texttt{OpImageWrite} as a result of this command, then the \texttt{Type} of the \texttt{Texel} operand of that instruction \textbf{must} have at least as many components as the buffer view’s format.

If a \texttt{VkImageView} with a \texttt{VkFormat} that has a 64-bit component width is accessed as a result of this command, the \texttt{SampledType} of the \texttt{OpTypeImage} operand of that instruction \textbf{must} have a \texttt{Width} of 64.

If a \texttt{VkBufferView} with a \texttt{VkFormat} that has a 64-bit component width is accessed as a result of this command, the \texttt{SampledType} of the \texttt{OpTypeImage} operand of that instruction \textbf{must} have a \texttt{Width} of 64.

If a \texttt{VkBufferView} with a \texttt{VkFormat} that has a component width less than 64-bit is accessed as a result of this command, the \texttt{SampledType} of the \texttt{OpTypeImage} operand of that instruction \textbf{must} have a \texttt{Width} of 32.

If a \texttt{VkBufferView} with a \texttt{VkFormat} that has a component width less than 64-bit is accessed as a result of this command, the \texttt{SampledType} of the \texttt{OpTypeImage} operand of that instruction \textbf{must} have a \texttt{Width} of 32.

If the \texttt{sparseImageInt64Atomics} feature is not enabled, \texttt{VkImage} objects created with the \texttt{VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT} flag \textbf{must} not be accessed by atomic instructions through an \texttt{OpTypeImage} with a \texttt{SampledType} with a \texttt{Width} of 64 by this command.

If the \texttt{sparseImageInt64Atomics} feature is not enabled, \texttt{VkBuffer} objects created with the \texttt{VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT} flag \textbf{must} not be accessed by atomic instructions through an \texttt{OpTypeImage} with a \texttt{SampledType} with a \texttt{Width} of 64 by this command.

If \texttt{buffer} is non-sparse then it \textbf{must} be bound completely and contiguously to a single \texttt{VkDeviceMemory} object.

\texttt{buffer} \textbf{must} have been created with the \texttt{VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT} bit set.

\texttt{offset} \textbf{must} be a multiple of 4.

\texttt{commandBuffer} \textbf{must} not be a protected command buffer.

The sum of \texttt{offset} and the size of \texttt{VkDispatchIndirectCommand} \textbf{must} be less than or equal to the size of \texttt{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

- VUID-vkCmdDispatchIndirect-commonparent
  Both of buffer, and commandBuffer must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Compute</td>
</tr>
<tr>
<td>Secondary</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 \(\text{VkDispatchIndirectCommand}\) have the same meaning as the corresponding parameters of \(\text{vkCmdDispatch}\).

### Valid Usage

- VUID-VkDispatchIndirectCommand-x-00417
  - \(x\) must be less than or equal to \(\text{VkPhysicalDeviceLimits}::\text{maxComputeWorkGroupCount}[0]\)
- VUID-VkDispatchIndirectCommand-y-00418
  - \(y\) must be less than or equal to \(\text{VkPhysicalDeviceLimits}::\text{maxComputeWorkGroupCount}[1]\)
- VUID-VkDispatchIndirectCommand-z-00419
  - \(z\) must be less than or equal to \(\text{VkPhysicalDeviceLimits}::\text{maxComputeWorkGroupCount}[2]\)

To record a dispatch using non-zero base values for the components of \(\text{WorkgroupId}\), call:

```c
// Provided by VK_VERSION_1_1
void vkCmdDispatchBase(
    VkCommandBuffer commandBuffer,
    uint32_t baseGroupX,
    uint32_t baseGroupY,
    uint32_t baseGroupZ,
    uint32_t groupCountX,
    uint32_t groupCountY,
    uint32_t groupCountZ);
```

- \(\text{commandBuffer}\) is the command buffer into which the command will be recorded.
- \(\text{baseGroupX}\) is the start value for the X component of \(\text{WorkgroupId}\).
- \(\text{baseGroupY}\) is the start value for the Y component of \(\text{WorkgroupId}\).
- \(\text{baseGroupZ}\) is the start value for the Z component of \(\text{WorkgroupId}\).
- \(\text{groupCountX}\) is the number of local workgroups to dispatch in the X dimension.
- \(\text{groupCountY}\) is the number of local workgroups to dispatch in the Y dimension.
- \(\text{groupCountZ}\) is the number of local workgroups to dispatch in the Z dimension.

When the command is executed, a global workgroup consisting of \(\text{groupCountX} \times \text{groupCountY} \times \text{groupCountZ}\) local workgroups is assembled, with \(\text{WorkgroupId}\) values ranging from \([\text{baseGroup*}, \text{baseGroup*} + \text{groupCount*})\) in each component. \(\text{vkCmdDispatch}\) is equivalent to \(\text{vkCmdDispatchBase}(0,0,0,\text{groupCountX},\text{groupCountY},\text{groupCountZ})\).

### Valid Usage

- VUID-vkCmdDispatchBase-magFilter-04553
  - If a \(\text{VkSampler}\) created with \(\text{magFilter}\) or \(\text{minFilter}\) equal to \(\text{VK_FILTER_LINEAR}\) and \(\text{compareEnable}\) equal to \(\text{VK_FALSE}\) is used to sample a \(\text{VkImageView}\) as a result of this
command, then the image view's format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT

- VUID-vkCmdDispatchBase-mipmapMode-04770
  If a VkSampler created with mipmapMode equal to VK_SAMPLER_MIPMAP_MODE_LINEAR and compareEnable equal to VK_FALSE is used to sample a VkImageView as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT

- VUID-vkCmdDispatchBase-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-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

- VUID-vkCmdDispatchBase-None-02692
  If a VkImageView is sampled with VK_FILTER_CUBIC_EXT as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT

- VUID-vkCmdDispatchBase-filterCubic-02694
  Any VkImageView being sampled with VK_FILTER_CUBIC_EXT as a result of this command must have a VkImageViewType and format that supports cubic filtering, as specified by VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic returned by vkGetPhysicalDeviceImageFormatProperties2

- VUID-vkCmdDispatchBase-filterCubicMinmax-02695
  Any VkImageView being sampled with VK_FILTER_CUBIC_EXT with a reduction mode of either VK_SAMPLER_REDUCTION_MODE_MIN or VK_SAMPLER_REDUCTION_MODE_MAX as a result of this command must have a VkImageViewType and format that supports cubic filtering together with minmax filtering, as specified by VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax returned by vkGetPhysicalDeviceImageFormatProperties2

- VUID-vkCmdDispatchBase-None-02697
  For each set n that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a descriptor set must have been bound to n at the same pipeline bind point, with a VkPipelineLayout that is compatible for set n, with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout Compatibility

- VUID-vkCmdDispatchBase-None-02698
  For each push constant that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a VkPipelineLayout that is compatible for push constants, with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout Compatibility

- VUID-vkCmdDispatchBase-None-02699
  Descriptors in each bound descriptor set, specified via vkCmdBindDescriptorSets, must be valid if they are statically used by the VkPipeline bound to the pipeline bind point used by this command
A valid pipeline must be bound to the pipeline bind point used by this command.

If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited (if the `VK_NV_inherited_viewport_scissor` extension is enabled) for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

If `commandBuffer` is an unprotected command buffer and `protectedNoFault` is not supported, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command must not be a protected resource.

If a `VkImageView` is accessed using `OpImageWrite` as a result of this command, then the
**Type** of the **Texel** operand of that instruction **must** have at least as many components as the image view's format.

- **VUID-vkCmdDispatchBase-OpImageWrite-04469**
  If a **VkBufferView** is accessed using **OpImageWrite** as a result of this command, then the **Type** of the **Texel** operand of that instruction **must** have at least as many components as the buffer view's format.

- **VUID-vkCmdDispatchBase-SampledType-04470**
  If a **VkImageView** with a **VkFormat** that has a 64-bit component width is accessed as a result of this command, the **SampledType** of the **OpTypeImage** operand of that instruction **must** have a **Width** of 64.

- **VUID-vkCmdDispatchBase-SampledType-04471**
  If a **VkImageView** with a **VkFormat** that has a component width less than 64-bit is accessed as a result of this command, the **SampledType** of the **OpTypeImage** operand of that instruction **must** have a **Width** of 32.

- **VUID-vkCmdDispatchBase-SampledType-04472**
  If a **VkBufferView** with a **VkFormat** that has a 64-bit component width is accessed as a result of this command, the **SampledType** of the **OpTypeImage** operand of that instruction **must** have a **Width** of 64.

- **VUID-vkCmdDispatchBase-SampledType-04473**
  If a **VkBufferView** with a **VkFormat** that has a component width less than 64-bit is accessed as a result of this command, the **SampledType** of the **OpTypeImage** operand of that instruction **must** have a **Width** of 32.

- **VUID-vkCmdDispatchBase-sparseImageInt64Atomics-04474**
  If the **sparseImageInt64Atomics** feature is not enabled, **VkImage** objects created with the **VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT** flag **must** not be accessed by atomic instructions through an **OpTypeImage** with a **SampledType** with a **Width** of 64 by this command.

- **VUID-vkCmdDispatchBase-sparseImageInt64Atomics-04475**
  If the **sparseImageInt64Atomics** feature is not enabled, **VkBuffer** objects created with the **VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT** flag **must** not be accessed by atomic instructions through an **OpTypeImage** with a **SampledType** with a **Width** of 64 by this command.

- **VUID-vkCmdDispatchBase-commandBuffer-02712**
  If **commandBuffer** is a protected command buffer and **protectedNoFault** is not supported, any resource written to by the **VkPipeline** object bound to the pipeline bind point used by this command **must** not be an unprotected resource.

- **VUID-vkCmdDispatchBase-commandBuffer-02713**
  If **commandBuffer** is a protected command buffer and **protectedNoFault** is not supported, pipeline stages other than the framebuffer-space and compute stages in the **VkPipeline** object bound to the pipeline bind point used by this command **must** not write to any resource.

- **VUID-vkCmdDispatchBase-commandBuffer-04617**
  If any of the shader stages of the **VkPipeline** bound to the pipeline bind point used by this command uses the **RayQueryKHR** capability, then **commandBuffer** **must** not be a protected command buffer.
• **VUID-vkCmdDispatchBase-baseGroupX-00421**
  baseGroupX must be less than VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0]

• **VUID-vkCmdDispatchBase-baseGroupY-00422**
  baseGroupY must be less than VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1]

• **VUID-vkCmdDispatchBase-baseGroupZ-00423**
  baseGroupZ must be less than VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2]

• **VUID-vkCmdDispatchBase-groupCountX-00424**
  groupCountX must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0] minus baseGroupX

• **VUID-vkCmdDispatchBase-groupCountY-00425**
  groupCountY must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1] minus baseGroupY

• **VUID-vkCmdDispatchBase-groupCountZ-00426**
  groupCountZ must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2] minus baseGroupZ

• **VUID-vkCmdDispatchBase-baseGroupX-00427**
  If any of baseGroupX, baseGroupY, or baseGroupZ are not zero, then the bound compute pipeline must have been created with the VK_PIPELINE_CREATE_DISPATCH_BASE flag

---

**Valid Usage (Implicit)**

• **VUID-vkCmdDispatchBase-commandBuffer-parameter**
  commandBuffer must be a valid VkCommandBuffer handle

• **VUID-vkCmdDispatchBase-commandBuffer-recording**
  commandBuffer must be in the recording state

• **VUID-vkCmdDispatchBase-commandBuffer-cmdpool**
  The VkCommandPool that commandBuffer was allocated from must support compute operations

• **VUID-vkCmdDispatchBase-renderpass**
  This command must only be called outside of a render pass instance

---

**Host Synchronization**

• Host access to commandBuffer must be externally synchronized

• Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized
<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Compute</td>
</tr>
<tr>
<td>Secondary</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.

- Sparse binding is the base feature, and provides the following capabilities:
  - Resources can be bound at some defined (sparse block) granularity.
  - The entire resource must be bound to memory before use regardless of regions actually accessed.
  - No specific mapping of image region to memory offset is defined, i.e. the location that each texel corresponds to in memory is implementation-dependent.
  - Sparse buffers have a well-defined mapping of buffer range to memory range, where an offset into a range of the buffer that is bound to a single contiguous range of memory corresponds to an identical offset within that range of memory.
    - Requested via the VK_IMAGE_CREATE_SPARSE_BINDING_BIT and VK_BUFFER_CREATE_SPARSE_BINDING_BIT bits.
    - A sparse image created using VK_IMAGE_CREATE_SPARSE_BINDING_BIT (but not VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT) supports all formats that non-sparse usage supports, and supports both VK_IMAGE_TILING_OPTIMAL and VK_IMAGE_TILING_LINEAR tiling.
- Sparse Residency builds on (and requires) the sparseBinding feature. It includes the following
capabilities:

◦ Resources do not have to be completely bound to memory before use on the device.

◦ Images have a prescribed sparse image block layout, allowing specific rectangular regions of the image to be bound to specific offsets in memory allocations.

◦ Consistency of access to unbound regions of the resource is defined by the absence or presence of `VkPhysicalDeviceSparseProperties::residencyNonResidentStrict`. If this property is present, accesses to unbound regions of the resource are well defined and behave as if the data bound is populated with all zeros; writes are discarded. When this property is absent, accesses are considered safe, but reads will return undefined values.

◦ Requested via the `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` and `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` bits.

◦ Sparse residency support is advertised on a finer grain via the following features:

  ▪ `sparseResidencyBuffer`: Support for creating `VkBuffer` objects with the `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT`.

  ▪ `sparseResidencyImage2D`: Support for creating 2D single-sampled `VkImage` objects with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.

  ▪ `sparseResidencyImage3D`: Support for creating 3D `VkImage` objects with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.

  ▪ `sparseResidency2Samples`: Support for creating 2D `VkImage` objects with 2 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.

  ▪ `sparseResidency4Samples`: Support for creating 2D `VkImage` objects with 4 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.

  ▪ `sparseResidency8Samples`: Support for creating 2D `VkImage` objects with 8 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.

  ▪ `sparseResidency16Samples`: Support for creating 2D `VkImage` objects with 16 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.

Implementations supporting `sparseResidencyImage2D` are only required to support sparse 2D, single-sampled images. Support for sparse 3D and MSAA images is optional and can be enabled via `sparseResidencyImage3D`, `sparseResidency2Samples`, `sparseResidency4Samples`, `sparseResidency8Samples`, and `sparseResidency16Samples`.

◦ A sparse image created using `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` supports all non-compressed color formats with power-of-two element size that non-sparse usage supports. Additional formats may also be supported and can be queried via `vkGetPhysicalDeviceSparseImageFormatProperties`. `VK_IMAGE_TILING_LINEAR` tiling is not supported.

• **Sparse aliasing** provides the following capability that can be enabled per resource:

  Allows physical memory ranges to be shared between multiple locations in the same sparse resource or between multiple sparse resources, with each binding of a memory location observing a consistent interpretation of the memory contents.

  See [Sparse Memory Aliasing](#) for more information.
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:

```c
// 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. Window System Integration (WSI)

This chapter discusses the window system integration (WSI) between the Vulkan API and the various forms of displaying the results of rendering to a user. Since the Vulkan API can be used without displaying results, WSI is provided through the use of optional Vulkan extensions. This chapter provides an overview of WSI. See the appendix for additional details of each WSI extension, including which extensions must be enabled in order to use each of the functions described in this chapter.

30.1. WSI Platform

A platform is an abstraction for a window system, OS, etc. Some examples include MS Windows, Android, and Wayland. The Vulkan API may be integrated in a unique manner for each platform.

The Vulkan API does not define any type of platform object. Platform-specific WSI extensions are defined, each containing platform-specific functions for using WSI. Use of these extensions is guarded by preprocessor symbols as defined in the Window System-Specific Header Control appendix.

In order for an application to be compiled to use WSI with a given platform, it must either:

- define the appropriate preprocessor symbol prior to including the vulkan_sc.h header file, or
- include vulkan_sc_core.h and any native platform headers, followed by the appropriate platform-specific header.

The preprocessor symbols and platform-specific headers are defined in the Window System Extensions and Headers table.

Each platform-specific extension is an instance extension. The application must enable instance extensions with vkCreateInstance before using them.

30.2. WSI Surface

Native platform surface or window objects are abstracted by surface objects, which are represented by VkSurfaceKHR handles:

```
// Provided by VK_KHR_surface
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSurfaceKHR)
```

The VK_KHR_surface extension declares the VkSurfaceKHR object, and provides a function for destroying VkSurfaceKHR objects. Separate platform-specific extensions each provide a function for creating a VkSurfaceKHR object for the respective platform. From the application’s perspective this is an opaque handle, just like the handles of other Vulkan objects.
On certain platforms, the Vulkan loader and ICDs may have conventions that treat the handle as a pointer to a structure containing the platform-specific information about the surface. This will be described in the documentation for the loader-ICD interface, and in the `vk_icd.h` header file of the LoaderAndTools source-code repository. This does not affect the loader-layer interface; layers may wrap VkSurfaceKHR objects.

### 30.2.1. Platform-Independent Information

Once created, VkSurfaceKHR objects can be used in this and other extensions, in particular the VK_KHR_swapchain extension.

Several WSI functions return VK_ERROR_SURFACE_LOST_KHR if the surface becomes no longer available. After such an error, the surface (and any child swapchain, if one exists) should be destroyed, as there is no way to restore them to a not-lost state. Applications may attempt to create a new VkSurfaceKHR using the same native platform window object, but whether such re-creation will succeed is platform-dependent and may depend on the reason the surface became unavailable. A lost surface does not otherwise cause devices to be lost.

To destroy a VkSurfaceKHR object, call:

```c
// Provided by VK_KHR_surface
void vkDestroySurfaceKHR(
    VkInstance instance,  // instance used to create the surface.
    VkSurfaceKHR surface, // the surface to destroy.
    const VkAllocationCallbacks* pAllocator);  // allocator used for host memory allocated for the surface object when there is no more specific allocator available.
```

- `instance` is the instance used to create the surface.
- `surface` is the surface to destroy.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see Memory Allocation).

Destroying a VkSurfaceKHR merely severs the connection between Vulkan and the native surface, and does not imply destroying the native surface, closing a window, or similar behavior.

#### Valid Usage

- **VUID-vkDestroySurfaceKHR-surface-01266**
  All VkSwapchainKHR objects created for `surface` must have been destroyed prior to destroying `surface`.

#### Valid Usage (Implicit)

- **VUID-vkDestroySurfaceKHR-instance-parameter**
  `instance` must be a valid VkInstance handle.
30.3. Presenting Directly to Display Devices

In some environments applications can also present Vulkan rendering directly to display devices without using an intermediate windowing system. This can be useful for embedded applications, or implementing the rendering/presentation backend of a windowing system using Vulkan. The VK_KHR_display extension provides the functionality necessary to enumerate display devices and create VkSurfaceKHR objects that target displays.

30.3.1. Display Enumeration

Displays are represented by VkDisplayKHR handles:

```c
// Provided by VK_KHR_display
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDisplayKHR)
```

Various functions are provided for enumerating the available display devices present on a Vulkan physical device. To query information about the available displays, call:

```c
// Provided by VK_KHR_display
VkResult vkGetPhysicalDeviceDisplayPropertiesKHR(
    VkPhysicalDevice physicalDevice,
    uint32_t* pPropertyCount,
    VkDisplayPropertiesKHR* pProperties);
```

- `physicalDevice` is a physical device.
- `pPropertyCount` is a pointer to an integer related to the number of display devices available or queried, as described below.
- `pProperties` is either NULL or a pointer to an array of VkDisplayPropertiesKHR structures.

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

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

Valid Usage (Implicit)

- `VUID-vkGetPhysicalDeviceDisplayPropertiesKHR-physicalDevice-parameter`: `physicalDevice` must be a valid `VkPhysicalDevice` handle.
- `VUID-vkGetPhysicalDeviceDisplayPropertiesKHR-pPropertyCount-parameter`: `pPropertyCount` must be a valid pointer to a `uint32_t` value.
- `VUID-vkGetPhysicalDeviceDisplayPropertiesKHR-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` `VkDisplayPropertiesKHR` structures.

Return Codes

Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDisplayPropertiesKHR` structure is defined as:

```c
// Provided by VK_KHR_display
typedef struct VkDisplayPropertiesKHR {
    VkDisplayKHR display;
    const char* displayName;
    VkExtent2D physicalDimensions;
    VkExtent2D physicalResolution;
    VkSurfaceTransformFlagsKHR supportedTransforms;
    VkBool32 planeReorderPossible;
    VkBool32 persistentContent;
} VkDisplayPropertiesKHR;
```

- `display` is a handle that is used to refer to the display described here. This handle will be valid for the lifetime of the Vulkan instance.
• **displayName** is NULL or a pointer to a null-terminated UTF-8 string containing the name of the display. Generally, this will be the name provided by the display’s EDID. If NULL, no suitable name is available. If not NULL, the string pointed to must remain accessible and unmodified as long as display is valid.

• **physicalDimensions** describes the physical width and height of the visible portion of the display, in millimeters.

• **physicalResolution** describes the physical, native, or preferred resolution of the display.

  *Note*
  
  For devices which have no natural value to return here, implementations should return the maximum resolution supported.

• **supportedTransforms** is a bitmask of VkSurfaceTransformFlagBitsKHR describing which transforms are supported by this display.

• **planeReorderPossible** tells whether the planes on this display can have their z order changed. If this is VK_TRUE, the application can re-arrange the planes on this display in any order relative to each other.

• **persistentContent** tells whether the display supports self-refresh/internal buffering. If this is true, the application can submit persistent present operations on swapchains created against this display.

  *Note*
  
  Persistent presents may have higher latency, and may use less power when the screen content is updated infrequently, or when only a portion of the screen needs to be updated in most frames.

To query information about the available displays, call:

```
// Provided by VK_KHR_get_display_properties2
VkResult vkGetPhysicalDeviceDisplayProperties2KHR(
    VkPhysicalDevice physicalDevice,
    uint32_t* pPropertyCount,
    VkDisplayProperties2KHR* pProperties);
```

• **physicalDevice** is a physical device.

• **pPropertyCount** is a pointer to an integer related to the number of display devices available or queried, as described below.

• **pProperties** is either NULL or a pointer to an array of VkDisplayProperties2KHR structures.

vkGetPhysicalDeviceDisplayProperties2KHR behaves similarly to vkGetPhysicalDeviceDisplayPropertiesKHR, with the ability to return extended information via chained output structures.

If VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations is VK_TRUE, vkGetPhysicalDeviceDisplayProperties2KHR must not return VK_ERROR_OUT_OF_HOST_MEMORY.
Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceDisplayProperties2KHR-physicalDevice-parameter
  physicalDevice must be a valid VkPhysicalDevice handle
- VUID-vkGetPhysicalDeviceDisplayProperties2KHR-pPropertyCount-parameter
  pPropertyCount must be a valid pointer to a uint32_t value
- VUID-vkGetPhysicalDeviceDisplayProperties2KHR-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 VkDisplayProperties2KHR structures

Return Codes

Success
- VK_SUCCESS
- VK_INCOMPLETE

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkDisplayProperties2KHR structure is defined as:

```c
// Provided by VK_KHR_get_display_properties2
typedef struct VkDisplayProperties2KHR {
    VkStructureType sType;
    void* pNext;    
    VkDisplayPropertiesKHR displayProperties;
} VkDisplayProperties2KHR;
```

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

Valid Usage (Implicit)

- VUID-VkDisplayProperties2KHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_DISPLAY_PROPERTIES_2_KHR
- VUID-VkDisplayProperties2KHR-pNext-pNext
  pNext must be NULL
Acquiring and Releasing Displays

On some platforms, access to displays is limited to a single process or native driver instance. On such platforms, some or all of the displays may not be available to Vulkan if they are already in use by a native windowing system or other application.

To release a previously acquired display, call:

```c
// Provided by VK_EXT_direct_mode_display
VkResult vkReleaseDisplayEXT(
    VkPhysicalDevice physicalDevice,
    VkDisplayKHR display);
```

- `physicalDevice` The physical device the display is on.
- `display` The display to release control of.

### Valid Usage (Implicit)

- VUID-vkReleaseDisplayEXT-physicalDevice-parameter `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkReleaseDisplayEXT-display-parameter `display` must be a valid `VkDisplayKHR` handle
- VUID-vkReleaseDisplayEXT-display-parent `display` must have been created, allocated, or retrieved from `physicalDevice`

### Return Codes

**Success**

- `VK_SUCCESS`

### Display Planes

Images are presented to individual planes on a display. Devices **must** support at least one plane on each display. Planes **can** be stacked and blended to composite multiple images on one display. Devices **may** support only a fixed stacking order and fixed mapping between planes and displays, or they **may** allow arbitrary application specified stacking orders and mappings between planes and displays. To query the properties of device display planes, call:

```c
// Provided by VK_KHR_display
VkResult vkGetPhysicalDeviceDisplayPlanePropertiesKHR(
    VkPhysicalDevice physicalDevice,
    uint32_t* pPropertyCount,
    VkDisplayPlanePropertiesKHR* pProperties);
```
• `physicalDevice` is a physical device.

• `pPropertyCount` is a pointer to an integer related to the number of display planes available or queried, as described below.

• `pProperties` is either `NULL` or a pointer to an array of `VkDisplayPlanePropertiesKHR` structures.

If `pProperties` is `NULL`, then the number of display planes available for `physicalDevice` is returned in `pPropertyCount`. Otherwise, `pPropertyCount` must point to a variable set by the user to the number of elements in the `pProperties` array, and on return the variable is overwritten with the number of structures actually written to `pProperties`. If the value of `pPropertyCount` is less than the number of display planes for `physicalDevice`, at most `pPropertyCount` structures will be written.

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

Valid Usage (Implicit)

• VUID-vkGetPhysicalDeviceDisplayPlanePropertiesKHR-physicalDevice-parameter `physicalDevice` must be a valid `VkPhysicalDevice` handle

• VUID-vkGetPhysicalDeviceDisplayPlanePropertiesKHR-pPropertyCount-parameter `pPropertyCount` must be a valid pointer to a `uint32_t` value

• VUID-vkGetPhysicalDeviceDisplayPlanePropertiesKHR-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` `VkDisplayPlanePropertiesKHR` structures

Return Codes

Success

• `VK_SUCCESS`

• `VK_INCOMPLETE`

Failure

• `VK_ERROR_OUT_OF_HOST_MEMORY`

• `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDisplayPlanePropertiesKHR` structure is defined as:

```
// Provided by VK_KHR_display
typedef struct VkDisplayPlanePropertiesKHR {
    VkDisplayKHR currentDisplay;
    uint32_t currentStackIndex;
} VkDisplayPlanePropertiesKHR;
```
• **currentDisplay** is the handle of the display the plane is currently associated with. If the plane is not currently attached to any displays, this will be **VK_NULL_HANDLE**.

• **currentStackIndex** is the current z-order of the plane. This will be between 0 and the value returned by `vkGetPhysicalDeviceDisplayPlanePropertiesKHR` in `pPropertyCount`.

To query the properties of a device’s display planes, call:

```c
// Provided by VK_KHR_get_display_properties2
VkResult vkGetPhysicalDeviceDisplayPlaneProperties2KHR(
    VkPhysicalDevice physicalDevice,
    uint32_t* pPropertyCount,
    VkDisplayPlaneProperties2KHR* pProperties);
```

• **physicalDevice** is a physical device.

• **pPropertyCount** is a pointer to an integer related to the number of display planes available or queried, as described below.

• **pProperties** is either **NULL** or a pointer to an array of `VkDisplayPlaneProperties2KHR` structures.

`vkGetPhysicalDeviceDisplayPlaneProperties2KHR` behaves similarly to `vkGetPhysicalDeviceDisplayPlanePropertiesKHR`, with the ability to return extended information via chained output structures.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is **VK_TRUE**, `vkGetPhysicalDeviceDisplayPlaneProperties2KHR` **must** not return **VK_ERROR_OUT_OF_HOST_MEMORY**.

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceDisplayPlaneProperties2KHR-physicalDevice-parameter physicalDevice **must** be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceDisplayPlaneProperties2KHR-pPropertyCount-parameter pPropertyCount **must** be a valid pointer to a `uint32_t` value
- VUID-vkGetPhysicalDeviceDisplayPlaneProperties2KHR-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` `VkDisplayPlaneProperties2KHR` structures

### Return Codes

**Success**

- **VK_SUCCESS**
- **VK_INCOMPLETE**

**Failure**

- **VK_ERROR_OUT_OF_HOST_MEMORY**
The **VkDisplayPlaneProperties2KHR** structure is defined as:

```c
// Provided by VK_KHR_get_display_properties2
typedef struct VkDisplayPlaneProperties2KHR {
    VkStructureType sType;
    void* pNext;
    VkDisplayPlanePropertiesKHR displayPlaneProperties;
} VkDisplayPlaneProperties2KHR;
```

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

### Valid Usage (Implicit)

- VUID-VkDisplayPlaneProperties2KHR-sType-sType
  - **sType** must be **VK_STRUCTURE_TYPE_DISPLAY_PLANE_PROPERTIES_2_KHR**
- VUID-VkDisplayPlaneProperties2KHR-pNext-pNext
  - **pNext** must be **NULL**

To determine which displays a plane is usable with, call

```c
// Provided by VK_KHR_display
VkResult vkGetDisplayPlaneSupportedDisplaysKHR(
    VkPhysicalDevice physicalDevice,          // physical device.
    uint32_t planeIndex,                      // the plane which the application wishes to use, and **must** be in the range [0, physical device plane count - 1].
    uint32_t* pDisplayCount,                  // a pointer to an integer related to the number of displays available or queried, as described below.
    VkDisplayKHR* pDisplays);                // **pDisplays** is either **NULL** or a pointer to an array of VkDisplayKHR handles.
```

- **physicalDevice** is a physical device.
- **planeIndex** is the plane which the application wishes to use, and **must** be in the range [0, physical device plane count - 1].
- **pDisplayCount** is a pointer to an integer related to the number of displays available or queried, as described below.
- **pDisplays** is either **NULL** or a pointer to an array of VkDisplayKHR handles.

If **pDisplays** is **NULL**, then the number of displays usable with the specified **planeIndex** for **physicalDevice** is returned in **pDisplayCount**. Otherwise, **pDisplayCount** **must** point to a variable set by the user to the number of elements in the **pDisplays** array, and on return the variable is overwritten with the number of handles actually written to **pDisplays**. If the value of **pDisplayCount** is less than the number of usable display-plane pairs for **physicalDevice**, at most **pDisplayCount**
handles will be written, and \texttt{VK\_INCOMPLETE} will be returned instead of \texttt{VK\_SUCCESS}, to indicate that not all the available pairs were returned.

If \texttt{VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations} is \texttt{VK\_TRUE}, \texttt{vkGetDisplayPlaneSupportedDisplaysKHR} must not return \texttt{VK\_ERROR\_OUT\_OF\_HOST\_MEMORY}.

### Valid Usage

- VUID-vkGetDisplayPlaneSupportedDisplaysKHR-planeIndex-01249
  - \texttt{planeIndex} must be less than the number of display planes supported by the device as determined by calling \texttt{vkGetPhysicalDeviceDisplayPlanePropertiesKHR}

### Valid Usage (Implicit)

- VUID-vkGetDisplayPlaneSupportedDisplaysKHR-physicalDevice-parameter
  - \texttt{physicalDevice} must be a valid \texttt{VkPhysicalDevice} handle
- VUID-vkGetDisplayPlaneSupportedDisplaysKHR-pDisplayCount-parameter
  - \texttt{pDisplayCount} must be a valid pointer to a \texttt{uint32\_t} value
- VUID-vkGetDisplayPlaneSupportedDisplaysKHR-pDisplays-parameter
  - If the value referenced by \texttt{pDisplayCount} is not \texttt{0}, and \texttt{pDisplays} is not \texttt{NULL}, \texttt{pDisplays} must be a valid pointer to an array of \texttt{pDisplayCount} \texttt{VkDisplayKHR} handles

### Return Codes

**Success**

- \texttt{VK\_SUCCESS}
- \texttt{VK\_INCOMPLETE}

**Failure**

- \texttt{VK\_ERROR\_OUT\_OF\_HOST\_MEMORY}
- \texttt{VK\_ERROR\_OUT\_OF\_DEVICE\_MEMORY}

Additional properties of displays are queried using specialized query functions.

### Display Modes

Display modes are represented by \texttt{VkDisplayModeKHR} handles:

```c
// Provided by VK\_KHR\_display
VK\_DEFINE\_NON\_DISPATCHABLE\_HANDLE(VkDisplayModeKHR)
```

Each display has one or more supported modes associated with it by default. These built-in modes are queried by calling:
VkResult vkGetDisplayModePropertiesKHR(
    VkPhysicalDevice physicalDevice,
    VkDisplayKHR display,
    uint32_t* pPropertyCount,
    VkDisplayModePropertiesKHR* pProperties);

- `physicalDevice` is the physical device associated with `display`.
- `display` is the display to query.
- `pPropertyCount` is a pointer to an integer related to the number of display modes available or queried, as described below.
- `pProperties` is either NULL or a pointer to an array of `VkDisplayModePropertiesKHR` structures.

If `pProperties` is NULL, then the number of display modes available on the specified `display` for `physicalDevice` is returned in `pPropertyCount`. Otherwise, `pPropertyCount` must point to a variable set by the user to the number of elements in the `pProperties` array, and on return the variable is overwritten with the number of structures actually written to `pProperties`. If the value of `pPropertyCount` is less than the number of display modes for `physicalDevice`, at most `pPropertyCount` structures will be written, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available display modes were returned.

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

### Valid Usage (Implicit)

- VUID-vkGetDisplayModePropertiesKHR-physicalDevice-parameter `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetDisplayModePropertiesKHR-display-parameter `display` must be a valid `VkDisplayKHR` handle
- VUID-vkGetDisplayModePropertiesKHR-pPropertyCount-parameter `pPropertyCount` must be a valid pointer to a `uint32_t` value
- VUID-vkGetDisplayModePropertiesKHR-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` `VkDisplayModePropertiesKHR` structures
- VUID-vkGetDisplayModePropertiesKHR-display-parent `display` must have been created, allocated, or retrieved from `physicalDevice`

### Return Codes

**Success**
- `VK_SUCCESS`
The `VkDisplayModePropertiesKHR` structure is defined as:

```c
// Provided by VK_KHR_display
typedef struct VkDisplayModePropertiesKHR {
    VkDisplayModeKHR displayMode;
    VkDisplayModeParametersKHR parameters;
} VkDisplayModePropertiesKHR;
```

- `displayMode` is a handle to the display mode described in this structure. This handle will be valid for the lifetime of the Vulkan instance.
- `parameters` is a `VkDisplayModeParametersKHR` structure describing the display parameters associated with `displayMode`.

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

To query the properties of a device's built-in display modes, call:

```c
// Provided by VK_KHR_get_display_properties2
VkResult vkGetDisplayModeProperties2KHR(
    VkPhysicalDevice physicalDevice,    // physicalDevice is the physical device associated with display.
    VkDisplayKHR display,               // display is the display to query.
    uint32_t* pPropertyCount,           // pPropertyCount is a pointer to an integer related to the number of display modes available or queried, as described below.
    VkDisplayModeProperties2KHR* pProperties); // pProperties is either NULL or a pointer to an array of VkDisplayModeProperties2KHR structures.
```

`vkGetDisplayModeProperties2KHR` behaves similarly to `vkGetDisplayModePropertiesKHR`, with the ability to return extended information via chained output structures.
vkGetDisplayModeProperties2KHR must not return VK_ERROR_OUT_OF_HOST_MEMORY.

**Valid Usage (Implicit)**

- VUID-vkGetDisplayModeProperties2KHR-physicalDevice-parameter
  physicalDevice must be a valid VkPhysicalDevice handle

- VUID-vkGetDisplayModeProperties2KHR-display-parameter
  display must be a valid VkDisplayKHR handle

- VUID-vkGetDisplayModeProperties2KHR-pPropertyCount-parameter
  pPropertyCount must be a valid pointer to a uint32_t value

- VUID-vkGetDisplayModeProperties2KHR-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 VkDisplayModeProperties2KHR structures

- VUID-vkGetDisplayModeProperties2KHR-display-parent
  display must have been created, allocated, or retrieved from physicalDevice

**Return Codes**

**Success**

- VK_SUCCESS
- VK_INCOMPLETE

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkDisplayModeProperties2KHR structure is defined as:

```
// Provided by VK_KHR_get_display_properties2
typedef struct VkDisplayModeProperties2KHR {
    VkStructureType sType;
    void* pNext;
    VkDisplayModePropertiesKHR displayModeProperties;
} VkDisplayModeProperties2KHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- displayModeProperties is a VkDisplayModePropertiesKHR structure.
Valid Usage (Implicit)

- VUID-VkDisplayModeProperties2KHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_DISPLAY_MODE_PROPERTIES_2_KHR

- VUID-VkDisplayModeProperties2KHR-pNext-pNext
  pNext must be NULL

The VkDisplayModeParametersKHR structure is defined as:

```c
// Provided by VK_KHR_display
typedef struct VkDisplayModeParametersKHR {
    VkExtent2D visibleRegion;
    uint32_t refreshRate;
} VkDisplayModeParametersKHR;
```

- visibleRegion is the 2D extents of the visible region.
- refreshRate is a uint32_t that is the number of times the display is refreshed each second multiplied by 1000.

**Note**
For example, a 60Hz display mode would report a refreshRate of 60,000.

Valid Usage

- VUID-VkDisplayModeParametersKHR-width-01990
  The width member of visibleRegion must be greater than 0

- VUID-VkDisplayModeParametersKHR-height-01991
  The height member of visibleRegion must be greater than 0

- VUID-VkDisplayModeParametersKHR-refreshRate-01992
  refreshRate must be greater than 0

Additional modes may also be created by calling:

```c
// Provided by VK_KHR_display
VkResult vkCreateDisplayModeKHR(
    VkPhysicalDevice physicalDevice,     // Provided by physicalDevice
    VkDisplayKHR display,               // Provided by display
    const VkDisplayModeCreateInfoKHR* pCreateInfo,  // Provided by pCreateInfo
    const VkAllocationCallbacks* pAllocator,    // Provided by pAllocator
    VkDisplayModeKHR* pMode);              // Provided by pMode
```

- physicalDevice is the physical device associated with display.
display is the display to create an additional mode for.

pCreateInfo is a pointer to a VkDisplayModeCreateInfoKHR structure describing the new mode to create.

pAllocator is the allocator used for host memory allocated for the display mode object when there is no more specific allocator available (see Memory Allocation).

pMode is a pointer to a VkDisplayModeKHR handle in which the mode created is returned.

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

Valid Usage (Implicit)

- VUID-vkCreateDisplayModeKHR-physicalDevice-parameter physicalDevice must be a valid VkPhysicalDevice handle
- VUID-vkCreateDisplayModeKHR-display-parameter display must be a valid VkDisplayKHR handle
- VUID-vkCreateDisplayModeKHR-pCreateInfo-parameter pCreateInfo must be a valid pointer to a valid VkDisplayModeCreateInfoKHR structure
- VUID-vkCreateDisplayModeKHR-pAllocator-null pAllocator must be NULL
- VUID-vkCreateDisplayModeKHR-pMode-parameter pMode must be a valid pointer to a VkDisplayModeKHR handle
- VUID-vkCreateDisplayModeKHR-display-parent display must have been created, allocated, or retrieved from physicalDevice

Host Synchronization

- Host access to display must be externally synchronized

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_INITIALIZATION_FAILED

The VkDisplayModeCreateInfoKHR structure is defined as:
typedef struct VkDisplayModeCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkDisplayModeCreateFlagsKHR flags;
    VkDisplayModeParametersKHR parameters;
} VkDisplayModeCreateInfoKHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is reserved for future use, and must be zero.
- **parameters** is a VkDisplayModeParametersKHR structure describing the display parameters to use in creating the new mode. If the parameters are not compatible with the specified display, the implementation must return VK_ERROR_INITIALIZATION_FAILED.

**Valid Usage (Implicit)**

- VUID-VkDisplayModeCreateInfoKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_DISPLAY_MODE_CREATE_INFO_KHR

- VUID-VkDisplayModeCreateInfoKHR-pNext-pNext
  pNext must be NULL

- VUID-VkDisplayModeCreateInfoKHR-flags-zerobitmask
  flags must be 0

- VUID-VkDisplayModeCreateInfoKHR-parameters-parameter
  parameters must be a valid VkDisplayModeParametersKHR structure

Applications that wish to present directly to a display must select which layer, or “plane” of the display they wish to target, and a mode to use with the display. Each display supports at least one plane. The capabilities of a given mode and plane combination are determined by calling:

```
// Provided by VK_KHR_display
VkResult vkGetDisplayPlaneCapabilitiesKHR(
    VkPhysicalDevice physicalDevice,  
    VkDisplayModeKHR mode,           
    uint32_t planeIndex,             
    VkDisplayPlaneCapabilitiesKHR* pCapabilities);
```

- **physicalDevice** is the physical device associated with the display specified by **mode**
- **mode** is the display mode the application intends to program when using the specified plane. Note this parameter also implicitly specifies a display.
- **planeIndex** is the plane which the application intends to use with the display, and is less than the number of display planes supported by the device.
pCapabilities is a pointer to a VkDisplayPlaneCapabilitiesKHR structure in which the capabilities are returned.

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

Valid Usage (Implicit)

- VUID-vkGetDisplayPlaneCapabilitiesKHR-physicalDevice-parameter
  physicalDevice must be a valid VkPhysicalDevice handle
- VUID-vkGetDisplayPlaneCapabilitiesKHR-mode-parameter
  mode must be a valid VkDisplayModeKHR handle
- VUID-vkGetDisplayPlaneCapabilitiesKHR-pCapabilities-parameter
  pCapabilities must be a valid pointer to a VkDisplayPlaneCapabilitiesKHR structure

Host Synchronization

- Host access to mode must be externally synchronized

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkDisplayPlaneCapabilitiesKHR structure is defined as:

```c
// Provided by VK_KHR_display
typedef struct VkDisplayPlaneCapabilitiesKHR {
    VkDisplayPlaneAlphaFlagsKHR supportedAlpha;
    VkOffset2D minSrcPosition;
    VkOffset2D maxSrcPosition;
    VkExtent2D minSrcExtent;
    VkExtent2D maxSrcExtent;
    VkOffset2D minDstPosition;
    VkOffset2D maxDstPosition;
    VkExtent2D minDstExtent;
    VkExtent2D maxDstExtent;
} VkDisplayPlaneCapabilitiesKHR;
```

- supportedAlpha is a bitmask of VkDisplayPlaneAlphaFlagBitsKHR describing the supported alpha
• **minSrcPosition** is the minimum source rectangle offset supported by this plane using the specified mode.

• **maxSrcPosition** is the maximum source rectangle offset supported by this plane using the specified mode. The $x$ and $y$ components of **maxSrcPosition** must each be greater than or equal to the $x$ and $y$ components of **minSrcPosition**, respectively.

• **minSrcExtent** is the minimum source rectangle size supported by this plane using the specified mode.

• **maxSrcExtent** is the maximum source rectangle size supported by this plane using the specified mode.

• **minDstPosition**, **maxDstPosition**, **minDstExtent**, **maxDstExtent** all have similar semantics to their corresponding *Src* equivalents, but apply to the output region within the mode rather than the input region within the source image. Unlike the *Src* offsets, **minDstPosition** and **maxDstPosition** may contain negative values.

The minimum and maximum position and extent fields describe the implementation limits, if any, as they apply to the specified display mode and plane. Vendors may support displaying a subset of a swapchain's presentable images on the specified display plane. This is expressed by returning **minSrcPosition**, **maxSrcPosition**, **minSrcExtent**, and **maxSrcExtent** values that indicate a range of possible positions and sizes which may be used to specify the region within the presentable images that source pixels will be read from when creating a swapchain on the specified display mode and plane.

Vendors may also support mapping the presentable images' content to a subset or superset of the visible region in the specified display mode. This is expressed by returning **minDstPosition**, **maxDstPosition**, **minDstExtent**, and **maxDstExtent** values that indicate a range of possible positions and sizes which may be used to describe the region within the display mode that the source pixels will be mapped to.

Other vendors may support only a 1-1 mapping between pixels in the presentable images and the display mode. This may be indicated by returning $(0,0)$ for **minSrcPosition**, **maxSrcPosition**, **minDstPosition**, and **maxDstPosition**, and (display mode width, display mode height) for **minSrcExtent**, **maxSrcExtent**, **minDstExtent**, and **maxDstExtent**.

The value **supportedAlpha** must contain at least one valid **VkDisplayPlaneAlphaFlagBitsKHR** bit.

These values indicate the limits of the implementation's individual fields. Not all combinations of values within the offset and extent ranges returned in **VkDisplayPlaneCapabilitiesKHR** are guaranteed to be supported. Presentation requests specifying unsupported combinations may fail.

To query the capabilities of a given mode and plane combination, call:

```c
// Provided by VK_KHR_get_display_properties2
VkResult vkGetDisplayPlaneCapabilities2KHR(
    VkPhysicalDevice physicalDevice,
    const VkDisplayPlaneInfo2KHR* pDisplayPlaneInfo,
```
- `physicalDevice` is the physical device associated with `pDisplayPlaneInfo`.
- `pDisplayPlaneInfo` is a pointer to a `VkDisplayPlaneInfo2KHR` structure describing the plane and mode.
- `pCapabilities` is a pointer to a `VkDisplayPlaneCapabilities2KHR` structure in which the capabilities are returned.

`vkGetDisplayPlaneCapabilities2KHR` behaves similarly to `vkGetDisplayPlaneCapabilitiesKHR`, with the ability to specify extended inputs via chained input structures, and to return extended information via chained output structures.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkGetDisplayPlaneCapabilities2KHR` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

### Valid Usage (Implicit)

- `VUID-vkGetDisplayPlaneCapabilities2KHR-physicalDevice-parameterphysicalDevice must be a valid VkPhysicalDevice handle`
- `VUID-vkGetDisplayPlaneCapabilities2KHR-pDisplayPlaneInfo-parameterpDisplayPlaneInfo must be a valid pointer to a valid VkDisplayPlaneInfo2KHR structure`
- `VUID-vkGetDisplayPlaneCapabilities2KHR-pCapabilities-parameterpCapabilities must be a valid pointer to a VkDisplayPlaneCapabilities2KHR structure`

### Return Codes

**Success**
- `VK_SUCCESS`

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

The `VkDisplayPlaneInfo2KHR` structure is defined as:

```c
// Provided by VK_KHR_get_display_properties2
typedef struct VkDisplayPlaneInfo2KHR {
    VkStructureType sType;
    const void* pNext;
    VkDisplayModeKHR mode;
    uint32_t planeIndex;
} VkDisplayPlaneInfo2KHR;
```

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

• **mode** is the display mode the application intends to program when using the specified plane.

*Note*
This parameter also implicitly specifies a display.

• **planeIndex** is the plane which the application intends to use with the display.

The members of **VkDisplayPlaneInfo2KHR** correspond to the arguments to *vkGetDisplayPlaneCapabilitiesKHR*, with **sType** and **pNext** added for extensibility.

---

**Valid Usage (Implicit)**

- VUID-VkDisplayPlaneInfo2KHR-sType-sType
  - **sType** must be **VK_STRUCTURE_TYPE_DISPLAY_PLANE_INFO_2_KHR**

- VUID-VkDisplayPlaneInfo2KHR-pNext-pNext
  - **pNext** must be **NULL**

- VUID-VkDisplayPlaneInfo2KHR-mode-parameter
  - **mode** must be a valid **VkDisplayModeKHR** handle

---

**Host Synchronization**

- Host access to **mode** must be externally synchronized

The **VkDisplayPlaneCapabilities2KHR** structure is defined as:

```c
// Provided by VK_KHR_get_display_properties2
typedef struct VkDisplayPlaneCapabilities2KHR {
    VkStructureType sType;
    void* pNext;
    VkDisplayPlaneCapabilitiesKHR capabilities;
} VkDisplayPlaneCapabilities2KHR;
```

- **sType** is the type of this structure.

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

- **capabilities** is a **VkDisplayPlaneCapabilitiesKHR** structure.

---

**Valid Usage (Implicit)**

- VUID-VkDisplayPlaneCapabilities2KHR-sType-sType
  - **sType** must be **VK_STRUCTURE_TYPE_DISPLAY_PLANE_CAPABILITIES_2_KHR**

- VUID-VkDisplayPlaneCapabilities2KHR-pNext-pNext
  - **pNext** must be **NULL**
30.3.2. Display Control

To set the power state of a display, call:

```c
// Provided by VK_EXT_display_control
VkResult vkDisplayPowerControlEXT(
    VkDevice device,           // device
    VkDisplayKHR display,      // display
    const VkDisplayPowerInfoEXT* pDisplayPowerInfo               // pDisplayPowerInfo
);
```

- **device** is a logical device associated with **display**.
- **display** is the display whose power state is modified.
- **pDisplayPowerInfo** is a pointer to a `VkDisplayPowerInfoEXT` structure specifying the new power state of **display**.

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

### Valid Usage (Implicit)

- VUID-vkDisplayPowerControlEXT-device-parameter
  - **device** must be a valid `VkDevice` handle
- VUID-vkDisplayPowerControlEXT-display-parameter
  - **display** must be a valid `VkDisplayKHR` handle
- VUID-vkDisplayPowerControlEXT-pDisplayPowerInfo-parameter
  - **pDisplayPowerInfo** must be a valid pointer to a valid `VkDisplayPowerInfoEXT` structure
- VUID-vkDisplayPowerControlEXT-commonparent
  - Both of **device**, and **display** must have been created, allocated, or retrieved from the same `VkPhysicalDevice`

### Return Codes

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkDisplayPowerInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_display_control
typedef struct VkDisplayPowerInfoEXT {
    VkStructureType sType;
    const void* pNext;
} VkDisplayPowerInfoEXT;
```
VkDisplayPowerStateEXT powerState;
} VkDisplayPowerInfoEXT;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **powerState** is a `VkDisplayPowerStateEXT` value specifying the new power state of the display.

### Valid Usage (Implicit)

- VUID-VkDisplayPowerInfoEXT-sType-sType
  
  *sType* must be `VK_STRUCTURE_TYPE_DISPLAY_POWER_INFO_EXT`

- VUID-VkDisplayPowerInfoEXT-pNext-pNext
  
  *pNext* must be NULL

- VUID-VkDisplayPowerInfoEXT-powerState-parameter
  
  *powerState* must be a valid `VkDisplayPowerStateEXT` value

Possible values of `VkDisplayPowerInfoEXT::powerState`, specifying the new power state of a display, are:

```c
// Provided by VK_EXT_display_control
typedef enum VkDisplayPowerStateEXT {
    VK_DISPLAY_POWER_STATE_OFF_EXT = 0,
    VK_DISPLAY_POWER_STATE_SUSPEND_EXT = 1,
    VK_DISPLAY_POWER_STATE_ON_EXT = 2,
} VkDisplayPowerStateEXT;
```

- **VK_DISPLAY_POWER_STATE_OFF_EXT** specifies that the display is powered down.
- **VK_DISPLAY_POWER_STATE_SUSPEND_EXT** specifies that the display is put into a low power mode, from which it **may** be able to transition back to `VK_DISPLAY_POWER_STATE_ON_EXT` more quickly than if it were in `VK_DISPLAY_POWER_STATE_OFF_EXT`. This state **may** be the same as `VK_DISPLAY_POWER_STATE_OFF_EXT`.
- **VK_DISPLAY_POWER_STATE_ON_EXT** specifies that the display is powered on.

### 30.3.3. Display Surfaces

A complete display configuration includes a mode, one or more display planes and any parameters describing their behavior, and parameters describing some aspects of the images associated with those planes. Display surfaces describe the configuration of a single plane within a complete display configuration. To create a `VkSurfaceKHR` object for a display plane, call:

```c
// Provided by VK_KHR_display
VkResult vkCreateDisplayPlaneSurfaceKHR(  
    VkInstance instance,  
    const VkDisplaySurfaceCreateInfoKHR* pCreateInfo,  
    ...)  
```
• `instance` is the instance corresponding to the physical device the targeted display is on.

• `pCreateInfo` is a pointer to a `VkDisplaySurfaceCreateInfoKHR` structure specifying which mode, plane, and other parameters to use, as described below.

• `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see Memory Allocation).

• `pSurface` is a pointer to a `VkSurfaceKHR` handle in which the created surface is returned.

### Valid Usage (Implicit)

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

- VUID-vkCreateDisplayPlaneSurfaceKHR-pCreateInfo-parameter
  `pCreateInfo` must be a valid pointer to a valid `VkDisplaySurfaceCreateInfoKHR` structure

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

- VUID-vkCreateDisplayPlaneSurfaceKHR-pSurface-parameter
  `pSurface` must be a valid pointer to a `VkSurfaceKHR` handle

### Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkDisplaySurfaceCreateInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_display
typedef struct VkDisplaySurfaceCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkDisplaySurfaceCreateFlagsKHR flags;
    VkDisplayModeKHR displayMode;
    uint32_t planeIndex;
    uint32_t planeStackIndex;
    VkSurfaceTransformFlagBitsKHR transform;
    float globalAlpha;
    VkDisplayPlaneAlphaFlagBitsKHR alphaMode;
} VkDisplaySurfaceCreateInfoKHR;
```
• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **flags** is reserved for future use, and **must** be zero.
• **displayMode** is a **VkDisplayModeKHR** handle specifying the mode to use when displaying this surface.
• **planeIndex** is the plane on which this surface appears.
• **planeStackIndex** is the z-order of the plane.
• **transform** is a **VkSurfaceTransformFlagBitsKHR** value specifying the transformation to apply to images as part of the scanout operation.
• **globalAlpha** is the global alpha value. This value is ignored if **alphaMode** is not **VK_DISPLAY_PLANE_ALPHA_GLOBAL_BIT_KHR**.
• **alphaMode** is a **VkDisplayPlaneAlphaFlagBitsKHR** value specifying the type of alpha blending to use.
• **imageExtent** is the size of the presentable images to use with the surface.

**Note**

Creating a display surface **must** not modify the state of the displays, planes, or other resources it names. For example, it **must** not apply the specified mode to be set on the associated display. Application of display configuration occurs as a side effect of presenting to a display surface.

**Valid Usage**

- **VUID-VkDisplaySurfaceCreateInfoKHR-planeIndex-01252**
  
  **planeIndex** **must** be less than the number of display planes supported by the device as determined by calling **vkGetPhysicalDeviceDisplayPlanePropertiesKHR**

- **VUID-VkDisplaySurfaceCreateInfoKHR-planeReorderPossible-01253**
  
  If the **planeReorderPossible** member of the **VkDisplayPropertiesKHR** structure returned by **vkGetPhysicalDeviceDisplayPropertiesKHR** for the display corresponding to **displayMode** is **VK_TRUE** then **planeStackIndex** **must** be less than the number of display planes supported by the device as determined by calling **vkGetPhysicalDeviceDisplayPlanePropertiesKHR**; otherwise **planeStackIndex** **must** equal the **currentStackIndex** member of **VkDisplayPlanePropertiesKHR** returned by **vkGetPhysicalDeviceDisplayPlanePropertiesKHR** for the display plane corresponding to **displayMode**

- **VUID-VkDisplaySurfaceCreateInfoKHR-alphaMode-01254**
  
  If **alphaMode** is **VK_DISPLAY_PLANE_ALPHA_GLOBAL_BIT_KHR** then **globalAlpha** **must** be between 0 and 1, inclusive

- **VUID-VkDisplaySurfaceCreateInfoKHR-alphaMode-01255**
  
  **alphaMode** **must** be one of the bits present in the **supportedAlpha** member of
VkDisplayPlaneCapabilitiesKHR for the display plane corresponding to displayMode

- VUID-VkDisplaySurfaceCreateInfoKHR-transform-06740
  transform must be one of the bits present in the supportedTransforms member of VkDisplayPropertiesKHR for the display corresponding to displayMode

- VUID-VkDisplaySurfaceCreateInfoKHR-width-01256
  The width and height members of imageExtent must be less than or equal to VkPhysicalDeviceLimits::maxImageDimension2D

### Valid Usage (Implicit)

- VUID-VkDisplaySurfaceCreateInfoKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_DISPLAY_SURFACE_CREATE_INFO_KHR

- VUID-VkDisplaySurfaceCreateInfoKHR-pNext-pNext
  pNext must be NULL

- VUID-VkDisplaySurfaceCreateInfoKHR-flags-zerobitmask
  flags must be 0

- VUID-VkDisplaySurfaceCreateInfoKHR-displayMode-parameter
  displayMode must be a valid VkDisplayModeKHR handle

- VUID-VkDisplaySurfaceCreateInfoKHR-transform-parameter
  transform must be a valid VkSurfaceTransformFlagBitsKHR value

- VUID-VkDisplaySurfaceCreateInfoKHR-alphaMode-parameter
  alphaMode must be a valid VkDisplayPlaneAlphaFlagBitsKHR value

// Provided by VK_KHR_display
typedef VkFlags VkDisplaySurfaceCreateFlagsKHR;

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

Possible values of VkDisplaySurfaceCreateInfoKHR::alphaMode, specifying the type of alpha blending to use on a display, are:

// Provided by VK_KHR_display
typedef enum VkDisplayPlaneAlphaFlagBitsKHR {
    VK_DISPLAY_PLANE_ALPHA_OPAQUE_BIT_KHR = 0x00000001,
    VK_DISPLAY_PLANE_ALPHA_GLOBAL_BIT_KHR = 0x00000002,
    VK_DISPLAY_PLANE_ALPHA_PER_PIXEL_BIT_KHR = 0x00000004,
    VK_DISPLAY_PLANE_ALPHA_PER_PIXEL_PREMULTIPLIED_BIT_KHR = 0x00000008,
} VkDisplayPlaneAlphaFlagBitsKHR;

- VK_DISPLAY_PLANE_ALPHA_OPAQUE_BIT_KHR specifies that the source image will be treated as opaque.
• **VK_DISPLAY_PLANE_ALPHA_GLOBAL_BIT_KHR** specifies that a global alpha value must be specified that will be applied to all pixels in the source image.

• **VK_DISPLAY_PLANE_ALPHA_PER_PIXEL_BIT_KHR** specifies that the alpha value will be determined by the alpha component of the source image's pixels. If the source format contains no alpha values, no blending will be applied. The source alpha values are not premultiplied into the source image's other color components.

• **VK_DISPLAY_PLANE_ALPHA_PER_PIXEL_PREMULTIPLIED_BIT_KHR** is equivalent to **VK_DISPLAY_PLANE_ALPHA_PER_PIXEL_BIT_KHR**, except the source alpha values are assumed to be premultiplied into the source image's other color components.

```c
// Provided by VK_KHR_display
typedef VkFlags VkDisplayPlaneAlphaFlagsKHR;
```

VkDisplayPlaneAlphaFlagsKHR is a bitmask type for setting a mask of zero or more VkDisplayPlaneAlphaFlagBitsKHR.

### 30.3.4. Presenting to headless surfaces

Vulkan rendering can be presented to a headless surface, where the presentation operation is a no-op producing no externally-visible result.

> **Note**
> Because there is no real presentation target, the headless presentation engine may be extended to impose an arbitrary or customisable set of restrictions and features. This makes it a useful portable test target for applications targeting a wide range of presentation engines where the actual target presentation engines might be scarce, unavailable or otherwise undesirable or inconvenient to use for general Vulkan application development.

The usual surface query mechanisms must be used to determine the actual restrictions and features of the implementation.

To create a headless **VkSurfaceKHR** object, call:

```c
// Provided by VK_EXT_headless_surface
VkResult vkCreateHeadlessSurfaceEXT(
    VkInstance instance, 
    const VkHeadlessSurfaceCreateInfoEXT* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkSurfaceKHR* pSurface);
```

- **instance** is the instance to associate the surface with.
- **pCreateInfo** is a pointer to a VkHeadlessSurfaceCreateInfoEXT structure containing parameters affecting the creation of the surface object.
- **pAllocator** is the allocator used for host memory allocated for the surface object when there is
no more specific allocator available (see Memory Allocation).

• pSurface is a pointer to a VkSurfaceKHR handle in which the created surface object is returned.

Valid Usage (Implicit)

• VUID-vkCreateHeadlessSurfaceEXT-instance-parameter
  instance must be a valid VkInstance handle

• VUID-vkCreateHeadlessSurfaceEXT-pCreateInfo-parameter
  pCreateInfo must be a valid pointer to a valid VkHeadlessSurfaceCreateInfoEXT structure

• VUID-vkCreateHeadlessSurfaceEXT-pAllocator-null
  pAllocator must be NULL

• VUID-vkCreateHeadlessSurfaceEXT-pSurface-parameter
  pSurface must be a valid pointer to a VkSurfaceKHR handle

Return Codes

Success

• VK_SUCCESS

Failure

• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkHeadlessSurfaceCreateInfoEXT structure is defined as:

```c
// Provided by VK_EXT_headless_surface
typedef struct VkHeadlessSurfaceCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkHeadlessSurfaceCreateFlagsEXT flags;
} VkHeadlessSurfaceCreateInfoEXT;
```

• sType is the type of this structure.

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

• flags is reserved for future use.

Valid Usage (Implicit)

• VUID-VkHeadlessSurfaceCreateInfoEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_HEADLESS_SURFACE_CREATE_INFO_EXT

• VUID-VkHeadlessSurfaceCreateInfoEXT-pNext-pNext
  pNext must be NULL
For headless surfaces, `currentExtent` is the reserved value (0xFFFFFFFF, 0xFFFFFFFF). Whatever the application sets a swapchain’s `imageExtent` to will be the size of the surface, after the first image is presented.

```c
// Provided by VK_EXT_headless_surface
typedef VkFlags VkHeadlessSurfaceCreateFlagsEXT;
```

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

### 30.4. Querying for WSI Support

Not all physical devices will include WSI support. Within a physical device, not all queue families will support presentation. WSI support and compatibility can be determined in a platform-neutral manner (which determines support for presentation to a particular surface object) and additionally may be determined in platform-specific manners (which determine support for presentation on the specified physical device but do not guarantee support for presentation to a particular surface object).

To determine whether a queue family of a physical device supports presentation to a given surface, call:

```c
// Provided by VK_KHR_surface
VkResult vkGetPhysicalDeviceSurfaceSupportKHR(
    VkPhysicalDevice physicalDevice,
    uint32_t queueFamilyIndex,
    VkSurfaceKHR surface,
    VkBool32* pSupported);
```

- `physicalDevice` is the physical device.
- `queueFamilyIndex` is the queue family.
- `surface` is the surface.
- `pSupported` is a pointer to a `VkBool32`, which is set to `VK_TRUE` to indicate support, and `VK_FALSE` otherwise.

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

### Valid Usage

- `VUID-vkGetPhysicalDeviceSurfaceSupportKHR-queueFamilyIndex-01269`
Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceSurfaceSupportKHR-physicalDevice-parameter
  physicalDevice must be a valid VkPhysicalDevice handle
- VUID-vkGetPhysicalDeviceSurfaceSupportKHR-surface-parameter
  surface must be a valid VkSurfaceKHR handle
- VUID-vkGetPhysicalDeviceSurfaceSupportKHR-pSupported-parameter
  pSupported must be a valid pointer to a VkBool32 value
- VUID-vkGetPhysicalDeviceSurfaceSupportKHR-commonparent
  Both of physicalDevice, and surface must have been created, allocated, or retrieved from the same VkInstance

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_SURFACE_LOST_KHR

30.5. Surface Queries

The capabilities of a swapchain targeting a surface are the intersection of the capabilities of the WSI platform, the native window or display, and the physical device. The resulting capabilities can be obtained with the queries listed below in this section.

Note
In addition to the surface capabilities as obtained by surface queries below, swapchain images are also subject to ordinary image creation limits as reported by vkGetPhysicalDeviceImageFormatProperties. As an application is instructed by the appropriate Valid Usage sections, both the surface capabilities and the image creation limits have to be satisfied whenever swapchain images are created.

30.5.1. Surface Capabilities

To query the basic capabilities of a surface, needed in order to create a swapchain, call:
// Provided by VK_KHR_surface
VkResult vkGetPhysicalDeviceSurfaceCapabilitiesKHR(
    VkPhysicalDevice physicalDevice,
    VkSurfaceKHR surface,
    VkSurfaceCapabilitiesKHR* pSurfaceCapabilities);

- **physicalDevice** is the physical device that will be associated with the swapchain to be created, as described for `vkCreateSwapchainKHR`.
- **surface** is the surface that will be associated with the swapchain.
- **pSurfaceCapabilities** is a pointer to a `VkSurfaceCapabilitiesKHR` structure in which the capabilities are returned.

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

---

**Valid Usage**

- **VUID-vkGetPhysicalDeviceSurfaceCapabilitiesKHR-surface-06211**
  
  `surface` must be supported by `physicalDevice`, as reported by `vkGetPhysicalDeviceSurfaceSupportKHR` or an equivalent platform-specific mechanism.

---

**Valid Usage (Implicit)**

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

- **VUID-vkGetPhysicalDeviceSurfaceCapabilitiesKHR-surface-parameter**
  
  `surface` must be a valid `VkSurfaceKHR` handle.

- **VUID-vkGetPhysicalDeviceSurfaceCapabilitiesKHR-pSurfaceCapabilities-parameter**
  
  `pSurfaceCapabilities` must be a valid pointer to a `VkSurfaceCapabilitiesKHR` structure.

- **VUID-vkGetPhysicalDeviceSurfaceCapabilitiesKHR-commonparent**
  
  Both of `physicalDevice`, and `surface` must have been created, allocated, or retrieved from the same `VkInstance`.

---

**Return Codes**

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_SURFACE_LOST_KHR`
The `VkSurfaceCapabilitiesKHR` structure is defined as:

```c
// Provided by VK_KHR_surface
typedef struct VkSurfaceCapabilitiesKHR {
    uint32_t minImageCount;
    uint32_t maxImageCount;
    VkExtent2D currentExtent;
    VkExtent2D minImageExtent;
    VkExtent2D maxImageExtent;
    uint32_t maxImageArrayLayers;
    VkSurfaceTransformFlagsKHR supportedTransforms;
    VkSurfaceTransformFlagBitsKHR currentTransform;
    VkCompositeAlphaFlagsKHR supportedCompositeAlpha;
    VkImageUsageFlags supportedUsageFlags;
} VkSurfaceCapabilitiesKHR;
```

- `minImageCount` is the minimum number of images the specified device supports for a swapchain created for the surface, and will be at least one.
- `maxImageCount` is the maximum number of images the specified device supports for a swapchain created for the surface, and will be either 0, or greater than or equal to `minImageCount`. A value of 0 means that there is no limit on the number of images, though there may be limits related to the total amount of memory used by presentable images.
- `currentExtent` is the current width and height of the surface, or the special value (0xFFFFFFFF, 0xFFFFFFFF) indicating that the surface size will be determined by the extent of a swapchain targeting the surface.
- `minImageExtent` contains the smallest valid swapchain extent for the surface on the specified device. The width and height of the extent will each be less than or equal to the corresponding width and height of `currentExtent`, unless `currentExtent` has the special value described above.
- `maxImageExtent` contains the largest valid swapchain extent for the surface on the specified device. The width and height of the extent will each be greater than or equal to the corresponding width and height of `minImageExtent`. The width and height of the extent will each be greater than or equal to the corresponding width and height of `currentExtent`, unless `currentExtent` has the special value described above.
- `maxImageArrayLayers` is the maximum number of layers presentable images can have for a swapchain created for this device and surface, and will be at least one.
- `supportedTransforms` is a bitmask of `VkSurfaceTransformFlagBitsKHR` indicating the presentation transforms supported for the surface on the specified device. At least one bit will be set.
- `currentTransform` is `VkSurfaceTransformFlagBitsKHR` value indicating the surface’s current transform relative to the presentation engine’s natural orientation.
- `supportedCompositeAlpha` is a bitmask of `VkCompositeAlphaFlagBitsKHR`, representing the alpha compositing modes supported by the presentation engine for the surface on the specified device, and at least one bit will be set. Opaque composition can be achieved in any alpha compositing mode by either using an image format that has no alpha component, or by
ensuring that all pixels in the presentable images have an alpha value of 1.0.

- **supportedUsageFlags** is a bitmask of `VkImageUsageFlagBits` representing the ways the application can use the presentable images of a swapchain created with `VkPresentModeKHR` set to `VK_PRESENT_MODE_IMMEDIATE_KHR`, `VK_PRESENT_MODE_MAILBOX_KHR`, `VK_PRESENT_MODE_FIFO_KHR` or `VK_PRESENT_MODE_FIFO_RELAXED_KHR` for the surface on the specified device. `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` must be included in the set. Implementations may support additional usages.

### Note

Supported usage flags of a presentable image when using `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR` or `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR` presentation mode are provided by `VkSharedPresentSurfaceCapabilitiesKHR::sharedPresentSupportedUsageFlags`.

### Note

Formulas such as `min(N, maxImageCount)` are not correct, since `maxImageCount` may be zero.

To query the basic capabilities of a surface defined by the core or extensions, call:

```c
// Provided by VK_KHR_get_surface_capabilities2
VkResult vkGetPhysicalDeviceSurfaceCapabilities2KHR(  
    VkPhysicalDevice physicalDevice,  
    const VkPhysicalDeviceSurfaceInfo2KHR* pSurfaceInfo,  
    VkSurfaceCapabilities2KHR* pSurfaceCapabilities);
```

- `physicalDevice` is the physical device that will be associated with the swapchain to be created, as described for `vkCreateSwapchainKHR`.
- `pSurfaceInfo` is a pointer to a `VkPhysicalDeviceSurfaceInfo2KHR` structure describing the surface and other fixed parameters that would be consumed by `vkCreateSwapchainKHR`.
- `pSurfaceCapabilities` is a pointer to a `VkSurfaceCapabilities2KHR` structure in which the capabilities are returned.

`vkGetPhysicalDeviceSurfaceCapabilities2KHR` behaves similarly to `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`, with the ability to specify extended inputs via chained input structures, and to return extended information via chained output structures.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkGetPhysicalDeviceSurfaceCapabilities2KHR` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceSurfaceCapabilities2KHR-physicalDevice-parameter
  `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceSurfaceCapabilities2KHR-pSurfaceInfo-parameter
pSurfaceInfo must be a valid pointer to a valid VkPhysicalDeviceSurfaceInfo2KHR structure

- VUID-vkGetPhysicalDeviceSurfaceCapabilities2KHR-pSurfaceCapabilities-parameter
  pSurfaceCapabilities must be a valid pointer to a VkSurfaceCapabilities2KHR structure

## Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_SURFACE_LOST_KHR

The VkPhysicalDeviceSurfaceInfo2KHR structure is defined as:

```c
// Provided by VK_KHR_get_surface_capabilities2
typedef struct VkPhysicalDeviceSurfaceInfo2KHR {
    VkStructureType sType;
    const void* pNext;
    VkSurfaceKHR surface;
} VkPhysicalDeviceSurfaceInfo2KHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **surface** is the surface that will be associated with the swapchain.

The members of VkPhysicalDeviceSurfaceInfo2KHR correspond to the arguments to vkGetPhysicalDeviceSurfaceCapabilitiesKHR, with sType and pNext added for extensibility.

## Valid Usage (Implicit)

- VUID-VkPhysicalDeviceSurfaceInfo2KHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICALDEVICE_SURFACE_INFO_2_KHR

- VUID-VkPhysicalDeviceSurfaceInfo2KHR-pNext-pNext
  pNext must be NULL

- VUID-VkPhysicalDeviceSurfaceInfo2KHR-surface-parameter
  surface must be a valid VkSurfaceKHR handle

The VkSurfaceCapabilities2KHR structure is defined as:
typedef struct VkSurfaceCapabilities2KHR {
    VkStructureType sType;
    void* pNext;
    VkSurfaceCapabilitiesKHR surfaceCapabilities;
} VkSurfaceCapabilities2KHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **surfaceCapabilities** is a VkSurfaceCapabilitiesKHR structure describing the capabilities of the specified surface.

### Valid Usage (Implicit)

- VUID-VkSurfaceCapabilities2KHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_KHR

- VUID-VkSurfaceCapabilities2KHR-pNext-pNext
  pNext must be NULL or a pointer to a valid instance of VkSharedPresentSurfaceCapabilitiesKHR

- VUID-VkSurfaceCapabilities2KHR-sType-unique
  The sType value of each struct in the pNext chain must be unique

The VkSharedPresentSurfaceCapabilitiesKHR structure is defined as:

typedef struct VkSharedPresentSurfaceCapabilitiesKHR {
    VkStructureType sType;
    void* pNext;
    VkImageUsageFlags sharedPresentSupportedUsageFlags;
} VkSharedPresentSurfaceCapabilitiesKHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **sharedPresentSupportedUsageFlags** is a bitmask of VkImageUsageFlagBits representing the ways the application can use the shared presentable image from a swapchain created with VkPresentModeKHR set to VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR or VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR for the surface on the specified device. **VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT** must be included in the set but implementations may support additional usages.

### Valid Usage (Implicit)

- VUID-VkSharedPresentSurfaceCapabilitiesKHR-sType-sType
sType must be VK_STRUCTURE_TYPE_SHARED_PRESENT_SURFACE_CAPABILITIES_KHR

To query the basic capabilities of a surface, needed in order to create a swapchain, call:

```c
// Provided by VK_EXT_display_surface_counter
VkResult vkGetPhysicalDeviceSurfaceCapabilities2EXT(
    VkPhysicalDevice physicalDevice,
    VkSurfaceKHR surface,
    VkSurfaceCapabilities2EXT* pSurfaceCapabilities);
```

- `physicalDevice` is the physical device that will be associated with the swapchain to be created, as described for `vkCreateSwapchainKHR`.
- `surface` is the surface that will be associated with the swapchain.
- `pSurfaceCapabilities` is a pointer to a `VkSurfaceCapabilities2EXT` structure in which the capabilities are returned.

`vkGetPhysicalDeviceSurfaceCapabilities2EXT` behaves similarly to `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`, with the ability to return extended information by adding extending structures to the `pNext` chain of its `pSurfaceCapabilities` parameter.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is VK_TRUE, `vkGetPhysicalDeviceSurfaceCapabilities2EXT` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

**Valid Usage**

- VUID-vkGetPhysicalDeviceSurfaceCapabilities2EXT-surface-06211
  surface must be supported by `physicalDevice`, as reported by `vkGetPhysicalDeviceSurfaceSupportKHR` or an equivalent platform-specific mechanism

**Valid Usage (Implicit)**

- VUID-vkGetPhysicalDeviceSurfaceCapabilities2EXT-physicalDevice-parameter
  `physicalDevice` must be a valid `VkPhysicalDevice` handle

- VUID-vkGetPhysicalDeviceSurfaceCapabilities2EXT-surface-parameter
  `surface` must be a valid `VkSurfaceKHR` handle

- VUID-vkGetPhysicalDeviceSurfaceCapabilities2EXT-pSurfaceCapabilities-parameter
  `pSurfaceCapabilities` must be a valid pointer to a `VkSurfaceCapabilities2EXT` structure

- VUID-vkGetPhysicalDeviceSurfaceCapabilities2EXT-commonparent
  Both of `physicalDevice`, and `surface` must have been created, allocated, or retrieved from the same `VkInstance`
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY
• VK_ERROR_SURFACE_LOST_KHR

The VkSurfaceCapabilities2EXT structure is defined as:

```c
// Provided by VK_EXT_display_surface_counter
typedef struct VkSurfaceCapabilities2EXT {
    VkStructureType sType;
    void* pNext;
    uint32_t minImageCount;
    uint32_t maxImageCount;
    VkExtent2D currentExtent;
    VkExtent2D minImageExtent;
    VkExtent2D maxImageExtent;
    uint32_t maxImageArrayLayers;
    VkSurfaceTransformFlagsKHR supportedTransforms;
    VkSurfaceTransformFlagBitsKHR currentTransform;
    VkCompositeAlphaFlagsKHR supportedCompositeAlpha;
    VkImageUsageFlags supportedUsageFlags;
    VkSurfaceCounterFlagsEXT supportedSurfaceCounters;
} VkSurfaceCapabilities2EXT;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **minImageCount** is the minimum number of images the specified device supports for a swapchain created for the surface, and will be at least one.
- **maxImageCount** is the maximum number of images the specified device supports for a swapchain created for the surface, and will be either 0, or greater than or equal to minImageCount. A value of 0 means that there is no limit on the number of images, though there **may** be limits related to the total amount of memory used by presentable images.
- **currentExtent** is the current width and height of the surface, or the special value (0xFFFFFFFF, 0xFFFFFFFF) indicating that the surface size will be determined by the extent of a swapchain targeting the surface.
- **minImageExtent** contains the smallest valid swapchain extent for the surface on the specified device. The width and height of the extent will each be less than or equal to the corresponding width and height of currentExtent, unless currentExtent has the special value described above.
- **maxImageExtent** contains the largest valid swapchain extent for the surface on the specified
device. The width and height of the extent will each be greater than or equal to the corresponding width and height of minImageExtent. The width and height of the extent will each be greater than or equal to the corresponding width and height of currentExtent, unless currentExtent has the special value described above.

- maxImageArrayLayers is the maximum number of layers presentable images can have for a swapchain created for this device and surface, and will be at least one.

- supportedTransforms is a bitmask of VkSurfaceTransformFlagBitsKHR indicating the presentation transforms supported for the surface on the specified device. At least one bit will be set.

- currentTransform is VkSurfaceTransformFlagBitsKHR value indicating the surface's current transform relative to the presentation engine's natural orientation.

- supportedCompositeAlpha is a bitmask of VkCompositeAlphaFlagBitsKHR, representing the alpha compositing modes supported by the presentation engine for the surface on the specified device, and at least one bit will be set. Opaque composition can be achieved in any alpha compositing mode by either using an image format that has no alpha component, or by ensuring that all pixels in the presentable images have an alpha value of 1.0.

- supportedUsageFlags is a bitmask of VkImageUsageFlagBits representing the ways the application can use the presentable images of a swapchain created with VkPresentModeKHR set to VK_PRESENT_MODE_IMMEDIATE_KHR, VK_PRESENT_MODE_MAILBOX_KHR, VK_PRESENT_MODE_FIFO_KHR or VK_PRESENT_MODE_FIFO_RELAXED_KHR for the surface on the specified device. VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT must be included in the set. Implementations may support additional usages.

- supportedSurfaceCounters is a bitmask of VkSurfaceCounterFlagBitsEXT indicating the supported surface counter types.

---

**Valid Usage**

- VUID-VkSurfaceCapabilities2EXT-supportedSurfaceCounters-01246
  supportedSurfaceCounters must not include VK_SURFACE_COUNTER_VBLANK_BIT_EXT unless the surface queried is a display surface

---

**Valid Usage (Implicit)**

- VUID-VkSurfaceCapabilities2EXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_EXT

- VUID-VkSurfaceCapabilities2EXT-pNext-pNext
  pNext must be NULL

---

Bits which can be set in VkSurfaceCapabilities2EXT::supportedSurfaceCounters, indicating supported surface counter types, are:

```c
// Provided by VK_EXT_display_surface_counter
typedef enum VkSurfaceCounterFlagBitsEXT {
```
• **VK_SURFACE_COUNTER_VBLANK_BIT_EXT** specifies a counter incrementing once every time a vertical blanking period occurs on the display associated with the surface.

```c
VK_SURFACE_COUNTER_VBLANK_BIT_EXT = 0x00000001,
VK_SURFACE_COUNTER_VBLANK_EXT = VK_SURFACE_COUNTER_VBLANK_BIT_EXT,
} VkSurfaceCounterFlagBitsEXT;
```

// Provided by VK_EXT_display_surface_counter
typedef VkFlags VkSurfaceCounterFlagsEXT;

**VkSurfaceCounterFlagsEXT** is a bitmask type for setting a mask of zero or more **VkSurfaceCounterFlagBitsEXT**.

Bits which **may** be set in **VkSurfaceCapabilitiesKHR::supportedTransforms** indicating the presentation transforms supported for the surface on the specified device, and possible values of **VkSurfaceCapabilitiesKHR::currentTransform** indicating the surface’s current transform relative to the presentation engine's natural orientation, are:

```c
// Provided by VK_KHR_surface
typedef enum VkSurfaceTransformFlagBitsKHR {
    VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR = 0x00000001,
    VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR = 0x00000002,
    VK_SURFACE_TRANSFORM_ROTATE_180_BIT_KHR = 0x00000004,
    VK_SURFACE_TRANSFORM_ROTATE_270_BIT_KHR = 0x00000008,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_BIT_KHR = 0x00000010,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_90_BIT_KHR = 0x00000020,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_180_BIT_KHR = 0x00000040,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_270_BIT_KHR = 0x00000080,
    VK_SURFACE_TRANSFORM_INHERIT_BIT_KHR = 0x00000100,
} VkSurfaceTransformFlagBitsKHR;
```

• **VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR** specifies that image content is presented without being transformed.

• **VK_SURFACE_TRANSFORM.Rotate_90_BIT_KHR** specifies that image content is rotated 90 degrees clockwise.

• **VK_SURFACE_TRANSFORM.Rotate_180_BIT_KHR** specifies that image content is rotated 180 degrees clockwise.

• **VK_SURFACE_TRANSFORM.Rotate_270_BIT_KHR** specifies that image content is rotated 270 degrees clockwise.

• **VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_BIT_KHR** specifies that image content is mirrored horizontally.

• **VK_SURFACE_TRANSFORM.Horizontal_MIRROR_ROTATE_90_BIT_KHR** specifies that image content is mirrored horizontally, then rotated 90 degrees clockwise.

• **VK_SURFACE_TRANSFORM.HORIZONTAL_MIRROR_ROTATE_180_BIT_KHR** specifies that image content is mirrored horizontally, then rotated 180 degrees clockwise.
mirrored horizontally, then rotated 180 degrees clockwise.

- **VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_270_BIT_KHR** specifies that image content is mirrored horizontally, then rotated 270 degrees clockwise.
- **VK_SURFACE_TRANSFORM_INHERIT_BIT_KHR** specifies that the presentation transform is not specified, and is instead determined by platform-specific considerations and mechanisms outside Vulkan.

```c
// Provided by VK_KHR_display
typedef VkFlags VkSurfaceTransformFlagsKHR;
```

**VkSurfaceTransformFlagsKHR** is a bitmask type for setting a mask of zero or more **VkSurfaceTransformFlagBitsKHR**.

The **supportedCompositeAlpha** member is of type **VkCompositeAlphaFlagBitsKHR**, containing the following values:

```c
// Provided by VK_KHR_surface
typedef enum VkCompositeAlphaFlagBitsKHR {
    VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR = 0x00000001,
    VK_COMPOSITE_ALPHA_PRE_MULTIPLIED_BIT_KHR = 0x00000002,
    VK_COMPOSITE_ALPHA_POST_MULTIPLIED_BIT_KHR = 0x00000004,
    VK_COMPOSITE_ALPHA_INHERIT_BIT_KHR = 0x00000008,
} VkCompositeAlphaFlagBitsKHR;
```

These values are described as follows:

- **VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR**: The alpha component, if it exists, of the images is ignored in the compositing process. Instead, the image is treated as if it has a constant alpha of 1.0.
- **VK_COMPOSITE_ALPHA_PRE_MULTIPLIED_BIT_KHR**: The alpha component, if it exists, of the images is respected in the compositing process. The non-alpha components of the image are expected to already be multiplied by the alpha component by the application.
- **VK_COMPOSITE_ALPHA_POST_MULTIPLIED_BIT_KHR**: The alpha component, if it exists, of the images is respected in the compositing process. The non-alpha components of the image are not expected to already be multiplied by the alpha component by the application; instead, the compositor will multiply the non-alpha components of the image by the alpha component during compositing.
- **VK_COMPOSITE_ALPHA_INHERIT_BIT_KHR**: The way in which the presentation engine treats the alpha component in the images is unknown to the Vulkan API. Instead, the application is responsible for setting the composite alpha blending mode using native window system commands. If the application does not set the blending mode using native window system commands, then a platform-specific default will be used.

```c
// Provided by VK_KHR_surface
typedef VkFlags VkCompositeAlphaFlagsKHR;
```

**VkCompositeAlphaFlagsKHR** is a bitmask type for setting a mask of zero or more
30.5.2. Surface Format Support

To query the supported swapchain format-color space pairs for a surface, call:

```c
// Provided by VK_KHR_surface
VkResult vkGetPhysicalDeviceSurfaceFormatsKHR(
    VkPhysicalDevice physicalDevice,
    VkSurfaceKHR surface,
    uint32_t* pSurfaceFormatCount,
    VkSurfaceFormatKHR* pSurfaceFormats);
```

- **physicalDevice** is the physical device that will be associated with the swapchain to be created, as described for `vkCreateSwapchainKHR`.
- **surface** is the surface that will be associated with the swapchain.
- **pSurfaceFormatCount** is a pointer to an integer related to the number of format pairs available or queried, as described below.
- **pSurfaceFormats** is either `NULL` or a pointer to an array of `VkSurfaceFormatKHR` structures.

If `pSurfaceFormats` is `NULL`, then the number of format pairs supported for the given `surface` is returned in `pSurfaceFormatCount`. Otherwise, `pSurfaceFormatCount` must point to a variable set by the user to the number of elements in the `pSurfaceFormats` array, and on return the variable is overwritten with the number of structures actually written to `pSurfaceFormats`. If the value of `pSurfaceFormatCount` is less than the number of format pairs supported, at most `pSurfaceFormatCount` structures will be written, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available format pairs were returned.

The number of format pairs supported must be greater than or equal to 1. `pSurfaceFormats` must not contain an entry whose value for `format` is `VK_FORMAT_UNDEFINED`.

If `pSurfaceFormats` includes an entry whose value for `colorSpace` is `VK_COLOR_SPACE_SRGB_NONLINEAR_KHR` and whose value for `format` is a UNORM (or SRGB) format and the corresponding SRGB (or UNORM) format is a color renderable format for `VK_IMAGE_TILING_OPTIMAL`, then `pSurfaceFormats` must also contain an entry with the same value for `colorSpace` and `format` equal to the corresponding SRGB (or UNORM) format.

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

**Valid Usage**

- **VUID-vkGetPhysicalDeviceSurfaceFormatsKHR-surface-06211**
  - `surface` must be supported by `physicalDevice`, as reported by `vkGetPhysicalDeviceSurfaceSupportKHR` or an equivalent platform-specific mechanism
**Valid Usage (Implicit)**

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

- VUID-vkGetPhysicalDeviceSurfaceFormatsKHR-surface-parameter
  - `surface` must be a valid `VkSurfaceKHR` handle

- VUID-vkGetPhysicalDeviceSurfaceFormatsKHR-pSurfaceFormatCount-parameter
  - `pSurfaceFormatCount` must be a valid pointer to a `uint32_t` value

- VUID-vkGetPhysicalDeviceSurfaceFormatsKHR-pSurfaceFormats-parameter
  - If the value referenced by `pSurfaceFormatCount` is not 0, and `pSurfaceFormats` is not NULL, `pSurfaceFormats` must be a valid pointer to an array of `pSurfaceFormatCount` `VkSurfaceFormatKHR` structures

- VUID-vkGetPhysicalDeviceSurfaceFormatsKHR-commonparent
  - Both of `physicalDevice`, and `surface` must have been created, allocated, or retrieved from the same `VkInstance`

---

**Return Codes**

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

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_SURFACE_LOST_KHR`

The `VkSurfaceFormatKHR` structure is defined as:

```c
// Provided by VK_KHR_surface
typedef struct VkSurfaceFormatKHR {
    VkFormat format;
    VkColorSpaceKHR colorSpace;
} VkSurfaceFormatKHR;
```

- `format` is a `VkFormat` that is compatible with the specified surface.
- `colorSpace` is a presentation `VkColorSpaceKHR` that is compatible with the surface.

To query the supported swapchain format tuples for a surface, call:

```c
// Provided by VK_KHR_get_surface_capabilities2
VkResult vkGetPhysicalDeviceSurfaceFormats2KHR(
```
Commit 7

vkGetPhysicalDeviceSurfaceFormats2KHR

const VkPhysicalDeviceSurfaceInfo2KHR* pSurfaceInfo,
uint32_t* pSurfaceFormatCount,
VkSurfaceFormat2KHR* pSurfaceFormats);

- **physicalDevice** is the physical device that will be associated with the swapchain to be created, as described for vkCreateSwapchainKHR.

- **pSurfaceInfo** is a pointer to a `VkPhysicalDeviceSurfaceInfo2KHR` structure describing the surface and other fixed parameters that would be consumed by vkCreateSwapchainKHR.

- **pSurfaceFormatCount** is a pointer to an integer related to the number of format tuples available or queried, as described below.

- **pSurfaceFormats** is either NULL or a pointer to an array of `VkSurfaceFormat2KHR` structures.

vkGetPhysicalDeviceSurfaceFormats2KHR behaves similarly to vkGetPhysicalDeviceSurfaceFormatsKHR, with the ability to be extended via pNext chains.

If pSurfaceFormats is NULL, then the number of format tuples supported for the given surface is returned in pSurfaceFormatCount. Otherwise, pSurfaceFormatCount must point to a variable set by the user to the number of elements in the pSurfaceFormats array, and on return the variable is overwritten with the number of structures actually written to pSurfaceFormats. If the value of pSurfaceFormatCount is less than the number of format tuples supported, at most pSurfaceFormatCount structures will be written, and VK_INCOMPLETE will be returned instead of VK_SUCCESS, to indicate that not all the available values were returned.

If VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations is VK_TRUE, vkGetPhysicalDeviceSurfaceFormats2KHR must not return VK_ERROR_OUT_OF_HOST_MEMORY.

**Valid Usage**

- VUID-vkGetPhysicalDeviceSurfaceFormats2KHR-pSurfaceInfo-06210 pSurfaceInfo->surface must be supported by physicalDevice, as reported by vkGetPhysicalDeviceSurfaceSupportKHR or an equivalent platform-specific mechanism

**Valid Usage (Implicit)**

- VUID-vkGetPhysicalDeviceSurfaceFormats2KHR-physicalDevice-parameter physicalDevice must be a valid VkPhysicalDevice handle

- VUID-vkGetPhysicalDeviceSurfaceFormats2KHR-pSurfaceInfo-parameter pSurfaceInfo must be a valid pointer to a valid VkPhysicalDeviceSurfaceInfo2KHR structure

- VUID-vkGetPhysicalDeviceSurfaceFormats2KHR-pSurfaceFormatCount-parameter pSurfaceFormatCount must be a valid pointer to a uint32_t value

- VUID-vkGetPhysicalDeviceSurfaceFormats2KHR-pSurfaceFormats-parameter If the value referenced by pSurfaceFormatCount is not 0, and pSurfaceFormats is not NULL,
**pSurfaceFormats must** be a valid pointer to an array of `pSurfaceFormatCount` `VkSurfaceFormat2KHR` structures.

### Return Codes

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

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_SURFACE_LOST_KHR`

The `VkSurfaceFormat2KHR` structure is defined as:

```c
// Provided by VK_KHR_get_surface_capabilities2
typedef struct VkSurfaceFormat2KHR {
    VkStructureType sType;
    void* pNext;
    VkSurfaceFormatKHR surfaceFormat;
} VkSurfaceFormat2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `surfaceFormat` is a `VkSurfaceFormatKHR` structure describing a format-color space pair that is compatible with the specified surface.

### Valid Usage (Implicit)

- `VUID-VkSurfaceFormat2KHR-sType-sType` `sType` must be `VK_STRUCTURE_TYPE_SURFACE_FORMAT_2_KHR`
- `VUID-VkSurfaceFormat2KHR-pNext-pNext` `pNext` must be `NULL`

While the format of a presentable image refers to the encoding of each pixel, the colorSpace determines how the presentation engine interprets the pixel values. A color space in this document refers to a specific color space (defined by the chromaticities of its primaries and a white point in CIE Lab), and a transfer function that is applied before storing or transmitting color data in the given color space.

Possible values of `VkSurfaceFormatKHR::colorSpace`, specifying supported color spaces of a presentation engine, are:
• **VK_COLOR_SPACE_SRGB_NONLINEAR_KHR** specifies support for the sRGB color space.

• **VK_COLOR_SPACE_DISPLAY_P3_NONLINEAR_EXT** specifies support for the Display-P3 color space to be displayed using an sRGB-like EOTF (defined below).

• **VK_COLOR_SPACE_EXTENDED_SRGB_LINEAR_EXT** specifies support for the extended sRGB color space to be displayed using a linear EOTF.

• **VK_COLOR_SPACE_EXTENDED_SRGB_NONLINEAR_EXT** specifies support for the extended sRGB color space to be displayed using an sRGB EOTF.

• **VK_COLOR_SPACE_DISPLAY_P3_LINEAR_EXT** specifies support for the Display-P3 color space to be displayed using a linear EOTF.

• **VK_COLOR_SPACE_DCI_P3_NONLINEAR_EXT** specifies support for the DCI-P3 color space to be displayed using the DCI-P3 EOTF. Note that values in such an image are interpreted as XYZ encoded color data by the presentation engine.

• **VK_COLOR_SPACE_BT709_LINEAR_EXT** specifies support for the BT709 color space to be displayed...
using a linear EOTF.

- **VK_COLOR_SPACE_BT709_NONLINEAR_EXT** specifies support for the BT709 color space to be displayed using the SMPTE 170M EOTF.

- **VK_COLOR_SPACE_BT2020_LINEAR_EXT** specifies support for the BT2020 color space to be displayed using a linear EOTF.

- **VK_COLOR_SPACE_HDR10_ST2084_EXT** specifies support for the HDR10 (BT2020 color) space to be displayed using the SMPTE ST2084 Perceptual Quantizer (PQ) EOTF.

- **VK_COLOR_SPACE_DOLBYVISION_EXT** specifies support for the Dolby Vision (BT2020 color space), proprietary encoding, to be displayed using the SMPTE ST2084 EOTF.

- **VK_COLOR_SPACE_HDR10_HLG_EXT** specifies support for the HDR10 (BT2020 color space) to be displayed using the Hybrid Log Gamma (HLG) EOTF.

- **VK_COLOR_SPACE_ADOBERGB_LINEAR_EXT** specifies support for the AdobeRGB color space to be displayed using a linear EOTF.

- **VK_COLOR_SPACE_ADOBERGB_NONLINEAR_EXT** specifies support for the AdobeRGB color space to be displayed using the Gamma 2.2 EOTF.

- **VK_COLOR_SPACE_PASS_THROUGH_EXT** specifies that color components are used “as is”. This is intended to allow applications to supply data for color spaces not described here.

The color components of non-linear color space swap chain images must have had the appropriate transfer function applied. The color space selected for the swap chain image will not affect the processing of data written into the image by the implementation. Vulkan requires that all implementations support the sRGB transfer function by use of an SRGB pixel format. Other transfer functions, such as SMPTE 170M or SMPTE2084, can be performed by the application shader. This extension defines enums for *VkColorSpaceKHR* that correspond to the following color spaces:

### Table 39. Color Spaces and Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Red Primary</th>
<th>Green Primary</th>
<th>Blue Primary</th>
<th>White-point</th>
<th>Transfer function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCI-P3</td>
<td>1.000, 0.000</td>
<td>0.000, 1.000</td>
<td>0.000, 0.000</td>
<td>0.3333, 0.3333</td>
<td>DCI P3</td>
</tr>
<tr>
<td>Display-P3</td>
<td>0.680, 0.320</td>
<td>0.265, 0.690</td>
<td>0.150, 0.060</td>
<td>0.3127, 0.3290 (D65)</td>
<td>Display-P3</td>
</tr>
<tr>
<td>BT709</td>
<td>0.640, 0.330</td>
<td>0.300, 0.600</td>
<td>0.150, 0.060</td>
<td>0.3127, 0.3290 (D65)</td>
<td>ITU (SMPTE 170M)</td>
</tr>
<tr>
<td>sRGB</td>
<td>0.640, 0.330</td>
<td>0.300, 0.600</td>
<td>0.150, 0.060</td>
<td>0.3127, 0.3290 (D65)</td>
<td>sRGB</td>
</tr>
<tr>
<td>extended sRGB</td>
<td>0.640, 0.330</td>
<td>0.300, 0.600</td>
<td>0.150, 0.060</td>
<td>0.3127, 0.3290 (D65)</td>
<td>extended sRGB</td>
</tr>
<tr>
<td>HDR10_ST2084</td>
<td>0.708, 0.292</td>
<td>0.170, 0.797</td>
<td>0.131, 0.046</td>
<td>0.3127, 0.3290 (D65)</td>
<td>ST2084 PQ</td>
</tr>
<tr>
<td>DOLBYVISION</td>
<td>0.708, 0.292</td>
<td>0.170, 0.797</td>
<td>0.131, 0.046</td>
<td>0.3127, 0.3290 (D65)</td>
<td>ST2084 PQ</td>
</tr>
</tbody>
</table>

1175
<table>
<thead>
<tr>
<th>Name</th>
<th>Red Primary</th>
<th>Green Primary</th>
<th>Blue Primary</th>
<th>White-point</th>
<th>Transfer function</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDR10_HLG</td>
<td>0.708, 0.292</td>
<td>0.170, 0.797</td>
<td>0.131, 0.046</td>
<td>0.3127, 0.3290</td>
<td>HLG</td>
</tr>
<tr>
<td>AdobeRGB</td>
<td>0.640, 0.330</td>
<td>0.210, 0.710</td>
<td>0.150, 0.060</td>
<td>0.3127, 0.3290</td>
<td>AdobeRGB</td>
</tr>
</tbody>
</table>

The transfer functions are described in the “Transfer Functions” chapter of the [Khronos Data Format Specification](https://www.khronos.org/registry/vulkan/specs/1.2-extensions/specs/vk_KHR_data_format_specification.txt).

Except Display-P3 OETF, which is:

\[
E = \begin{cases} 
1.055 \times L^{2.4} - 0.055 & \text{for } 0.0030186 \leq L \leq 1 \\
12.92 \times L & \text{for } 0 \leq L < 0.0030186
\end{cases}
\]

where \(L\) is the linear value of a color component and \(E\) is the encoded value (as stored in the image in memory).

**Note**

For most uses, the sRGB OETF is equivalent.

### 30.5.3. Surface Presentation Mode Support

To query the supported presentation modes for a surface, call:

```c
// Provided by VK_KHR_surface
VkResult vkGetPhysicalDeviceSurfacePresentModesKHR(
    VkPhysicalDevice physicalDevice,
    VkSurfaceKHR surface,
    uint32_t* pPresentModeCount,
    VkPresentModeKHR* pPresentModes);
```

- `physicalDevice` is the physical device that will be associated with the swapchain to be created, as described for `vkCreateSwapchainKHR`.
- `surface` is the surface that will be associated with the swapchain.
- `pPresentModeCount` is a pointer to an integer related to the number of presentation modes available or queried, as described below.
- `pPresentModes` is either `NULL` or a pointer to an array of `VkPresentModeKHR` values, indicating the supported presentation modes.

If `pPresentModes` is `NULL`, then the number of presentation modes supported for the given `surface` is returned in `pPresentModeCount`. Otherwise, `pPresentModeCount` must point to a variable set by the user to the number of elements in the `pPresentModes` array, and on return the variable is overwritten with the number of values actually written to `pPresentModes`. If the value of
pPresentModeCount is less than the number of presentation modes supported, at most pPresentModeCount values will be written, and VK_INCOMPLETE will be returned instead of VK_SUCCESS, to indicate that not all the available modes were returned.

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

**Valid Usage**

- VUID-vkGetPhysicalDeviceSurfacePresentModesKHR-surface-06211
  surface must be supported by physicalDevice, as reported by vkGetPhysicalDeviceSurfaceSupportKHR or an equivalent platform-specific mechanism

**Valid Usage (Implicit)**

- VUID-vkGetPhysicalDeviceSurfacePresentModesKHR-physicalDevice-parameter
  physicalDevice must be a valid VkPhysicalDevice handle

- VUID-vkGetPhysicalDeviceSurfacePresentModesKHR-surface-parameter
  surface must be a valid VkSurfaceKHR handle

- VUID-vkGetPhysicalDeviceSurfacePresentModesKHR-pPresentModeCount-parameter
  pPresentModeCount must be a valid pointer to a uint32_t value

- VUID-vkGetPhysicalDeviceSurfacePresentModesKHR-pPresentModes-parameter
  If the value referenced by pPresentModeCount is not 0, and pPresentModes is not NULL, pPresentModes must be a valid pointer to an array of pPresentModeCount VkPresentModeKHR values

- VUID-vkGetPhysicalDeviceSurfacePresentModesKHR-commonparent
  Both of physicalDevice, and surface must have been created, allocated, or retrieved from the same VkInstance

**Return Codes**

**Success**

- VK_SUCCESS
- VK_INCOMPLETE

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_SURFACE_LOST_KHR

Possible values of elements of the vkGetPhysicalDeviceSurfacePresentModesKHR::pPresentModes array, indicating the supported presentation modes for a surface, are:
typedef enum VkPresentModeKHR {
    VK_PRESENT_MODE_IMMEDIATE_KHR = 0,
    VK_PRESENT_MODE_MAILBOX_KHR = 1,
    VK_PRESENT_MODE_FIFO_KHR = 2,
    VK_PRESENT_MODE_FIFO_RELAXED_KHR = 3,
} VkPresentModeKHR;

- **VK_PRESENT_MODE_IMMEDIATE_KHR** specifies that the presentation engine does not wait for a vertical blanking period to update the current image, meaning this mode may result in visible tearing. No internal queuing of presentation requests is needed, as the requests are applied immediately.

- **VK_PRESENT_MODE_MAILBOX_KHR** specifies that the presentation engine waits for the next vertical blanking period to update the current image. Tearing cannot be observed. An internal single-entry queue is used to hold pending presentation requests. If the queue is full when a new presentation request is received, the new request replaces the existing entry, and any images associated with the prior entry become available for re-use by the application. One request is removed from the queue and processed during each vertical blanking period in which the queue is non-empty.

- **VK_PRESENT_MODE_FIFO_KHR** specifies that the presentation engine waits for the next vertical blanking period to update the current image. Tearing cannot be observed. An internal queue is used to hold pending presentation requests. New requests are appended to the end of the queue, and one request is removed from the beginning of the queue and processed during each vertical blanking period in which the queue is non-empty. This is the only value of presentMode that is required to be supported.

- **VK_PRESENT_MODE_FIFO_RELAXED_KHR** specifies that the presentation engine generally waits for the next vertical blanking period to update the current image. If a vertical blanking period has already passed since the last update of the current image then the presentation engine does not wait for another vertical blanking period for the update, meaning this mode may result in visible tearing in this case. This mode is useful for reducing visual stutter with an application that will mostly present a new image before the next vertical blanking period, but may occasionally be late, and present a new image just after the next vertical blanking period. An internal queue is used to hold pending presentation requests. New requests are appended to the end of the queue, and one request is removed from the beginning of the queue and processed during or after each vertical blanking period in which the queue is non-empty.

- **VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR** specifies that the presentation engine and application have concurrent access to a single image, which is referred to as a shared presentable image. The presentation engine is only required to update the current image after a new presentation request is received. Therefore the application must make a presentation request whenever an update is required. However, the presentation engine may update the current image at any point, meaning this mode may result in visible tearing.
VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR specifies that the presentation engine and application have concurrent access to a single image, which is referred to as a shared presentable image. The presentation engine periodically updates the current image on its regular refresh cycle. The application is only required to make one initial presentation request, after which the presentation engine must update the current image without any need for further presentation requests. The application can indicate the image contents have been updated by making a presentation request, but this does not guarantee the timing of when it will be updated. This mode may result in visible tearing if rendering to the image is not timed correctly.

The supported VkImageUsageFlagBits of the presentable images of a swapchain created for a surface may differ depending on the presentation mode, and can be determined as per the table below:

**Table 40. Presentable image usage queries**

<table>
<thead>
<tr>
<th>Presentation mode</th>
<th>Image usage flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_PRESENT_MODE_IMMEDIATE_KHR</td>
<td>VkSurfaceCapabilitiesKHR::supportedUsageFlags</td>
</tr>
<tr>
<td>VK_PRESENT_MODE_MAILBOX_KHR</td>
<td>VkSurfaceCapabilitiesKHR::supportedUsageFlags</td>
</tr>
<tr>
<td>VK_PRESENT_MODE_FIFO_KHR</td>
<td>VkSurfaceCapabilitiesKHR::supportedUsageFlags</td>
</tr>
<tr>
<td>VK_PRESENT_MODE_FIFO_RELAXED_KHR</td>
<td>VkSurfaceCapabilitiesKHR::supportedUsageFlags</td>
</tr>
<tr>
<td>VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR</td>
<td>VkSharedPresentSurfaceCapabilitiesKHR::sharedPresentSupportedUsageFlags</td>
</tr>
<tr>
<td>VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR</td>
<td>VkSharedPresentSurfaceCapabilitiesKHR::sharedPresentSupportedUsageFlags</td>
</tr>
</tbody>
</table>

**Note**

For reference, the mode indicated by VK_PRESENT_MODE_FIFO_KHR is equivalent to the behavior of {wgl|glX|egl}SwapBuffers with a swap interval of 1, while the mode indicated by VK_PRESENT_MODE_FIFO_RELAXED_KHR is equivalent to the behavior of {wgl|glX}SwapBuffers with a swap interval of -1 (from the {WGL|GLX}_EXT_swap_control_tear extensions).

### 30.6. Device Group Queries

A logical device that represents multiple physical devices may support presenting from images on more than one physical device, or combining images from multiple physical devices.

To query these capabilities, call:

```c
// Provided by VK_VERSION_1_1 with VK_KHR_swapchain
VkResult vkGetDeviceGroupPresentCapabilitiesKHR(
    VkDevice device,
    VkDeviceGroupPresentCapabilitiesKHR* pDeviceGroupPresentCapabilities);
```
• **device** is the logical device.

• **pDeviceGroupPresentCapabilities** is a pointer to a *VkDeviceGroupPresentCapabilitiesKHR* structure in which the device’s capabilities are returned.

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

### Valid Usage (Implicit)

- VUID-vkGetDeviceGroupPresentCapabilitiesKHR-device-parameter
  
  **device** must be a valid *VkDevice* handle

- VUID-vkGetDeviceGroupPresentCapabilitiesKHR-pDeviceGroupPresentCapabilities-parameter
  
  **pDeviceGroupPresentCapabilities** must be a valid pointer to a *VkDeviceGroupPresentCapabilitiesKHR* structure

### Return Codes

**Success**

- **VK_SUCCESS**

**Failure**

- **VK_ERROR_OUT_OF_HOST_MEMORY**
  
  - **VK_ERROR_OUT_OF_DEVICE_MEMORY**

The **VkDeviceGroupPresentCapabilitiesKHR** structure is defined as:

```c
// Provided by VK_VERSION_1_1 with VK_KHR_swapchain
typedef struct VkDeviceGroupPresentCapabilitiesKHR {
    VkStructureType sType;
    void* pNext;
    uint32_t presentMask[VK_MAX_DEVICE_GROUP_SIZE];
    VkDeviceGroupPresentModeFlagsKHR modes;
} VkDeviceGroupPresentCapabilitiesKHR;
```

- **sType** is the type of this structure.

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

- **presentMask** is an array of **VK_MAX_DEVICE_GROUP_SIZE uint32_t** masks, where the mask at element i is non-zero if physical device i has a presentation engine, and where bit j is set in element i if physical device i can present swapchain images from physical device j. If element i is non-zero, then bit i must be set.

- **modes** is a bitmask of *VkDeviceGroupPresentModeFlagBitsKHR* indicating which device group presentation modes are supported.
modes always has VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR set.

The present mode flags are also used when presenting an image, in VkDeviceGroupPresentInfoKHR::mode.

If a device group only includes a single physical device, then modes must equal VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR.

Valid Usage (Implicit)

- VUID-VkDeviceGroupPresentCapabilitiesKHR-sType-sType
  sType must be VK_STRUCTURE_TYPEDEVICE_GROUP_PRESENT_CAPABILITIES_KHR
- VUID-VkDeviceGroupPresentCapabilitiesKHR-pNext-pNext
  pNext must be NULL

Bits which may be set in VkDeviceGroupPresentCapabilitiesKHR::modes to indicate which device group presentation modes are supported are:

```c
// Provided by VK_VERSION_1_1 with VK_KHR_swapchain
typedef enum VkDeviceGroupPresentModeFlagBitsKHR {
    VKDEVICEGROUP_PRESENT_MODE_LOCAL_BIT_KHR = 0x00000001,
    VKDEVICEGROUP_PRESENT_MODE_REMOTE_BIT_KHR = 0x00000002,
    VKDEVICEGROUP_PRESENT_MODE_SUM_BIT_KHR = 0x00000004,
    VKDEVICEGROUP_PRESENT_MODELOCAL_MULTI_DEVICE_BIT_KHR = 0x00000008,
} VkDeviceGroupPresentModeFlagBitsKHR;
```

- VKDEVICEGROUP_PRESENT_MODE_LOCAL_BIT_KHR specifies that any physical device with a presentation engine can present its own swapchain images.
- VKDEVICEGROUP_PRESENT_MODE_REMOTE_BIT_KHR specifies that any physical device with a presentation engine can present swapchain images from any physical device in its presentMask.
- VKDEVICEGROUP_PRESENT_MODE_SUM_BIT_KHR specifies that any physical device with a presentation engine can present the sum of swapchain images from any physical devices in its presentMask.
- VKDEVICEGROUP_PRESENT_MODELOCAL_MULTI_DEVICE_BIT_KHR specifies that multiple physical devices with a presentation engine can each present their own swapchain images.

```c
// Provided by VK_VERSION_1_1 with VK_KHR_swapchain
typedef VkFlags VkDeviceGroupPresentModeFlagsKHR;
```

VkDeviceGroupPresentModeFlagsKHR is a bitmask type for setting a mask of zero or more VkDeviceGroupPresentModeFlagBitsKHR.

Some surfaces may not be capable of using all the device group present modes.

To query the supported device group present modes for a particular surface, call:
VkResult vkGetDeviceGroupSurfacePresentModesKHR(
    VkDevice device, 
    VkSurfaceKHR surface, 
    VkDeviceGroupPresentModeFlagsKHR* pModes);

- `device` is the logical device.
- `surface` is the surface.
- `pModes` is a pointer to a `VkDeviceGroupPresentModeFlagsKHR` in which the supported device group present modes for the surface are returned.

The modes returned by this command are not invariant, and may change in response to the surface being moved, resized, or occluded. These modes must be a subset of the modes returned by `vkGetDeviceGroupPresentCapabilitiesKHR`.

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

### Valid Usage

- VUID-vkGetDeviceGroupSurfacePresentModesKHR-surface-06212
  - `surface` must be supported by all physical devices associated with `device`, as reported by `vkGetPhysicalDeviceSurfaceSupportKHR` or an equivalent platform-specific mechanism

### Valid Usage (Implicit)

- VUID-vkGetDeviceGroupSurfacePresentModesKHR-device-parameter
  - `device` must be a valid `VkDevice` handle
- VUID-vkGetDeviceGroupSurfacePresentModesKHR-surface-parameter
  - `surface` must be a valid `VkSurfaceKHR` handle
- VUID-vkGetDeviceGroupSurfacePresentModesKHR-pModes-parameter
  - `pModes` must be a valid pointer to a `VkDeviceGroupPresentModeFlagsKHR` value
- VUID-vkGetDeviceGroupSurfacePresentModesKHR-commonparent
  - Both of `device`, and `surface` must have been created, allocated, or retrieved from the same `VkInstance`

### Host Synchronization

- Host access to `surface` must be externally synchronized
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY
• VK_ERROR_SURFACE_LOST_KHR

When using `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_MULTI_DEVICE_BIT_KHR`, the application may need to know which regions of the surface are used when presenting locally on each physical device. Presentation of swapchain images to this surface need only have valid contents in the regions returned by this command.

To query a set of rectangles used in presentation on the physical device, call:

```c
// Provided by VK_VERSION_1_1 with VK_KHR_swapchain
VkResult vkGetPhysicalDevicePresentRectanglesKHR(
    VkPhysicalDevice physicalDevice,
    VkSurfaceKHR surface,
    uint32_t* pRectCount,
    VkRect2D* pRects);
```

• `physicalDevice` is the physical device.
• `surface` is the surface.
• `pRectCount` is a pointer to an integer related to the number of rectangles available or queried, as described below.
• `pRects` is either `NULL` or a pointer to an array of `VkRect2D` structures.

If `pRects` is `NULL`, then the number of rectangles used when presenting the given `surface` is returned in `pRectCount`. Otherwise, `pRectCount` must point to a variable set by the user to the number of elements in the `pRects` array, and on return the variable is overwritten with the number of structures actually written to `pRects`. If the value of `pRectCount` is less than the number of rectangles, at most `pRectCount` structures will be written, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available rectangles were returned.

The values returned by this command are not invariant, and may change in response to the surface being moved, resized, or occluded.

The rectangles returned by this command must not overlap.

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

- VUID-vkGetPhysicalDevicePresentRectanglesKHR-surface-06211
  
  surface must be supported by physicalDevice, as reported by
  vkGetPhysicalDeviceSurfaceSupportKHR or an equivalent platform-specific mechanism

Valid Usage (Implicit)

- VUID-vkGetPhysicalDevicePresentRectanglesKHR-physicalDevice-parameter
  physicalDevice must be a valid VkPhysicalDevice handle

- VUID-vkGetPhysicalDevicePresentRectanglesKHR-surface-parameter
  surface must be a valid VkSurfaceKHR handle

- VUID-vkGetPhysicalDevicePresentRectanglesKHR-pRectCount-parameter
  pRectCount must be a valid pointer to a uint32_t value

- VUID-vkGetPhysicalDevicePresentRectanglesKHR-pRects-parameter
  If the value referenced by pRectCount is not 0, and pRects is not NULL, pRects must be a valid pointer to an array of pRectCount VkRect2D structures

- VUID-vkGetPhysicalDevicePresentRectanglesKHR-commonparent
  Both of physicalDevice, and surface must have been created, allocated, or retrieved from the same VkInstance

Host Synchronization

- Host access to surface must be externally synchronized

Return Codes

Success

- VK_SUCCESS
- VK_INCOMPLETE

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

30.7. WSI Swapchain

A swapchain object (a.k.a. swapchain) provides the ability to present rendering results to a surface. Swapchain objects are represented by VkSwapchainKHR handles:
A swapchain is an abstraction for an array of presentable images that are associated with a surface. The presentable images are represented by `VkImage` objects created by the platform. One image (which can be an array image for multiview/stereoscopic-3D surfaces) is displayed at a time, but multiple images can be queued for presentation. An application renders to the image, and then queues the image for presentation to the surface.

A native window cannot be associated with more than one non-retired swapchain at a time. Further, swapchains cannot be created for native windows that have a non-Vulkan graphics API surface associated with them.

The presentation engine is an abstraction for the platform’s compositor or display engine. The presentation engine may be synchronous or asynchronous with respect to the application and/or logical device.

Some implementations may use the device’s graphics queue or dedicated presentation hardware to perform presentation.

The presentable images of a swapchain are owned by the presentation engine. An application can acquire use of a presentable image from the presentation engine. Use of a presentable image must occur only after the image is returned by `vkAcquireNextImageKHR`, and before it is released by `vkQueuePresentKHR`. This includes transitioning the image layout and rendering commands.

An application can acquire use of a presentable image with `vkAcquireNextImageKHR`. After acquiring a presentable image and before modifying it, the application must use a synchronization primitive to ensure that the presentation engine has finished reading from the image. The application can then transition the image's layout, queue rendering commands to it, etc. Finally, the application presents the image with `vkQueuePresentKHR`, which releases the acquisition of the image.

The presentation engine controls the order in which presentable images are acquired for use by the application.

This allows the platform to handle situations which require out-of-order return of images after presentation. At the same time, it allows the application to generate command buffers referencing all of the images in the swapchain at initialization time, rather than in its main loop.

How this all works is described below.

If a swapchain is created with `presentMode` set to either `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR` or `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`, a single presentable image can be acquired, referred to as a shared presentable image. A shared presentable image may be concurrently....
accessed by the application and the presentation engine, without transitioning the image's layout after it is initially presented.

- With `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR`, the presentation engine is only required to update to the latest contents of a shared presentable image after a present. The application **must** call `vkQueuePresentKHR` to guarantee an update. However, the presentation engine **may** update from it at any time.

- With `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`, the presentation engine will automatically present the latest contents of a shared presentable image during every refresh cycle. The application is only required to make one initial call to `vkQueuePresentKHR`, after which the presentation engine will update from it without any need for further present calls. The application **can** indicate the image contents have been updated by calling `vkQueuePresentKHR`, but this does not guarantee the timing of when updates will occur.

The presentation engine **may** access a shared presentable image at any time after it is first presented. To avoid tearing, an application **should** coordinate access with the presentation engine. This requires presentation engine timing information through platform-specific mechanisms and ensuring that color attachment writes are made available during the portion of the presentation engine's refresh cycle they are intended for.

**Note**

The `VK_KHR_shared_presentable_image` extension does not provide functionality for determining the timing of the presentation engine's refresh cycles.

In order to query a swapchain's status when rendering to a shared presentable image, call:

```c
// Provided by VK_KHR_shared_presentable_image
VkResult vkGetSwapchainStatusKHR(
    VkDevice device,
    VkSwapchainKHR swapchain);
```

- `device` is the device associated with `swapchain`.
- `swapchain` is the swapchain to query.

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

**Valid Usage (Implicit)**

- VUID-vkGetSwapchainStatusKHR-device-parameter
  `device` **must** be a valid `VkDevice` handle
- VUID-vkGetSwapchainStatusKHR-swapchain-parameter
  `swapchain` **must** be a valid `VkSwapchainKHR` handle
- VUID-vkGetSwapchainStatusKHR-commonparent
  Both of `device`, and `swapchain` **must** have been created, allocated, or retrieved from the same `VkInstance`
Host Synchronization

- Host access to swapchain must be externally synchronized

Return Codes

Success

- VK_SUCCESS
- VK_SUBOPTIMAL_KHR

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST
- VK_ERROR_OUT_OF_DATE_KHR
- VK_ERROR_SURFACE_LOST_KHR

The possible return values for `vkGetSwapchainStatusKHR` should be interpreted as follows:

- **VK_SUCCESS** specifies the presentation engine is presenting the contents of the shared presentable image, as per the swapchain’s `VkPresentModeKHR`.
- **VK_SUBOPTIMAL_KHR** the swapchain no longer matches the surface properties exactly, but the presentation engine is presenting the contents of the shared presentable image, as per the swapchain’s `VkPresentModeKHR`.
- **VK_ERROR_OUT_OF_DATE_KHR** the surface has changed in such a way that it is no longer compatible with the swapchain.
- **VK_ERROR_SURFACE_LOST_KHR** the surface is no longer available.

Note

The swapchain state may be cached by implementations, so applications should regularly call `vkGetSwapchainStatusKHR` when using a swapchain with `VkPresentModeKHR` set to `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`.

To create a swapchain, call:

```c
// Provided by VK_KHR_swapchain
VkResult vkCreateSwapchainKHR(
    VkDevice device,
    const VkSwapchainCreateInfoKHR* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkSwapchainKHR* pSwapchain);
```
• **device** is the device to create the swapchain for.

• **pCreateInfo** is a pointer to a `VkSwapchainCreateInfoKHR` structure specifying the parameters of the created swapchain.

• **pAllocator** is the allocator used for host memory allocated for the swapchain object when there is no more specific allocator available (see Memory Allocation).

• **pSwapchain** is a pointer to a `VkSwapchainKHR` handle in which the created swapchain object will be returned.

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

### Valid Usage

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

### Valid Usage (Implicit)

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

- VUID-vkCreateSwapchainKHR-pCreateInfo-parameter
  `pCreateInfo` **must** be a valid pointer to a valid `VkSwapchainCreateInfoKHR` structure

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

- VUID-vkCreateSwapchainKHR-pSwapchain-parameter
  `pSwapchain` **must** be a valid pointer to a `VkSwapchainKHR` handle

### Host Synchronization

- Host access to `pCreateInfo->surface` **must** be externally synchronized

### Return Codes

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
The `VkSwapchainCreateInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_swapchain
typedef struct VkSwapchainCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkSwapchainCreateFlagsKHR flags;
    VkSurfaceKHR surface;
    uint32_t minImageCount;
    VkFormat imageFormat;
    VkColorSpaceKHR imageColorSpace;
    VkExtent2D imageExtent;
    uint32_t imageArrayLayers;
    VkImageUsageFlags imageUsage;
    VkSharingMode imageSharingMode;
    uint32_t queueFamilyIndexCount;
    const uint32_t* pQueueFamilyIndices;
    VkSurfaceTransformFlagBitsKHR preTransform;
    VkCompositeAlphaFlagBitsKHR compositeAlpha;
    VkPresentModeKHR presentMode;
    VkBool32 clipped;
    VkSwapchainKHR oldSwapchain;
} VkSwapchainCreateInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **flags** is a bitmask of `VkSwapchainCreateFlagBitsKHR` indicating parameters of the swapchain creation.
- **surface** is the surface onto which the swapchain will present images. If the creation succeeds, the swapchain becomes associated with `surface`.
- **minImageCount** is the minimum number of presentable images that the application needs. The implementation will either create the swapchain with at least that many images, or it will fail to create the swapchain.
- **imageFormat** is a `VkFormat` value specifying the format the swapchain image(s) will be created with.
- **imageColorSpace** is a `VkColorSpaceKHR` value specifying the way the swapchain interprets image data.
- **imageExtent** is the size (in pixels) of the swapchain image(s). The behavior is platform-dependent if the image extent does not match the surface’s `currentExtent` as returned by
Note
On some platforms, it is normal that maxImageExtent may become \((0, 0)\), for example when the window is minimized. In such a case, it is not possible to create a swapchain due to the Valid Usage requirements.

- **imageArrayLayers** is the number of views in a multiview/stereo surface. For non-stereoscopic-3D applications, this value is 1.
- **imageUsage** is a bitmask of VkImageUsageFlagBits describing the intended usage of the (acquired) swapchain images.
- **imageSharingMode** is the sharing mode used for the image(s) of the swapchain.
- **queueFamilyIndexCount** is the number of queue families having access to the image(s) of the swapchain when imageSharingMode is VK_SHARING_MODE_CONCURRENT.
- **pQueueFamilyIndices** is a pointer to an array of queue family indices having access to the images(s) of the swapchain when imageSharingMode is VK_SHARING_MODE_CONCURRENT.
- **preTransform** is a VkSurfaceTransformFlagBitsKHR value describing the transform, relative to the presentation engine's natural orientation, applied to the image content prior to presentation. If it does not match the currentTransform value returned by vkGetPhysicalDeviceSurfaceCapabilitiesKHR, the presentation engine will transform the image content as part of the presentation operation.
- **compositeAlpha** is a VkCompositeAlphaFlagBitsKHR value indicating the alpha compositing mode to use when this surface is composited together with other surfaces on certain window systems.
- **presentMode** is the presentation mode the swapchain will use. A swapchain's present mode determines how incoming present requests will be processed and queued internally.
- **clipped** specifies whether the Vulkan implementation is allowed to discard rendering operations that affect regions of the surface that are not visible.
  - If set to VK_TRUE, the presentable images associated with the swapchain may not own all of their pixels. Pixels in the presentable images that correspond to regions of the target surface obscured by another window on the desktop, or subject to some other clipping mechanism will have undefined content when read back. Fragment shaders may not execute for these pixels, and thus any side effects they would have had will not occur. Setting VK_TRUE does not guarantee any clipping will occur, but allows more efficient presentation methods to be used on some platforms.
  - If set to VK_FALSE, presentable images associated with the swapchain will own all of the pixels they contain.

Note
Applications should set this value to VK_TRUE if they do not expect to read back the content of presentable images before presenting them or after reacquiring them, and if their fragment shaders do not have any side effects that require them to run for all pixels in the presentable image.
• oldSwapchain must be VK_NULL_HANDLE in Vulkan SC [SCID-4].

Valid Usage

• VUID-VkSwapchainCreateInfoKHR-surface-01270
  surface must be a surface that is supported by the device as determined using vkGetPhysicalDeviceSurfaceSupportKHR

• VUID-VkSwapchainCreateInfoKHR-minImageCount-01272
  minImageCount must be less than or equal to the value returned in the maxImageCount member of the VkSurfaceCapabilitiesKHR structure returned by vkGetPhysicalDeviceSurfaceCapabilitiesKHR for the surface if the returned maxImageCount is not zero

• VUID-VkSwapchainCreateInfoKHR-presentMode-02839
  If presentMode is not VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR nor VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR, then minImageCount must be greater than or equal to the value returned in the minImageCount member of the VkSurfaceCapabilitiesKHR structure returned by vkGetPhysicalDeviceSurfaceCapabilitiesKHR for the surface

• VUID-VkSwapchainCreateInfoKHR-minImageCount-01383
  minImageCount must be 1 if presentMode is either VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR or VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR

• VUID-VkSwapchainCreateInfoKHR-imageFormat-01273
  imageFormat and imageColorSpace must match the format and colorSpace members, respectively, of one of the VkSurfaceFormatKHR structures returned by vkGetPhysicalDeviceSurfaceFormatsKHR for the surface

• VUID-VkSwapchainCreateInfoKHR-imageExtent-01274
  imageExtent must be between minImageExtent and maxImageExtent, inclusive, where minImageExtent and maxImageExtent are members of the VkSurfaceCapabilitiesKHR structure returned by vkGetPhysicalDeviceSurfaceCapabilitiesKHR for the surface

• VUID-VkSwapchainCreateInfoKHR-imageExtent-01689
  imageExtent members width and height must both be non-zero

• VUID-VkSwapchainCreateInfoKHR-imageArrayLayers-01275
  imageArrayLayers must be greater than 0 and less than or equal to the maxImageArrayLayers member of the VkSurfaceCapabilitiesKHR structure returned by vkGetPhysicalDeviceSurfaceCapabilitiesKHR for the surface

• VUID-VkSwapchainCreateInfoKHR-presentMode-01427
  If presentMode is VK_PRESENT_MODE_IMMEDIATE_KHR, VK_PRESENT_MODE_MAILBOX_KHR, VK_PRESENT_MODE_FIFO_KHR or VK_PRESENT_MODE_FIFO_RELAXED_KHR, imageUsage must be a subset of the supported usage flags present in the supportedUsageFlags member of the VkSurfaceCapabilitiesKHR structure returned by vkGetPhysicalDeviceSurfaceCapabilitiesKHR for surface

• VUID-VkSwapchainCreateInfoKHR-imageUsage-01384
  If presentMode is VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR or
VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR, imageUsage must be a subset of the supported usage flags present in the sharedPresentSupportedUsageFlags member of the VkSharedPresentSurfaceCapabilitiesKHR structure returned by vkGetPhysicalDeviceSurfaceCapabilities2KHR for surface

- VUID-VkSwapchainCreateInfoKHR-imageSharingMode-01277
  If imageSharingMode is VK_SHARING_MODE_CONCURRENT, pQueueFamilyIndices must be a valid pointer to an array of queueFamilyIndexCount uint32_t values

- VUID-VkSwapchainCreateInfoKHR-imageSharingMode-01278
  If imageSharingMode is VK_SHARING_MODE_CONCURRENT, queueFamilyIndexCount must be greater than 1

- VUID-VkSwapchainCreateInfoKHR-imageSharingMode-01428
  If imageSharingMode is VK_SHARING_MODE_CONCURRENT, each element of pQueueFamilyIndices must be unique and must be less than pQueueFamilyPropertyCount returned by either vkGetPhysicalDeviceQueueFamilyProperties or vkGetPhysicalDeviceQueueFamilyProperties2 for the physicalDevice that was used to create device

- VUID-VkSwapchainCreateInfoKHR-preTransform-01279
  preTransform must be one of the bits present in the supportedTransforms member of the VkSurfaceCapabilitiesKHR structure returned by vkGetPhysicalDeviceSurfaceCapabilitiesKHR for the surface

- VUID-VkSwapchainCreateInfoKHR-compositeAlpha-01280
  compositeAlpha must be one of the bits present in the supportedCompositeAlpha member of the VkSurfaceCapabilitiesKHR structure returned by vkGetPhysicalDeviceSurfaceCapabilitiesKHR for the surface

- VUID-VkSwapchainCreateInfoKHR-presentMode-01281
  presentMode must be one of the VkPresentModeKHR values returned by vkGetPhysicalDeviceSurfacePresentModesKHR for the surface

- VUID-VkSwapchainCreateInfoKHR-flags-05072
  flags must not contain VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR

- VUID-VkSwapchainCreateInfoKHR-oldSwapchain-05073
  oldSwapchain must be VK_NULL_HANDLE

- VUID-VkSwapchainCreateInfoKHR-imageFormat-01778
  The implied image creation parameters of the swapchain must be supported as reported by vkGetPhysicalDeviceImageFormatProperties

- VUID-VkSwapchainCreateInfoKHR-flags-03168
  If flags contains VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR then the pNext chain must include a VkImageFormatListCreateInfo structure with a viewFormatCount greater than zero and pViewFormats must have an element equal to imageFormat

- VUID-VkSwapchainCreateInfoKHR-pNext-04099
  If a VkImageFormatListCreateInfo structure was included in the pNext chain and VkImageFormatListCreateInfo::viewFormatCount is not zero then all of the formats in VkImageFormatListCreateInfo::pViewFormats must be compatible with the format as described in the compatibility table
If `flags` does not contain `VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR` and the `pNext` chain include a `VkImageFormatListCreateInfo` structure then `VkImageFormatListCreateInfo::viewFormatCount` must be `0` or `1`.

**Valid Usage (Implicit)**

- **VUID-VkSwapchainCreateInfoKHR-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR`

- **VUID-VkSwapchainCreateInfoKHR-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 `VkDeviceGroupSwapchainCreateInfoKHR`, `VkImageFormatListCreateInfo`, or `VkSwapchainCounterCreateInfoEXT`

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

- **VUID-VkSwapchainCreateInfoKHR-flags-parameter**
  `flags` must be a valid combination of `VkSwapchainCreateFlagBitsKHR` values

- **VUID-VkSwapchainCreateInfoKHR-surface-parameter**
  `surface` must be a valid `VkSurfaceKHR` handle

- **VUID-VkSwapchainCreateInfoKHR-imageFormat-parameter**
  `imageFormat` must be a valid `VkFormat` value

- **VUID-VkSwapchainCreateInfoKHR-imageColorSpace-parameter**
  `imageColorSpace` must be a valid `VkColorSpaceKHR` value

- **VUID-VkSwapchainCreateInfoKHR-imageUsage-parameter**
  `imageUsage` must be a valid combination of `VkImageUsageFlagBits` values

- **VUID-VkSwapchainCreateInfoKHR-imageUsage-requiredbitmask**
  `imageUsage` must not be `0`

- **VUID-VkSwapchainCreateInfoKHR-imageSharingMode-parameter**
  `imageSharingMode` must be a valid `VkSharingMode` value

- **VUID-VkSwapchainCreateInfoKHR-preTransform-parameter**
  `preTransform` must be a valid `VkSurfaceTransformFlagBitsKHR` value

- **VUID-VkSwapchainCreateInfoKHR-compositeAlpha-parameter**
  `compositeAlpha` must be a valid `VkCompositeAlphaFlagBitsKHR` value

- **VUID-VkSwapchainCreateInfoKHR-presentMode-parameter**
  `presentMode` must be a valid `VkPresentModeKHR` value

- **VUID-VkSwapchainCreateInfoKHR-oldSwapchain-parent**
  If `oldSwapchain` is a valid handle, it must have been created, allocated, or retrieved from `surface`

- **VUID-VkSwapchainCreateInfoKHR-commonparent**
  Both of `oldSwapchain`, and `surface` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkInstance`
Bits which can be set in VkSwapchainCreateInfoKHR::flags, specifying parameters of swapchain creation, are:

```cpp
// Provided by VK_KHR_swapchain
typedef enum VkSwapchainCreateFlagBitsKHR {
    // Provided by VK_VERSION_1_1 with VK_KHR_swapchain
    VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR = 0x00000001,
    // Provided by VK_VERSION_1_1 with VK_KHR_swapchain
    VK_SWAPCHAIN_CREATE_PROTECTED_BIT_KHR = 0x00000002,
    // Provided by VK_KHR_swapchain_mutable_format
    VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR = 0x00000004,
} VkSwapchainCreateFlagBitsKHR;
```

- **VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR** specifies that images created from the swapchain (i.e. with the `swapchain` member of `VkImageSwapchainCreateInfoKHR` set to this swapchain's handle) must use `VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT`. This flag is not supported in Vulkan SC [SCID-8].

- **VK_SWAPCHAIN_CREATE_PROTECTED_BIT_KHR** specifies that images created from the swapchain are protected images.

- **VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR** specifies that the images of the swapchain can be used to create a `VkImageView` with a different format than what the swapchain was created with. The list of allowed image view formats is specified by adding a `VkImageFormatListCreateInfo` structure to the `pNext` chain of `VkSwapchainCreateInfoKHR`. In addition, this flag also specifies that the swapchain can be created with usage flags that are not supported for the format the swapchain is created with but are supported for at least one of the allowed image view formats.

```cpp
// Provided by VK_KHR_swapchain
typedef VkFlags VkSwapchainCreateFlagsKHR;
```

`VkSwapchainCreateFlagsKHR` is a bitmask type for setting a mask of zero or more `VkSwapchainCreateFlagBitsKHR`.

If the `pNext` chain of `VkSwapchainCreateInfoKHR` includes a `VkDeviceGroupSwapchainCreateInfoKHR` structure, then that structure includes a set of device group present modes that the swapchain can be used with.

The `VkDeviceGroupSwapchainCreateInfoKHR` structure is defined as:

```cpp
// Provided by VK_VERSION_1_1 with VK_KHR_swapchain
typedef struct VkDeviceGroupSwapchainCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkDeviceGroupPresentModeFlagsKHR modes;
} VkDeviceGroupSwapchainCreateInfoKHR;
```

- **sType** is the type of this structure.
• `pNext` is NULL or a pointer to a structure extending this structure.
• `modes` is a bitfield of modes that the swapchain can be used with.

If this structure is not present, `modes` is considered to be `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR`.

### Valid Usage (Implicit)

- VUID-VkDeviceGroupSwapchainCreateInfoKHR-sType-sType  
  `sType` must be `VK_STRUCTURE_TYPE_DEVICE_GROUP_SWAPCHAIN_CREATE_INFO_KHR`
- VUID-VkDeviceGroupSwapchainCreateInfoKHR-modes-parameter  
  `modes` must be a valid combination of `VkDeviceGroupPresentModeFlagBitsKHR` values
- VUID-VkDeviceGroupSwapchainCreateInfoKHR-modes-requiredbitmask  
  `modes` must not be 0

To enable surface counters when creating a swapchain, add a `VkSwapchainCounterCreateInfoEXT` structure to the `pNext` chain of `VkSwapchainCreateInfoKHR`. `VkSwapchainCounterCreateInfoEXT` is defined as:

```c
// Provided by VK_EXT_display_control
typedef struct VkSwapchainCounterCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkSurfaceCounterFlagsEXT surfaceCounters;
} VkSwapchainCounterCreateInfoEXT;
```

• `sType` is the type of this structure.
• `pNext` is NULL or a pointer to a structure extending this structure.
• `surfaceCounters` is a bitmask of `VkSurfaceCounterFlagBitsEXT` specifying surface counters to enable for the swapchain.

### Valid Usage

- VUID-VkSwapchainCounterCreateInfoEXT-surfaceCounters-01244  
  The bits in `surfaceCounters` must be supported by `VkSwapchainCreateInfoKHR::surface`, as reported by `vkGetPhysicalDeviceSurfaceCapabilities2EXT`

### Valid Usage (Implicit)

- VUID-VkSwapchainCounterCreateInfoEXT-sType-sType  
  `sType` must be `VK_STRUCTURE_TYPE_SWAPCHAIN_COUNTER_CREATE_INFO_EXT`
- VUID-VkSwapchainCounterCreateInfoEXT-surfaceCounters-parameter  
  `surfaceCounters` must be a valid combination of `VkSurfaceCounterFlagBitsEXT` values
The requested counters become active when the first presentation command for the associated swapchain is processed by the presentation engine. To query the value of an active counter, use:

```c
// Provided by VK_EXT_display_control
VkResult vkGetSwapchainCounterEXT(
    VkDevice device,  
    VkSwapchainKHR swapchain,  
    VkSurfaceCounterFlagBitsEXT counter,  
    uint64_t* pCounterValue);
```

- `device` is the `VkDevice` associated with `swapchain`.
- `swapchain` is the swapchain from which to query the counter value.
- `counter` is a `VkSurfaceCounterFlagBitsEXT` value specifying the counter to query.
- `pCounterValue` will return the current value of the counter.

If a counter is not available because the swapchain is out of date, the implementation may return `VK_ERROR_OUT_OF_DATE_KHR`.

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

### Valid Usage

- VUID-vkGetSwapchainCounterEXT-swapchain-01245
  One or more present commands on `swapchain` must have been processed by the presentation engine

### Valid Usage (Implicit)

- VUID-vkGetSwapchainCounterEXT-device-parameter
  `device` must be a valid `VkDevice` handle
- VUID-vkGetSwapchainCounterEXT-swapchain-parameter
  `swapchain` must be a valid `VkSwapchainKHR` handle
- VUID-vkGetSwapchainCounterEXT-counter-parameter
  `counter` must be a valid `VkSurfaceCounterFlagBitsEXT` value
- VUID-vkGetSwapchainCounterEXT-pCounterValue-parameter
  `pCounterValue` must be a valid pointer to a `uint64_t` value
- VUID-vkGetSwapchainCounterEXT-commonparent
  Both of `device`, and `swapchain` must have been created, allocated, or retrieved from the same `VkInstance`
As mentioned above, if \texttt{vkCreateSwapchainKHR} succeeds, it will return a handle to a swapchain containing an array of at least \texttt{minImageCount} presentable images.

While acquired by the application, presentable images \textbf{can} be used in any way that equivalent non-presentable images \textbf{can} be used. A presentable image is equivalent to a non-presentable image created with the following \texttt{VkImageCreateInfo} parameters:

### VkImageCreateInfo Field | Value
--- | ---
flags | \texttt{VK\_IMAGE\_CREATE\_SPLIT\_INSTANCE\_BIND\_REGIONS\_BIT} is set if \texttt{VK\_SWAPCHAIN\_CREATE\_SPLIT\_INSTANCE\_BIND\_REGIONS\_BIT\_KHR} is set \texttt{VK\_IMAGE\_CREATE\_PROTECTED\_BIT} is set if \texttt{VK\_SWAPCHAIN\_CREATE\_PROTECTED\_BIT\_KHR} is set \texttt{VK\_IMAGE\_CREATE\_MUTABLE\_FORMAT\_BIT} and \texttt{VK\_IMAGE\_CREATE\_EXTENDED\_USAGE\_BIT\_KHR} are both set if \texttt{VK\_SWAPCHAIN\_CREATE\_MUTABLE\_FORMAT\_BIT\_KHR} is set all other bits are unset
imageType | \texttt{VK\_IMAGE\_TYPE\_2D}
format | \texttt{pCreateInfo->imageFormat}
extent | \{\texttt{pCreateInfo->imageExtent.width}, \texttt{pCreateInfo->imageExtent.height}, 1\}
mipLevels | 1
arrayLayers | \texttt{pCreateInfo->imageArrayLayers}
samples | \texttt{VK\_SAMPLE\_COUNT\_1\_BIT}
tiling | \texttt{VK\_IMAGE\_TILING\_OPTIMAL}
usage | \texttt{pCreateInfo->imageUsage}
sharingMode | \texttt{pCreateInfo->imageSharingMode}
queueFamilyIndexCount | \texttt{pCreateInfo->queueFamilyIndexCount}
VkImageCreateInfo

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pQueueFamilyIndices</td>
<td>pCreateInfo-&gt;pQueueFamilyIndices</td>
</tr>
<tr>
<td>initialLayout</td>
<td>VK_IMAGE_LAYOUT_UNDEFINED</td>
</tr>
</tbody>
</table>

The **surface** must not be destroyed until after the swapchain is destroyed.

If the native window referred to by **surface** is already associated with a Vulkan swapchain, **VK_ERROR_NATIVE_WINDOW_IN_USE_KHR** must be returned.

If the native window referred to by **surface** is already associated with a non-Vulkan graphics API surface, **VK_ERROR_NATIVE_WINDOW_IN_USE_KHR** must be returned.

The native window referred to by **surface** must not become associated with a non-Vulkan graphics API surface before all associated Vulkan swapchains have been destroyed.

Like core functions, several WSI functions, including `vkCreateSwapchainKHR` return **VK_ERROR_DEVICE_LOST** if the logical device was lost. See Lost Device. However, VkSurfaceKHR is not a child of any VkDevice and is not otherwise affected by the lost device. After successfully recreating a VkDevice, the same VkSurfaceKHR can be used to create a new VkSwapchainKHR, provided the previous one was destroyed.

**Note**

As mentioned in Lost Device, after a lost device event, the VkPhysicalDevice may also be lost. If other VkPhysicalDevice are available, they can be used together with the same VkSurfaceKHR to create the new VkSwapchainKHR, however the application must query the surface capabilities again, because they may differ on a per-physical device basis.

Swapchains cannot be destroyed [SCID-4]. If VkPhysicalDeviceVulkanSC10Properties::deviceDestroyFreesMemory is VK_TRUE, the memory for swapchain images is returned to the system when the device is destroyed.

When the **VK_KHR_display_swapchain** extension is enabled, multiple swapchains that share presentable images are created by calling:

```cpp
// Provided by VK_KHR_display_swapchain
VkResult vkCreateSharedSwapchainsKHR(
    VkDevice device,
    uint32_t swapchainCount,
    const VkSwapchainCreateInfoKHR* pCreateInfos,
    const VkAllocationCallbacks* pAllocator,
    VkSwapchainKHR* pSwapchains);
```

- **device** is the device to create the swapchains for.
- **swapchainCount** is the number of swapchains to create.
- **pCreateInfos** is a pointer to an array of VkSwapchainCreateInfoKHR structures specifying the parameters of the created swapchains.
• `pAllocator` is the allocator used for host memory allocated for the swapchain objects when there is no more specific allocator available (see `Memory Allocation`).

• `pSwapchains` is a pointer to an array of `VkSwapchainKHR` handles in which the created swapchain objects will be returned.

`vkCreateSharedSwapchainsKHR` is similar to `vkCreateSwapchainKHR`, except that it takes an array of `VkSwapchainCreateInfoKHR` structures, and returns an array of swapchain objects.

The swapchain creation parameters that affect the properties and number of presentable images must match between all the swapchains. If the displays used by any of the swapchains do not use the same presentable image layout or are incompatible in a way that prevents sharing images, swapchain creation will fail with the result code `VK_ERROR_INCOMPATIBLE_DISPLAY_KHR`. If any error occurs, no swapchains will be created. Images presented to multiple swapchains must be re-acquired from all of them before transitioning away from `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`. After destroying one or more of the swapchains, the remaining swapchains and the presentable images can continue to be used.

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

---

### Valid Usage

- **VUID-vkCreateSharedSwapchainsKHR-device-05068**
  - The number of swapchains currently allocated from `device` plus `swapchainCount` must be less than or equal to the total number of swapchains requested via `VkDeviceObjectReservationCreateInfo::swapchainRequestCount` specified when `device` was created.

### Valid Usage (Implicit)

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

- **VUID-vkCreateSharedSwapchainsKHR-pCreateInfos-parameter**
  - `pCreateInfos` must be a valid pointer to an array of `swapchainCount` valid `VkSwapchainCreateInfoKHR` structures

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

- **VUID-vkCreateSharedSwapchainsKHR-pSwapchains-parameter**
  - `pSwapchains` must be a valid pointer to an array of `swapchainCount` `VkSwapchainKHR` handles

- **VUID-vkCreateSharedSwapchainsKHR-swapchainCount-arraylength**
  - `swapchainCount` must be greater than `0`
Host Synchronization

- Host access to `pCreateInfos[].surface` must be externally synchronized

Return Codes

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INCOMPATIBLE_DISPLAY_KHR`
- `VK_ERROR_DEVICE_LOST`
- `VK_ERROR_SURFACE_LOST_KHR`

To obtain the array of presentable images associated with a swapchain, call:

```c
// Provided by VK_KHR_swapchain
VkResult vkGetSwapchainImagesKHR(
    VkDevice device,         // Provided by VK_KHR_swapchain
    VkSwapchainKHR swapchain, // Provided by VK_KHR_swapchain
    uint32_t* pSwapchainImageCount, // Provided by VK_KHR_swapchain
    VkImage* pSwapchainImages
);                           // Provided by VK_KHR_swapchain
```

- `device` is the device associated with `swapchain`.
- `swapchain` is the swapchain to query.
- `pSwapchainImageCount` is a pointer to an integer related to the number of presentable images available or queried, as described below.
- `pSwapchainImages` is either `NULL` or a pointer to an array of `VkImage` handles.

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

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkGetSwapchainImagesKHR` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`. 
Valid Usage (Implicit)

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

• VUID-vkGetSwapchainImagesKHR-swapchain-parameter
swapchain must be a valid VkSwapchainKHR handle

• VUID-vkGetSwapchainImagesKHR-pSwapchainImageCount-parameter
pSwapchainImageCount must be a valid pointer to a uint32_t value

• VUID-vkGetSwapchainImagesKHR-pSwapchainImages-parameter
If the value referenced by pSwapchainImageCount is not 0, and pSwapchainImages is not NULL, pSwapchainImages must be a valid pointer to an array of pSwapchainImageCount VkImage handles

• VUID-vkGetSwapchainImagesKHR-commonparent
Both of device, and swapchain must have been created, allocated, or retrieved from the same VkInstance

Return Codes

Success
• VK_SUCCESS
• VK_INCOMPLETE

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

Note

By knowing all presentable images used in the swapchain, the application can create command buffers that reference these images prior to entering its main rendering loop.

Images returned by vkGetSwapchainImagesKHR are fully backed by memory before they are passed to the application. All presentable images are initially in the VK_IMAGE_LAYOUT_UNDEFINED layout, thus before using presentable images, the application must transition them to a valid layout for the intended use.

Images can also be created by using vkCreateImage with VkImageSwapchainCreateInfoKHR and bound to swapchain memory using vkBindImageMemory2 with VkBindImageMemorySwapchainInfoKHR. These images can be used anywhere swapchain images are used, and are useful in logical devices with multiple physical devices to create peer memory bindings of swapchain memory. These images and bindings have no effect on what memory is presented. Unlike images retrieved from vkGetSwapchainImagesKHR, these images must be destroyed with vkDestroyImage.
To acquire an available presentable image to use, and retrieve the index of that image, call:

```c
// Provided by VK_KHR_swapchain
VkResult vkAcquireNextImageKHR(
    VkDevice device,
    VkSwapchainKHR swapchain,
    uint64_t timeout,
    VkSemaphore semaphore,
    VkFence fence,
    uint32_t*pImageIndex);
```

- **device** is the device associated with **swapchain**.
- **swapchain** is the non-retired swapchain from which an image is being acquired.
- **timeout** specifies how long the function waits, in nanoseconds, if no image is available.
- **semaphore** is **VK_NULL_HANDLE** or a semaphore to signal.
- **fence** is **VK_NULL_HANDLE** or a fence to signal.
- **pImageIndex** is a pointer to a **uint32_t** in which the index of the next image to use (i.e. an index into the array of images returned by `vkGetSwapchainImagesKHR`) is returned.

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

### Valid Usage

- **VUID-vkAcquireNextImageKHR-swapchain-01285**
  **swapchain** must not be in the retired state

- **VUID-vkAcquireNextImageKHR-semaphore-01286**
  If **semaphore** is not **VK_NULL_HANDLE** it must be unsignalned

- **VUID-vkAcquireNextImageKHR-semaphore-01779**
  If **semaphore** is not **VK_NULL_HANDLE** it must not have any uncompleted signal or wait operations pending

- **VUID-vkAcquireNextImageKHR-fence-01287**
  If **fence** is not **VK_NULL_HANDLE** it must be unsignaled and must not be associated with any other queue command that has not yet completed execution on that queue

- **VUID-vkAcquireNextImageKHR-semaphore-01780**
  **semaphore** and **fence** must not both be equal to **VK_NULL_HANDLE**

- **VUID-vkAcquireNextImageKHR-swapchain-01802**
  If the number of currently acquired images is greater than the difference between the number of images in **swapchain** and the value of **VkSurfaceCapabilitiesKHR::minImageCount** as returned by a call to `vkGetPhysicalDeviceSurfaceCapabilities2KHR` with the **surface** used to create **swapchain**, **timeout** must not be **UINT64_MAX**

- **VUID-vkAcquireNextImageKHR-semaphore-03265**
  **semaphore** must have a **VkSemaphoreType** of **VK_SEMAPHORE_TYPE_BINARY**
Valid Usage (Implicit)

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

• VUID-vkAcquireNextImageKHR-swapchain-parameter
  swapchain must be a valid VkSwapchainKHR handle

• VUID-vkAcquireNextImageKHR-semaphore-parameter
  If semaphore is not VK_NULL_HANDLE, semaphore must be a valid VkSemaphore handle

• VUID-vkAcquireNextImageKHR-fence-parameter
  If fence is not VK_NULL_HANDLE, fence must be a valid VkFence handle

• VUID-vkAcquireNextImageKHR-pImageIndex-parameter
  pImageIndex must be a valid pointer to a uint32_t value

• VUID-vkAcquireNextImageKHR-semaphore-parent
  If semaphore is a valid handle, it must have been created, allocated, or retrieved from device

• VUID-vkAcquireNextImageKHR-fence-parent
  If fence is a valid handle, it must have been created, allocated, or retrieved from device

• VUID-vkAcquireNextImageKHR-commonparent
  Both of device, and swapchain that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same VkInstance

Host Synchronization

• Host access to swapchain must be externally synchronized

• Host access to semaphore must be externally synchronized

• Host access to fence must be externally synchronized

Return Codes

Success

• VK_SUCCESS
• VK_TIMEOUT
• VK_NOT_READY
• VK_SUBOPTIMAL_KHR

Failure

• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY
• VK_ERROR_DEVICE_LOST
When successful, `vkAcquireNextImageKHR` acquires a presentable image from *swapchain* that an application can use, and sets `pImageIndex` to the index of that image within the swapchain. The presentation engine may not have finished reading from the image at the time it is acquired, so the application must use semaphore and/or fence to ensure that the image layout and contents are not modified until the presentation engine reads have completed. If semaphore is not `VK_NULL_HANDLE`, the application may assume that, once `vkAcquireNextImageKHR` returns, the semaphore signal operation referenced by semaphore has been submitted for execution. The order in which images are acquired is implementation-dependent, and may be different than the order the images were presented.

If `timeout` is zero, then `vkAcquireNextImageKHR` does not wait, and will either successfully acquire an image, or fail and return `VK_NOT_READY` if no image is available.

If the specified timeout period expires before an image is acquired, `vkAcquireNextImageKHR` returns `VK_TIMEOUT`. If `timeout` is `UINT64_MAX`, the timeout period is treated as infinite, and `vkAcquireNextImageKHR` will block until an image is acquired or an error occurs.

`vkAcquireNextImageKHR` should not be called if the number of images that the application has currently acquired is greater than the difference between the number of images in *swapchain* and the value of `VkSurfaceCapabilitiesKHR::minImageCount`. If `vkAcquireNextImageKHR` is called when the number of images that the application has currently acquired is less or equal than the difference between the number of images in *swapchain* and the value of `VkSurfaceCapabilitiesKHR::minImageCount`, `vkAcquireNextImageKHR` must return in finite time with an allowed VkResult code.

**Note**

Returning a result in finite time guarantees that the implementation cannot deadlock an application, or suspend its execution indefinitely with correct API usage. Acquiring too many images at once may block indefinitely, which is covered by valid usage when attempting to use `UINT64_MAX`. For example, a scenario here is when a compositor holds on to images which are currently being presented, and there are not any vacant images left to be acquired.

If an image is acquired successfully, `vkAcquireNextImageKHR` must either return `VK_SUCCESS` or `VK_SUBOPTIMAL_KHR`. The implementation may return `VK_SUBOPTIMAL_KHR` if the swapchain no longer matches the surface properties exactly, but can still be used for presentation.

**Note**

`VK_SUBOPTIMAL_KHR` may happen, for example, if the platform surface has been resized but the platform is able to scale the presented images to the new size to produce valid surface updates. It is up to the application to decide whether it prefers to continue using the current swapchain in this state, or to re-create the swapchain to better match the platform surface properties.

If the swapchain images no longer match native surface properties, either `VK_SUBOPTIMAL_KHR` or...
VK_ERROR_OUT_OF_DATE_KHR must be returned. If VK_ERROR_OUT_OF_DATE_KHR is returned, no image is acquired and attempts to present previously acquired images to the swapchain will also fail with VK_ERROR_OUT_OF_DATE_KHR. Applications need to create a new swapchain for the surface to continue presenting if VK_ERROR_OUT_OF_DATE_KHR is returned.

If device loss occurs (see Lost Device) before the timeout has expired, vkAcquireNextImageKHR must return in finite time with either one of the allowed success codes, or VK_ERROR_DEVICE_LOST.

If semaphore is not VK_NULL_HANDLE, the semaphore must be unsignaled, with no signal or wait operations pending. It will become signaled when the application can use the image.

Note
Use of semaphore allows rendering operations to be recorded and submitted before the presentation engine has completed its use of the image.

If fence is not equal to VK_NULL_HANDLE, the fence must be unsignaled, with no signal operations pending. It will become signaled when the application can use the image.

Note
Applications should not rely on vkAcquireNextImageKHR blocking in order to meter their rendering speed. The implementation may return from this function immediately regardless of how many presentation requests are queued, and regardless of when queued presentation requests will complete relative to the call. Instead, applications can use fence to meter their frame generation work to match the presentation rate.

An application must wait until either the semaphore or fence is signaled before accessing the image’s data.

Note
When the presentable image will be accessed by some stage S, the recommended idiom for ensuring correct synchronization is:

- The VkSubmitInfo used to submit the image layout transition for execution includes vkAcquireNextImageKHR::semaphore in its pWaitSemaphores member, with the corresponding element of pWaitDstStageMask including S.
- The synchronization command that performs any necessary image layout transition includes S in both the srcStageMask and dstStageMask.

After a successful return, the image indicated by pImageIndex and its data will be unmodified compared to when it was presented.

Note
Exclusive ownership of presentable images corresponding to a swapchain created with VK_SHARING_MODE_EXCLUSIVE as defined in Resource Sharing is not altered by a call to vkAcquireNextImageKHR. That means upon the first acquisition from such a swapchain presentable images are not owned by any queue family, while at
subsequent acquisitions the presentable images remain owned by the queue family the image was previously presented on.

The possible return values for `vkAcquireNextImageKHR` depend on the `timeout` provided:

- **VK_SUCCESS** is returned if an image became available.
- **VK_ERROR_SURFACE_LOST_KHR** is returned if the surface becomes no longer available.
- **VK_NOT_READY** is returned if `timeout` is zero and no image was available.
- **VK_TIMEOUT** is returned if `timeout` is greater than zero and less than `UINT64_MAX`, and no image became available within the time allowed.
- **VK_SUBOPTIMAL_KHR** is returned if an image became available, and the swapchain no longer matches the surface properties exactly, but can still be used to present to the surface successfully.

**Note**

This may happen, for example, if the platform surface has been resized but the platform is able to scale the presented images to the new size to produce valid surface updates. It is up to the application to decide whether it prefers to continue using the current swapchain indefinitely or temporarily in this state, or to recreate the swapchain to better match the platform surface properties.

- **VK_ERROR_OUT_OF_DATE_KHR** is returned if the surface has changed in such a way that it is no longer compatible with the swapchain, and further presentation requests using the swapchain will fail. Applications must query the new surface properties and recreate their swapchain if they wish to continue presenting to the surface.

If the native surface and presented image sizes no longer match, presentation may fail. If presentation does succeed, the mapping from the presented image to the native surface is implementation-defined. It is the application’s responsibility to detect surface size changes and react appropriately. If presentation fails because of a mismatch in the surface and presented image sizes, a **VK_ERROR_OUT_OF_DATE_KHR** error will be returned.

**Note**

For example, consider a 4x3 window/surface that gets resized to be 3x4 (taller than wider). On some window systems, the portion of the window/surface that was previously and still is visible (the 3x3 part) will contain the same contents as before, while the remaining parts of the window will have undefined contents. Other window systems may squash/stretch the image to fill the new window size without any undefined contents, or apply some other mapping.

To acquire an available presentable image to use, and retrieve the index of that image, call:

```c
// Provided by VK_VERSION_1_1 with VK_KHR_swapchain
VkResult vkAcquireNextImage2KHR(
    VkDevice device,
    const VkAcquireNextImageInfoKHR* pAcquireInfo,
);```
uint32_t* pImageIndex);

- `device` is the device associated with `swapchain`.
- `pAcquireInfo` is a pointer to a `VkAcquireNextImageInfoKHR` structure containing parameters of the acquire.
- `pImageIndex` is a pointer to a `uint32_t` that is set to the index of the next image to use.

If `VkPhysicalDeviceVulkanSC10Properties::deviceNoDynamicHostAllocations` is `VK_TRUE`, `vkAcquireNextImage2KHR` must not return `VK_ERROR_OUT_OF_HOST_MEMORY`.

### Valid Usage

- VUID-vkAcquireNextImage2KHR-swapchain-01803
  If the number of currently acquired images is greater than the difference between the number of images in the `swapchain` member of `pAcquireInfo` and the value of `VkSurfaceCapabilitiesKHR::minImageCount` as returned by a call to `vkGetPhysicalDeviceSurfaceCapabilities2KHR` with the `surface` used to create `swapchain`, the `timeout` member of `pAcquireInfo` must not be `UINT64_MAX`

### Valid Usage (Implicit)

- VUID-vkAcquireNextImage2KHR-device-parameter
  `device` must be a valid `VkDevice` handle
- VUID-vkAcquireNextImage2KHR-pAcquireInfo-parameter
  `pAcquireInfo` must be a valid pointer to a valid `VkAcquireNextImageInfoKHR` structure
- VUID-vkAcquireNextImage2KHR-pImageIndex-parameter
  `pImageIndex` must be a valid pointer to a `uint32_t` value

### Return Codes

**Success**

- `VK_SUCCESS`
- `VK_TIMEOUT`
- `VK_NOT_READY`
- `VK_SUBOPTIMAL_KHR`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`
- `VK_ERROR_OUT_OF_DATE_KHR`
The `VkAcquireNextImageInfoKHR` structure is defined as:

```c
// Provided by VK_VERSION_1_1 with VK_KHR_swapchain
typedef struct VkAcquireNextImageInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkSwapchainKHR swapchain;
    uint64_t timeout;
    VkSemaphore semaphore;
    VkFence fence;
    uint32_t deviceMask;
} VkAcquireNextImageInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **swapchain** is a non-retired swapchain from which an image is acquired.
- **timeout** specifies how long the function waits, in nanoseconds, if no image is available.
- **semaphore** is `VK_NULL_HANDLE` or a semaphore to signal.
- **fence** is `VK_NULL_HANDLE` or a fence to signal.
- **deviceMask** is a mask of physical devices for which the swapchain image will be ready to use when the semaphore or fence is signaled.

If `vkAcquireNextImageKHR` is used, the device mask is considered to include all physical devices in the logical device.

**Note**

`vkAcquireNextImage2KHR` signals at most one semaphore, even if the application requests waiting for multiple physical devices to be ready via the `deviceMask`. However, only a single physical device can wait on that semaphore, since the semaphore becomes unsignaled when the wait succeeds. For other physical devices to wait for the image to be ready, it is necessary for the application to submit semaphore signal operation(s) to that first physical device to signal additional semaphore(s) after the wait succeeds, which the other physical device(s) can wait upon.

**Valid Usage**

- VUID-VkAcquireNextImageInfoKHR-swapchain-01675
  `swapchain` must not be in the retired state
- VUID-VkAcquireNextImageInfoKHR-semaphore-01288
  If `semaphore` is not `VK_NULL_HANDLE` it must be unsignaled
• **VUID-VkAcquireNextImageInfoKHR-semaphore-01781**
  If `semaphore` is not `VK_NULL_HANDLE` it **must** not have any uncompleted signal or wait operations pending

• **VUID-VkAcquireNextImageInfoKHR-fence-01289**
  If `fence` is not `VK_NULL_HANDLE` it **must** be unsignaled and **must** not be associated with any other queue command that has not yet completed execution on that queue

• **VUID-VkAcquireNextImageInfoKHR-semaphore-01782**
  `semaphore` and `fence` **must** not both be equal to `VK_NULL_HANDLE`

• **VUID-VkAcquireNextImageInfoKHR-deviceMask-01290**
  `deviceMask` **must** be a valid device mask

• **VUID-VkAcquireNextImageInfoKHR-deviceMask-01291**
  `deviceMask` **must** not be zero

• **VUID-VkAcquireNextImageInfoKHR-semaphore-03266**
  `semaphore` **must** have a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_BINARY`

---

**Valid Usage (Implicit)**

• **VUID-VkAcquireNextImageInfoKHR-sType-sType**
  `sType` **must** be `VK_STRUCTURE_TYPE_ACQUIRE_NEXT_IMAGE_INFO_KHR`

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

• **VUID-VkAcquireNextImageInfoKHR-swapchain-parameter**
  `swapchain` **must** be a valid `VkSwapchainKHR` handle

• **VUID-VkAcquireNextImageInfoKHR-semaphore-parameter**
  If `semaphore` is not `VK_NULL_HANDLE`, `semaphore` **must** be a valid `VkSemaphore` handle

• **VUID-VkAcquireNextImageInfoKHR-fence-parameter**
  If `fence` is not `VK_NULL_HANDLE`, `fence` **must** be a valid `VkFence` handle

• **VUID-VkAcquireNextImageInfoKHR-commonparent**
  Each of `fence`, `semaphore`, and `swapchain` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkInstance`

---

**Host Synchronization**

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

• Host access to `semaphore` **must** be externally synchronized

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

---

After queueing all rendering commands and transitioning the image to the correct layout, to queue an image for presentation, call:
// Provided by VK_KHR_swapchain

```c
VkResult vkQueuePresentKHR(
    VkQueue queue,
    const VkPresentInfoKHR* pPresentInfo);
```

- **queue** is a queue that is capable of presentation to the target surface’s platform on the same device as the image’s swapchain.
- **pPresentInfo** is a pointer to a `VkPresentInfoKHR` structure specifying parameters of the presentation.

**Note**

There is no requirement for an application to present images in the same order that they were acquired - applications can arbitrarily present any image that is currently acquired.

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

### Valid Usage

- **VUID-vkQueuePresentKHR-pSwapchains-01292**
  Each element of `pSwapchains` member of `pPresentInfo` must be a swapchain that is created for a surface for which presentation is supported from `queue` as determined using a call to `vkGetPhysicalDeviceSurfaceSupportKHR`.

- **VUID-vkQueuePresentKHR-pSwapchains-01293**
  If more than one member of `pSwapchains` was created from a display surface, all display surfaces referenced that refer to the same display must use the same display mode.

- **VUID-vkQueuePresentKHR-pWaitSemaphores-01294**
  When a semaphore wait operation referring to a binary semaphore defined by the elements of the `pWaitSemaphores` member of `pPresentInfo` executes on `queue`, there must be no other queues waiting on the same semaphore.

- **VUID-vkQueuePresentKHR-pWaitSemaphores-01295**
  All elements of the `pWaitSemaphores` member of `pPresentInfo` must be semaphores that are signaled, or have semaphore signal operations previously submitted for execution.

- **VUID-vkQueuePresentKHR-pWaitSemaphores-03267**
  All elements of the `pWaitSemaphores` member of `pPresentInfo` must be created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_BINARY`.

- **VUID-vkQueuePresentKHR-pWaitSemaphores-03268**
  All elements of the `pWaitSemaphores` member of `pPresentInfo` must reference a semaphore signal operation that has been submitted for execution and any semaphore signal operations on which it depends (if any) must have also been submitted for execution.

Any writes to memory backing the images referenced by the `pImageIndices` and `pSwapchains`
members of \( \text{pPresentInfo} \), that are available before \text{vkQueuePresentKHR} \ is executed, are automatically made visible to the read access performed by the presentation engine. This automatic visibility operation for an image happens-after the semaphore signal operation, and happens-before the presentation engine accesses the image.

Queueing an image for presentation defines a set of \text{queue operations}, including waiting on the semaphores and submitting a presentation request to the presentation engine. However, the scope of this set of queue operations does not include the actual processing of the image by the presentation engine.

![Note]

The origin of the native orientation of the surface coordinate system is not specified in the Vulkan specification; it depends on the platform. For most platforms the origin is by default upper-left, meaning the pixel of the presented \text{VkImage} \ at coordinates \((0,0)\) would appear at the upper left pixel of the platform surface (assuming \text{VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR}, and the display standing the right way up).

If \text{vkQueuePresentKHR} \ fails to enqueue the corresponding set of queue operations, it \textbf{may} return \text{VK_ERROR_OUT_OF_HOST_MEMORY} or \text{VK_ERROR_OUT_OF_DEVICE_MEMORY}. If it does, the implementation \textbf{must} ensure that the state and contents of any resources or synchronization primitives referenced is unaffected by the call or its failure.

If \text{vkQueuePresentKHR} \ fails in such a way that the implementation is unable to make that guarantee, the implementation \textbf{must} return \text{VK_ERROR_DEVICE_LOST}.

However, if the presentation request is rejected by the presentation engine with an error \text{VK_ERROR_OUT_OF_DATE_KHR}, or \text{VK_ERROR_SURFACE_LOST_KHR}, the set of queue operations are still considered to be enqueued and thus any semaphore wait operation specified in \text{VkPresentInfoKHR} will execute when the corresponding queue operation is complete.

Calls to \text{vkQueuePresentKHR} \textbf{may} block, but \textbf{must} return in finite time.

### Valid Usage (Implicit)

- VUID-vkQueuePresentKHR-queue-parameter: \text{queue} \textbf{must} be a valid \text{VkQueue} handle
- VUID-vkQueuePresentKHR-pPresentInfo-parameter: \text{pPresentInfo} \textbf{must} be a valid pointer to a valid \text{VkPresentInfoKHR} structure

### Host Synchronization

- Host access to \text{queue} \textbf{must} be externally synchronized
- Host access to \text{pPresentInfo->pWaitSemaphores[]} \textbf{must} be externally synchronized
- Host access to \text{pPresentInfo->pSwapchains[]} \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>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>Any</td>
</tr>
</tbody>
</table>

### Return Codes

**Success**
- `VK_SUCCESS`
- `VK_SUBOPTIMAL_KHR`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`
- `VK_ERROR_OUT_OF_DATE_KHR`
- `VK_ERROR_SURFACE_LOST_KHR`

The `VkPresentInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_swapchain
typedef struct VkPresentInfoKHR {
    VkStructureType sType;
    const void* pNext;
    uint32_t waitSemaphoreCount;
    const VkSemaphore* pWaitSemaphores;
    uint32_t swapchainCount;
    const VkSwapchainKHR* pSwapchains;
    const uint32_t* pImageIndices;
    VkResult* pResults;
} VkPresentInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `waitSemaphoreCount` is the number of semaphores to wait for before issuing the present request. The number may be zero.
- `pWaitSemaphores` is `NULL` or a pointer to an array of `VkSemaphore` objects with `waitSemaphoreCount` entries, and specifies the semaphores to wait for before issuing the present request.
- `swapchainCount` is the number of swapchains being presented to by this command.
- `pSwapchains` is a pointer to an array of `VkSwapchainKHR` objects with `swapchainCount` entries. A given swapchain must not appear in this list more than once.
• **pImageIndices** is a pointer to an array of indices into the array of each swapchain’s presentable images, with `swapchainCount` entries. Each entry in this array identifies the image to present on the corresponding entry in the `pSwapchains` array.

• **pResults** is a pointer to an array of `VkResult` typed elements with `swapchainCount` entries. Applications that do not need per-swapchain results can use `NULL` for `pResults`. If non-`NULL`, each entry in `pResults` will be set to the `VkResult` for presenting the swapchain corresponding to the same index in `pSwapchains`.

Before an application can present an image, the image's layout must be transitioned to the `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` layout, or for a shared presentable image the `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR` layout.

**Note**
When transitioning the image to `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR` or `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`, there is no need to delay subsequent processing, or perform any visibility operations (as `vkQueuePresentKHR` performs automatic visibility operations). To achieve this, the `dstAccessMask` member of the `VkImageMemoryBarrier` should be set to 0, and the `dstStageMask` parameter should be set to `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`.

### Valid Usage

- **VUID-VkPresentInfoKHR-pImageIndices-01430**
  Each element of `pImageIndices` must be the index of a presentable image acquired from the swapchain specified by the corresponding element of the `pSwapchains` array, and the presented image subresource must be in the `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` or `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR` layout at the time the operation is executed on a `VkDevice`.

### Valid Usage (Implicit)

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

- **VUID-VkPresentInfoKHR-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 `VkDeviceGroupPresentInfoKHR`, `VkDisplayPresentInfoKHR`, or `VkPresentRegionsKHR`.

- **VUID-VkPresentInfoKHR-pWaitSemaphores-parameter**
  If `waitSemaphoreCount` is not 0, `pWaitSemaphores` must be a valid pointer to an array of `VkSemaphore` handles.

- **VUID-VkPresentInfoKHR-pSwapchains-parameter**
  `pSwapchains` must be a valid pointer to an array of `swapchainCount` valid `VkSwapchainKHR`.
When the `VK_KHR_incremental_present` extension is enabled, additional fields can be specified that allow an application to specify that only certain rectangular regions of the presentable images of a swapchain are changed. This is an optimization hint that a presentation engine may use to only update the region of a surface that is actually changing. The application still must ensure that all pixels of a presented image contain the desired values, in case the presentation engine ignores this hint. An application can provide this hint by adding a `VkPresentRegionsKHR` structure to the `pNext` chain of the `VkPresentInfoKHR` structure.

The `VkPresentRegionsKHR` structure is defined as:

```c
// Provided by VK_KHR_incremental_present
typedef struct VkKHR_incremental_present

typedef struct VkPresentRegionsKHR {
    VkStructureType sType;
    const void* pNext;
    uint32_t swapchainCount;
    const VkPresentRegionKHR* pRegions;
} VkPresentRegionsKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `swapchainCount` is the number of swapchains being presented to by this command.
- `pRegions` is `NULL` or a pointer to an array of `VkPresentRegionKHR` elements with `swapchainCount` entries. If not `NULL`, each element of `pRegions` contains the region that has changed since the last present to the swapchain in the corresponding entry in the `VkPresentInfoKHR::pSwapchains` array.

---

**Valid Usage**

- VUID-VkPresentRegionsKHR-swapchainCount-01260 `swapchainCount` must be the same value as `VkPresentInfoKHR::swapchainCount`, where `VkPresentInfoKHR` is included in the `pNext` chain of this `VkPresentRegionsKHR` structure.
Valid Usage (Implicit)

- VUID-VkPresentRegionsKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_PRESENT_REGIONS_KHR

- VUID-VkPresentRegionsKHR-pRegions-parameter
  If pRegions is not NULL, pRegions must be a valid pointer to an array of swapchainCount valid VkPresentRegionKHR structures

- VUID-VkPresentRegionsKHR-swapchainCount-arraylength
  swapchainCount must be greater than 0

For a given image and swapchain, the region to present is specified by the VkPresentRegionKHR structure, which is defined as:

```c
// Provided by VK_KHR_incremental_present
typedef struct VkPresentRegionKHR {
    uint32_t rectangleCount;
    const VkRectLayerKHR* pRectangles;
} VkPresentRegionKHR;
```

- rectangleCount is the number of rectangles in pRectangles, or zero if the entire image has changed and should be presented.

- pRectangles is either NULL or a pointer to an array of VkRectLayerKHR structures. The VkRectLayerKHR structure is the framebuffer coordinates, plus layer, of a portion of a presentable image that has changed and must be presented. If non-NULL, each entry in pRectangles is a rectangle of the given image that has changed since the last image was presented to the given swapchain. The rectangles must be specified relative to VkSurfaceCapabilitiesKHR::currentTransform, regardless of the swapchain’s preTransform. The presentation engine will apply the preTransform transformation to the rectangles, along with any further transformation it applies to the image content.

Valid Usage (Implicit)

- VUID-VkPresentRegionKHR-pRectangles-parameter
  If rectangleCount is not 0, and pRectangles is not NULL, pRectangles must be a valid pointer to an array of rectangleCount valid VkRectLayerKHR structures

The VkRectLayerKHR structure is defined as:

```c
// Provided by VK_KHR_incremental_present
typedef struct VkRectLayerKHR {
    VkOffset2D offset;
    VkExtent2D extent;
    uint32_t layer;
} VkRectLayerKHR;
```
• **offset** is the origin of the rectangle, in pixels.
• **extent** is the size of the rectangle, in pixels.
• **layer** is the layer of the image. For images with only one layer, the value of **layer** must be 0.

Some platforms allow the size of a surface to change, and then scale the pixels of the image to fit the surface. **VkRectLayerKHR** specifies pixels of the swapchain's image(s), which will be constant for the life of the swapchain.

**Valid Usage**

- **VUID-VkRectLayerKHR-offset-04864**
  The sum of **offset** and **extent**, after being transformed according to the **preTransform** member of the **VkSwapchainCreateInfoKHR** structure, must be no greater than the **imageExtent** member of the **VkSwapchainCreateInfoKHR** structure passed to **vkCreateSwapchainKHR**

- **VUID-VkRectLayerKHR-layer-01262**
  **layer** must be less than the **imageArrayLayers** member of the **VkSwapchainCreateInfoKHR** structure passed to **vkCreateSwapchainKHR**

When the **VK_KHR_display_swapchain** extension is enabled additional fields can be specified when presenting an image to a swapchain by setting **VkPresentInfoKHR::pNext** to point to a **VkDisplayPresentInfoKHR** structure.

The **VkDisplayPresentInfoKHR** structure is defined as:

```c
// Provided by VK_KHR_display_swapchain
typedef struct VkDisplayPresentInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkRect2D srcRect;
    VkRect2D dstRect;
    VkBool32 persistent;
} VkDisplayPresentInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **srcRect** is a rectangular region of pixels to present. It must be a subset of the image being presented. If **VkDisplayPresentInfoKHR** is not specified, this region will be assumed to be the entire presentable image.
- **dstRect** is a rectangular region within the visible region of the swapchain's display mode. If **VkDisplayPresentInfoKHR** is not specified, this region will be assumed to be the entire visible region of the swapchain's mode. If the specified rectangle is a subset of the display mode's visible region, the pixels of that region are scaled to fit the visible region.
visible region, content from display planes below the swapchain's plane will be visible outside
the rectangle. If there are no planes below the swapchain's, the area outside the specified
rectangle will be black. If portions of the specified rectangle are outside of the display's visible
region, pixels mapping only to those portions of the rectangle will be discarded.

• persistent: If this is VK_TRUE, the display engine will enable buffered mode on displays that
support it. This allows the display engine to stop sending content to the display until a new
image is presented. The display will instead maintain a copy of the last presented image. This
allows less power to be used, but may increase presentation latency. If VkDisplayPresentInfoKHR
is not specified, persistent mode will not be used.

If the extent of the srcRect and dstRect are not equal, the presented pixels will be scaled
accordingly.

Valid Usage

• VUID-VkDisplayPresentInfoKHR-srcRect-01257
  srcRect must specify a rectangular region that is a subset of the image being presented

• VUID-VkDisplayPresentInfoKHR-dstRect-01258
  dstRect must specify a rectangular region that is a subset of the visibleRegion parameter
  of the display mode the swapchain being presented uses

• VUID-VkDisplayPresentInfoKHR-persistentContent-01259
  If the persistentContent member of the VkDisplayPropertiesKHR structure returned by
  vkGetPhysicalDeviceDisplayPropertiesKHR for the display the present operation targets is
  VK_FALSE, then persistent must be VK_FALSE

Valid Usage (Implicit)

• VUID-VkDisplayPresentInfoKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_DISPLAY_PRESENT_INFO_KHR

If the pNext chain of VkPresentInfoKHR includes a VkDeviceGroupPresentInfoKHR structure, then that
structure includes an array of device masks and a device group present mode.

The VkDeviceGroupPresentInfoKHR structure is defined as:

```c
// Provided by VK_VERSION_1_1 with VK_KHR_swapchain
typedef struct VkDeviceGroupPresentInfoKHR {
    VkStructureType sType;
    const void* pNext;
    uint32_t swapchainCount;
    const uint32_t* pDeviceMasks;
    VkDeviceGroupPresentModeFlagBitsKHR mode;
} VkDeviceGroupPresentInfoKHR;
```

• sType is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `swapchainCount` is zero or the number of elements in `pDeviceMasks`.
- `pDeviceMasks` is a pointer to an array of device masks, one for each element of `VkPresentInfoKHR::pSwapchains`.
- `mode` is a `VkDeviceGroupPresentModeFlagBitsKHR` value specifying the device group present mode that will be used for this present.

If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR`, then each element of `pDeviceMasks` selects which instance of the swapchain image is presented. Each element of `pDeviceMasks` must have exactly one bit set, and the corresponding physical device must have a presentation engine as reported by `VkDeviceGroupPresentCapabilitiesKHR`.

If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_REMOTE_BIT_KHR`, then each element of `pDeviceMasks` selects which instance of the swapchain image is presented. Each element of `pDeviceMasks` must have exactly one bit set, and some physical device in the logical device must include that bit in its `VkDeviceGroupPresentCapabilitiesKHR::presentMask`.

If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_SUM_BIT_KHR`, then each element of `pDeviceMasks` selects which instances of the swapchain image are component-wise summed and the sum of those images is presented. If the sum in any component is outside the representable range, the value of that component is undefined. Each element of `pDeviceMasks` must have a value for which all set bits are set in one of the elements of `VkDeviceGroupPresentCapabilitiesKHR::presentMask`.

If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_MULTI_DEVICE_BIT_KHR`, then each element of `pDeviceMasks` selects which instance(s) of the swapchain images are presented. For each bit set in each element of `pDeviceMasks`, the corresponding physical device must have a presentation engine as reported by `VkDeviceGroupPresentCapabilitiesKHR`.

If `VkDeviceGroupPresentInfoKHR` is not provided or `swapchainCount` is zero then the masks are considered to be 1. If `VkDeviceGroupPresentInfoKHR` is not provided, `mode` is considered to be `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR`.

### Valid Usage

- VUID-VkDeviceGroupPresentInfoKHR-swapchainCount-01297
  - `swapchainCount` must equal 0 or `VkPresentInfoKHR::swapchainCount`

- VUID-VkDeviceGroupPresentInfoKHR-mode-01298
  - If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR`, then each element of `pDeviceMasks` must have exactly one bit set, and the corresponding element of `VkDeviceGroupPresentCapabilitiesKHR::presentMask` must be non-zero

- VUID-VkDeviceGroupPresentInfoKHR-mode-01299
  - If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_REMOTE_BIT_KHR`, then each element of `pDeviceMasks` must have exactly one bit set, and some physical device in the logical device must include that bit in its `VkDeviceGroupPresentCapabilitiesKHR::presentMask`.

- VUID-VkDeviceGroupPresentInfoKHR-mode-01300
  - If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_SUM_BIT_KHR`, then each element of `pDeviceMasks`
must have a value for which all set bits are set in one of the elements of VkDeviceGroupPresentCapabilitiesKHR::presentMask

- VUID-VkDeviceGroupPresentInfoKHR-mode-01301
  If mode is VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_MULTI_DEVICE_BIT_KHR, then for each bit set in each element of pDeviceMasks, the corresponding element of VkDeviceGroupPresentCapabilitiesKHR::presentMask must be non-zero

- VUID-VkDeviceGroupPresentInfoKHR-pDeviceMasks-01302
  The value of each element of pDeviceMasks must be equal to the device mask passed in VkAcquireNextImageInfoKHR::deviceMask when the image index was last acquired

- VUID-VkDeviceGroupPresentInfoKHR-mode-01303
  mode must have exactly one bit set, and that bit must have been included in VkDeviceGroupSwapchainCreateInfoKHR::modes

Valid Usage (Implicit)

- VUID-VkDeviceGroupPresentInfoKHR-sType-sType
  sType must be VK_STRUCTURE_TYPEDEVICEGROUP_PRESENT_INFO_KHR

- VUID-VkDeviceGroupPresentInfoKHR-pDeviceMasks-parameter
  If swapchainCount is not 0, pDeviceMasks must be a valid pointer to an array of swapchainCount uint32_t values

- VUID-VkDeviceGroupPresentInfoKHR-mode-parameter
  mode must be a valid VkDeviceGroupPresentModeFlagBitsKHR value

\texttt{vkQueuePresentKHR}, releases the acquisition of the images referenced by imageIndices. The queue family corresponding to the queue \texttt{vkQueuePresentKHR} is executed on must have ownership of the presented images as defined in Resource Sharing. \texttt{vkQueuePresentKHR} does not alter the queue family ownership, but the presented images must not be used again before they have been reacquired using \texttt{vkAcquireNextImageKHR}.

The processing of the presentation happens in issue order with other queue operations, but semaphores have to be used to ensure that prior rendering and other commands in the specified queue complete before the presentation begins. The presentation command itself does not delay processing of subsequent commands on the queue, however, presentation requests sent to a particular queue are always performed in order. Exact presentation timing is controlled by the semantics of the presentation engine and native platform in use.

If an image is presented to a swapchain created from a display surface, the mode of the associated display will be updated, if necessary, to match the mode specified when creating the display surface. The mode switch and presentation of the specified image will be performed as one atomic operation.

The result codes VK_ERROR_OUT_OF_DATE_KHR and VK_SUBOPTIMAL_KHR have the same meaning when returned by \texttt{vkQueuePresentKHR} as they do when returned by \texttt{vkAcquireNextImageKHR}. If multiple swapchains are presented, the result code is determined applying the following rules in order:
• If the device is lost, VK_ERROR_DEVICE_LOST is returned.

• If any of the target surfaces are no longer available the error VK_ERROR_SURFACE_LOST_KHR is returned.

• If any of the presents would have a result of VK_ERROR_OUT_OF_DATE_KHR if issued separately then VK_ERROR_OUT_OF_DATE_KHR is returned.

• If any of the presents would have a result of VK_SUBOPTIMAL_KHR if issued separately then VK_SUBOPTIMAL_KHR is returned.

• Otherwise VK_SUCCESS is returned.

Presentation is a read-only operation that will not affect the content of the presentable images. Upon reacquiring the image and transitioning it away from the VK_IMAGE_LAYOUT_PRESENT_SRC_KHR layout, the contents will be the same as they were prior to transitioning the image to the present source layout and presenting it. However, if a mechanism other than Vulkan is used to modify the platform window associated with the swapchain, the content of all presentable images in the swapchain becomes undefined.

Note

The application can continue to present any acquired images from a retired swapchain as long as the swapchain has not entered a state that causes vkQueuePresentKHR to return VK_ERROR_OUT_OF_DATE_KHR.

### 30.8. Hdr Metadata

To improve color reproduction of content it is useful to have information that can be used to better reproduce the colors as seen on the reference monitor. That information can be provided to an implementation by calling vkSetHdrMetadataEXT. The metadata will be applied to the specified VkSwapchainKHR objects at the next vkQueuePresentKHR call using that VkSwapchainKHR object. The metadata will persist until a subsequent vkSetHdrMetadataEXT changes it. The definitions below are from the associated SMPTE 2086, CTA 861.3 and CIE 15:2004 specifications.

The definition of vkSetHdrMetadataEXT is:

```c
// Provided by VK_EXT_hdr_metadata
void vkSetHdrMetadataEXT(
    VkDevice device, 
    uint32_t swapchainCount, 
    const VkSwapchainKHR* pSwapchains, 
    const VkHdrMetadataEXT* pMetadata);
```

- **device** is the logical device where the swapchain(s) were created.
- **swapchainCount** is the number of swapchains included in pSwapchains.
- **pSwapchains** is a pointer to an array of swapchainCount VkSwapchainKHR handles.
- **pMetadata** is a pointer to an array of swapchainCount VkHdrMetadataEXT structures.
Valid Usage (Implicit)

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

- **VUID-vkSetHdrMetadataEXT-pSwapchains-parameter**
  
  `pSwapchains` **must** be a valid pointer to an array of `swapchainCount` valid `VkSwapchainKHR` handles

- **VUID-vkSetHdrMetadataEXT-pMetadata-parameter**
  
  `pMetadata` **must** be a valid pointer to an array of `swapchainCount` valid `VkHdrMetadataEXT` structures

- **VUID-vkSetHdrMetadataEXT-swapchainCount-arraylength**
  
  `swapchainCount` **must** be greater than 0

- **VUID-vkSetHdrMetadataEXT-commonparent**
  
  Both of `device`, and the elements of `pSwapchains` **must** have been created, allocated, or retrieved from the same `VkInstance`

---

```c
// Provided by VK_EXT_hdr_metadata
typedef struct VkXYColorEXT {
    float  x;
    float  y;
} VkXYColorEXT;
```

Chromaticity coordinates x and y are as specified in CIE 15:2004 “Calculation of chromaticity coordinates” (Section 7.3) and are limited to between 0 and 1 for real colors for the reference monitor.

```c
// Provided by VK_EXT_hdr_metadata
typedef struct VkHdrMetadataEXT {
    VkStructureType   sType;
    const void*       pNext;
    VkXYColorEXT      displayPrimaryRed;
    VkXYColorEXT      displayPrimaryGreen;
    VkXYColorEXT      displayPrimaryBlue;
    VkXYColorEXT      whitePoint;
    float             maxLuminance;
    float             minLuminance;
    float             maxContentLightLevel;
    float             maxFrameAverageLightLevel;
} VkHdrMetadataEXT;
```

- `sType` is the type of this structure.
- `pNext` is **NULL** or a pointer to a structure extending this structure.
- `displayPrimaryRed` is the reference monitor’s red primary in chromaticity coordinates.
• displayPrimaryGreen is the reference monitor's green primary in chromaticity coordinates
• displayPrimaryBlue is the reference monitor's blue primary in chromaticity coordinates
• whitePoint is the reference monitor’s white-point in chromaticity coordinates
• maxLuminance is the maximum luminance of the reference monitor in nits
• minLuminance is the minimum luminance of the reference monitor in nits
• maxContentLightLevel is content's maximum luminance in nits
• maxFrameAverageLightLevel is the maximum frame average light level in nits

Valid Usage (Implicit)

• VUID-VkHdrMetadataEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_HDR_METADATA_EXT
• VUID-VkHdrMetadataEXT-pNext-pNext
  pNext must be NULL

Note
The validity and use of this data is outside the scope of Vulkan.
Chapter 31. Extending Vulkan

New functionality may be added to Vulkan via either new extensions or new versions of the core, or new versions of an extension in some cases.

This chapter describes how Vulkan is versioned, how compatibility is affected between different versions, and compatibility rules that are followed by the Vulkan Working Group.

31.1. Instance and Device Functionality

Commands that enumerate instance properties, or that accept a `VkInstance` object as a parameter, are considered instance-level functionality. Commands that enumerate physical device properties, or that accept a `VkDevice` object or any of a device's child objects as a parameter, are considered device-level functionality.

Note

Applications usually interface to Vulkan using a loader that implements only instance-level functionality, passing device-level functionality to implementations of the full Vulkan API on the system. In some circumstances, as these may be implemented independently, it is possible that the loader and device implementations on a given installation will support different versions. To allow for this and call out when it happens, the Vulkan specification enumerates device and instance level functionality separately - they have independent version queries.

Note

Vulkan 1.0 initially specified new physical device enumeration functionality as instance-level, requiring it to be included in an instance extension. As the capabilities of device-level functionality require discovery via physical device enumeration, this led to the situation where many device extensions required an instance extension as well. To alleviate this extra work, `VK_KHR_get_physical_device_properties2` (and subsequently Vulkan 1.1) redefined device-level functionality to include physical device enumeration.

31.2. Core Versions

The Vulkan Specification is regularly updated with bug fixes and clarifications. Occasionally new functionality is added to the core and at some point it is expected that there will be a desire to perform a large, breaking change to the API. In order to indicate to developers how and when these changes are made to the specification, and to provide a way to identify each set of changes, the Vulkan API maintains a version number.

31.2.1. Version Numbers

The Vulkan version number comprises four parts indicating the variant, major, minor and patch version of the Vulkan API Specification.
The **variant** indicates the variant of the Vulkan API supported by the implementation. This is always 1 for the Vulkan SC API. The Base Vulkan API is variant 0.

![Note]

A non-zero variant indicates the API is a variant of the Vulkan API and applications will typically need to be modified to run against it. The variant field was a later addition to the version number, added in version 1.2.175 of the Base Vulkan Specification.

The **major version** indicates a significant change in the API, which will encompass a wholly new version of the specification.

The **minor version** indicates the incorporation of new functionality into the core specification.

The **patch version** indicates bug fixes, clarifications, and language improvements have been incorporated into the specification.

Compatibility guarantees made about versions of the API sharing any of the same version numbers are documented in [Core Versions](#).

The version number is used in several places in the API. In each such use, the version numbers are packed into a 32-bit integer as follows:

- The 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:
VK_MAKE_API_VERSION constructs an API version number.

• `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.

**VK_API_VERSION_1_1** returns the API version number for Vulkan 1.1.0.

**VK_API_VERSION_1_2** returns the API version number for Vulkan 1.2.0.

**VKSC_API_VARIANT** returns the API variant number for Vulkan SC.
VKSC_API_VERSION_1_0 returns the API version number for Vulkan SC 1.0.0.

```c
// Provided by VKSC_VERSION_1_0
// Vulkan SC 1.0 version number
#define VKSC_API_VERSION_1_0 VK_MAKE_API_VERSION(VKSC_API_VARIANT, 1, 0, 0)// Patch version should always be set to 0
```

### 31.2.2. Querying Version Support

The version of instance-level functionality can be queried by calling `vkEnumerateInstanceVersion`.

The version of device-level functionality can be queried by calling `vkGetPhysicalDeviceProperties` or `vkGetPhysicalDeviceProperties2`, and is returned in `VkPhysicalDeviceProperties::apiVersion`, encoded as described in Version Numbers.

### 31.3. Layers

When a layer is enabled, it inserts itself into the call chain for Vulkan commands the layer is interested in. Layers can be used for a variety of tasks that extend the base behavior of Vulkan beyond what is required by the specification - such as call logging, tracing, validation, or providing additional extensions.

- **Note**
  For example, an implementation is not expected to check that the value of enums used by the application fall within allowed ranges. Instead, a validation layer would do those checks and flag issues. This avoids a performance penalty during production use of the application because those layers would not be enabled in production.

- **Note**
  Vulkan layers may wrap object handles (i.e. return a different handle value to the application than that generated by the implementation). This is generally discouraged, as it increases the probability of incompatibilities with new extensions. The validation layers wrap handles in order to track the proper use and destruction of each object. See the “Architecture of the Vulkan Loader Interfaces” document for additional information.

To query the available layers, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkEnumerateInstanceLayerProperties(
    uint32_t* pPropertyCount,
    VkLayerProperties* pProperties);
```

- **pPropertyCount** is a pointer to an integer related to the number of layer properties available or
queried, as described below.

- `pProperties` is either `NULL` or a pointer to an array of `VkLayerProperties` structures.

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

The list of available layers may change at any time due to actions outside of the Vulkan implementation, so two calls to `vkEnumerateInstanceLayerProperties` with the same parameters may return different results, or retrieve different `pPropertyCount` values or `pProperties` contents. Once an instance has been created, the layers enabled for that instance will continue to be enabled and valid for the lifetime of that instance, even if some of them become unavailable for future instances.

Valid Usage (Implicit)

- `VUID-vkEnumerateInstanceLayerProperties-pPropertyCount-parameter` `pPropertyCount` must be a valid pointer to a `uint32_t` value

- `VUID-vkEnumerateInstanceLayerProperties-pProperties-parameter` If the value referenced by `pPropertyCount` is not 0, and `pProperties` is not `NULL`, `pProperties` must be a valid pointer to an array of `pPropertyCount` `VkLayerProperties` structures

Return Codes

Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkLayerProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkLayerProperties {
    char     layerName[VK_MAX_EXTENSION_NAME_SIZE];
    uint32_t specVersion;
    uint32_t implementationVersion;
    char     description[VK_MAX_DESCRIPTION_SIZE];
} VkLayerProperties;
```
• **layerName** is an array of **VK_MAX_EXTENSION_NAME_SIZE** char containing a null-terminated UTF-8 string which is the name of the layer. Use this name in the **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.

```c
#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.

```c
#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.

### 31.3.1. Device Layer Deprecation

Previous versions of this specification distinguished between instance and device layers. Instance layers were only able to intercept commands that operate on **VkInstance** and **VkPhysicalDevice**, except they were not able to intercept **vkCreateDevice**. Device layers were enabled for individual devices when they were created, and could only intercept commands operating on that device or its child objects.

Device-only layers are now deprecated, and this specification no longer distinguishes between instance and device layers. Layers are enabled during instance creation, and are able to intercept all commands operating on that instance or any of its child objects. At the time of deprecation there
were no known device-only layers and no compelling reason to create one.

To enumerate device layers, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkEnumerateDeviceLayerProperties(
    VkPhysicalDevice physicalDevice,
    uint32_t* pPropertyCount,
    VkLayerProperties* pProperties);
```

- `pPropertyCount` is a pointer to an integer related to the number of layer properties available or queried.
- `pProperties` is either `NULL` or a pointer to an array of `VkLayerProperties` structures.

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.

### 31.4. Extensions

Extensions **may** define new Vulkan commands, structures, and enumerants. For compilation purposes, the interfaces defined by registered extensions, including new structures and enumerants as well as function pointer types for new commands, are defined in the Khronos-supplied `vulkan_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.

31.4.1. Instance Extensions

Instance extensions add new instance-level functionality to the API, outside of the core specification.

To query the available instance extensions, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkEnumerateInstanceExtensionProperties(
    const char* pLayerName,
    uint32_t* pPropertyCount, 
    VkExtensionProperties* pProperties);
```

- pLayerName is either NULL or a pointer to a null-terminated UTF-8 string naming the layer to retrieve extensions from.
- pPropertyCount is a pointer to an integer related to the number of extension properties available or queried, as described below.
• **Properties** 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**
31.4.2. Device Extensions

Device extensions add new device-level functionality to the API, outside of the core specification.

To query the extensions available to a given physical device, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkEnumerateDeviceExtensionProperties(
    VkPhysicalDevice physicalDevice,
    const char* pLayerName,
    uint32_t* pPropertyCount,
    VkExtensionProperties* pProperties);
```

- `physicalDevice` is the physical device that will be queried.
- `pLayerName` is either NULL or a pointer to a null-terminated UTF-8 string naming the layer to retrieve extensions from.
- `pPropertyCount` is a pointer to an integer related to the number of extension properties available or queried, and is treated in the same fashion as the `vkEnumerateInstanceExtensionProperties::pPropertyCount` parameter.
- `pProperties` is either NULL or a pointer to an array of `VkExtensionProperties` structures.

When `pLayerName` parameter is NULL, only extensions provided by the Vulkan implementation or by implicitly enabled layers are returned. When `pLayerName` is the name of a layer, the device extensions provided by that layer are returned.

Implementations must not advertise any pair of extensions that cannot be enabled together due to behavioral differences, or any extension that cannot be enabled against the advertised version.

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

### Valid Usage (Implicit)

- VUID-vkEnumerateDeviceExtensionProperties-physicalDevice-parameter `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkEnumerateDeviceExtensionProperties-pLayerName-parameter If `pLayerName` is not NULL, `pLayerName` must be a null-terminated UTF-8 string
- VUID-vkEnumerateDeviceExtensionProperties-pPropertyCount-parameter `pPropertyCount` must be a valid pointer to a `uint32_t` value
- VUID-vkEnumerateDeviceExtensionProperties-pProperties-parameter If the value referenced by `pPropertyCount` is not 0, and `pProperties` is not NULL, `pProperties` must be a valid pointer to an array of `pPropertyCount` `VkExtensionProperties` structures
Return Codes

Success
• VK_SUCCESS
• VK_INCOMPLETE

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY
• VK_ERROR_LAYER_NOT_PRESENT

The VkExtensionProperties structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkExtensionProperties {
    char extensionName[VK_MAX_EXTENSION_NAME_SIZE];
    uint32_t specVersion;
} VkExtensionProperties;
```

• extensionName is an array of VK_MAX_EXTENSION_NAME_SIZE char containing a null-terminated UTF-8 string which is the name of the extension.

• specVersion is the version of this extension. It is an integer, incremented with backward compatible changes.

31.5. Extension Dependencies

Some extensions are dependent on other extensions, or on specific core API versions, to function. To enable extensions with dependencies, any required extensions must also be enabled through the same API mechanisms when creating an instance with `vkCreateInstance` or a device with `vkCreateDevice`. Each extension which has such dependencies documents them in the appendix summarizing that extension.

If an extension is supported (as queried by `vkEnumerateInstanceExtensionProperties` or `vkEnumerateDeviceExtensionProperties`), then required extensions of that extension must also be supported for the same instance or physical device.

Any device extension that has an instance extension dependency that is not enabled by `vkCreateInstance` is considered to be unsupported, hence it must not be returned by `vkEnumerateDeviceExtensionProperties` for any `VkPhysicalDevice` child of the instance. Instance extensions do not have dependencies on device extensions.

If a required extension has been promoted to another extension or to a core API version, then as a general rule, the dependency is also satisfied by the promoted extension or core version. This will be true so long as any features required by the original extension are also required or enabled by the promoted extension or core version. However, in some cases an extension is promoted while...
making some of its features optional in the promoted extension or core version. In this case, the dependency may not be satisfied. The only way to be certain is to look at the descriptions of the original dependency and the promoted version in the Layers & Extensions and Core Revisions appendices.

**Note**

There is metadata in vk.xml describing some aspects of promotion, especially requires, promotedto and deprecatedby attributes of <extension> tags. However, the metadata does not yet fully describe this scenario. In the future, we may extend the XML schema to describe the full set of extensions and versions satisfying a dependency.

### 31.6. Compatibility Guarantees (Informative)

This section is marked as informal as there is no binding responsibility on implementations of the Vulkan API - these guarantees are however a contract between the Vulkan Working Group and developers using this Specification.

#### 31.6.1. Core Versions

Each of the major, minor, and patch versions of the Vulkan specification provide different compatibility guarantees.

**Patch Versions**

A difference in the patch version indicates that a set of bug fixes or clarifications have been made to the Specification. Informative enums returned by Vulkan commands that will not affect the runtime behavior of a valid application may be added in a patch version (e.g. VkVendorId).

The specification’s patch version is strictly increasing for a given major version of the specification; any change to a specification as described above will result in the patch version being increased by 1. Patch versions are applied to all minor versions, even if a given minor version is not affected by the provoking change.

Specifications with different patch versions but the same major and minor version are fully compatible with each other - such that a valid application written against one will work with an implementation of another.

**Note**

If a patch version includes a bug fix or clarification that could have a significant impact on developer expectations, these will be highlighted in the change log. Generally the Vulkan Working Group tries to avoid these kinds of changes, instead fixing them in either an extension or core version.

**Minor Versions**

Changes in the minor version of the specification indicate that new functionality has been added to the core specification. This will usually include new interfaces in the header, and may also include
behavior changes and bug fixes. Core functionality may be deprecated in a minor version, but will not be obsoleted or removed.

The specification’s minor version is strictly increasing for a given major version of the specification; any change to a specification as described above will result in the minor version being increased by 1. Changes that can be accommodated in a patch version will not increase the minor version.

Specifications with a lower minor version are backwards compatible with an implementation of a specification with a higher minor version for core functionality and extensions issued with the KHR vendor tag. Vendor and multi-vendor extensions are not guaranteed to remain functional across minor versions, though in general they are with few exceptions - see Obsoletion for more information.

**Major Versions**

A difference in the major version of specifications indicates a large set of changes which will likely include interface changes, behavioral changes, removal of deprecated functionality, and the modification, addition, or replacement of other functionality.

The specification’s major version is monotonically increasing; any change to the specification as described above will result in the major version being increased. Changes that can be accommodated in a patch or minor version will not increase the major version.

The Vulkan Working Group intends to only issue a new major version of the Specification in order to realise significant improvements to the Vulkan API that will necessarily require breaking compatibility.

A new major version will likely include a wholly new version of the specification to be issued - which could include an overhaul of the versioning semantics for the minor and patch versions. The patch and minor versions of a specification are therefore not meaningful across major versions. If a major version of the specification includes similar versioning semantics, it is expected that the patch and the minor version will be reset to 0 for that major version.

**31.6.2. Extensions**

A KHR extension must be able to be enabled alongside any other KHR extension, and for any minor or patch version of the core Specification beyond the minimum version it requires. A multi-vendor extension should be able to be enabled alongside any KHR extension or other multi-vendor extension, and for any minor or patch version of the core Specification beyond the minimum version it requires. A vendor extension should be able to be enabled alongside any KHR extension, multi-vendor extension, or other vendor extension from the same vendor, and for any minor or patch version of the core Specification beyond the minimum version it requires. A vendor extension may be able to be enabled alongside vendor extensions from another vendor.

The one other exception to this is if a vendor or multi-vendor extension is made obsolete by either a core version or another extension, which will be highlighted in the extension appendix.
Promotion

Extensions, or features of an extension, may be promoted to a new core version of the API, or a newer extension which an equal or greater number of implementors are in favour of.

When extension functionality is promoted, minor changes may be introduced, limited to the following:

- Naming
- Non-intrusive parameters changes
- Feature advertisement/enablement
- Combining structure parameters into larger structures
- Author ID suffixes changed or removed

**Note**

If extension functionality is promoted, there is no guarantee of direct compatibility, however it should require little effort to port code from the original feature to the promoted one.

The Vulkan Working Group endeavours to ensure that larger changes are marked as either deprecated or obsoleted as appropriate, and can do so retroactively if necessary.

Extensions that are promoted are listed as being promoted in their extension appendices, with reference to where they were promoted to.

When an extension is promoted, any backwards compatibility aliases which exist in the extension will not be promoted.

**Note**

As a hypothetical example, if the VK_KHR_surface extension were promoted to part of a future core version, the VK_COLOR_SPACE_SRGB_NONLINEAR_KHR token defined by that extension would be promoted to VK_COLOR_SPACE_SRGB_NONLINEAR. However, the VK_COLORSPACE_SRGB_NONLINEAR_KHR token aliases VK_COLOR_SPACE_SRGB_NONLINEAR_KHR. The VK_COLORSPACE_SRGB_NONLINEAR_KHR would not be promoted, because it is a backwards compatibility alias that exists only due to a naming mistake when the extension was initially published.

Deprecation

Extensions may be marked as deprecated when the intended use cases either become irrelevant or can be solved in other ways. Generally, a new feature will become available to solve the use case in another extension or core version of the API, but it is not guaranteed.

**Note**

Features that are intended to replace deprecated functionality have no guarantees of compatibility, and applications may require drastic modification in order to
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 41. 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>Special Use</td>
<td>XML Tag</td>
<td>Full Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------</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>glemination</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 32. Features

Features describe functionality which is not supported on all implementations. Features are properties of the physical device. Features are optional, and must be explicitly enabled before use. Support for features is reported and enabled on a per-feature basis.

Note
Features are reported via the basic VkPhysicalDeviceFeatures structure, as well as the extensible structure VkPhysicalDeviceFeatures2, which was added in the VK_KHR_get_physical_device_properties2 extension and included in Vulkan 1.1. When new features are added in future Vulkan versions or extensions, each extension should introduce one new feature structure, if needed. This structure can be added to the pNext chain of the VkPhysicalDeviceFeatures2 structure.

For convenience, new core versions of Vulkan may introduce new unified feature structures for features promoted from extensions. At the same time, the extension’s original feature structure (if any) is also promoted to the core API, and is an alias of the extension’s structure. This results in multiple names for the same feature: in the original extension’s feature structure and the promoted structure alias, in the unified feature structure. When a feature was implicitly supported and enabled in the extension, but an explicit name was added during promotion, then the extension itself acts as an alias for the feature as listed in the table below.

All aliases of the same feature in the core API must be reported consistently: either all must be reported as supported, or none of them. When a promoted extension is available, any corresponding feature aliases must be supported.

Table 42. 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:

```
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceFeatures2 {
    VkStructureType sType;
    void* pNext;
    VkPhysicalDeviceFeatures features;
} VkPhysicalDeviceFeatures2;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `features` is a `VkPhysicalDeviceFeatures` structure describing the fine-grained features of the Vulkan 1.0 API.

The `pNext` chain of this structure is used to extend the structure with features defined by extensions. This structure can be used in `vkGetPhysicalDeviceFeatures2` or can be included in the `pNext` chain of a `VkDeviceCreateInfo` structure, in which case it controls which features are enabled in 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:

```
// Provided by VK_VERSION_1_0
typedef struct VkPhysicalDeviceFeatures {
    VkBool32 robustBufferAccess;
    VkBool32 fullDrawIndexUint32;
    VkBool32 imageCubeArray;
    VkBool32 independentBlend;
    VkBool32 geometryShader;
    VkBool32 tessellationShader;
    VkBool32 sampleRateShading;
    VkBool32 dualSrcBlend;
    VkBool32 logicOp;
    VkBool32 multiDrawIndirect;
    VkBool32 drawIndirectFirstInstance;
    VkBool32 depthClamp;
    VkBool32 depthBiasClamp;
    VkBool32 fillModeNonSolid;
    VkBool32 depthBounds;
    VkBool32 wideLines;
    VkBool32 largePoints;
    VkBool32 alphaToOne;
    VkBool32 multiViewport;
} VkPhysicalDeviceFeatures;
```
This structure describes the following features:

- **robustBufferAccess** specifies that accesses to buffers are bounds-checked against the range of the buffer descriptor (as determined by `VkDescriptorBufferInfo::range`, `VkBufferViewCreateInfo::range`, or the size of the buffer). Out of bounds accesses must not cause application termination, and the effects of shader loads, stores, and atomics must conform to an implementation-dependent behavior as described below.

  A buffer access is considered to be out of bounds if any of the following are true:

  - The pointer was formed by `OpImageTexelPointer` and the coordinate is less than zero or greater than or equal to the number of whole elements in the bound range.
  
  - The pointer was not formed by `OpImageTexelPointer` and the object pointed to is not
wholly contained within the bound range. This includes accesses performed via variable pointers where the buffer descriptor being accessed cannot be statically determined. Uninitialized pointers and pointers equal to $\text{OpConstantNull}$ 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 $\text{OpLoad}$ 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 $\text{robustBufferAccess2}$ is not enabled and 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.

  ◦ Accesses to descriptors written with a $\text{VK_NULL_HANDLE}$ resource or view are not considered to be out of bounds. Instead, each type of descriptor access defines a specific behavior for accesses to a null descriptor.

  ◦ Out-of-bounds buffer loads will return any of the following values:

    - If the access is to a uniform buffer and $\text{robustBufferAccess2}$ is enabled, loads of offsets between the end of the descriptor range and the end of the descriptor range rounded up to a multiple of $\text{robustUniformBufferAccessSizeAlignment}$ bytes must return either zero values or the contents of the memory at the offset being loaded. Loads of offsets past the descriptor range rounded up to a multiple of $\text{robustUniformBufferAccessSizeAlignment}$ bytes must return zero values.

    - If the access is to a storage buffer and $\text{robustBufferAccess2}$ is enabled, loads of offsets between the end of the descriptor range and the end of the descriptor range rounded up to a multiple of $\text{robustStorageBufferAccessSizeAlignment}$ bytes must return either zero values or the contents of the memory at the offset being loaded. Loads of offsets past the descriptor range rounded up to a multiple of $\text{robustStorageBufferAccessSizeAlignment}$ bytes must return zero values. Similarly, stores to addresses between the end of the descriptor range and the end of the descriptor range rounded up to a multiple of $\text{robustStorageBufferAccessSizeAlignment}$ bytes may be discarded.

    - Non-atomic accesses to storage buffers that are a multiple of 32 bits may be decomposed into 32-bit accesses that are individually bounds-checked.

    - If the access is to an index buffer and $\text{robustBufferAccess2}$ is enabled, zero values must be returned.

    - If the access is to a uniform texel buffer or storage texel buffer and $\text{robustBufferAccess2}$
is enabled, zero values **must** be returned, and then Conversion to RGBA is applied based on the buffer view’s format.

- 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.
    - If `robustBufferAccess2` is enabled, out of bounds writes **must** not modify any 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.
      - If `robustBufferAccess2` is enabled, out of bounds atomics **must** not modify any memory, and return an undefined value.
  - If `robustBufferAccess2` is disabled, 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 `robustBufferAccess2` is enabled, 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 the byte size of the memory range bound to the vertex buffer binding.
    - If a vertex input attribute is out of bounds, the raw data extracted are zero values, and missing G, B, or A components are filled with (0,0,1).
  - 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`, and `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:
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_R16G16_SINT
- VK_FORMAT_R8G8_SINT
- VK_FORMAT_R16_SINT
- VK_FORMAT_R8_SINT
- VK_FORMAT_A2B10G10R10_UINT_PACK32
- VK_FORMAT_R16G16_UINT
- VK_FORMAT_R8G8_UINT
- VK_FORMAT_R16_UINT
- VK_FORMAT_R8_UINT

Note
shaderStorageImageExtendedFormats feature only adds a guarantee of format support, which is specified for the whole physical device. Therefore enabling or disabling the feature via vkCreateDevice has no practical effect.

To query for additional properties, or if the feature is not supported, vkGetPhysicalDeviceFormatProperties and vkGetPhysicalDeviceImageFormatProperties can be used to check for supported properties of individual formats, as usual rules allow.

VK_FORMAT_R32G32_UINT, VK_FORMAT_R32G32_SINT, and VK_FORMAT_R32G32_SFLOAT from StorageImageExtendedFormats SPIR-V capability, are already covered by core Vulkan mandatory format support.

- shaderStorageImageMultisample specifies whether multisampled storage images are supported. If this feature is not enabled, images that are created with a usage that includes VK_IMAGE_USAGE_STORAGE_BIT must be created with samples equal to VK_SAMPLE_COUNT_1_BIT. This also specifies whether shader modules can declare the StorageImageMultisample and ImageMSArray capabilities.
- shaderStorageImageReadWithoutFormat specifies whether storage images require a format qualifier to be specified when reading.
- shaderStorageImageWriteWithoutFormat specifies whether storage images 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
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceVulkan11Features {
    VkStructureType sType;
    void* pNext;
    VkBool32 storageBuffer16BitAccess;
    VkBool32 uniformAndStorageBuffer16BitAccess;
    VkBool32 storagePushConstant16;
    VkBool32 storageInputOutput16;
    VkBool32 multiview;
    VkBool32 multiviewGeometryShader;
    VkBool32 multiviewTessellationShader;
    VkBool32 variablePointersStorageBuffer;
    VkBool32 variablePointers;
    VkBool32 protectedMemory;
    VkBool32 samplerYcbcrConversion;
    VkBool32 shaderDrawParameters;
} VkPhysicalDeviceVulkan11Features;
```

This structure describes the following features:

- **sType** is the type of this structure.

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

- **storageBuffer16BitAccess** specifies whether objects in the StorageBuffer, or
PhysicalStorageBuffer storage class with the Block decoration can have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members must not be used in such objects. This also specifies whether shader modules can declare the StorageBuffer16BitAccess capability.

- uniformAndStorageBuffer16BitAccess specifies whether objects in the Uniform storage class with the Block decoration can have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members must not be used in such objects. This also specifies whether shader modules can declare the UniformAndStorageBuffer16BitAccess capability.

- storagePushConstant16 specifies whether objects in the PushConstant storage class can have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members must not be used in such objects. This also specifies whether shader modules can declare the StoragePushConstant16 capability.

- storageInputOutput16 specifies whether objects in the Input and Output storage classes can have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members must not be used in such objects. This also specifies whether shader modules can declare the StorageInputOutput16 capability.

- multiview specifies whether the implementation supports multiview rendering within a render pass. If this feature is not enabled, the view mask of each subpass must always be zero.

- multiviewGeometryShader specifies whether the implementation supports multiview rendering within a render pass, with geometry shaders. If this feature is not enabled, then a pipeline compiled against a subpass with a non-zero view mask must not include a geometry shader.

- multiviewTessellationShader specifies whether the implementation supports multiview rendering within a render pass, with tessellation shaders. If this feature is not enabled, then a pipeline compiled against a subpass with a non-zero view mask must not include any tessellation shaders.

- variablePointersStorageBuffer specifies whether the implementation supports the SPIR-V VariablePointersStorageBuffer capability. When this feature is not enabled, shader modules must not declare the SPV_KHR_variable_pointers extension or the VariablePointersStorageBuffer capability.

- variablePointers specifies whether the implementation supports the SPIR-V VariablePointers capability. When this feature is not enabled, shader modules must not declare the VariablePointers capability.

- protectedMemory specifies whether protected memory is supported.

- samplerYcbcrConversion specifies whether the implementation supports sampler Y’C_bC_r conversion. If samplerYcbcrConversion is VK_FALSE, sampler Y’C_bC_r conversion is not supported, and samplers using sampler Y’C_bC_r conversion must not be used.

- shaderDrawParameters specifies whether the implementation supports the SPIR-V DrawParameters capability. When this feature is not enabled, shader modules must not declare the SPV_KHR_shader_draw_parameters extension or the DrawParameters capability.

If the VkPhysicalDeviceVulkan11Features structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceVulkan11Features can
also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

**Valid Usage (Implicit)**

- **VUID-VkPhysicalDeviceVulkan11Features-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_1_FEATURES`

The `VkPhysicalDeviceVulkan12Features` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceVulkan12Features {
    VkStructureType sType;
    void* pNext;
    VkBool32 samplerMirrorClampToEdge;
    VkBool32 drawIndirectCount;
    VkBool32 storageBuffer8BitAccess;
    VkBool32 uniformAndStorageBuffer8BitAccess;
    VkBool32 storagePushConstant8;
    VkBool32 shaderBufferInt64Atomics;
    VkBool32 shaderSharedInt64Atomics;
    VkBool32 shaderFloat16;
    VkBool32 shaderInt8;
    VkBool32 descriptorIndexing;
    VkBool32 shaderInputAttachmentArrayDynamicIndexing;
    VkBool32 shaderUniformTexelBufferArrayDynamicIndexing;
    VkBool32 shaderStorageTexelBufferArrayDynamicIndexing;
    VkBool32 shaderUniformBufferArrayNonUniformIndexing;
    VkBool32 shaderSampledImageArrayNonUniformIndexing;
    VkBool32 shaderStorageBufferArrayNonUniformIndexing;
    VkBool32 descriptorBindingUniformBufferUpdateAfterBind;
    VkBool32 descriptorBindingSampledImageUpdateAfterBind;
    VkBool32 descriptorBindingStorageBufferUpdateAfterBind;
    VkBool32 descriptorBindingUniformTexelBufferUpdateAfterBind;
    VkBool32 descriptorBindingStorageTexelBufferUpdateAfterBind;
    VkBool32 descriptorBindingUpdateUnusedWhilePending;
    VkBool32 descriptorBindingPartiallyBound;
    VkBool32 descriptorBindingVariableDescriptorCount;
    VkBool32 runtimeDescriptorArray;
    VkBool32 samplerFilterMinmax;
    VkBool32 scalarBlockLayout;
    VkBool32 imagelessFramebuffer;
    VkBool32 uniformBufferStandardLayout;
    VkBool32 shaderSubgroupExtendedTypes;
    VkBool32 separateDepthStencilLayouts;
} VkPhysicalDeviceVulkan12Features;
```
VkBool32 hostQueryReset;
VkBool32 timelineSemaphore;
VkBool32 bufferDeviceAddress;
VkBool32 bufferDeviceAddressCaptureReplay;
VkBool32 bufferDeviceAddressMultiDevice;
VkBool32 vulkanMemoryModel;
VkBool32 vulkanMemoryModelDeviceScope;
VkBool32 vulkanMemoryModelAvailabilityVisibilityChains;
VkBool32 shaderOutputViewportIndex;
VkBool32 shaderOutputLayer;
VkBool32 subgroupBroadcastDynamicId;

} VkPhysicalDeviceVulkan12Features;

This structure describes the following features:

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

- **samplerMirrorClampToEdge** indicates whether the implementation supports the
  VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE sampler address mode. If this feature is not
  enabled, the VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE sampler address mode must not be used.

- **drawIndirectCount** indicates whether the implementation supports the
  vkCmdDrawIndirectCount and vkCmdDrawIndexedIndirectCount functions. If this feature is
  not enabled, these functions must not be used.

- **storageBuffer8BitAccess** indicates whether objects in the StorageBuffer, or
  PhysicalStorageBuffer storage class with the Block decoration can have 8-bit integer members. If this feature is not
  enabled, 8-bit integer members must not be used in such objects. This also indicates whether shader modules can declare the StorageBuffer8BitAccess capability.

- **uniformAndStorageBuffer8BitAccess** indicates whether objects in the Uniform storage class with
  the Block decoration can have 8-bit integer members. If this feature is not enabled, 8-bit integer
  members must not be used in such objects. This also indicates whether shader modules can declare the UniformAndStorageBuffer8BitAccess capability.

- **storagePushConstant8** indicates whether objects in the PushConstant storage class can have 8-bit
  integer members. If this feature is not enabled, 8-bit integer members must not be used in such objects. This also indicates whether shader modules can declare the StoragePushConstant8 capability.

- **shaderBufferInt64Atomics** indicates whether shaders can perform 64-bit unsigned and signed
  integer atomic operations on buffers.

- **shaderSharedInt64Atomics** indicates whether shaders can perform 64-bit unsigned and signed
  integer atomic operations on shared memory.

- **shaderFloat16** indicates whether 16-bit floats (halves) are supported in shader code. This also
  indicates whether shader modules can declare the Float16 capability. However, this only
  enables a subset of the storage classes that SPIR-V allows for the Float16 SPIR-V capability:
  Declaring and using 16-bit floats in the Private, Workgroup, and Function storage classes is
enabled, while declaring them in the interface storage classes (e.g., UniformConstant, Uniform, StorageBuffer, Input, Output, and PushConstant) is not enabled.

- **shaderInt8** indicates whether 8-bit integers (signed and unsigned) are supported in shader code. This also indicates whether shader modules can declare the Int8 capability. However, this only enables a subset of the storage classes that SPIR-V allows for the Int8 SPIR-V capability: Declaring and using 8-bit integers in the Private, Workgroup, and Function storage classes is enabled, while declaring them in the interface storage classes (e.g., UniformConstant, Uniform, StorageBuffer, Input, Output, and PushConstant) is not enabled.

- **descriptorIndexing** indicates whether the implementation supports the minimum set of descriptor indexing features as described in the Feature Requirements section. Enabling the descriptorIndexing member when vkCreateDevice is called does not imply the other minimum descriptor indexing features are also enabled. Those other descriptor indexing features must be enabled individually as needed by the application.

- **shaderInputAttachmentArrayDynamicIndexing** indicates whether arrays of input attachments can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the InputAttachmentArrayDynamicIndexing capability.

- **shaderUniformTexelBufferArrayDynamicIndexing** indicates whether arrays of uniform texel buffers can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the UniformTexelBufferArrayDynamicIndexing capability.

- **shaderStorageTexelBufferArrayDynamicIndexing** indicates whether arrays of storage texel buffers can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the StorageTexelBufferArrayDynamicIndexing capability.

- **shaderUniformBufferArrayNonUniformIndexing** indicates whether arrays of uniform buffers can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the UniformBufferArrayNonUniformIndexing capability.

- **shaderSampledImageArrayNonUniformIndexing** indicates whether arrays of samplers or sampled images can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_SAMPLER, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, or VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the SampledImageArrayNonUniformIndexing capability.

- **shaderStorageBufferArrayNonUniformIndexing** indicates whether arrays of storage buffers can be
indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `StorageBufferArrayNonUniformIndexing` capability.

- `shaderStorageImageArrayNonUniformIndexing` indicates whether arrays of storage images can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE` must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `StorageImageArrayNonUniformIndexing` capability.

- `shaderInputAttachmentArrayNonUniformIndexing` indicates whether arrays of input attachments can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `InputAttachmentArrayNonUniformIndexing` capability.

- `shaderUniformTexelBufferArrayNonUniformIndexing` indicates whether arrays of uniform texel buffers can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `UniformTexelBufferArrayNonUniformIndexing` capability.

- `shaderStorageTexelBufferArrayNonUniformIndexing` indicates whether arrays of storage texel buffers can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `StorageTexelBufferArrayNonUniformIndexing` capability.

- `descriptorBindingUniformBufferUpdateAfterBind` indicates whether the implementation supports updating uniform buffer descriptors after a set is bound. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` must not be used with `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`.

- `descriptorBindingSampledImageUpdateAfterBind` indicates whether the implementation supports updating sampled image descriptors after a set is bound. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` must not be used with `VK_DESCRIPTOR_TYPE_SAMPLER`, `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, or `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`.

- `descriptorBindingStorageImageUpdateAfterBind` indicates whether the implementation supports updating storage image descriptors after a set is bound. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` must not be used with `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`.

- `descriptorBindingStorageBufferUpdateAfterBind` indicates whether the implementation supports updating storage buffer descriptors after a set is bound. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` must not be used with
VK_DESCRIPTOR_TYPE_STORAGE_BUFFER.

- descriptorBindingUniformTexelBufferUpdateAfterBind indicates whether the implementation supports updating uniform texel buffer descriptors after a set is bound. If this feature is not enabled, **VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT must not be used with VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER.**

- descriptorBindingStorageTexelBufferUpdateAfterBind indicates whether the implementation supports updating storage texel buffer descriptors after a set is bound. If this feature is not enabled, **VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT must not be used with VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER.**

- descriptorBindingUpdateUnusedWhilePending indicates whether the implementation supports updating descriptors while the set is in use. If this feature is not enabled, **VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT must not be used.**

- descriptorBindingPartiallyBound indicates whether the implementation supports statically using a descriptor set binding in which some descriptors are not valid. If this feature is not enabled, **VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT must not be used.**

- descriptorBindingVariableDescriptorCount indicates whether the implementation supports descriptor sets with a variable-sized last binding. If this feature is not enabled, **VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT must not be used.**

- runtimeDescriptorArray indicates whether the implementation supports the SPIR-V RuntimeDescriptorArray capability. If this feature is not enabled, descriptors must not be declared in runtime arrays.

- samplerFilterMinmax indicates whether the implementation supports a minimum set of required formats supporting min/max filtering as defined by the filterMinmaxSingleComponentFormats property minimum requirements. If this feature is not enabled, then no VkSamplerCreateInfo pNext chain can include a VkSamplerReductionModeCreateInfo structure.

- scalarBlockLayout indicates that the implementation supports the layout of resource blocks in shaders using scalar alignment.

- imagelessFramebuffer indicates that the implementation supports specifying the image view for attachments at render pass begin time via VkRenderPassAttachmentBeginInfo.

- uniformBufferStandardLayout indicates that the implementation supports the same layouts for uniform buffers as for storage and other kinds of buffers. See Standard Buffer Layout.

- shaderSubgroupExtendedTypes is a boolean specifying whether subgroup operations can use 8-bit integer, 16-bit integer, 64-bit integer, 16-bit floating-point, and vectors of these types in group operations with subgroup scope, if the implementation supports the types.

- separateDepthStencilLayouts indicates whether the implementation supports a VkImageMemoryBarrier for a depth/stencil image with only one of VK_IMAGE_ASPECT_DEPTH_BIT or VK_IMAGE_ASPECT_STENCIL_BIT set, and whether VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL can be used.

- hostQueryReset indicates that the implementation supports resetting queries from the host with vkResetQueryPool.

- timelineSemaphore indicates whether semaphores created with a VkSemaphoreType of
VK_SEMAPHORE_TYPE_TIMELINE are supported.

- **bufferDeviceAddress** indicates that the implementation supports accessing buffer memory in shaders as storage buffers via an address queried from *vkGetBufferDeviceAddress*.
- **bufferDeviceAddressCaptureReplay** indicates that the implementation supports saving and reusing buffer and device addresses, e.g. for trace capture and replay.
- **bufferDeviceAddressMultiDevice** indicates that the implementation supports the bufferDeviceAddress feature for logical devices created with multiple physical devices. If this feature is not supported, buffer addresses must not be queried on a logical device created with more than one physical device.
- **vulkanMemoryModel** indicates whether the Vulkan Memory Model is supported, as defined in Vulkan Memory Model. This also indicates whether shader modules can declare the VulkanMemoryModel capability.
- **vulkanMemoryModelDeviceScope** indicates whether the Vulkan Memory Model can use Device scope synchronization. This also indicates whether shader modules can declare the VulkanMemoryModelDeviceScope capability.
- **vulkanMemoryModelAvailabilityVisibilityChains** indicates whether the Vulkan Memory Model can use availability and visibility chains with more than one element.
- **shaderOutputViewportIndex** indicates whether the implementation supports the ShaderViewportIndex SPIR-V capability enabling variables decorated with the ViewportIndex built-in to be exported from vertex or tessellation evaluation shaders. If this feature is not enabled, the ViewportIndex built-in decoration must not be used on outputs in vertex or tessellation evaluation shaders.
- **shaderOutputLayer** indicates whether the implementation supports the ShaderLayer SPIR-V capability enabling variables decorated with the Layer built-in to be exported from vertex or tessellation evaluation shaders. If this feature is not enabled, the Layer built-in decoration must not be used on outputs in vertex or tessellation evaluation shaders.
- If subgroupBroadcastDynamicId is VK_TRUE, the “Id” operand of OpGroupNonUniformBroadcast can be dynamically uniform within a subgroup, and the “Index” operand of OpGroupNonUniformQuadBroadcast can be dynamically uniform within the derivative group. If it is VK_FALSE, these operands must be constants.

If the VkPhysicalDeviceVulkan12Features structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceVulkan12Features can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

### Valid Usage (Implicit)
- VUID-VkPhysicalDeviceVulkan12Features-sType-sType
  - sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_2_FEATURES

The VkPhysicalDeviceVariablePointersFeatures structure is defined as:
typedef struct VkPhysicalDeviceVariablePointersFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 variablePointersStorageBuffer;
    VkBool32 variablePointers;
} VkPhysicalDeviceVariablePointersFeatures;

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **variablePointersStorageBuffer** specifies whether the implementation supports the SPIR-V VariablePointersStorageBuffer capability. When this feature is not enabled, shader modules must not declare the SPV_KHR_variable_pointers extension or the VariablePointersStorageBuffer capability.
- **variablePointers** specifies whether the implementation supports the SPIR-V VariablePointers capability. When this feature is not enabled, shader modules must not declare the VariablePointers capability.

If the `VkPhysicalDeviceVariablePointersFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceVariablePointersFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

Valid Usage

- VUID-VkPhysicalDeviceVariablePointersFeatures-variablePointers-01431
  If `variablePointers` is enabled then `variablePointersStorageBuffer` must also be enabled

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceVariablePointersFeatures-sType-sType
  `sType` must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTERS_FEATURES

The `VkPhysicalDeviceMultiviewFeatures` structure is defined as:

```
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceMultiviewFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 multiview;
    VkBool32 multiviewGeometryShader;
} VkPhysicalDeviceMultiviewFeatures;
```
This structure describes the following features:

- **sType** is the type of this structure.

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

- **multiview** specifies whether the implementation supports multiview rendering within a render pass. If this feature is not enabled, the view mask of each subpass must always be zero.

- **multiviewGeometryShader** specifies whether the implementation supports multiview rendering within a render pass, with geometry shaders. If this feature is not enabled, then a pipeline compiled against a subpass with a non-zero view mask must not include a geometry shader.

- **multiviewTessellationShader** specifies whether the implementation supports multiview rendering within a render pass, with tessellation shaders. If this feature is not enabled, then a pipeline compiled against a subpass with a non-zero view mask must not include any tessellation shaders.

If the **VkPhysicalDeviceMultiviewFeatures** structure is included in the **pNext** chain of the **VkPhysicalDeviceFeatures2** structure passed to **vkGetPhysicalDeviceFeatures2**, it is filled in to indicate whether each corresponding feature is supported. **VkPhysicalDeviceMultiviewFeatures** can also be used in the **pNext** chain of **VkDeviceCreateInfo** to selectively enable these features.

### Valid Usage

- VUID-VkPhysicalDeviceMultiviewFeatures-multiviewGeometryShader-00580
  If **multiviewGeometryShader** is enabled then **multiview** must also be enabled

- VUID-VkPhysicalDeviceMultiviewFeatures-multiviewTessellationShader-00581
  If **multiviewTessellationShader** is enabled then **multiview** must also be enabled

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceMultiviewFeatures-sType-sType
  **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_FEATURES**

The **VkPhysicalDeviceShaderAtomicFloatFeaturesEXT** structure is defined as:

```c
// Provided by VK_EXT_shader_atomic_float
typedef struct VkPhysicalDeviceShaderAtomicFloatFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderBufferFloat32Atomics;
    VkBool32 shaderBufferFloat32AtomicAdd;
    VkBool32 shaderBufferFloat64Atomics;
    VkBool32 shaderBufferFloat64AtomicAdd;
} VkPhysicalDeviceShaderAtomicFloatFeaturesEXT;
```
This structure describes the following features:

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

- `shaderBufferFloat32Atomics` indicates whether shaders can perform 32-bit floating-point load, store and exchange atomic operations on storage buffers.

- `shaderBufferFloat32AtomicAdd` indicates whether shaders can perform 32-bit floating-point add atomic operations on storage buffers.

- `shaderBufferFloat64Atomics` indicates whether shaders can perform 64-bit floating-point load, store and exchange atomic operations on storage buffers.

- `shaderBufferFloat64AtomicAdd` indicates whether shaders can perform 64-bit floating-point add atomic operations on storage buffers.

- `shaderSharedFloat32Atomics` indicates whether shaders can perform 32-bit floating-point load, store and exchange atomic operations on shared memory.

- `shaderSharedFloat32AtomicAdd` indicates whether shaders can perform 32-bit floating-point add atomic operations on shared memory.

- `shaderSharedFloat64Atomics` indicates whether shaders can perform 64-bit floating-point load, store and exchange atomic operations on shared memory.

- `shaderSharedFloat64AtomicAdd` indicates whether shaders can perform 64-bit floating-point add atomic operations on shared memory.

- `shaderImageFloat32Atomics` indicates whether shaders can perform 32-bit floating-point load, store and exchange atomic image operations.

- `shaderImageFloat32AtomicAdd` indicates whether shaders can perform 32-bit floating-point add atomic image operations.

- `sparseImageFloat32Atomics` indicates whether 32-bit floating-point load, store and exchange atomic operations can be used on sparse images.

- `sparseImageFloat32AtomicAdd` indicates whether 32-bit floating-point add atomic operations can be used on sparse images.

If the `VkPhysicalDeviceShaderAtomicFloatFeaturesEXT` 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. `VkPhysicalDeviceShaderAtomicFloatFeaturesEXT` can also be used in the `pNext` chain of
VkDeviceCreateInfo to selectively enable these features.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceShaderAtomicFloatFeaturesEXT-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_FLOAT_FEATURES_EXT

The VkPhysicalDeviceShaderAtomicInt64Features structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceShaderAtomicInt64Features {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderBufferInt64Atomics;
    VkBool32 shaderSharedInt64Atomics;
} VkPhysicalDeviceShaderAtomicInt64Features;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **shaderBufferInt64Atomics** indicates whether shaders can perform 64-bit unsigned and signed integer atomic operations on buffers.
- **shaderSharedInt64Atomics** indicates whether shaders can perform 64-bit unsigned and signed integer atomic operations on shared memory.

If the VkPhysicalDeviceShaderAtomicInt64Features structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceShaderAtomicInt64Features can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceShaderAtomicInt64Features-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_INT64_FEATURES

The VkPhysicalDeviceShaderImageAtomicInt64FeaturesEXT structure is defined as:

```c
// Provided by VK_EXT_shader_image_atomic_int64
typedef struct VkPhysicalDeviceShaderImageAtomicInt64FeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderImageInt64Atomics;
} VkPhysicalDeviceShaderImageAtomicInt64FeaturesEXT;
```
VkBool32 sparseImageInt64Atomics;
} VkPhysicalDeviceShaderImageAtomicInt64FeaturesEXT;

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **shaderImageInt64Atomics** indicates whether shaders can support 64-bit unsigned and signed integer atomic operations on images.
- **sparseImageInt64Atomics** indicates whether 64-bit integer atomics can be used on sparse images.

If the `VkPhysicalDeviceShaderImageAtomicInt64FeaturesEXT` 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. `VkPhysicalDeviceShaderImageAtomicInt64FeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceShaderImageAtomicInt64FeaturesEXT-sType-sType be

The `VkPhysicalDevice8BitStorageFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDevice8BitStorageFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 storageBuffer8BitAccess;
    VkBool32 uniformAndStorageBuffer8BitAccess;
    VkBool32 storagePushConstant8;
} VkPhysicalDevice8BitStorageFeatures;
```

This structure describes the following features:

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

- **storageBuffer8BitAccess** indicates whether objects in the StorageBuffer, or PhysicalStorageBuffer storage class with the Block decoration can have 8-bit integer members. If this feature is not enabled, 8-bit integer members must not be used in such objects. This also indicates whether shader modules can declare the StorageBuffer8BitAccess capability.

- **uniformAndStorageBuffer8BitAccess** indicates whether objects in the Uniform storage class with the Block decoration can have 8-bit integer members. If this feature is not enabled, 8-bit integer members must not be used in such objects.
members **must** not be used in such objects. This also indicates whether shader modules **can** declare the `UniformAndStorageBuffer8BitAccess` capability.

- **storagePushConstant8** indicates whether objects in the `PushConstant` storage class **can** have 8-bit integer members. If this feature is not enabled, 8-bit integer members **must** not be used in such objects. This also indicates whether shader modules **can** declare the `StoragePushConstant8` capability.

If the `VkPhysicalDevice8BitStorageFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDevice8BitStorageFeatures` **can** also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDevice8BitStorageFeatures-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_8BIT_STORAGE_FEATURES`

The `VkPhysicalDevice16BitStorageFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDevice16BitStorageFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 storageBuffer16BitAccess;
    VkBool32 uniformAndStorageBuffer16BitAccess;
    VkBool32 storagePushConstant16;
    VkBool32 storageInputOutput16;
} VkPhysicalDevice16BitStorageFeatures;
```

This structure describes the following features:

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

- **storageBuffer16BitAccess** specifies whether objects in the `StorageBuffer`, or `PhysicalStorageBuffer` storage class with the `Block` decoration **can** have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members **must** not be used in such objects. This also specifies whether shader modules **can** declare the `StorageBuffer16BitAccess` capability.

- **uniformAndStorageBuffer16BitAccess** specifies whether objects in the `Uniform` storage class with the `Block` decoration **can** have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members **must** not be used in such objects. This also specifies whether shader modules **can** declare the `UniformAndStorageBuffer16BitAccess` capability.

- **storagePushConstant16** specifies whether objects in the `PushConstant` storage class **can** have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or
floating-point members **must** not be used in such objects. This also specifies whether shader modules **can** declare the `StoragePushConstant16` capability.

- `storageInputOutput16` specifies whether objects in the `Input` and `Output` storage classes **can** have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members **must** not be used in such objects. This also specifies whether shader modules **can** declare the `StorageInputOutput16` capability.

If the `VkPhysicalDevice16BitStorageFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDevice16BitStorageFeatures` **can** also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDevice16BitStorageFeatures-sType-sType
  
  **sType** **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_16BIT_STORAGE_FEATURES`

The `VkPhysicalDeviceShaderFloat16Int8Features` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceShaderFloat16Int8Features {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderFloat16;
    VkBool32 shaderInt8;
} VkPhysicalDeviceShaderFloat16Int8Features;
```

This structure describes the following features:

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

- **shaderFloat16** indicates whether 16-bit floats (halves) are supported in shader code. This also indicates whether shader modules **can** declare the `Float16` capability. However, this only enables a subset of the storage classes that SPIR-V allows for the `Float16` SPIR-V capability: Declaring and using 16-bit floats in the `Private`, `Workgroup`, and `Function` storage classes is enabled, while declaring them in the interface storage classes (e.g., `UniformConstant`, `Uniform`, `StorageBuffer`, `Input`, `Output`, and `PushConstant`) is not enabled.

- **shaderInt8** indicates whether 8-bit integers (signed and unsigned) are supported in shader code. This also indicates whether shader modules **can** declare the `Int8` capability. However, this only enables a subset of the storage classes that SPIR-V allows for the `Int8` SPIR-V capability: Declaring and using 8-bit integers in the `Private`, `Workgroup`, and `Function` storage classes is enabled, while declaring them in the interface storage classes (e.g., `UniformConstant`, `Uniform`, `StorageBuffer`, `Input`, `Output`, and `PushConstant`) is not enabled.

If the `VkPhysicalDeviceShaderFloat16Int8Features` structure is included in the `pNext` chain of the
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 `VkPhysicalDeviceShaderClockFeaturesKHR` structure is defined as:

```c
// Provided by VK_KHR_shader_clock
typedef struct VkPhysicalDeviceShaderClockFeaturesKHR {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderSubgroupClock;
    VkBool32 shaderDeviceClock;
} VkPhysicalDeviceShaderClockFeaturesKHR;
```

This structure describes the following features:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `shaderSubgroupClock` indicates whether shaders can perform `Subgroup` scoped clock reads.
- `shaderDeviceClock` indicates whether shaders can perform `Device` scoped clock reads.

If the `VkPhysicalDeviceShaderClockFeaturesKHR` 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. `VkPhysicalDeviceShaderClockFeaturesKHR` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceShaderClockFeaturesKHR-sType-sType
  - `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_CLOCK_FEATURES_KHR`

The `VkPhysicalDeviceSamplerYcbcrConversionFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceSamplerYcbcrConversionFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 samplerYcbcrConversion;
} VkPhysicalDeviceSamplerYcbcrConversionFeatures;
```
This structure describes the following feature:

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

- **samplerYcbcrConversion** specifies whether the implementation supports **sampler Y’C_bC_r conversion**. If **samplerYcbcrConversion** is **VK_FALSE**, sampler Y’C_bC_r conversion is not supported, and samplers using sampler Y’C_bC_r conversion **must** not be used.

If the **VkPhysicalDeviceSamplerYcbcrConversionFeatures** structure is included in the **pNext** chain of the **VkPhysicalDeviceFeatures2** structure passed to **vkGetPhysicalDeviceFeatures2**, it is filled in to indicate whether each corresponding feature is supported. **VkPhysicalDeviceSamplerYcbcrConversionFeatures** can also be used in the **pNext** chain of **VkDeviceCreateInfo** to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceSamplerYcbcrConversionFeatures-sType-sType
  
  **sType** **must** be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_YCBCR_CONVERSION_FEATURES**

The **VkPhysicalDeviceProtectedMemoryFeatures** structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceProtectedMemoryFeatures {
    VkStructureType   sType;
    void*             pNext;
    VkBool32          protectedMemory;
} VkPhysicalDeviceProtectedMemoryFeatures;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **protectedMemory** specifies whether protected memory is supported.

If the **VkPhysicalDeviceProtectedMemoryFeatures** structure is included in the **pNext** chain of the **VkPhysicalDeviceFeatures2** structure passed to **vkGetPhysicalDeviceFeatures2**, it is filled in to indicate whether each corresponding feature is supported. **VkPhysicalDeviceProtectedMemoryFeatures** can also be used in the **pNext** chain of **VkDeviceCreateInfo** to selectively enable these features.
The `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT` structure is defined as:

```c
// Provided by VK_EXT_blend_operation_advanced
typedef struct VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 advancedBlendCoherentOperations;
} VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **advancedBlendCoherentOperations** specifies whether blending using advanced blend operations is guaranteed to execute atomically and in primitive order. If this is `VK_TRUE`, `VK_ACCESS_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT` is treated the same as `VK_ACCESS_COLOR_ATTACHMENT_READ_BIT`, and advanced blending needs no additional synchronization over basic blending. If this is `VK_FALSE`, then memory dependencies are required to guarantee order between two advanced blending operations that occur on the same sample.

If the `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

The `VkPhysicalDeviceShaderDrawParametersFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceShaderDrawParametersFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderDrawParameters;
} VkPhysicalDeviceShaderDrawParametersFeatures;
```
VkPhysicalDeviceShaderDrawParametersFeatures;

This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `shaderDrawParameters` specifies whether the implementation supports the SPIR-V DrawParameters capability. When this feature is not enabled, shader modules must not declare the SPV_KHR_shader_draw_parameters extension or the DrawParameters capability.

If the `VkPhysicalDeviceShaderDrawParametersFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceShaderDrawParametersFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceShaderDrawParametersFeatures-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DRAW_PARAMETERS_FEATURES`

The `VkPhysicalDeviceDescriptorIndexingFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VK_VERSION_1_2
VkPhysicalDeviceDescriptorIndexingFeatures {
  VkStructureType sType;
  void* pNext;
  VkBool32 shaderInputAttachmentArrayDynamicIndexing;
  VkBool32 shaderUniformTexelBufferArrayDynamicIndexing;
  VkBool32 shaderStorageTexelBufferArrayDynamicIndexing;
  VkBool32 shaderUniformBufferArrayNonUniformIndexing;
  VkBool32 shaderSampledImageArrayNonUniformIndexing;
  VkBool32 shaderStorageBufferArrayNonUniformIndexing;
  VkBool32 shaderInputAttachmentArrayNonUniformIndexing;
  VkBool32 shaderUniformTexelBufferArrayNonUniformIndexing;
  VkBool32 shaderStorageTexelBufferArrayNonUniformIndexing;
  VkBool32 descriptorBindingUniformBufferUpdateAfterBind;
  VkBool32 descriptorBindingSampledImageUpdateAfterBind;
  VkBool32 descriptorBindingStorageImageUpdateAfterBind;
  VkBool32 descriptorBindingStorageBufferUpdateAfterBind;
  VkBool32 descriptorBindingUpdateUnusedWhilePending;
  VkBool32 descriptorBindingPartiallyBound;
  VkBool32 descriptorBindingVariableDescriptorCount;
  VkBool32 runtimeDescriptorArray;
};
```
This structure describes the following features:

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

- **shaderInputAttachmentArrayDynamicIndexing** indicates whether arrays of input attachments can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the InputAttachmentArrayDynamicIndexing capability.

- **shaderUniformTexelBufferArrayDynamicIndexing** indicates whether arrays of uniform texel buffers can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the UniformTexelBufferArrayDynamicIndexing capability.

- **shaderStorageTexelBufferArrayDynamicIndexing** indicates whether arrays of storage texel buffers can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the StorageTexelBufferArrayDynamicIndexing capability.

- **shaderUniformBufferArrayNonUniformIndexing** indicates whether arrays of uniform buffers can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the UniformBufferArrayNonUniformIndexing capability.

- **shaderSampledImageArrayNonUniformIndexing** indicates whether arrays of samplers or sampled images can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_SAMPLER, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, or VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the SampledImageArrayNonUniformIndexing capability.

- **shaderStorageBufferArrayNonUniformIndexing** indicates whether arrays of storage buffers can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the StorageBufferArrayNonUniformIndexing capability.

- **shaderStorageImageArrayNonUniformIndexing** indicates whether arrays of storage images can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_STORAGE_IMAGE must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the StorageImageArrayNonUniformIndexing capability.
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_PARTIALLYBOUND_BIT must not be used.

• descriptorBindingVariableDescriptorCount indicates whether the implementation supports descriptor sets with a variable-sized last binding. If this feature is not enabled, VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT must not be used.

• runtimeDescriptorArray indicates whether the implementation supports the SPIR-V RuntimeDescriptorArray capability. If this feature is not enabled, descriptors must not be declared in runtime arrays.

If the VkPhysicalDeviceDescriptorIndexingFeatures structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceDescriptorIndexingFeatures can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

• VUID-VkPhysicalDeviceDescriptorIndexingFeatures-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_FEATURES

The VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT structure is defined as:

```c
// Provided by VK_EXT_vertex_attribute_divisor
typedef struct VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT {
    VkStructureType   sType;
    void*              pNext;
    VkBool32           vertexAttributeInstanceRateDivisor;
    VkBool32           vertexAttributeInstanceRateZeroDivisor;
} VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT;
```

This structure describes the following features:

• sType is the type of this structure.

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

• vertexAttributeInstanceRateDivisor specifies whether vertex attribute fetching may be repeated in case of instanced rendering.
vertexAttributeInstanceRateZeroDivisor specifies whether a zero value for VkVertexInputBindingDivisorDescriptionEXT::divisor is supported.

If the VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT-sType-sType

The VkPhysicalDeviceASTCDecodeFeaturesEXT structure is defined as:

```c
// Provided by VK_EXT_astc_decode_mode
typedef struct VkPhysicalDeviceASTCDecodeFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 decodeModeSharedExponent;
} VkPhysicalDeviceASTCDecodeFeaturesEXT;
```

This structure describes the following feature:

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- decodeModeSharedExponent indicates whether the implementation supports decoding ASTC compressed formats to VK_FORMAT_E5B9G9R9_UFLOAT_PACK32 internal precision.

If the VkPhysicalDeviceASTCDecodeFeaturesEXT structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceASTCDecodeFeaturesEXT can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceASTCDecodeFeaturesEXT-sType-sType

The VkPhysicalDeviceVulkanMemoryModelFeatures structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceVulkanMemoryModelFeatures {
    VkStructureType sType;
```
This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **vulkanMemoryModel** indicates whether the Vulkan Memory Model is supported, as defined in Vulkan Memory Model. This also indicates whether shader modules can declare the VulkanMemoryModel capability.
- **vulkanMemoryModelDeviceScope** indicates whether the Vulkan Memory Model can use Device scope synchronization. This also indicates whether shader modules can declare the VulkanMemoryModelDeviceScope capability.
- **vulkanMemoryModelAvailabilityVisibilityChains** indicates whether the Vulkan Memory Model can use availability and visibility chains with more than one element.

If the VkPhysicalDeviceVulkanMemoryModelFeaturesKHR structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceVulkanMemoryModelFeaturesKHR can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceVulkanMemoryModelFeatures-sType-sType
  
  **sType** must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_MEMORY_MODEL_FEATURES

The VkPhysicalDeviceScalarBlockLayoutFeatures structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceScalarBlockLayoutFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 scalarBlockLayout;
} VkPhysicalDeviceScalarBlockLayoutFeatures;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **scalarBlockLayout** indicates that the implementation supports the layout of resource blocks in
shaders using scalar alignment.

If the VkPhysicalDeviceScalarBlockLayoutFeatures structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceScalarBlockLayoutFeatures can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceScalarBlockLayoutFeatures-sType-sType
  
  *sType* must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SCALAR_BLOCK_LAYOUT_FEATURES

The VkPhysicalDeviceUniformBufferStandardLayoutFeatures structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceUniformBufferStandardLayoutFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 uniformBufferStandardLayout;
} VkPhysicalDeviceUniformBufferStandardLayoutFeatures;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **uniformBufferStandardLayout** indicates that the implementation supports the same layouts for uniform buffers as for storage and other kinds of buffers. See Standard Buffer Layout.

If the VkPhysicalDeviceUniformBufferStandardLayoutFeatures structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceUniformBufferStandardLayoutFeatures can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceUniformBufferStandardLayoutFeatures-sType-sType
  
  *sType* must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_UNIFORM_BUFFER_STANDARD_LAYOUT_FEATURES

The VkPhysicalDeviceDepthClipEnableFeaturesEXT structure is defined as:

```c
// Provided by VK_EXT_depth_clip_enable
typedef struct VkPhysicalDeviceDepthClipEnableFeaturesEXT {
```

---

1277
This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **depthClipEnable** indicates that the implementation supports setting the depth clipping operation explicitly via the `VkPipelineRasterizationDepthClipStateCreateInfoEXT` pipeline state. Otherwise depth clipping is only enabled when `VkPipelineRasterizationStateCreateInfo::depthClampEnable` is set to **VK_FALSE**.

If the `VkPhysicalDeviceDepthClipEnableFeaturesEXT` structure is included in the **pNext** chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceDepthClipEnableFeaturesEXT` can also be used in the **pNext** chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceDepthClipEnableFeaturesEXT-sType-sType
  
  **sType** **must** be **VK_STRUCTURE_TYPE_PHYSICALDEVICEDEPTHCLIPENABLEFEATURES_EXT**

The `VkPhysicalDeviceBufferDeviceAddressFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceBufferDeviceAddressFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 bufferDeviceAddress;
    VkBool32 bufferDeviceAddressCaptureReplay;
    VkBool32 bufferDeviceAddressMultiDevice;
} VkPhysicalDeviceBufferDeviceAddressFeatures;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **bufferDeviceAddress** indicates that the implementation supports accessing buffer memory in shaders as storage buffers via an address queried from `vkGetBufferDeviceAddress`.
- **bufferDeviceAddressCaptureReplay** indicates that the implementation supports saving and reusing buffer and device addresses, e.g. for trace capture and replay.
• `bufferDeviceAddressMultiDevice` indicates that the implementation supports the `bufferDeviceAddress` feature for logical devices created with multiple physical devices. If this feature is not supported, buffer addresses must not be queried on a logical device created with more than one physical device.

**Note**

`bufferDeviceAddressMultiDevice` exists to allow certain legacy platforms to be able to support `bufferDeviceAddress` without needing to support shared GPU virtual addresses for multi-device configurations.

See `vkGetBufferDeviceAddress` for more information.

If the `VkPhysicalDeviceBufferDeviceAddressFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceBufferDeviceAddressFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceBufferDeviceAddressFeatures-sType-sType
  
sType must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES`

The `VkPhysicalDeviceImagelessFramebufferFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceImagelessFramebufferFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 imagelessFramebuffer;
} VkPhysicalDeviceImagelessFramebufferFeatures;
```

This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `imagelessFramebuffer` indicates that the implementation supports specifying the image view for attachments at render pass begin time via `VkRenderPassAttachmentBeginInfo`.

If the `VkPhysicalDeviceImagelessFramebufferFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceImagelessFramebufferFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceImagelessFramebufferFeatures-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGELESS_FRAMEBUFFER_FEATURES

The VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT structure is defined as:

```c
// Provided by VK_EXT_fragment_shader_interlock
typedef struct VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 fragmentShaderSampleInterlock;
    VkBool32 fragmentShaderPixelInterlock;
    VkBool32 fragmentShaderShadingRateInterlock;
} VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **fragmentShaderSampleInterlock** indicates that the implementation supports the FragmentShaderSampleInterlockEXT SPIR-V capability.
- **fragmentShaderPixelInterlock** indicates that the implementation supports the FragmentShaderPixelInterlockEXT SPIR-V capability.
- **fragmentShaderShadingRateInterlock** indicates that the implementation supports the FragmentShaderShadingRateInterlockEXT SPIR-V capability.

If the VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADER_INTERLOCK_FEATURES_EXT

The VkPhysicalDeviceYcbcrImageArraysFeaturesEXT structure is defined as:

```c
// Provided by VK_EXT_ycbcr_image_arrays
typedef struct VkPhysicalDeviceYcbcrImageArraysFeaturesEXT {
    VkStructureType sType;
    // ... other fields...
} VkPhysicalDeviceYcbcrImageArraysFeaturesEXT;
```
This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `ycbcrImageArrays` indicates that the implementation supports creating images with a format that requires Y’CbCr conversion and has multiple array layers.

If the `VkPhysicalDeviceYcbcrImageArraysFeaturesEXT` 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. `VkPhysicalDeviceYcbcrImageArraysFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

---

**Valid Usage (Implicit)**

- `VUID-VkPhysicalDeviceYcbcrImageArraysFeaturesEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_YCBCR_IMAGE_ARRAYS_FEATURES_EXT`
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures-sType-sType
  must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SUBGROUP_EXTENDED_TYPES_FEATURES

The `VkPhysicalDeviceHostQueryResetFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceHostQueryResetFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 hostQueryReset;
} VkPhysicalDeviceHostQueryResetFeatures;
```

This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `hostQueryReset` indicates that the implementation supports resetting queries from the host with `vkResetQueryPool`.

If the `VkPhysicalDeviceHostQueryResetFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceHostQueryResetFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceHostQueryResetFeatures-sType-sType
  `sType` must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_HOST_QUERY_RESET_FEATURES

The `VkPhysicalDeviceTimelineSemaphoreFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceTimelineSemaphoreFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 timelineSemaphore;
} VkPhysicalDeviceTimelineSemaphoreFeatures;
```

This structure describes the following feature:

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

• `timelineSemaphore` indicates whether semaphores created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` are supported.

If the `VkPhysicalDeviceTimelineSemaphoreFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceTimelineSemaphoreFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

• VUID-VkPhysicalDeviceTimelineSemaphoreFeatures-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_FEATURES`

The `VkPhysicalDeviceExternalSciSyncFeaturesNV` structure is defined as:

```c
// Provided by VK_NV_external_sci_sync
typedef struct VkPhysicalDeviceExternalSciSyncFeaturesNV {
    VkStructureType         sType;
    void*                   pNext;
    VkBool32                sciSyncFence;
    VkBool32                sciSyncSemaphore;
    VkBool32                sciSyncImport;
    VkBool32                sciSyncExport;
} VkPhysicalDeviceExternalSciSyncFeaturesNV;
```

The members of the `VkPhysicalDeviceExternalSciSyncFeaturesNV` structure describe the following features:

• `sciSyncFence` indicates whether external fences created with a handle type of `VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_OBJ_BIT_NV` and `VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_FENCE_BIT_NV` are supported for import and/or export.

• `sciSyncSemaphore` indicates whether external semaphores created with a handle type of `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SCI_SYNC_OBJ_BIT_NV` are supported for import and/or export.

• `sciSyncImport` indicates whether `NvSciSyncObj` import functionality is supported. If `sciSyncImport` is set to `VK_TRUE`, `VkFence` and/or `VkSemaphore` support importing `NvSciSyncObj` from applications. In this case, the application is responsible for the resource management of the `NvSciSyncObj`.

• `sciSyncExport` indicates whether `NvSciSyncObj` export functionality is supported. If `sciSyncExport` is set to `VK_TRUE`, `VkFence` and/or `VkSemaphore` support exporting `NvSciSyncObj` created by the driver to applications. In this case, the driver is responsible for the resource management of the `NvSciSyncObj`.

*Table 43. Functionality supported for NvSciSync features*
### Features

<table>
<thead>
<tr>
<th>Features</th>
<th>sciSyncImport</th>
<th>sciSyncExport</th>
<th>Always supported¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>sciSyncFence</td>
<td>vkImportFenceSciSyncFenceNV,</td>
<td>VkExportFenceSciSyncInfoNV</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vkImportFenceSciSyncObjNV</td>
<td></td>
<td>vkGetFenceSciSyncFenceNV,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>vkGetFenceSciSyncObjNV,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>vkGetPhysicalDeviceSci</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SyncAttributesNV (with</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VK_SCI_SYNC_PRIMITIVE_</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TYPE_FENCE_NV)</td>
</tr>
<tr>
<td>sciSyncSemaphore</td>
<td>vkImportSemaphoreSciSyncObjNV</td>
<td>VkExportSemaphoreSciSyncInfoNV</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>vkGetSemaphoreSciSyn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cObjNV,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>vkGetPhysicalDeviceSci</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SyncAttributesNV (with</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VK_SCI_SYNC_PRIMITIVE_</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TYPE_SEMAPHORE_NV)</td>
</tr>
</tbody>
</table>

1

Functionality in this column is always available.

The **Functionality supported for NvSciSync features** table summarizes the functionality enabled by the *VkPhysicalDeviceExternalSciSyncFeaturesNV* structure. There are two orthogonal pieces of functionality: fence and semaphore support; import and export support. Each entry in the body of the table summarizes the functionality that **can** be used when the given features are supported and enabled. This summarizes Valid Usage statements that are added elsewhere in this specification.

If the *VkPhysicalDeviceExternalSciSyncFeaturesNV* 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. *VkPhysicalDeviceExternalSciSyncFeaturesNV* **can** also be used in the *pNext* chain of *VkDeviceCreateInfo* to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceExternalSciSyncFeaturesNV-sType-sType
  sType **must** be *VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SCI_SYNC_FEATURES_NV*

The *VkPhysicalDeviceExternalSciSync2FeaturesNV* structure is defined as:

```c
// Provided by VK_NV_external_sci_sync2
typedef struct VkPhysicalDeviceExternalSciSync2FeaturesNV {
    VkStructureType     sType;
    void*                pNext;
    VkBool32             sciSyncFence;
    VkBool32             sciSyncSemaphore2;
    VkBool32             sciSyncImport;
    VkBool32             sciSyncExport;
} VkPhysicalDeviceExternalSciSync2FeaturesNV;
```
The members of the `VkPhysicalDeviceExternalSciSync2FeaturesNV` structure describe the following features:

- `sciSyncFence` indicates whether external fences created with a handle type of `VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_OBJ_BIT_NV` and `VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_FENCE_BIT_NV` are supported for import and/or export.

- `sciSyncSemaphore2` indicates whether semaphore SciSync pools are supported and semaphores can be created from `NvSciSyncObj` via `VkSemaphoreSciSyncPoolNV` objects. In this case, the application is responsible for the resource management of the `NvSciSyncObj`.

- `sciSyncImport` indicates whether `NvSciSyncObj` import functionality is supported. If `sciSyncImport` is set to `VK_TRUE`, `VkFence` and/or `VkSemaphore` support importing `NvSciSyncObj` from applications. In this case, the application is responsible for the resource management of the `NvSciSyncObj`.

- `sciSyncExport` indicates whether `NvSciSyncObj` export functionality is supported. If `sciSyncExport` is set to `VK_TRUE`, `VkFence` supports exporting `NvSciSyncObj` created by the driver to applications. In this case, the driver is responsible for the resource management of the `NvSciSyncObj`.

### Table 44. Functionality supported for NvSciSync features

<table>
<thead>
<tr>
<th>Features</th>
<th>sciSyncImport</th>
<th>sciSyncExport</th>
<th>Always supported(^1)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sciSyncSemaphore2</code></td>
<td><code>vkCreateSemaphoreSciSyncPoolNV</code>, <code>VkSemaphoreSciSyncCreateInfoNV</code></td>
<td>n/a</td>
<td><code>vkGetPhysicalDeviceSciSyncAttributesNV</code> (with <code>VK_SCI_SYNC_PRIMITIVE_TYPE_SEMAPHORE_NV</code>)</td>
</tr>
</tbody>
</table>

\(^1\) Functionality in this column is always available.

The **Functionality supported for NvSciSync features** table summarizes the functionality enabled by the `VkPhysicalDeviceExternalSciSync2FeaturesNV` structure. There are two orthogonal pieces of functionality: fence and semaphore support; import and export support. Each entry in the body of the table summarizes the functionality that can be used when the given features are supported and enabled. This summarizes Valid Usage statements that are added elsewhere in this specification.

If the `VkPhysicalDeviceExternalSciSync2FeaturesNV` 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.
VkPhysicalDeviceExternalSciSync2FeaturesNV can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceExternalSciSync2FeaturesNV-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SCI_SYNC_2_FEATURES_NV

The VkPhysicalDeviceExternalMemorySciBufFeaturesNV structure is defined as:

```c
// Provided by VK_NV_external_memory_sci_buf
typedef struct VkPhysicalDeviceExternalMemorySciBufFeaturesNV {
    VkStructureType sType;
    void* pNext;
    VkBool32 sciBufImport;
    VkBool32 sciBufExport;
} VkPhysicalDeviceExternalMemorySciBufFeaturesNV;
```

The members of the VkPhysicalDeviceExternalMemorySciBufFeaturesNV structure describe the following features:

- sciBufImport indicates whether NvSciBufObj import functionality is supported. If sciBufImport is set to VK_TRUE, VkDeviceMemory supports importing NvSciBufObj from applications. In this case, the application is responsible for the resource management of the NvSciBufObj.

- sciBufExport indicates whether NvSciBufObj export functionality is supported. If sciBufExport is set to VK_TRUE, VkDeviceMemory supports exporting NvSciBufObj created by the driver to applications. In this case, the driver is responsible for the resource management of the NvSciBufObj.

<table>
<thead>
<tr>
<th>Features</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>sciBufImport</td>
<td>VkImportMemorySciBufInfoNV,</td>
</tr>
<tr>
<td></td>
<td>vkGetPhysicalDeviceExternalMemorySciBufFeaturesNV</td>
</tr>
<tr>
<td>sciBufExport</td>
<td>VkExportMemorySciBufInfoNV</td>
</tr>
<tr>
<td>Always supported¹</td>
<td>vkGetPhysicalDeviceSciBufAttributesNV,</td>
</tr>
<tr>
<td></td>
<td>vkGetMemorySciBufNV</td>
</tr>
</tbody>
</table>

¹ Functionality in this row is always available.
The Functionality supported for NvSciBuf features table summarizes the functionality enabled by the VkPhysicalDeviceExternalMemorySciBufFeaturesNV structure. Each entry in the body of the table summarizes the functionality that can be used when the given features are supported and enabled. This summarizes Valid Usage statements that are added elsewhere in this specification.

If the VkPhysicalDeviceExternalMemorySciBufFeaturesNV 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. VkPhysicalDeviceExternalMemorySciBufFeaturesNV can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceExternalMemorySciBufFeaturesNV-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_MEMORY_SCI_BUF_FEATURES_NV

The VkPhysicalDeviceIndexTypeUint8FeaturesEXT structure is defined as:

```c
// Provided by VK_EXT_index_type_uint8
typedef struct VkPhysicalDeviceIndexTypeUint8FeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 indexTypeUint8;
} VkPhysicalDeviceIndexTypeUint8FeaturesEXT;
```

This structure describes the following feature:

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- indexTypeUint8 indicates that VK_INDEX_TYPE_UINT8_EXT can be used with vkCmdBindIndexBuffer.

If the VkPhysicalDeviceIndexTypeUint8FeaturesEXT 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. VkPhysicalDeviceIndexTypeUint8FeaturesEXT can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceIndexTypeUint8FeaturesEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INDEX_TYPE_UINT8_FEATURES_EXT

The VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures structure is defined as:

```c
// Provided by VK_VERSION_1_2
```
typedef struct VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 separateDepthStencilLayouts;
} VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures;

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **separateDepthStencilLayouts** indicates whether the implementation supports a VkImageMemoryBarrier for a depth/stencil image with only one of VK_IMAGE_ASPECT_DEPTH_BIT or VK_IMAGE_ASPECT_STENCIL_BIT set, and whether VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL can be used.

If the VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures-sType-sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SEPARATE_DEPTH_STENCIL_LAYOUTS_FEATURES

The VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT structure is defined as:

```
// Provided by VK_EXT_shader_demote_to_helper_invocation
typedef struct VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderDemoteToHelperInvocation;
} VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **shaderDemoteToHelperInvocation** indicates whether the implementation supports the SPIR-V DemoteToHelperInvocationEXT capability.

If the VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT 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. VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.
chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT-sType-sType sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DEMOTE_TO_HELPER_INVOCATION_FEATURES_EXT

The VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT structure is defined as:

```c
typedef struct VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 texelBufferAlignment;
} VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **texelBufferAlignment** indicates whether the implementation uses more specific alignment requirements advertised in VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT rather than VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment.

If the VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT-sType-sType sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_FEATURES_EXT

The VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT structure is defined as:

```c
// Provided by VK_EXT_texture_compression_astc_hdr
typedef struct VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT {
    VkStructureType sType;
    void* pNext;
} VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT;
```
This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **textureCompressionASTC_HDR** indicates whether all of the ASTC HDR compressed texture formats are supported. If this feature is enabled, then the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `VK_FORMAT_FEATURE_BLIT_SRC_BIT` and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` features must be supported in `optimalTilingFeatures` for the following formats:
  - `VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK_EXT`

To query for additional properties, or if the feature is not enabled, `vkGetPhysicalDeviceFormatProperties` and `vkGetPhysicalDeviceImageFormatProperties` can be used to check for supported properties of individual formats as normal.

If the `VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT-sType-sType** must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXTURE_COMPRESSION_ASTC_HDR_FEATURES_EXT`
The `VkPhysicalDeviceLineRasterizationFeaturesEXT` structure is defined as:

```c
// Provided by VK_EXT_line_rasterization
typedef struct VkPhysicalDeviceLineRasterizationFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 rectangularLines;
    VkBool32 bresenhamLines;
    VkBool32 smoothLines;
    VkBool32 stippledRectangularLines;
    VkBool32 stippledBresenhamLines;
    VkBool32 stippledSmoothLines;
} VkPhysicalDeviceLineRasterizationFeaturesEXT;
```

This structure describes the following features:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `rectangularLines` indicates whether the implementation supports rectangular line rasterization.
- `bresenhamLines` indicates whether the implementation supports Bresenham-style line rasterization.
- `smoothLines` indicates whether the implementation supports smooth line rasterization.
- `stippledRectangularLines` indicates whether the implementation supports stippled line rasterization with `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT` lines, or with `VK_LINE_RASTERIZATION_MODE_DEFAULT_EXT` lines if `VkPhysicalDeviceLimits::strictLines` is `VK_TRUE`.
- `stippledBresenhamLines` indicates whether the implementation supports stippled line rasterization with `VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT` lines.
- `stippledSmoothLines` indicates whether the implementation supports stippled line rasterization with `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT` lines.

If the `VkPhysicalDeviceLineRasterizationFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceLineRasterizationFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

---

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceLineRasterizationFeaturesEXT-sType-sType` sType must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_LINE_RASTERIZATION_FEATURES_EXT`
typedef struct VkPhysicalDeviceSubgroupSizeControlFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 subgroupSizeControl;
    VkBool32 computeFullSubgroups;
} VkPhysicalDeviceSubgroupSizeControlFeaturesEXT;

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **subgroupSizeControl** indicates whether the implementation supports controlling shader subgroup sizes via the VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT flag and the VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT structure.
- **computeFullSubgroups** indicates whether the implementation supports requiring full subgroups in compute shaders via the VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT flag.

If the VkPhysicalDeviceSubgroupSizeControlFeaturesEXT structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceSubgroupSizeControlFeaturesEXT can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceSubgroupSizeControlFeaturesEXT-sType-sType
  sType **must** be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_FEATURES_EXT

The VkPhysicalDeviceExtendedDynamicStateFeaturesEXT structure is defined as:

```
// Provided by VK_EXT_extended_dynamic_state
typedef struct VkPhysicalDeviceExtendedDynamicStateFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 extendedDynamicState;
} VkPhysicalDeviceExtendedDynamicStateFeaturesEXT;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **extendedDynamicState** indicates that the implementation supports the following dynamic states:
  - **VK_DYNAMIC_STATE_CULL_MODE_EXT**
- VK_DYNAMIC_STATE_FRONT_FACE_EXT
- VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT
- VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT
- VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT
- VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT
- VK_DYNAMIC_STATE_DEPTH_TEST_ENABLE_EXT
- VK_DYNAMIC_STATE_DEPTH_WRITE_ENABLE_EXT
- VK_DYNAMIC_STATE_DEPTH_COMPARE_OP_EXT
- VK_DYNAMIC_STATE_DEPTH_BOUNDS_TEST_ENABLE_EXT
- VK_DYNAMIC_STATE_STENCIL_TEST_ENABLE_EXT
- VK_DYNAMIC_STATE_STENCIL_OP_EXT

If the VkPhysicalDeviceExtendedDynamicStateFeaturesEXT 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. VkPhysicalDeviceExtendedDynamicStateFeaturesEXT can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceExtendedDynamicStateFeaturesEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTENDED_DYNAMIC_STATE_FEATURES_EXT

The VkPhysicalDeviceExtendedDynamicState2FeaturesEXT structure is defined as:

```c
// Provided by VK_EXT_extended_dynamic_state2
typedef struct VkPhysicalDeviceExtendedDynamicState2FeaturesEXT {
  VkStructureType sType;
  void* pNext;
  VkBool32 extendedDynamicState2;
  VkBool32 extendedDynamicState2LogicOp;
  VkBool32 extendedDynamicState2PatchControlPoints;
} VkPhysicalDeviceExtendedDynamicState2FeaturesEXT;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **extendedDynamicState2** indicates that the implementation supports the following dynamic states:
  - VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT
  - VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT
• **VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT**

  - `extendedDynamicState2LogicOp` indicates that the implementation supports the following dynamic state:
    - **VK_DYNAMIC_STATE_LOGIC_OP_EXT**

  - `extendedDynamicState2PatchControlPoints` indicates that the implementation supports the following dynamic state:
    - **VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT**

If the `VkPhysicalDeviceExtendedDynamicState2FeaturesEXT` 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. `VkPhysicalDeviceExtendedDynamicState2FeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceExtendedDynamicState2FeaturesEXT-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTENDED_DYNAMIC_STATE_2_FEATURES_EXT`

The `VkPhysicalDeviceRobustness2FeaturesEXT` structure is defined as:

```c
// Provided by VK_EXT_robustness2
typedef struct VkPhysicalDeviceRobustness2FeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 robustBufferAccess2;
    VkBool32 robustImageAccess2;
    VkBool32 nullDescriptor;
} VkPhysicalDeviceRobustness2FeaturesEXT;
```

This structure describes the following features:

- **sType** is the type of this structure.

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

- **robustBufferAccess2** indicates whether buffer accesses are tightly bounds-checked against the range of the descriptor. Uniform buffers must be bounds-checked to the range of the descriptor, where the range is rounded up to a multiple of `robustUniformBufferAccessSizeAlignment`. Storage buffers must be bounds-checked to the range of the descriptor, where the range is rounded up to a multiple of `robustStorageBufferAccessSizeAlignment`. Out of bounds buffer loads will return zero values, and formatted loads will have (0,0,1) values inserted for missing G, B, or A components based on the format.

- **robustImageAccess2** indicates whether image accesses are tightly bounds-checked against the dimensions of the image view. Out of bounds image loads will return zero values, with (0,0,1) values inserted for missing G, B, or A components based on the format.
nullDescriptor indicates whether descriptors can be written with a VK_NULL_HANDLE resource or view, which are considered valid to access and act as if the descriptor were bound to nothing.

If the VkPhysicalDeviceRobustness2FeaturesEXT 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. VkPhysicalDeviceRobustness2FeaturesEXT can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage

- VUID-VkPhysicalDeviceRobustness2FeaturesEXT-robustBufferAccess2-04000
  If robustBufferAccess2 is enabled then robustBufferAccess must also be enabled

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceRobustness2FeaturesEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ROBUSTNESS_2_FEATURES_EXT

The VkPhysicalDeviceImageRobustnessFeaturesEXT structure is defined as:

```c
// Provided by VK_EXT_image_robustness
typedef struct VkPhysicalDeviceImageRobustnessFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 robustImageAccess;
} VkPhysicalDeviceImageRobustnessFeaturesEXT;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **robustImageAccess** indicates whether image accesses are tightly bounds-checked against the dimensions of the image view. Invalid texels resulting from out of bounds image loads will be replaced as described in Texel Replacement, with either (0,0,1) or (0,0,0) values inserted for missing G, B, or A components based on the format.

If the VkPhysicalDeviceImageRobustnessFeaturesEXT 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. VkPhysicalDeviceImageRobustnessFeaturesEXT can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceImageRobustnessFeaturesEXT-sType-sType
  sType **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_ROBUSTNESS_FEATURES_EXT`

The `VkPhysicalDeviceShaderTerminateInvocationFeaturesKHR` structure is defined as:

```c
// Provided by VK_KHR_shader_terminate_invocation
typedef struct VkPhysicalDeviceShaderTerminateInvocationFeaturesKHR {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderTerminateInvocation;
} VkPhysicalDeviceShaderTerminateInvocationFeaturesKHR;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **shaderTerminateInvocation** specifies whether the implementation supports SPIR-V modules that use the `SPV_KHR_terminate_invocation` extension.

If the `VkPhysicalDeviceShaderTerminateInvocationFeaturesKHR` 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. `VkPhysicalDeviceShaderTerminateInvocationFeaturesKHR` **can** also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceShaderTerminateInvocationFeaturesKHR-sType-sType
  sType **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_TERMINATE_INVOCATION_FEATURES_KHR`

The `VkPhysicalDeviceCustomBorderColorFeaturesEXT` structure is defined as:

```c
// Provided by VK_EXT_custom_border_color
typedef struct VkPhysicalDeviceCustomBorderColorFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 customBorderColors;
    VkBool32 customBorderColorWithoutFormat;
} VkPhysicalDeviceCustomBorderColorFeaturesEXT;
```

This structure describes the following features:
• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• customBorderColor indicates that the implementation supports providing a borderColor value with one of the following values at sampler creation time:
  ◦ VK_BORDER_COLOR_FLOAT_CUSTOM_EXT
  ◦ VK_BORDER_COLOR_INT_CUSTOM_EXT
• customBorderColorWithoutFormat indicates that explicit formats are not required for custom border colors and the value of the format member of the VkSamplerCustomBorderColorCreateInfoEXT structure may be VK_FORMAT_UNDEFINED. If this feature bit is not set, applications must provide the VkFormat of the image view(s) being sampled by this sampler in the format member of the VkSamplerCustomBorderColorCreateInfoEXT structure.

If the VkPhysicalDeviceCustomBorderColorFeaturesEXT 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. VkPhysicalDeviceCustomBorderColorFeaturesEXT can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

• VUID-VkPhysicalDeviceCustomBorderColorFeaturesEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CUSTOM_BORDER_COLOR_FEATURES_EXT

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 the type of 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
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_SC_1_0_FEATURES

The VkPhysicalDevicePerformanceQueryFeaturesKHR structure is defined as:

```c
// Provided by VK_KHR_performance_query
typedef struct VkPhysicalDevicePerformanceQueryFeaturesKHR {
    VkStructureType sType;
    void* pNext;
    VkBool32 performanceCounterQueryPools;
    VkBool32 performanceCounterMultipleQueryPools;
} VkPhysicalDevicePerformanceQueryFeaturesKHR;
```

This structure describes the following features:

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `performanceCounterQueryPools` indicates whether the implementation supports performance counter query pools.
- `performanceCounterMultipleQueryPools` indicates whether the implementation supports using multiple performance query pools in a primary command buffer and secondary command buffers executed within it.

If the VkPhysicalDevicePerformanceQueryFeaturesKHR 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. VkPhysicalDevicePerformanceQueryFeaturesKHR can also be used in the `pNext` chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDevicePerformanceQueryFeaturesKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PERFORMANCE_QUERY_FEATURES_KHR

The VkPhysicalDevice4444FormatsFeaturesEXT structure is defined as:

```c
// Provided by VK_EXT_4444_formats
typedef struct VkPhysicalDevice4444FormatsFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 formatA4R4G4B4;
    VkBool32 formatA4B4G4R4;
} VkPhysicalDevice4444FormatsFeaturesEXT;
```
This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **formatA4R4G4B4** indicates that the implementation **must** support using a `VkFormat` of `VK_FORMAT_A4R4G4B4_UNORM_PACK16_EXT` with at least the following `VkFormatFeatureFlagBits`:
  - `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`
  - `VK_FORMAT_FEATURE_BLIT_SRC_BIT`
  - `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`
- **formatA4B4G4R4** indicates that the implementation **must** support using a `VkFormat` of `VK_FORMAT_A4B4G4R4_UNORM_PACK16_EXT` with at least the following `VkFormatFeatureFlagBits`:
  - `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`
  - `VK_FORMAT_FEATURE_BLIT_SRC_BIT`
  - `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

If the `VkPhysicalDevice4444FormatsFeaturesEXT` 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. `VkPhysicalDevice4444FormatsFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDevice4444FormatsFeaturesEXT-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_4444_FORMATS_FEATURES_EXT`

The `VkPhysicalDeviceSynchronization2FeaturesKHR` structure is defined as:

```c
// Provided by VK_KHR_synchronization2
typedef struct VkPhysicalDeviceSynchronization2FeaturesKHR {
    VkStructureType sType;
    void* pNext;
    VkBool32 synchronization2;
} VkPhysicalDeviceSynchronization2FeaturesKHR;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **synchronization2** indicates whether the implementation supports the new set of synchronization commands introduced in `VK_KHR_synchronization2`.  

1299
If the `VkPhysicalDeviceSynchronization2FeaturesKHR` 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. `VkPhysicalDeviceSynchronization2FeaturesKHR` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceSynchronization2FeaturesKHR-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SYNCHRONIZATION_2FEATURES_KHR`

The `VkPhysicalDeviceVertexInputDynamicStateFeaturesEXT` structure is defined as:

```c
// Provided by VK_EXT_vertex_input_dynamic_state
typedef struct VkPhysicalDeviceVertexInputDynamicStateFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 vertexInputDynamicState;
} VkPhysicalDeviceVertexInputDynamicStateFeaturesEXT;
```

This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `vertexInputDynamicState` indicates that the implementation supports the following dynamic states:
  - `VK_DYNAMIC_STATE_VERTEX_INPUT_EXT`

If the `VkPhysicalDeviceVertexInputDynamicStateFeaturesEXT` 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. `VkPhysicalDeviceVertexInputDynamicStateFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceVertexInputDynamicStateFeaturesEXT-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VERTEX_INPUT_DYNAMIC_STATE_FEATURES_EXT`

The `VkPhysicalDeviceFragmentShadingRateFeaturesKHR` structure is defined as:

```c
// Provided by VK_KHR_fragment_shading_rate
typedef struct VkPhysicalDeviceFragmentShadingRateFeaturesKHR {
```

If the `VkPhysicalDeviceFragmentShadingRateFeaturesKHR` 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. `VkPhysicalDeviceFragmentShadingRateFeaturesKHR` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.
This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **pipelineFragmentShadingRate** indicates that the implementation supports the pipeline fragment shading rate.
- **primitiveFragmentShadingRate** indicates that the implementation supports the primitive fragment shading rate.
- **attachmentFragmentShadingRate** indicates that the implementation supports the attachment fragment shading rate.

If the `VkPhysicalDeviceFragmentShadingRateFeaturesKHR` 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. `VkPhysicalDeviceFragmentShadingRateFeaturesKHR` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceFragmentShadingRateFeaturesKHR-sType-sType

  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADING_RATE_FEATURES_KHR`.

The `VkPhysicalDeviceYcbcr2Plane444FormatsFeaturesEXT` structure is defined as:

```c
// Provided by VK_EXT_ycbcr_2plane_444_formats
typedef struct VkPhysicalDeviceYcbcr2Plane444FormatsFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 ycbcr2plane444Formats;
} VkPhysicalDeviceYcbcr2Plane444FormatsFeaturesEXT;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **ycbcr2plane444Formats** indicates that the implementation supports the following 2-plane 444 YC\_b\_c\_r formats:
VK_FORMAT_G8_B8R8_2PLANE_444_UNORM_EXT
VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16_EXT
VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_EXT
VK_FORMAT_G16_B16R16_2PLANE_444_UNORM_EXT

If the `VkPhysicalDeviceYcbcr2Plane444FormatsFeaturesEXT` 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. `VkPhysicalDeviceYcbcr2Plane444FormatsFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceYcbcr2Plane444FormatsFeaturesEXT-sType-sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_YCBCR_2_PLANE_444_FORMATS_FEATURES_EXT`

The `VkPhysicalDeviceColorWriteEnableFeaturesEXT` structure is defined as:

```c
// Provided by VK_EXT_color_write_enable
typedef struct VkPhysicalDeviceColorWriteEnableFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 colorWriteEnable;
} VkPhysicalDeviceColorWriteEnableFeaturesEXT;
```

This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `colorWriteEnable` indicates that the implementation supports the dynamic state `VK_DYNAMIC_STATE_COLOR_WRITE_ENABLE_EXT`.

If the `VkPhysicalDeviceColorWriteEnableFeaturesEXT` 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. `VkPhysicalDeviceColorWriteEnableFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceColorWriteEnableFeaturesEXT-sType-sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COLOR_WRITE_ENABLE_FEATURES_EXT`
32.1. 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.
- **subgroupSizeControl**, if the VK_EXT_subgroup_size_control extension is supported.
- **computeFullSubgroups**, if the VK_EXT_subgroup_size_control extension is supported.
- **imagelessFramebuffer**, if Vulkan 1.2 or the VK_KHR_imageless_framebuffer extension is supported.
- **separateDepthStencilLayouts**, if Vulkan 1.2 or the VK_KHR_separate_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.
- `textureCompressionASTC_HDR`, if the `VK_EXT_texture_compression_astc_hdr` extension is supported.
- `depthClipEnable`, if the `VK_EXT_depth_clip_enable` extension is supported.
- `ycbcrImageArrays`, if the `VK_EXT_ycbcr_image_arrays` extension is supported.
- `indexTypeUint8`, if the `VK_EXT_index_type_uint8` extension is supported.
- `shaderDemoteToHelperInvocation`, if the `VK_EXT_shader_demote_to_helper_invocation` extension is supported.
- `texelBufferAlignment`, if the `VK_EXT_texel_buffer_alignment` extension is supported.
- `vulkanMemoryModel`, if Vulkan SC 1.0 [SCID-5] or if the `VK_KHR_vulkan_memory_model` extension is supported.
- `performanceCounterQueryPools`, if the `VK_KHR_performance_query` extension is supported.
- `vertexAttributeInstanceRateDivisor`, if the `VK_EXT_vertex_attribute_divisor` extension is supported.
- `shaderSubgroupClock`, if the `VK_KHR_shader_clock` extension is supported.
- `fragmentShaderSampleInterlock` or `fragmentShaderPixelInterlock` or `fragmentShaderShadingRateInterlock`, if the `VK_EXT_fragment_shader_interlock` extension is supported.
- `rectangularLines` or `bresenhamLines` or `smoothLines` or `stippledRectangularLines` or `stippledBresenhamLines` or `stippledSmoothLines`, if the `VK_EXT_line_rasterization` extension is supported.
- `storageBuffer16BitAccess`, if `uniformAndStorageBuffer16BitAccess` is enabled.
- `robustImageAccess`, if the `VK_EXT_image_robustness` extension is supported.
- `formatA4R4G4B4`, if the `VK_EXT_4444_formats` extension is supported.
- `shaderInt64` and `shaderImageInt64Atomics`, if the `VK_EXT_shader_image_atomic_int64` extension is supported.
- `shaderImageInt64Atomics`, if the `sparseImageInt64Atomics` feature is supported.
- `shaderImageFloat32Atomics`, if the `sparseImageFloat32Atomics` feature is supported.
- `shaderImageFloat32AtomicAdd`, if the `sparseImageFloat32AtomicAdd` feature is supported.
- `pipelineFragmentShadingRate`, if the `VK_KHR_fragment_shading_rate` extension is supported.
- `shaderTerminateInvocation` if the `VK_KHR_shader_terminate_invocation` extension is supported.
- `vertexInputDynamicState`, if the `VK_EXT_vertex_input_dynamic_state` extension is supported.
- `synchronization2` if the `VK_KHR_synchronization2` extension is supported.
- At least one of `sciSyncFence` and `sciSyncSemaphore`, and at least one of `sciSyncImport` and `sciSyncExport`, if the `VK_NV_external_sci_sync` extension is supported.
- At least one of `sciSyncFence` and `sciSyncSemaphore2`, and at least one of `sciSyncImport` and `sciSyncExport`, if the `VK_NV_external_sci_sync2` extension is supported.
• At least one of `sciBufImport` and `sciBufExport`, if the `VK_NV_external_memory_sci_buf` extension is supported.

All other features defined in the Specification are **optional**.
Chapter 33. Limits

Limits are implementation-dependent minimums, maximums, and other device characteristics that an application may need to be aware of.

Note

Limits are reported via the basic VkPhysicalDeviceLimits structure as well as the extensible structure VkPhysicalDeviceProperties2, which was added in VK_KHR_get_physical_device_properties2 and included in Vulkan 1.1. When limits are added in future Vulkan versions or extensions, each extension should introduce one new limit structure, if needed. This structure can be added to the pNext chain of the VkPhysicalDeviceProperties2 structure.

The VkPhysicalDeviceLimits structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPhysicalDeviceLimits {
    uint32_t maxImageDimension1D;
    uint32_t maxImageDimension2D;
    uint32_t maxImageDimension3D;
    uint32_t maxImageDimensionCube;
    uint32_t maxImageArrayLayers;
    uint32_t maxTexelBufferElements;
    uint32_t maxUniformBufferRange;
    uint32_t maxStorageBufferRange;
    uint32_t maxPushConstantsSize;
    uint32_t maxMemoryAllocationCount;
    uint32_t maxSamplerAllocationCount;
    VkDeviceSize bufferImageGranularity;
    VkDeviceSize sparseAddressSpaceSize;
    uint32_t maxBoundDescriptorSets;
    uint32_t maxPerStageDescriptorSamplers;
    uint32_t maxPerStageDescriptorUniformBuffers;
    uint32_t maxPerStageDescriptorStorageBuffers;
    uint32_t maxPerStageDescriptorSampledImages;
    uint32_t maxPerStageDescriptorStorageImages;
    uint32_t maxPerStageDescriptorInputAttachments;
    uint32_t 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 maxComputeWorkSize[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 maxViewportsDimensions[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_UNIFORM_TEXEL_BUFFER**, **VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER**, **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. 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.
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`. 

---

1311
structure. See **Vertex Attributes** and **Vertex Input Description**.

- **maxVertexInputBindings** is the maximum number of vertex buffers that can be specified for providing vertex attributes to a graphics pipeline. These are described in the array of `VkVertexInputBindingDescription` structures that are provided at graphics pipeline creation time via the `pVertexBindingDescriptions` member of the `VkPipelineVertexInputStateCreateInfo` structure. The `binding` member of `VkVertexInputBindingDescription` must be less than this limit. See **Vertex Input Description**.

- **maxVertexInputAttributeOffset** is the maximum vertex input attribute offset that can be added to the vertex input binding stride. The `offset` member of the `VkVertexInputAttributeDescription` structure must be less than or equal to this limit. See **Vertex Input Description**.

- **maxVertexInputBindingStride** is the maximum vertex input binding stride that can be specified in a vertex input binding. The `stride` member of the `VkVertexInputBindingDescription` structure must be less than or equal to this limit. See **Vertex Input Description**.

- **maxVertexOutputComponents** is the maximum number of components of output variables which can be output by a vertex shader. See **Vertex Shaders**.

- **maxTessellationGenerationLevel** is the maximum tessellation generation level supported by the fixed-function tessellation primitive generator. See **Tessellation**.

- **maxTessellationPatchSize** is the maximum patch size, in vertices, of patches that can be processed by the tessellation control shader and tessellation primitive generator. The `patchControlPoints` member of the `VkPipelineTessellationStateCreateInfo` structure specified at pipeline creation time and the value provided in the **OutputVertices** execution mode of shader modules must be less than or equal to this limit. See **Tessellation**.

- **maxTessellationControlPerVertexInputComponents** is the maximum number of components of input variables which can be provided as per-vertex inputs to the tessellation control shader stage.

- **maxTessellationControlPerVertexOutputComponents** is the maximum number of components of per-vertex output variables which can be output from the tessellation control shader stage.

- **maxTessellationControlPerPatchOutputComponents** is the maximum number of components of per-patch output variables which can be output from the tessellation control shader stage.

- **maxTessellationControlTotalOutputComponents** is the maximum total number of components of per-vertex and per-patch output variables which can be output from the tessellation control shader stage.

- **maxTessellationEvaluationInputComponents** is the maximum number of components of input variables which can be provided as per-vertex inputs to the tessellation evaluation shader stage.

- **maxTessellationEvaluationOutputComponents** is the maximum number of components of per-vertex output variables which can be output from the tessellation evaluation shader stage.

- **maxGeometryShaderInvocations** is the maximum invocation count supported for instanced geometry shaders. The value provided in the **Invocations** execution mode of shader modules must be less than or equal to this limit. See **Geometry Shading**.

- **maxGeometryInputComponents** is the maximum number of components of input variables which can be provided as inputs to the geometry shader stage.
• maxGeometryOutputComponents is the maximum number of components of output variables which can be output from the geometry shader stage.

• maxGeometryOutputVertices is the maximum number of vertices which can be emitted by any geometry shader.

• maxGeometryTotalOutputComponents is the maximum total number of components of output variables, across all emitted vertices, which can be output from the geometry shader stage.

• maxFragmentInputComponents is the maximum number of components of input variables which can be provided as inputs to the fragment shader stage.

• maxFragmentOutputAttachments is the maximum number of output attachments which can be written to by the fragment shader stage.

• maxFragmentDualSrcAttachments is the maximum number of output attachments which can be written to by the fragment shader stage when blending is enabled and one of the dual source blend modes is in use. See Dual-Source Blending and dualSrcBlend.

• maxFragmentCombinedOutputResources is the total number of storage buffers, storage images, and output Location decorated color attachments (described in Fragment Output Interface) which can be used in the fragment shader stage.

• maxComputeSharedMemorySize is the maximum total storage size, in bytes, available for variables declared with the Workgroup storage class in shader modules (or with the shared storage qualifier in GLSL) in the compute shader stage. The amount of storage consumed by the variables declared with the Workgroup storage class is implementation-dependent. However, the amount of storage consumed may not exceed the largest block size that would be obtained if all active variables declared with Workgroup storage class were assigned offsets in an arbitrary order by successively taking the smallest valid offset according to the Standard Storage Buffer Layout rules. (This is equivalent to using the GLSL std430 layout rules.)

• maxComputeWorkGroupCount[3] is the maximum number of local workgroups that can be dispatched by a single dispatching command. These three values represent the maximum number of local workgroups for the X, Y, and Z dimensions, respectively. The workgroup count parameters to the dispatching commands must be less than or equal to the corresponding limit. See Dispatching Commands.

• maxComputeWorkGroupInvocations is the maximum total number of compute shader invocations in a single local workgroup. The product of the X, Y, and Z sizes, as specified by the LocalSize 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^{mipmapPrecisionBits} is the actual number of divisions.

- **maxDrawIndexedIndexValue** is the maximum index value that can be used for indexed draw calls when using 32-bit indices. This excludes the primitive restart index value of 0xFFFFFFFF. See fullDrawIndexUint32.

- **maxDrawIndirectCount** is the maximum draw count that is supported for indirect drawing calls. See multiDrawIndirect.

- **maxSamplerLodBias** is the maximum absolute sampler LOD bias. The sum of the mipLodBias member of the VkSamplerCreateInfo structure and the Bias operand of image sampling operations in shader modules (or 0 if no Bias operand is provided to an image sampling operation) are clamped to the range \([-\text{maxSamplerLodBias},+\text{maxSamplerLodBias}]\). See [samplers-mipLodBias].

- **maxSamplerAnisotropy** is the maximum degree of sampler anisotropy. The maximum degree of anisotropic filtering used for an image sampling operation is the minimum of the maxAnisotropy member of the VkSamplerCreateInfo structure and this limit. See [samplers-maxAnisotropy].

- **maxViewports** is the maximum number of active viewports. The viewportCount member of the VkPipelineViewportStateCreateInfo structure that is provided at pipeline creation must be less than or equal to this limit.

- **maxViewportDimensions[2]** are the maximum viewport dimensions in the X (width) and Y (height) dimensions, respectively. The maximum viewport dimensions must be greater than or equal to the largest image which can be created and used as a framebuffer attachment. See Controlling the Viewport.

- **viewportBoundsRange[2]** is the [minimum, maximum] range that the corners of a viewport must be contained in. This range must be at least \([-2 \times \text{size}, 2 \times \text{size} - 1]\), where \(\text{size} = \max(\maxViewportDimensions[0], \maxViewportDimensions[1])\). See Controlling the Viewport.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

- **viewportSubPixelBits** is the number of bits of subpixel precision for viewport bounds. The subpixel precision that floating-point viewport bounds are interpreted at is given by this limit.

- **minMemoryMapAlignment** is the minimum required alignment, in bytes, of host visible memory allocations within the host address space. When mapping a memory allocation with vkMapMemory, subtracting offset bytes from the returned pointer will always produce an integer multiple of this limit. See Host Access to Device Memory Objects. The value must be a power of two.
- minTexelBufferOffsetAlignment is the minimum required alignment, in bytes, for the offset member of the VkBufferViewCreateInfo structure for texel buffers. The value must be a power of two. If texelBufferAlignment is enabled, this limit is equivalent to the maximum of the uniformTexelBufferOffsetAlignmentBytes and storageTexelBufferOffsetAlignmentBytes members of VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT, but smaller alignment is optionally allowed by storageTexelBufferOffsetSingleTexelAlignment and uniformTexelBufferOffsetSingleTexelAlignment. If texelBufferAlignment is not enabled, VkBufferViewCreateInfo::offset must be a multiple of this value.

- minUniformBufferOffsetAlignment is the minimum required alignment, in bytes, for the offset member of the VkDescriptorBufferInfo structure for uniform buffers. When a descriptor of type VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC is updated, the offset must be an integer multiple of this limit. Similarly, dynamic offsets for uniform buffers must be multiples of this limit. 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** 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** 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 `vkCmdCopyBufferToImage2KHR`, `vkCmdCopyBufferToImage`, `vkCmdCopyImageToBuffer2KHR`, 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 `vkCmdCopyBufferToImage2KHR`, `vkCmdCopyBufferToImage`, `vkCmdCopyImageToBuffer2KHR`, 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.

For all bitmasks of `VkSampleCountFlagBits`, the sample count limits defined above represent the minimum supported sample counts for each image type. Individual images **may** support additional sample counts, which are queried using `vkGetPhysicalDeviceImageFormatProperties` as described in Supported Sample Counts.

Bits which **may** be set in the sample count limits returned by `VkPhysicalDeviceLimits`, as well as in
other queries and structures representing image sample counts, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkSampleCountFlagBits {
    VK_SAMPLE_COUNT_1_BIT = 0x00000001,
    VK_SAMPLE_COUNT_2_BIT = 0x00000002,
    VK_SAMPLE_COUNT_4_BIT = 0x00000004,
    VK_SAMPLE_COUNT_8_BIT = 0x00000008,
    VK_SAMPLE_COUNT_16_BIT = 0x00000010,
    VK_SAMPLE_COUNT_32_BIT = 0x00000020,
    VK_SAMPLE_COUNT_64_BIT = 0x00000040,
} VkSampleCountFlagBits;
```

- `VK_SAMPLE_COUNT_1_BIT` specifies an image with one sample per pixel.
- `VK_SAMPLE_COUNT_2_BIT` specifies an image with 2 samples per pixel.
- `VK_SAMPLE_COUNT_4_BIT` specifies an image with 4 samples per pixel.
- `VK_SAMPLE_COUNT_8_BIT` specifies an image with 8 samples per pixel.
- `VK_SAMPLE_COUNT_16_BIT` specifies an image with 16 samples per pixel.
- `VK_SAMPLE_COUNT_32_BIT` specifies an image with 32 samples per pixel.
- `VK_SAMPLE_COUNT_64_BIT` specifies an image with 64 samples per pixel.

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

`VkSampleCountFlags` is a bitmask type for setting a mask of zero or more `VkSampleCountFlagBits`.

The `VkPhysicalDeviceMultiviewProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceMultiviewProperties {
    VkStructureType sType;
    void* pNext;
    uint32_t maxMultiviewViewCount;
    uint32_t maxMultiviewInstanceIndex;
} VkPhysicalDeviceMultiviewProperties;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `maxMultiviewViewCount` is one greater than the maximum view index that can be used in a subpass.
- `maxMultiviewInstanceIndex` is the maximum valid value of instance index allowed to be generated by a drawing command recorded within a subpass of a multiview render pass 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 shaderRoundingModeRTEFloat64;
    VkBool32 shaderRoundingModeRTZFloat16;
    VkBool32 shaderRoundingModeRTZFloat32;
    VkBool32 shaderRoundingModeRTZFloat64;
} VkPhysicalDeviceFloatControlsProperties;
```

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

- `denormBehaviorIndependence` is a `VkShaderFloatControlsIndependence` value indicating whether, and how, denorm behavior can be set independently for different bit widths.
- `roundingModeIndependence` is a `VkShaderFloatControlsIndependence` value indicating whether, and how, rounding modes can be set independently for different bit widths.

- `shaderSignedZeroInfNanPreserveFloat16` is a boolean value indicating whether sign of a zero, Nans and ±∞ can be preserved in 16-bit floating-point computations. It also indicates whether the `SignedZeroInfNanPreserve` execution mode can be used for 16-bit floating-point types.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceFloatControlsProperties-sType-sType
  
  *sType* **must** be *VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FLOAT_CONTROLS_PROPERTIES*

Values which **may** be returned in the *denormBehaviorIndependence* and *roundingModeIndependence* fields of *VkPhysicalDeviceFloatControlsProperties* are:

```c
// Provided by VK_VERSION_1_2
typedef enum VkShaderFloatControlsIndependence {
    VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_32_BIT_ONLY = 0,
    VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_ALL = 1,
    VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_NONE = 2,
} VkShaderFloatControlsIndependence;
```

- **VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_32_BIT_ONLY** specifies that shader float controls for 32-bit floating point can be set independently; other bit widths must be set identically to each other.
- **VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_ALL** specifies that shader float controls for all bit widths can be set independently.
- **VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_NONE** specifies that shader float controls for all bit widths must be set identically.

The *VkPhysicalDeviceDiscardRectanglePropertiesEXT* structure is defined as:

```c
// Provided by VK_EXT_discard_rectangles
typedef struct VkPhysicalDeviceDiscardRectanglePropertiesEXT {
    VkStructureType sType;
    void* pNext;
    uint32_t maxDiscardRectangles;
} VkPhysicalDeviceDiscardRectanglePropertiesEXT;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **maxDiscardRectangles** is the maximum number of active discard rectangles that can be specified.

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

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceDiscardRectanglePropertiesEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DISCARD_RECTANGLE_PROPERTIES_EXT

The VkPhysicalDeviceSampleLocationsPropertiesEXT structure is defined as:

```c
typedef struct VkPhysicalDeviceSampleLocationsPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    VkSampleCountFlags sampleLocationSampleCounts;
    VkExtent2D maxSampleLocationGridSize;
    float sampleLocationCoordinateRange[2];
    uint32_t sampleLocationSubPixelBits;
    VkBool32 variableSampleLocations;
} VkPhysicalDeviceSampleLocationsPropertiesEXT;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- sampleLocationSampleCounts is a bitmask of VkSampleCountFlagBits indicating the sample counts supporting custom sample locations.
- maxSampleLocationGridSize is the maximum size of the pixel grid in which sample locations can vary that is supported for all sample counts in sampleLocationSampleCounts.
- sampleLocationCoordinateRange[2] is the range of supported sample location coordinates.
- sampleLocationSubPixelBits is the number of bits of subpixel precision for sample locations.
- variableSampleLocations specifies whether the sample locations used by all pipelines that will be bound to a command buffer during a subpass must match. If set to VK_TRUE, the implementation supports variable sample locations in a subpass. If set to VK_FALSE, then the sample locations must stay constant in each subpass.

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

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceSampleLocationsPropertiesEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLE_LOCATIONS_PROPERTIES_EXT

The VkPhysicalDeviceExternalMemoryHostPropertiesEXT structure is defined as:
typedef struct VkPhysicalDeviceExternalMemoryHostPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    VkDeviceSize minImportedHostPointerAlignment;
} VkPhysicalDeviceExternalMemoryHostPropertiesEXT;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **minImportedHostPointerAlignment** is the minimum required alignment, in bytes, for the base address and size of host pointers that can be imported to a Vulkan memory object. The value must be a power of two.

If the `VkPhysicalDeviceExternalMemoryHostPropertiesEXT` 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-VkPhysicalDeviceExternalMemoryHostPropertiesEXT-sType-sType
  
  **sType** must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_MEMORY_HOST_PROPERTIES_EXT`.

---

The `VkPhysicalDevicePointClippingProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDevicePointClippingProperties {
    VkStructureType sType;
    void* pNext;
    VkPointClippingBehavior pointClippingBehavior;
} VkPhysicalDevicePointClippingProperties;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **pointClippingBehavior** is a `VkPointClippingBehavior` value specifying the point clipping behavior supported by the implementation.

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

### Valid Usage (Implicit)

- VUID-VkPhysicalDevicePointClippingProperties-sType-sType
  
  **sType** must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_POINT_CLIPPING_PROPERTIES`.
The **VkPhysicalDeviceSubgroupProperties** structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceSubgroupProperties {
    VkStructureType sType;
    void* pNext;
    uint32_t subgroupSize;
    VkShaderStageFlags supportedStages;
    VkSubgroupFeatureFlags supportedOperations;
    VkBool32 quadOperationsInAllStages;
} VkPhysicalDeviceSubgroupProperties;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **subgroupSize** is the default number of invocations in each subgroup. **subgroupSize** is at least 1 if any of the physical device's queues support **VK_QUEUE_GRAPHICS_BIT** or **VK_QUEUE_COMPUTE_BIT**. **subgroupSize** is a power-of-two.
- **supportedStages** is a bitfield of **VkShaderStageFlagBits** describing the shader stages that **group operations** with **subgroup scope** are supported in. **supportedStages** will have the **VK_SHADER_STAGE_COMPUTE_BIT** bit set if any of the physical device's queues support **VK_QUEUE_COMPUTE_BIT**.
- **supportedOperations** is a bitmask of **VkSubgroupFeatureFlagBits** specifying the sets of **group operations** with **subgroup scope** supported on this device. **supportedOperations** will have the **VK_SUBGROUP_FEATURE_BASIC_BIT** bit set if any of the physical device's queues support **VK_QUEUE_GRAPHICS_BIT** or **VK_QUEUE_COMPUTE_BIT**.
- **quadOperationsInAllStages** is a boolean specifying whether **quad group operations** are available in all stages, or are restricted to fragment and compute stages.

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

If **supportedOperations** includes **VK_SUBGROUP_FEATURE_QUAD_BIT**, **subgroupSize** must be greater than or equal to 4.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceSubgroupProperties-sType-sType**
  
  **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_PROPERTIES**

Bits which can be set in **VkPhysicalDeviceSubgroupProperties::supportedOperations** and **VkPhysicalDeviceVulkan11Properties::subgroupSupportedOperations** to specify supported **group operations** with **subgroup scope** are:
typedef enum VkSubgroupFeatureFlagBits {
    VK_SUBGROUP_FEATURE_BASIC_BIT = 0x00000001,
    VK_SUBGROUP_FEATURE_VOTE_BIT = 0x00000002,
    VK_SUBGROUP_FEATURE_ARITHMETIC_BIT = 0x00000004,
    VK_SUBGROUP_FEATURE_BALLOT_BIT = 0x00000008,
    VK_SUBGROUP_FEATURE_SHUFFLE_BIT = 0x00000010,
    VK_SUBGROUP_FEATURE_SHUFFLE_RELATIVE_BIT = 0x00000020,
    VK_SUBGROUP_FEATURE_CLUSTERED_BIT = 0x00000040,
    VK_SUBGROUP_FEATURE_QUAD_BIT = 0x00000080,
} VkSubgroupFeatureFlagBits;

• VK_SUBGROUP_FEATURE_BASIC_BIT specifies the device will accept SPIR-V shader modules containing the GroupNonUniform capability.
• VK_SUBGROUP_FEATURE_VOTE_BIT specifies the device will accept SPIR-V shader modules containing the GroupNonUniformVote capability.
• VK_SUBGROUP_FEATURE_ARITHMETIC_BIT specifies the device will accept SPIR-V shader modules containing the GroupNonUniformArithmetic capability.
• VK_SUBGROUP_FEATURE_BALLOT_BIT specifies the device will accept SPIR-V shader modules containing the GroupNonUniformBallot capability.
• VK_SUBGROUP_FEATURE_SHUFFLE_BIT specifies the device will accept SPIR-V shader modules containing the GroupNonUniformShuffle capability.
• VK_SUBGROUP_FEATURE_SHUFFLE_RELATIVE_BIT specifies the device will accept SPIR-V shader modules containing the GroupNonUniformShuffleRelative capability.
• VK_SUBGROUP_FEATURE_CLUSTERED_BIT specifies the device will accept SPIR-V shader modules containing the GroupNonUniformClustered capability.
• VK_SUBGROUP_FEATURE_QUAD_BIT specifies the device will accept SPIR-V shader modules containing the GroupNonUniformQuad capability.

typedef VkFlags VkSubgroupFeatureFlags;

VkSubgroupFeatureFlags is a bitmask type for setting a mask of zero or more VkSubgroupFeatureFlagBits.

The VkPhysicalDeviceSubgroupSizeControlPropertiesEXT structure is defined as:

typedef struct VkPhysicalDeviceSubgroupSizeControlPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    uint32_t minSubgroupSize;
    uint32_t maxSubgroupSize;
    uint32_t maxComputeWorkgroupSubgroups;
};
VkShaderStageFlags requiredSubgroupSizeStages;
} VkPhysicalDeviceSubgroupSizeControlPropertiesEXT;

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `minSubgroupSize` is the minimum subgroup size supported by this device. `minSubgroupSize` is at least one if any of the physical device's queues support `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT`. `minSubgroupSize` is a power-of-two. `minSubgroupSize` is less than or equal to `maxSubgroupSize`. `minSubgroupSize` is less than or equal to `subgroupSize`.
- `maxSubgroupSize` is the maximum subgroup size supported by this device. `maxSubgroupSize` is at least one if any of the physical device's queues support `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT`. `maxSubgroupSize` is a power-of-two. `maxSubgroupSize` is greater than or equal to `minSubgroupSize`. `maxSubgroupSize` is greater than or equal to `subgroupSize`.
- `maxComputeWorkgroupSubgroups` is the maximum number of subgroups supported by the implementation within a workgroup.
- `requiredSubgroupSizeStages` is a bitfield of what shader stages support having a required subgroup size specified.

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

If `VkPhysicalDeviceSubgroupProperties::supportedOperations` includes `VK_SUBGROUP_FEATURE_QUAD_BIT`, `minSubgroupSize` must be greater than or equal to 4.

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceSubgroupSizeControlPropertiesEXT-sType-sType` `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_PROPERTIES_EXT`

The `VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT` structure is defined as:

```c
// Provided by VK_EXT_blend_operation_advanced
typedef struct VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    uint32_t advancedBlendMaxColorAttachments;
    VkBool32 advancedBlendIndependentBlend;
    VkBool32 advancedBlendNonPremultipliedSrcColor;
    VkBool32 advancedBlendNonPremultipliedDstColor;
    VkBool32 advancedBlendCorrelatedOverlap;
    VkBool32 advancedBlendAllOperations;
} VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT;

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

• **advancedBlendMaxColorAttachments** is one greater than the highest color attachment index that can be used in a subpass, for a pipeline that uses an advanced blend operation.

• **advancedBlendIndependentBlend** specifies whether advanced blend operations can vary per-attachment.

• **advancedBlendNonPremultipliedSrcColor** specifies whether the source color can be treated as non-premultiplied. If this is VK_FALSE, then **VkPipelineColorBlendAdvancedStateCreateInfo**::**srcPremultiplied** must be VK_TRUE.

• **advancedBlendNonPremultipliedDstColor** specifies whether the destination color can be treated as non-premultiplied. If this is VK_FALSE, then **VkPipelineColorBlendAdvancedStateCreateInfo**::**dstPremultiplied** must be VK_TRUE.

• **advancedBlendCorrelatedOverlap** specifies whether the overlap mode can be treated as correlated. If this is VK_FALSE, then **VkPipelineColorBlendAdvancedStateCreateInfo**::**blendOverlap** must be VK_BLEND_OVERLAP_UNCORRELATED_EXT.

• **advancedBlendAllOperations** specifies whether all advanced blend operation enums are supported. See the valid usage of **VkPipelineColorBlendAttachmentState**.

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

---

**Valid Usage (Implicit)**

- **VUID-VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT-sType-sType**
  - **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BLEND_OPERATION_ADVANCED_PROPERTIES_EXT**

The **VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT** structure is defined as:

```c
// Provided by VK_EXT_vertex_attribute_divisor
typedef struct VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    uint32_t maxVertexAttribDivisor;
} VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT;
```

• **sType** is the type of this structure.

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

• **maxVertexAttribDivisor** is the maximum value of the number of instances that will repeat the value of vertex attribute data when instanced rendering is enabled.

If the **VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT** structure is included in the **pNext** chain of the **VkPhysicalDeviceProperties2** structure passed to **vkGetPhysicalDeviceProperties2**, it is filled in with each corresponding implementation-dependent property.
Valid Usage (Implicit)

• VUID-VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT-sType-sType must be
  VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VERTEX_ATTRIBUTE_DIVISOR_PROPERTIES_EXT

The `VkPhysicalDeviceSamplerFilterMinmaxProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceSamplerFilterMinmaxProperties {
    VkStructureType sType;
    void* pNext;
    VkBool32 filterMinmaxSingleComponentFormats;
    VkBool32 filterMinmaxImageComponentMapping;
} VkPhysicalDeviceSamplerFilterMinmaxProperties;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `filterMinmaxSingleComponentFormats` is a boolean value indicating whether a minimum set of required formats support min/max filtering.
- `filterMinmaxImageComponentMapping` is a boolean value indicating whether the implementation supports non-identity component mapping of the image when doing min/max filtering.

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

If `filterMinmaxSingleComponentFormats` is `VK_TRUE`, the following formats must support the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT` feature with `VK_IMAGE_TILING_OPTIMAL`, if they support `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`:

- `VK_FORMAT_R8_UNORM`
- `VK_FORMAT_R8_SNORM`
- `VK_FORMAT_R16_UNORM`
- `VK_FORMAT_R16_SNORM`
- `VK_FORMAT_R16_SFLOAT`
- `VK_FORMAT_R32_SFLOAT`
- `VK_FORMAT_D16_UNORM`
- `VK_FORMAT_D16_UNORM_S8_UINT`
- `VK_FORMAT_X8_D24_UNORM_PACK32`
- `VK_FORMAT_D32_SFLOAT`
- `VK_FORMAT_D16_UNORM_S8_UINT`
• VK_FORMAT_D24_UNORM_S8_UINT
• VK_FORMAT_D32_SFLOAT_S8_UINT

If the format is a depth/stencil format, this bit only specifies that the depth aspect (not the stencil aspect) of an image of this format supports min/max filtering, and that min/max filtering of the depth aspect is supported when depth compare is disabled in the sampler.

If `filterMinmaxImageComponentMapping` is `VK_FALSE` the component mapping of the image view used with min/max filtering must have been created with the `r` component set to the identity swizzle. Only the `r` component of the sampled image value is defined and the other component values are undefined. If `filterMinmaxImageComponentMapping` is `VK_TRUE` this restriction does not apply and image component mapping works as normal.

Valid Usage (Implicit)

• VUID-VkPhysicalDeviceSamplerFilterMinmaxProperties-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_FILTER_MINMAX_PROPERTIES`

The `VkPhysicalDeviceProtectedMemoryProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceProtectedMemoryProperties {
    VkStructureType sType;
    void* pNext;
    VkBool32 protectedNoFault;
} VkPhysicalDeviceProtectedMemoryProperties;
```

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

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

Valid Usage (Implicit)

• VUID-VkPhysicalDeviceProtectedMemoryProperties-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_PROPERTIES`
The **VkPhysicalDeviceMaintenance3Properties** structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceMaintenance3Properties {
    VkStructureType sType;
    void* pNext;
    uint32_t maxPerSetDescriptors;
    VkDeviceSize maxMemoryAllocationSize;
} VkPhysicalDeviceMaintenance3Properties;
```

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

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

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

If the **VkPhysicalDeviceMaintenance3Properties** structure is included in the **pNext** chain of the **VkPhysicalDeviceProperties2** structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceMaintenance3Properties-sType-sType
  - sType must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_3_PROPERTIES**

The **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 maxPerStageDescriptorUpdateAfterBindStorageBuffers;
    uint32_t maxPerStageDescriptorUpdateAfterBindSampledImages;
} VkPhysicalDeviceDescriptorIndexingProperties;
```
uint32_t maxPerStageDescriptorUpdateAfterBindStorageImages;
uint32_t maxPerStageDescriptorUpdateAfterBindInputAttachments;
uint32_t maxPerStageUpdateAfterBindResources;
uint32_t maxPerStageUpdateAfterBindSamplers;
uint32_t maxDescriptorSetUpdateAfterBindUniformBuffers;
uint32_t maxDescriptorSetUpdateAfterBindUniformBuffersDynamic;
uint32_t maxDescriptorSetUpdateAfterBindStorageBuffers;
uint32_t maxDescriptorSetUpdateAfterBindStorageBuffersDynamic;
uint32_t maxDescriptorSetUpdateAfterBindSampledImages;
uint32_t maxDescriptorSetUpdateAfterBindStorageImages;
uint32_t maxDescriptorSetUpdateAfterBindInputAttachments;

VkPhysicalDeviceDescriptorIndexingProperties

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

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

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

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

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

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

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

• robustBufferAccessUpdateAfterBind is a boolean value indicating whether robustBufferAccess can be enabled in a device simultaneously with descriptorBindingUniformBufferUpdateAfterBind, descriptorBindingStorageBufferUpdateAfterBind, and/or
descriptorBindingStorageTexelBufferUpdateAfterBind. If this is VK_FALSE, then either robustBufferAccess must be disabled or all of these update-after-bind features must be disabled.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceDescriptorIndexingProperties-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_PROPERTIES

The VkPhysicalDeviceConservativeRasterizationPropertiesEXT structure is defined as:

```c
// Provided by VK_EXT_conservative_rasterization
typedef struct VkPhysicalDeviceConservativeRasterizationPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    float primitiveOverestimationSize;
    float maxExtraPrimitiveOverestimationSize;
    float extraPrimitiveOverestimationSizeGranularity;
    VkBool32 primitiveUnderestimation;
    VkBool32 conservativePointAndLineRasterization;
    VkBool32 degenerateTrianglesRasterized;
    VkBool32 degenerateLinesRasterized;
    VkBool32 fullyCoveredFragmentShaderInputVariable;
    VkBool32 conservativeRasterizationPostDepthCoverage;
} VkPhysicalDeviceConservativeRasterizationPropertiesEXT;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
• **primitiveOverestimationSize** is the size in pixels the generating primitive is increased at each of its edges during conservative rasterization overestimation mode. Even with a size of 0.0, conservative rasterization overestimation rules still apply and if any part of the pixel rectangle is covered by the generating primitive, fragments are generated for the entire pixel. However implementations may make the pixel coverage area even more conservative by increasing the size of the generating primitive.

• **maxExtraPrimitiveOverestimationSize** is the maximum size in pixels of extra overestimation the implementation supports in the pipeline state. A value of 0.0 means the implementation does not support any additional overestimation of the generating primitive during conservative rasterization. A value above 0.0 allows the application to further increase the size of the generating primitive during conservative rasterization overestimation.

• **extraPrimitiveOverestimationSizeGranularity** is the granularity of extra overestimation that can be specified in the pipeline state between 0.0 and **maxExtraPrimitiveOverestimationSize** inclusive. A value of 0.0 means the implementation can use the smallest representable non-zero value in the screen space pixel fixed-point grid.

• **primitiveUnderestimation** is **VK_TRUE** if the implementation supports the **VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT** conservative rasterization mode in addition to **VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT**. Otherwise the implementation only supports **VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT**.

• **conservativePointAndLineRasterization** is **VK_TRUE** if the implementation supports conservative rasterization of point and line primitives as well as triangle primitives. Otherwise the implementation only supports triangle primitives.

• **degenerateTrianglesRasterized** is **VK_FALSE** if the implementation culls primitives generated from triangles that become zero area after they are quantized to the fixed-point rasterization pixel grid. **degenerateTrianglesRasterized** is **VK_TRUE** if these primitives are not culled and the provoking vertex attributes and depth value are used for the fragments. The primitive area calculation is done on the primitive generated from the clipped triangle if applicable. Zero area primitives are backfacing and the application can enable backface culling if desired.

• **degenerateLinesRasterized** is **VK_FALSE** if the implementation culls lines that become zero length after they are quantized to the fixed-point rasterization pixel grid. **degenerateLinesRasterized** is **VK_TRUE** if zero length lines are not culled and the provoking vertex attributes and depth value are used for the fragments.

• **fullyCoveredFragmentShaderInputVariable** is **VK_TRUE** if the implementation supports the SPIR-V builtin fragment shader input variable **FullyCoveredEXT** specifying that conservative rasterization is enabled and the fragment area is fully covered by the generating primitive.

• **conservativeRasterizationPostDepthCoverage** is **VK_TRUE** if the implementation supports conservative rasterization with the **PostDepthCoverage** execution mode enabled. Otherwise the **PostDepthCoverage** execution mode must not be used when conservative rasterization is enabled.

If the **VkPhysicalDeviceConservativeRasterizationPropertiesEXT** 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 `VkPhysicalDeviceDepthStencilResolveProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceDepthStencilResolveProperties {
    VkStructureType sType;
    void* pNext;
    VkResolveModeFlags supportedDepthResolveModes;
    VkResolveModeFlags supportedStencilResolveModes;
    VkBool32 independentResolveNone;
    VkBool32 independentResolve;
} VkPhysicalDeviceDepthStencilResolveProperties;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `supportedDepthResolveModes` is a bitmask of `VkResolveModeFlagBits` indicating the set of supported depth resolve modes. 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.
The `VkPhysicalDevicePerformanceQueryPropertiesKHR` structure is defined as:

```c
// Provided by VK_KHR_performance_query
typedef struct VkPhysicalDevicePerformanceQueryPropertiesKHR {
    VkStructureType sType;
    void* pNext;
    VkBool32 allowCommandBufferQueryCopies;
} VkPhysicalDevicePerformanceQueryPropertiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `allowCommandBufferQueryCopies` is `VK_TRUE` if the performance query pools are allowed to be used with `vkCmdCopyQueryPoolResults`.

If the `VkPhysicalDevicePerformanceQueryPropertiesKHR` 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-VkPhysicalDevicePerformanceQueryPropertiesKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PERFORMANCE_QUERY_PROPERTIES_KHR`

The `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT` structure is defined as:

```c
// Provided by VK_EXT_texel_buffer_alignment
typedef struct VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    VkDeviceSize storageTexelBufferOffsetAlignmentBytes;
    VkBool32 storageTexelBufferOffsetSingleTexelAlignment;
    VkDeviceSize uniformTexelBufferOffsetAlignmentBytes;
    VkBool32 uniformTexelBufferOffsetSingleTexelAlignment;
} VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `storageTexelBufferOffsetAlignmentBytes` is a byte alignment that is sufficient for a storage texel buffer of any format. The value **must** be a power of two.
- `storageTexelBufferOffsetSingleTexelAlignment` indicates whether single texel alignment is sufficient for a storage texel buffer of any format. The value **must** be a power of two.
• **uniformTexelBufferOffsetAlignmentBytes** is a byte alignment that is sufficient for a uniform texel buffer of any format. The value **must** be a power of two.

• **uniformTexelBufferOffsetSingleTexelAlignment** indicates whether single texel alignment is sufficient for a uniform texel buffer of any format. The value **must** be a power of two.

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

If the single texel alignment property is **VK_FALSE**, then the buffer view’s offset **must** be aligned to the corresponding byte alignment value. If the single texel alignment property is **VK_TRUE**, then the buffer view’s offset **must** be aligned to the lesser of the corresponding byte alignment value or the size of a single texel, based on **VkBufferViewCreateInfo::format**. If the size of a single texel is a multiple of three bytes, then the size of a single component of the format is used instead.

These limits **must** not advertise a larger alignment than the required maximum minimum value of **VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment**, for any format that supports use as a texel buffer.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT-sType-sType**
  
  **sType** **must** be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_PROPERTIES_EXT**

The **VkPhysicalDeviceTimelineSemaphoreProperties** structure is defined as:

```c
// Provided by VK_VERSION_1_2
typedef struct VkPhysicalDeviceTimelineSemaphoreProperties {
    VkStructureType sType;
    void* pNext;
    uint64_t maxTimelineSemaphoreValueDifference;
} VkPhysicalDeviceTimelineSemaphoreProperties;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **maxTimelineSemaphoreValueDifference** indicates the maximum difference allowed by the implementation between the current value of a timeline semaphore and any pending signal or wait operations.

If the **VkPhysicalDeviceTimelineSemaphoreProperties** structure is included in the **pNext** chain of the **VkPhysicalDeviceProperties2** structure passed to **vkGetPhysicalDeviceProperties2**, it is filled in with each corresponding implementation-dependent property.
The `VkPhysicalDeviceLineRasterizationPropertiesEXT` structure is defined as:

```c
// Provided by VK_EXT_line_rasterization
typedef struct VkPhysicalDeviceLineRasterizationPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    uint32_t lineSubPixelPrecisionBits;
} VkPhysicalDeviceLineRasterizationPropertiesEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `lineSubPixelPrecisionBits` is the number of bits of subpixel precision in framebuffer coordinates \( x_f \) and \( y_f \) when rasterizing line segments.

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

The `VkPhysicalDeviceRobustness2PropertiesEXT` structure is defined as:

```c
// Provided by VK_EXT_robustness2
typedef struct VkPhysicalDeviceRobustness2PropertiesEXT {
    VkStructureType sType;
    void* pNext;
    VkDeviceSize robustStorageBufferAccessSizeAlignment;
    VkDeviceSize robustUniformBufferAccessSizeAlignment;
} VkPhysicalDeviceRobustness2PropertiesEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `robustStorageBufferAccessSizeAlignment` is the number of bytes that the range of a storage buffer descriptor is rounded up to when used for bounds-checking when `robustBufferAccess2` is enabled. This value **must** be either 1 or 4.
robustUniformBufferAccessSizeAlignment is the number of bytes that the range of a uniform buffer descriptor is rounded up to when used for bounds-checking when robustBufferAccess2 is enabled. This value must be a power of two in the range [1, 256].

If the VkPhysicalDeviceRobustness2PropertiesEXT 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-VkPhysicalDeviceRobustness2PropertiesEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ROBUSTNESS_2_PROPERTIES_EXT

The VkPhysicalDeviceFragmentShadingRatePropertiesKHR structure is defined as:

```c
// Provided by VK_KHR_fragment_shading_rate
typedef struct VkPhysicalDeviceFragmentShadingRatePropertiesKHR {
    VkStructureType sType;
    void* pNext;
    VkExtent2D minFragmentShadingRateAttachmentTexelSize;
    VkExtent2D maxFragmentShadingRateAttachmentTexelSize;
    uint32_t maxFragmentShadingRateAttachmentTexelSizeAspectRatio;
    VkBool32 primitiveFragmentShadingRateWithMultipleViewports;
    VkBool32 layeredShadingRateAttachments;
    VkBool32 fragmentShadingRateNonTrivialCombinerOps;
    VkExtent2D maxFragmentSize;
    uint32_t maxFragmentSizeAspectRatio;
    uint32_t maxFragmentShadingRateCoverageSamples;
    VkSampleCountFlagBits maxFragmentShadingRateRasterizationSamples;
    VkBool32 fragmentShadingRateWithShaderDepthStencilWrites;
    VkBool32 fragmentShadingRateWithSampleMask;
    VkBool32 fragmentShadingRateWithConservativeRasterization;
    VkBool32 fragmentShadingRateWithFragmentShaderInterlock;
    VkBool32 fragmentShadingRateWithCustomSampleLocations;
    VkBool32 fragmentShadingRateStrictMultiplyCombiner;
} VkPhysicalDeviceFragmentShadingRatePropertiesKHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- minFragmentShadingRateAttachmentTexelSize indicates minimum supported width and height of the portion of the framebuffer corresponding to each texel in a fragment shading rate attachment. Each value must be less than or equal to the values in maxFragmentShadingRateAttachmentTexelSize. Each value must be a power-of-two. It must be (0,0) if the attachmentFragmentShadingRate feature is not supported.
- maxFragmentShadingRateAttachmentTexelSize indicates maximum supported width and height of
the portion of the framebuffer corresponding to each texel in a fragment shading rate attachment. Each value must be greater than or equal to the values in minFragmentShadingRateAttachmentTexelSize. Each value must be a power-of-two. It must be (0,0) if the attachmentFragmentShadingRate feature is not supported.

• maxFragmentShadingRateAttachmentTexelSizeAspectRatio indicates the maximum ratio between the width and height of the portion of the framebuffer corresponding to each texel in a fragment shading rate attachment. maxFragmentShadingRateAttachmentTexelSizeAspectRatio must be a power-of-two value, and must be less than or equal to max(maxFragmentShadingRateAttachmentTexelSize.width / minFragmentShadingRateAttachmentTexelSize.height, maxFragmentShadingRateAttachmentTexelSize.height / minFragmentShadingRateAttachmentTexelSize.width). It must be 0 if the attachmentFragmentShadingRate feature is not supported.

• primitiveFragmentShadingRateWithMultipleViewports specifies whether the primitive fragment shading rate can be used when multiple viewports are used. If this value is VK_FALSE, only a single viewport must be used, and applications must not write to the ViewportIndex built-in when setting PrimitiveShadingRateKHR. It must be VK_FALSE if the shaderOutputViewportIndex feature, or the geometryShader feature is not supported, or if the primitiveFragmentShadingRate feature is not supported.

• layeredShadingRateAttachments specifies whether a shading rate attachment image view can be created with multiple layers. If this value is VK_FALSE, when creating an image view with a usage that includes VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR, layerCount must be 1. It must be VK_FALSE if the multiview feature, the shaderOutputViewportIndex feature, or the geometryShader feature is not supported, or if the attachmentFragmentShadingRate feature is not supported.

• fragmentShadingRateNonTrivialCombinerOps specifies whether VkFragmentShadingRateCombinerOpKHR enums other than VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR or VK_FRAGMENT_SHADING_RATE_COMBINER_OP_REPLACE_KHR can be used. It must be VK_FALSE unless either the primitiveFragmentShadingRate or attachmentFragmentShadingRate feature is supported.

• maxFragmentSize indicates the maximum supported width and height of a fragment. Its width and height members must both be power-of-two values. This limit is purely informational, and is not validated.

• maxFragmentSizeAspectRatio indicates the maximum ratio between the width and height of a fragment. maxFragmentSizeAspectRatio must be a power-of-two value, and must be less than or equal to the maximum of the width and height members of maxFragmentSize. This limit is purely informational, and is not validated.

• maxFragmentShadingRateCoverageSamples specifies the maximum number of coverage samples supported in a single fragment. maxFragmentShadingRateCoverageSamples must be less than or equal to the product of the width and height members of maxFragmentSize, and the sample count reported by maxFragmentShadingRateRasterizationSamples. maxFragmentShadingRateCoverageSamples must be less than or equal to maxSampleMaskWords × 32 if fragmentShadingRateWithShaderSampleMask is supported. This limit is purely informational, and is not validated.

• maxFragmentShadingRateRasterizationSamples is a VkSampleCountFlagBits value specifying the
maximum sample rate supported when a fragment covers multiple pixels. This limit is purely informational, and is not validated.

- `fragmentShadingRateWithShaderDepthStencilWrites` specifies whether the implementation supports writing `FragDepth` or `FragStencilRefEXT` from a fragment shader for multi-pixel fragments. If this value is `VK_FALSE`, writing to those built-ins will clamp the fragment shading rate to (1,1).

- `fragmentShadingRateWithSampleMask` specifies whether the implementation supports setting valid bits of `VkPipelineMultisampleStateCreateInfo::pSampleMask` to 0 for multi-pixel fragments. If this value is `VK_FALSE`, zeroing valid bits in the sample mask will clamp the fragment shading rate to (1,1).

- `fragmentShadingRateWithShaderSampleMask` specifies whether the implementation supports reading or writing `SampleMask` for multi-pixel fragments. If this value is `VK_FALSE`, using that built-in will clamp the fragment shading rate to (1,1).

- `fragmentShadingRateWithConservativeRasterization` specifies whether conservative rasterization is supported for multi-pixel fragments. It must be `VK_FALSE` if `VK_EXT_conservative_rasterization` is not supported. If this value is `VK_FALSE`, using conservative rasterization will clamp the fragment shading rate to (1,1).

- `fragmentShadingRateWithFragmentShaderInterlock` specifies whether fragment shader interlock is supported for multi-pixel fragments. It must be `VK_FALSE` if `VK_EXT_fragment_shader_interlock` is not supported. If this value is `VK_FALSE`, using fragment shader interlock will clamp the fragment shading rate to (1,1).

- `fragmentShadingRateWithCustomSampleLocations` specifies whether custom sample locations are supported for multi-pixel fragments. It must be `VK_FALSE` if `VK_EXT_sample_locations` is not supported. If this value is `VK_FALSE`, using custom sample locations will clamp the fragment shading rate to (1,1).

- `fragmentShadingRateStrictMultiplyCombiner` specifies whether `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MUL_KHR` accurately performs a multiplication or not. Implementations where this value is `VK_FALSE` will instead combine rates with an addition. If `fragmentShadingRateNonTrivialCombinerOps` is `VK_FALSE`, implementations must report this as `VK_FALSE`. If `fragmentShadingRateNonTrivialCombinerOps` is `VK_TRUE`, implementations should report this as `VK_TRUE`.

**Note**

Multiplication of the combiner rates using the fragment width/height in linear space is equivalent to an addition of those values in log2 space. Some implementations inadvertently implemented an addition in linear space due to unclear requirements originating outside of this specification. This resulted in `fragmentShadingRateStrictMultiplyCombiner` being added. Fortunately, this only affects situations where a rate of 1 in either dimension is combined with another rate of 1. All other combinations result in the exact same result as if multiplication was performed in linear space due to the clamping logic, and the fact that both the sum and product of 2 and 2 are equal. In many cases, this limit will not affect the correct operation of applications.

If the `VkPhysicalDeviceFragmentShadingRatePropertiesKHR` 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 are related to fragment shading rates.

**Valid Usage (Implicit)**

- `VUID-VkPhysicalDeviceFragmentShadingRatePropertiesKHR-sType-sType`
  
  `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADING_RATE_PROPERTIES_KHR`

The `VkPhysicalDeviceCustomBorderColorPropertiesEXT` structure is defined as:

```c
// Provided by VK_EXT_custom_border_color
typedef struct VkPhysicalDeviceCustomBorderColorPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    uint32_t maxCustomBorderColorSamplers;
} VkPhysicalDeviceCustomBorderColorPropertiesEXT;
```

- `maxCustomBorderColorSamplers` indicates the maximum number of samplers with custom border colors which can simultaneously exist on a device.

If the `VkPhysicalDeviceCustomBorderColorPropertiesEXT` 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-VkPhysicalDeviceCustomBorderColorPropertiesEXT-sType-sType`
  
  `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CUSTOM_BORDER_COLOR_PROPERTIES_EXT`

### 33.1. Limit Requirements

The following table specifies the **required** minimum/maximum for all Vulkan graphics implementations. Where a limit corresponds to a fine-grained device feature which is **optional**, the feature name is listed with two **required** limits, one when the feature is supported and one when it is not supported. If an implementation supports a feature, the limits reported are the same whether or not the feature is enabled.

**Table 46. Required Limit Types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Limit</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxImageDimension1D</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxImageDimension2D</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxImageDimension3D</code></td>
<td>-</td>
</tr>
<tr>
<td>Type</td>
<td>Limit</td>
<td>Feature</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxImageDimensionCube</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxImageArrayLayers</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTexelBufferElements</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxUniformBufferRange</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxStorageBufferRange</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPushConstantsSize</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxMemoryAllocationCount</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxSamplerAllocationCount</td>
<td></td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>bufferImageGranularity</td>
<td></td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>sparseAddressSpaceSize</td>
<td>sparseBinding</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxBoundDescriptorSets</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorSamplers</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorUniformBuffers</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorStorageBuffers</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorSampledImages</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorStorageImages</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorInputAttachments</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageResources</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetSamplers</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetUniformBuffers</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetUniformBuffersDynamic</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetStorageBuffers</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetStorageBuffersDynamic</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetSampledImages</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetStorageImages</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetInputAttachments</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxVertexInputAttributes</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxVertexInputBindings</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxVertexInputAttributeOffset</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxVertexInputBindingStride</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxVertexOutputComponents</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>Type</td>
<td>Limit</td>
<td>Feature</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------------------------------------------------------</td>
<td>------------------------------</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>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>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>Type</td>
<td>Limit</td>
<td>Feature</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------------</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>
<tr>
<td>uint32_t</td>
<td>maxCullDistances</td>
<td>shaderCullDistance</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxCombinedClipAndCullDistances</td>
<td>shaderCullDistance</td>
</tr>
<tr>
<td>uint32_t</td>
<td>discreteQueuePriorities</td>
<td></td>
</tr>
<tr>
<td>2 × float</td>
<td>pointSizeRange</td>
<td>largePoints</td>
</tr>
<tr>
<td>Type</td>
<td>Limit</td>
<td>Feature</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>2 × float</td>
<td>lineWidthRange</td>
<td>wideLines</td>
</tr>
<tr>
<td>float</td>
<td>pointSizeGranularity</td>
<td>largePoints</td>
</tr>
<tr>
<td>float</td>
<td>lineWidthGranularity</td>
<td>wideLines</td>
</tr>
<tr>
<td>VkBool32</td>
<td>strictLines</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>standardSampleLocations</td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>optimalBufferCopyOffsetAlignment</td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>optimalBufferCopyRowPitchAlignment</td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>nonCoherentAtomSize</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDiscardRectangles</td>
<td>VK_EXT_discard_rectangles</td>
</tr>
<tr>
<td>VkBool32</td>
<td>filterMinmaxSingleComponentFormats</td>
<td>samplerFilterMinmax</td>
</tr>
<tr>
<td>VkBool32</td>
<td>filterMinmaxImageComponentMapping</td>
<td>samplerFilterMinmax</td>
</tr>
<tr>
<td>float</td>
<td>primitiveOverestimationSize</td>
<td>VK_EXT_conservative_rasterization</td>
</tr>
<tr>
<td>VkBool32</td>
<td>maxExtraPrimitiveOverestimationSize</td>
<td>VK_EXT_conservative_rasterization</td>
</tr>
<tr>
<td>float</td>
<td>extraPrimitiveOverestimationSizeGranularity</td>
<td>VK_EXT_conservative_rasterization</td>
</tr>
<tr>
<td>VkBool32</td>
<td>degenerateTriangleRasterized</td>
<td>VK_EXT_conservative_rasterization</td>
</tr>
<tr>
<td>float</td>
<td>degenerateLinesRasterized</td>
<td>VK_EXT_conservative_rasterization</td>
</tr>
<tr>
<td>VkBool32</td>
<td>fullyCoveredFragmentShaderInputVariable</td>
<td>VK_EXT_conservative_rasterization</td>
</tr>
<tr>
<td>VkBool32</td>
<td>conservativeRasterizationPostDepthCoverage</td>
<td>VK_EXT_conservative_rasterization</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxUpdateAfterBindDescriptorsInAllPools</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>VkBool32</td>
<td>shaderUniformBufferArrayNonUniformIndexingNative</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>shaderSampledImageArrayNonUniformIndexingNative</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>shaderStorageBufferArrayNonUniformIndexingNative</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>shaderStorageImageArrayNonUniformIndexingNative</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>shaderInputAttachmentArrayNonUniformIndexingNative</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorUpdateAfterBindSamplers</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorUpdateAfterBindUniformBuffers</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorUpdateAfterBindStorageBuffers</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>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>maxPerStageDescriptorUpdateAfterBindSampledImages</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxPerStageDescriptorUpdateAfterBindStorageImages</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxPerStageDescriptorUpdateAfterBindInputAttachments</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxPerStageUpdateAfterBindResources</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxDescriptorSetUpdateAfterBindSampledImages</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxDescriptorSetUpdateAfterBindUniformBuffers</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxDescriptorSetUpdateAfterBindUniformBuffersDynamic</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxDescriptorSetUpdateAfterBindStorageBuffers</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxDescriptorSetUpdateAfterBindStorageBuffersDynamic</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxDescriptorSetUpdateAfterBindSampledImages</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxDescriptorSetUpdateAfterBindStorageImages</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxVertexAttribDivisor</td>
<td>VK_EXT_vertex_attribute_divisor</td>
</tr>
<tr>
<td><code>uint64_t</code></td>
<td>maxTimelineSemaphoreValueDifference</td>
<td>timelineSemaphore</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>lineSubPixelPrecisionBits</td>
<td>VK_EXT_line_rasterization</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxCustomBorderColorSamplers</td>
<td>VK_EXT_custom_border_color</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>robustStorageBufferAccessSizeAlignment</td>
<td>VK_EXT_robustness2</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>robustUniformBufferAccessSizeAlignment</td>
<td>VK_EXT_robustness2</td>
</tr>
<tr>
<td>$2 \times \text{uint32}_t$</td>
<td>minFragmentShadingRateAttachmentTexelSize</td>
<td>attachmentFragmentShadingRate</td>
</tr>
<tr>
<td>$2 \times \text{uint32}_t$</td>
<td>maxFragmentShadingRateAttachmentTexelSize</td>
<td>attachmentFragmentShadingRate</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxFragmentShadingRateAttachmentTexelSizeAspectRatio</td>
<td>attachmentFragmentShadingRate</td>
</tr>
<tr>
<td>VkBool32</td>
<td>primitiveFragmentShadingRateWithMultipleViewports</td>
<td>primitiveFragmentShadingRate</td>
</tr>
<tr>
<td>VkBool32</td>
<td>layeredShadingRateAttachments</td>
<td>attachmentFragmentShadingRate</td>
</tr>
<tr>
<td>VkBool32</td>
<td>fragmentShadingRateNonTrivialCombinerOps</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td>$2 \times \text{uint32}_t$</td>
<td>maxFragmentSize</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxFragmentSizeAspectRatio</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxFragmentSizeCoverageSamples</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td>VkSampleCountFlagBits</td>
<td>maxFragmentShadingRateRasterizationSamples</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td>Type</td>
<td>Limit</td>
<td>Feature</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>VkBool32</td>
<td>fragmentShadingRateWithShaderDepthStencilWrites</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td>VkBool32</td>
<td>fragmentShadingRateWithSampleMask</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td>VkBool32</td>
<td>fragmentShadingRateWithShaderSampleMask</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td>VkBool32</td>
<td>fragmentShadingRateWithConservativeRasterization</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td>VkBool32</td>
<td>fragmentShadingRateWithFragmentShaderInterlock</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td>VkBool32</td>
<td>fragmentShadingRateStrictMultiplyCombiner</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td>VkBool32</td>
<td>deviceNoDynamicHostAllocations</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>deviceDestroyFreesMemory</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>commandPoolMultipleCommandBuffersRecording</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>commandPoolResetCommandBuffer</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>commandBufferSimultaneousUse</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>secondaryCommandBufferNullOrImagelessFramebuffer</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>recycleDescriptorSetMemory</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>recyclePipelineMemory</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxRenderPassSubpasses</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxRenderPassDependencies</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxSubpassInputAttachments</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxSubpassPreserveAttachments</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxFramebufferAttachments</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetLayoutBindings</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxQueryFaultCount</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxCallbackFaultCount</td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>maxCommandPoolCommandBuffers</td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>maxCommandBufferSize</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 47. Required Limits**

<table>
<thead>
<tr>
<th>Limit</th>
<th>Unsupported Limit</th>
<th>Supported Limit</th>
<th>Limit Type¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxImageDimension1D</td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td>maxImageDimension2D</td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td>maxImageDimension3D</td>
<td>-</td>
<td>256</td>
<td>min</td>
</tr>
<tr>
<td>maxImageDimensionCube</td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td>Limit</td>
<td>Unsupported Limit</td>
<td>Supported Limit</td>
<td>Limit Type</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-------------------</td>
<td>-----------------</td>
<td>------------</td>
</tr>
<tr>
<td><code>maxImageArrayLayers</code></td>
<td>-</td>
<td>256</td>
<td>min</td>
</tr>
<tr>
<td><code>maxTexelBufferElements</code></td>
<td>-</td>
<td>65536</td>
<td>min</td>
</tr>
<tr>
<td><code>maxUniformBufferRange</code></td>
<td>-</td>
<td>16384</td>
<td>min</td>
</tr>
<tr>
<td><code>maxStorageBufferRange</code></td>
<td>-</td>
<td>$2^{27}$</td>
<td>min</td>
</tr>
<tr>
<td><code>maxPushConstantsSize</code></td>
<td>-</td>
<td>128</td>
<td>min</td>
</tr>
<tr>
<td><code>maxMemoryAllocationCount</code></td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td><code>maxSamplerAllocationCount</code></td>
<td>-</td>
<td>4000</td>
<td>min</td>
</tr>
<tr>
<td><code>bufferImageGranularity</code></td>
<td>-</td>
<td>131072</td>
<td>max</td>
</tr>
<tr>
<td><code>sparseAddressSpaceSize</code></td>
<td>0</td>
<td>$2^{31}$</td>
<td>min</td>
</tr>
<tr>
<td><code>maxBoundDescriptorSets</code></td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td><code>maxPerStageDescriptorSamplers</code></td>
<td>-</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td><code>maxPerStageDescriptorUniformBuffers</code></td>
<td>-</td>
<td>12</td>
<td>min</td>
</tr>
<tr>
<td><code>maxPerStageDescriptorStorageBuffers</code></td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td><code>maxPerStageDescriptorSampledImages</code></td>
<td>-</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td><code>maxPerStageDescriptorStorageImages</code></td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td><code>maxPerStageDescriptorInputAttachments</code></td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td><code>maxPerStageResources</code></td>
<td>-</td>
<td>$128^2$</td>
<td>min</td>
</tr>
<tr>
<td><code>maxDescriptorSetSamplers</code></td>
<td>-</td>
<td>$96^8$</td>
<td>min, $n \times$ PerStage</td>
</tr>
<tr>
<td><code>maxDescriptorSetUniformBuffers</code></td>
<td>-</td>
<td>$72^8$</td>
<td>min, $n \times$ PerStage</td>
</tr>
<tr>
<td><code>maxDescriptorSetUniformBuffersDynamic</code></td>
<td>-</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td><code>maxDescriptorSetStorageBuffers</code></td>
<td>-</td>
<td>$24^8$</td>
<td>min, $n \times$ PerStage</td>
</tr>
<tr>
<td><code>maxDescriptorSetStorageBuffersDynamic</code></td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td><code>maxDescriptorSetSampledImages</code></td>
<td>-</td>
<td>$96^8$</td>
<td>min, $n \times$ PerStage</td>
</tr>
<tr>
<td><code>maxDescriptorSetStorageImages</code></td>
<td>-</td>
<td>$24^8$</td>
<td>min, $n \times$ PerStage</td>
</tr>
<tr>
<td><code>maxDescriptorSetInputAttachments</code></td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td><code>maxVertexInputAttributes</code></td>
<td>-</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td><code>maxVertexInputBindings</code></td>
<td>-</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td><code>maxVertexInputAttributeOffset</code></td>
<td>-</td>
<td>2047</td>
<td>min</td>
</tr>
<tr>
<td><code>maxVertexInputBindingStride</code></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>maxGeometryTotalOutputVertices</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>maxCompute WorkGroup Size</td>
<td>-</td>
<td>(128,128,64)</td>
<td>min</td>
</tr>
<tr>
<td>subPixelPrecisionBits</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>subTexelPrecisionBits</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>mipmapPrecisionBits</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxDrawIndexedIndexValue</td>
<td>$2^{24}$-1</td>
<td>$2^{32}$-1</td>
<td>min</td>
</tr>
<tr>
<td>maxDrawIndirectCount</td>
<td>1</td>
<td>$2^{16}$-1</td>
<td>min</td>
</tr>
<tr>
<td>maxSamplerLodBias</td>
<td>-</td>
<td>2</td>
<td>min</td>
</tr>
<tr>
<td>maxSamplerAnisotropy</td>
<td>1</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxViewports</td>
<td>1</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxViewportDimensions</td>
<td>-</td>
<td>(4096,4096)</td>
<td>min</td>
</tr>
<tr>
<td>viewportBoundsRange</td>
<td>-</td>
<td>(-8192,8191)</td>
<td>(max,min)</td>
</tr>
<tr>
<td>Limit</td>
<td>Unsupported Limit</td>
<td>Supported Limit</td>
<td>Limit 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>-0.5³</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>4³</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>min</td>
</tr>
<tr>
<td></td>
<td></td>
<td></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>min</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td>framebufferStencilSampleCounts</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>framebufferNoAttachmentsSampleCounts</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>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>min</td>
</tr>
<tr>
<td></td>
<td></td>
<td></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><code>sampledImageIntegerSampleCounts</code></td>
<td>-</td>
<td>VK_SAMPLE_COUNT_1_BIT</td>
<td>min</td>
</tr>
<tr>
<td><code>sampledImageDepthSampleCounts</code></td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>min</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td><code>sampledImageStencilSampleCounts</code></td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>min</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td><code>storageImageSampleCounts</code></td>
<td>VK_SAMPLE_COUNT_1_BIT</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>min</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td><code>maxSampleMaskWords</code></td>
<td>-</td>
<td>1</td>
<td>min</td>
</tr>
<tr>
<td><code>timestampComputeAndGraphics</code></td>
<td>-</td>
<td>-</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td><code>timestampPeriod</code></td>
<td>-</td>
<td>-</td>
<td>duration</td>
</tr>
<tr>
<td><code>maxClipDistances</code></td>
<td>0</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td><code>maxCullDistances</code></td>
<td>0</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td><code>maxCombinedClipAndCullDistances</code></td>
<td>0</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td><code>discreteQueuePriorities</code></td>
<td>-</td>
<td>2</td>
<td>min</td>
</tr>
<tr>
<td><code>pointSizeRange</code></td>
<td>(1.0,1.0)</td>
<td>(1.0,64.0 - ULP)</td>
<td>(max,min)</td>
</tr>
<tr>
<td><code>lineWidthRange</code></td>
<td>(1.0,1.0)</td>
<td>(1.0,8.0 - ULP)</td>
<td>(max,min)</td>
</tr>
<tr>
<td><code>pointSizeGranularity</code></td>
<td>0.0</td>
<td>1.0 ^6</td>
<td>max, fixed point increment</td>
</tr>
<tr>
<td><code>lineWidthGranularity</code></td>
<td>0.0</td>
<td>1.0 ^7</td>
<td>max, fixed point increment</td>
</tr>
<tr>
<td><code>strictLines</code></td>
<td>-</td>
<td>-</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td><code>standardSampleLocations</code></td>
<td>-</td>
<td>-</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td><code>optimalBufferCopyOffsetAlignment</code></td>
<td>-</td>
<td>-</td>
<td>recommendation</td>
</tr>
<tr>
<td><code>optimalBufferCopyRowPitchAlignment</code></td>
<td>-</td>
<td>-</td>
<td>recommendation</td>
</tr>
<tr>
<td>Limit</td>
<td>Unsupported Limit</td>
<td>Supported Limit</td>
<td>Limit Type¹</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-------------------</td>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>nonCoherentAtomSize</td>
<td>-</td>
<td>256</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>maxDiscardRectangles</td>
<td>0</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>sampleLocationSampleCounts</td>
<td>-</td>
<td>VK_SAMPLE_COUNT_4_BIT</td>
<td>min</td>
</tr>
<tr>
<td>maxSampleLocationGridSize</td>
<td>-</td>
<td>(1,1)</td>
<td>min</td>
</tr>
<tr>
<td>sampleLocationCoordinateRange</td>
<td>-</td>
<td>(0.0, 0.9375)</td>
<td>(max,min)</td>
</tr>
<tr>
<td>sampleLocationSubPixelBits</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>variableSampleLocations</td>
<td>-</td>
<td>false</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>minImportedHostPointerAlignment</td>
<td>-</td>
<td>65536</td>
<td>max</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>advancedBlendMaxColorAttachments</td>
<td>-</td>
<td>1</td>
<td>min</td>
</tr>
<tr>
<td>advancedBlendIndependentBlend</td>
<td>-</td>
<td>false</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>advancedBlendNonPremultipliedSrcColor</td>
<td>-</td>
<td>false</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>advancedBlendNonPremultipliedDstColor</td>
<td>-</td>
<td>false</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>advancedBlendCorrelatedOverlap</td>
<td>-</td>
<td>false</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>advancedBlendAllOperations</td>
<td>-</td>
<td>false</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>primitiveOverestimationSize</td>
<td>-</td>
<td>0.0</td>
<td>min</td>
</tr>
<tr>
<td>maxExtraPrimitiveOverestimationSize</td>
<td>-</td>
<td>0.0</td>
<td>min</td>
</tr>
<tr>
<td>extraPrimitiveOverestimationSizeGranularity</td>
<td>-</td>
<td>0.0</td>
<td>min</td>
</tr>
<tr>
<td>primitiveUnderestimation</td>
<td>-</td>
<td>false</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>conservativePointAndLineRasterization</td>
<td>-</td>
<td>false</td>
<td>implementation-dependent</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>degenerateTrianglesRasterized</td>
<td>-</td>
<td>false</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>degenerateLinesRasterized</td>
<td>-</td>
<td>false</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>fullyCoveredFragmentShaderInputVariable</td>
<td>-</td>
<td>false</td>
<td>implementation-dependent</td>
</tr>
<tr>
<td>conservativeRasterizationPostDepthCoverage</td>
<td>-</td>
<td>false</td>
<td>implementation-dependent</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>$72^89$</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>maxDescriptorSetUpdateAfterBindStorageBuffersDynamic</td>
<td>$0^9$</td>
<td>$4^9$</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>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>maxVertexAttribDivisor</td>
<td>-</td>
<td>2¹⁶-1</td>
<td>min</td>
</tr>
<tr>
<td>maxTimelineSemaphoreValueDifference</td>
<td>-</td>
<td>2³¹-1</td>
<td>min</td>
</tr>
<tr>
<td>lineSubPixelPrecisionBits</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxCustomBorderColorSamplers</td>
<td>-</td>
<td>32</td>
<td>min</td>
</tr>
<tr>
<td>robustStorageBufferAccessSizeAlignment</td>
<td>-</td>
<td>4</td>
<td>max</td>
</tr>
<tr>
<td>robustUniformBufferAccessSizeAlignment</td>
<td>-</td>
<td>256</td>
<td>max</td>
</tr>
<tr>
<td>minFragmentShadingRateAttachmentTexelSize</td>
<td>(0,0)</td>
<td>(32,32)</td>
<td>max</td>
</tr>
<tr>
<td>maxFragmentShadingRateAttachmentTexelSize</td>
<td>(0,0)</td>
<td>(8,8)</td>
<td>min</td>
</tr>
<tr>
<td>maxFragmentShadingRateAttachmentTexelSizeAspectRatio</td>
<td>0</td>
<td>1</td>
<td>min</td>
</tr>
</tbody>
</table>
| primitiveFragmentShadingRateWithMultipleViewports| false             | false           | implementatio
| layeredShadingRateAttachments                    | false             | false           | implementatio
| fragmentShadingRateNonTrivialCombinerOps         | -                 | false           | implementatio
| maxFragmentSize                                  | -                 | (2,2)           | min        |
| maxFragmentSizeAspectRatio                       | -                 | 2               | min        |
| maxFragmentShadingRateCoverageSamples            | -                 | 16              | min        |
| maxFragmentShadingRateRasterizationSamples       | -                 | VK_SAMPLE_COUNT_4_BIT | min        |
| fragmentShadingRateWithShaderDepthStencilWrites | -                 | false           | implementatio
| fragmentShadingRateWithSampleMask                | -                 | false           | implementatio
| fragmentShadingRateWithShaderSampleMask          | -                 | false           | implementatio
| fragmentShadingRateWithConservativeRasterization | -                 | false           | implementatio
| fragmentShadingRateWithFragmentShaderInterlock   | -                 | false           | implementatio
| fragmentShadingRateWithCustomSampleLocations     | -                 | false           | implementatio
<table>
<thead>
<tr>
<th>Limit</th>
<th>Unsupported Limit</th>
<th>Supported Limit</th>
<th>Limit Type¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>fragmentShadingRateStrictMultiplyCombiner</td>
<td>-</td>
<td>false</td>
<td>implementation-dependent</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: [minInterpolationOffset, maxInterpolationOffset]. The ULP is determined by subPixelInterpolationOffsetBits. If subPixelInterpolationOffsetBits is 4, this provides increments of $(1/2^4) = 0.0625$, and thus the range of supported interpolation offsets would be $[-0.5, 0.4375]$.

6

The point size ULP is determined by pointSizeGranularity. If the pointSizeGranularity is 0.125, the range of supported point sizes must be at least $[1.0, 63.875]$.

7

The line width ULP is determined by lineWidthGranularity. If the lineWidthGranularity is 0.0625, the range of supported line widths must be at least $[1.0, 7.9375]$.

8

The 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 $2^{32} - 1$.

33.2. Additional Multisampling Capabilities

To query additional multisampling capabilities which may be supported for a specific sample count, beyond the minimum capabilities described for Limits above, call:
void vkGetPhysicalDeviceMultisamplePropertiesEXT(
    VkPhysicalDevice physicalDevice,
    VkSampleCountFlagBits samples,
    VkMultisamplePropertiesEXT* pMultisampleProperties);

- **physicalDevice** is the physical device from which to query the additional multisampling capabilities.
- **samples** is a VkSampleCountFlagBits value specifying the sample count to query capabilities for.
- **pMultisampleProperties** is a pointer to a VkMultisamplePropertiesEXT structure in which information about additional multisampling capabilities specific to the sample count is returned.

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceMultisamplePropertiesEXT-physicalDevice-parameter
  - physicalDevice must be a valid VkPhysicalDevice handle
- VUID-vkGetPhysicalDeviceMultisamplePropertiesEXT-samples-parameter
  - samples must be a valid VkSampleCountFlagBits value
- VUID-vkGetPhysicalDeviceMultisamplePropertiesEXT-pMultisampleProperties-parameter
  - pMultisampleProperties must be a valid pointer to a VkMultisamplePropertiesEXT structure

The VkMultisamplePropertiesEXT structure is defined as

```c
typedef struct VkMultisamplePropertiesEXT {
    VkStructureType sType;
    void* pNext;
    VkExtent2D maxSampleLocationGridSize;
} VkMultisamplePropertiesEXT;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **maxSampleLocationGridSize** is the maximum size of the pixel grid in which sample locations can vary.

### Valid Usage (Implicit)

- VUID-VkMultisamplePropertiesEXT-sType-sType
  - sType must be VK_STRUCTURE_TYPE_MULTISAMPLE_PROPERTIES_EXT
- VUID-VkMultisamplePropertiesEXT-pNext-pNext
pNext must be NULL

If the sample count for which additional multisampling capabilities are requested using vkGetPhysicalDeviceMultisamplePropertiesEXT is set in VkPhysicalDeviceSampleLocationsPropertiesEXT::sampleLocationSampleCounts the width and height members of VkMultisamplePropertiesEXT::maxSampleLocationGridSize must be greater than or equal to the corresponding members of VkPhysicalDeviceSampleLocationsPropertiesEXT::maxSampleLocationGridSize, respectively, otherwise both members must be 0.
Chapter 34. Formats

Supported buffer and image formats may vary across implementations. A minimum set of format features are guaranteed, but others must be explicitly queried before use to ensure they are supported by the implementation.

The features for the set of formats (VkFormat) supported by the implementation are queried individually using the vkGetPhysicalDeviceFormatProperties command.

34.1. Format Definition

The following image formats can be passed to, and may be returned from Vulkan commands. The memory required to store each format is discussed with that format, and also summarized in the Representation and Texel Block Size section and the Compatible formats table.

```c
// Provided by VK_VERSION_1_0
typedef enum VkFormat {
    VK_FORMAT_UNDEFINED = 0,
    VK_FORMAT_R4G4_UNORM_PACK8 = 1,
    VK_FORMAT_R4G4B4A4_UNORM_PACK16 = 2,
    VK_FORMAT_B4G4R4A4_UNORM_PACK16 = 3,
    VK_FORMAT_R5G6B5_UNORM_PACK16 = 4,
    VK_FORMAT_B5G6R5_UNORM_PACK16 = 5,
    VK_FORMAT_R5G5B5A1_UNORM_PACK16 = 6,
    VK_FORMAT_B5G5R5A1_UNORM_PACK16 = 7,
    VK_FORMAT_A1R5G5B5_UNORM_PACK16 = 8,
    VK_FORMAT_R8_UNORM = 9,
    VK_FORMAT_R8_SNORM = 10,
    VK_FORMAT_R8_USCALED = 11,
    VK_FORMAT_R8_SSCALED = 12,
    VK_FORMAT_R8_UINT = 13,
    VK_FORMAT_R8_SINT = 14,
    VK_FORMAT_R8_SRGB = 15,
    VK_FORMAT_R8G8_UNORM = 16,
    VK_FORMAT_R8G8_SNORM = 17,
    VK_FORMAT_R8G8_USCALED = 18,
    VK_FORMAT_R8G8_SSCALED = 19,
    VK_FORMAT_R8G8_UINT = 20,
    VK_FORMAT_R8G8_SINT = 21,
    VK_FORMAT_R8G8_SRGB = 22,
    VK_FORMAT_R8G8B8_UNORM = 23,
    VK_FORMAT_R8G8B8_SNORM = 24,
    VK_FORMAT_R8G8B8_USCALED = 25,
    VK_FORMAT_R8G8B8_SSCALED = 26,
    VK_FORMAT_R8G8B8_UINT = 27,
    VK_FORMAT_R8G8B8_SINT = 28,
    VK_FORMAT_R8G8B8_SRGB = 29,
    VK_FORMAT_B8G8R8_UNORM = 30,
    VK_FORMAT_B8G8R8_SNORM = 31,
};
```
VK_FORMAT_B8G8R8_USCALED = 32,
VK_FORMAT_B8G8R8_SSCALED = 33,
VK_FORMAT_B8G8R8_UINT = 34,
VK_FORMAT_B8G8R8_SINT = 35,
VK_FORMAT_B8G8R8_SRGB = 36,
VK_FORMAT_R8G8B8A8_UNORM = 37,
VK_FORMAT_R8G8B8A8_SNORM = 38,
VK_FORMAT_R8G8B8A8_USCALED = 39,
VK_FORMAT_R8G8B8A8_SSCALED = 40,
VK_FORMAT_R8G8B8A8_UINT = 41,
VK_FORMAT_R8G8B8A8_SINT = 42,
VK_FORMAT_R8G8B8A8_SRGB = 43,
VK_FORMAT_B8G8R8A8_USCALED = 44,
VK_FORMAT_B8G8R8A8_SSCALED = 45,
VK_FORMAT_B8G8R8A8_UINT = 46,
VK_FORMAT_B8G8R8A8_SINT = 47,
VK_FORMAT_B8G8R8A8_SRGB = 48,
VK_FORMAT_A8B8G8R8_UNORM_PACK32 = 51,
VK_FORMAT_A8B8G8R8_SNORM_PACK32 = 52,
VK_FORMAT_A8B8G8R8_USCALED_PACK32 = 53,
VK_FORMAT_A8B8G8R8_SSCALED_PACK32 = 54,
VK_FORMAT_A8B8G8R8_UINT_PACK32 = 55,
VK_FORMAT_A8B8G8R8_SINT_PACK32 = 56,
VK_FORMAT_A8B8G8R8_SRGB_PACK32 = 57,
VK_FORMAT_A2R10G10B10_UNORM_PACK32 = 58,
VK_FORMAT_A2R10G10B10_SNORM_PACK32 = 59,
VK_FORMAT_A2R10G10B10_USCALED_PACK32 = 60,
VK_FORMAT_A2R10G10B10_SSCALED_PACK32 = 61,
VK_FORMAT_A2R10G10B10_UINT_PACK32 = 62,
VK_FORMAT_A2R10G10B10_SINT_PACK32 = 63,
VK_FORMAT_A2B10G10R10_UNORM_PACK32 = 64,
VK_FORMAT_A2B10G10R10_SNORM_PACK32 = 65,
VK_FORMAT_A2B10G10R10_USCALED_PACK32 = 66,
VK_FORMAT_A2B10G10R10_SSCALED_PACK32 = 67,
VK_FORMAT_A2B10G10R10_UINT_PACK32 = 68,
VK_FORMAT_A2B10G10R10_SINT_PACK32 = 69,
VK_FORMAT_R16_UNORM = 70,
VK_FORMAT_R16_SNORM = 71,
VK_FORMAT_R16_USCALED = 72,
VK_FORMAT_R16_SSCALED = 73,
VK_FORMAT_R16_UINT = 74,
VK_FORMAT_R16_SINT = 75,
VK_FORMAT_R16_SFLOAT = 76,
VK_FORMAT_R16G16_UNORM = 77,
VK_FORMAT_R16G16_SNORM = 78,
VK_FORMAT_R16G16_USCALED = 79,
VK_FORMAT_R16G16_SSCALED = 80,
VK_FORMAT_R16G16_SFLOAT = 83,
VK_FORMAT_R16G16B16_UNORM = 84,
VK_FORMAT_R16G16B16_SNORM = 85,
VK_FORMAT_R16G16B16_USCALED = 86,
VK_FORMAT_R16G16B16_SSCALED = 87,
VK_FORMAT_R16G16B16_UINT = 88,
VK_FORMAT_R16G16B16_SINT = 89,
VK_FORMAT_R16G16B16_SFLOAT = 90,
VK_FORMAT_R16G16B16A16_UNORM = 91,
VK_FORMAT_R16G16B16A16_SNORM = 92,
VK_FORMAT_R16G16B16A16_USCALED = 93,
VK_FORMAT_R16G16B16A16_SSCALED = 94,
VK_FORMAT_R16G16B16A16_UINT = 95,
VK_FORMAT_R16G16B16A16_SINT = 96,
VK_FORMAT_R16G16B16A16_SFLOAT = 97,
VK_FORMAT_R32_UINT = 98,
VK_FORMAT_R32_SINT = 99,
VK_FORMAT_R32_SFLOAT = 100,
VK_FORMAT_R32G32_UINT = 101,
VK_FORMAT_R32G32_SINT = 102,
VK_FORMAT_R32G32_SFLOAT = 103,
VK_FORMAT_R32G32B32_UINT = 104,
VK_FORMAT_R32G32B32_SINT = 105,
VK_FORMAT_R32G32B32_SFLOAT = 106,
VK_FORMAT_R32G32B32A32_UINT = 107,
VK_FORMAT_R32G32B32A32_SINT = 108,
VK_FORMAT_R32G32B32A32_SFLOAT = 109,
VK_FORMAT_R64_UINT = 110,
VK_FORMAT_R64_SINT = 111,
VK_FORMAT_R64_SFLOAT = 112,
VK_FORMAT_R64G64_UINT = 113,
VK_FORMAT_R64G64_SINT = 114,
VK_FORMAT_R64G64_SFLOAT = 115,
VK_FORMAT_R64G64B64_UINT = 116,
VK_FORMAT_R64G64B64_SINT = 117,
VK_FORMAT_R64G64B64_SFLOAT = 118,
VK_FORMAT_R64G64B64A64_UINT = 119,
VK_FORMAT_R64G64B64A64_SINT = 120,
VK_FORMAT_R64G64B64A64_SFLOAT = 121,
VK_FORMAT_B10G11R11_UFLOAT_PACK32 = 122,
VK_FORMAT_E5B9G9R9_UFLOAT_PACK32 = 123,
VK_FORMAT_D16_UNORM = 124,
VK_FORMAT_X8_D24_UNORM_PACK32 = 125,
VK_FORMAT_D32_SFLOAT = 126,
VK_FORMAT_S8_UINT = 127,
VK_FORMAT_D16_UNORM_S8_UINT = 128,
VK_FORMAT_D24_UNORM_S8_UINT = 129,
VK_FORMAT_D32_SFLOAT_S8_UINT = 130,
VK_FORMAT_BC1_RGB_UNORM_BLOCK = 131,
VK_FORMAT_BC1_RGB_SRGB_BLOCK = 132,
VK_FORMAT_BC1_RGBA_UNORM_BLOCK = 133,
VK_FORMAT_BC1_RGBA_SRGB_BLOCK = 134,
VK_FORMAT_BC2_UNORM_BLOCK = 135,
VK_FORMAT_BC2_SRGB_BLOCK = 136,
VK_FORMAT_BC3_UNORM_BLOCK = 137,
VK_FORMAT_BC3_SRGB_BLOCK = 138,
VK_FORMAT_BC4_UNORM_BLOCK = 139,
VK_FORMAT_BC4_SNORM_BLOCK = 140,
VK_FORMAT_BC5_UNORM_BLOCK = 141,
VK_FORMAT_BC5_SNORM_BLOCK = 142,
VK_FORMAT_BCGH_UFLOAT_BLOCK = 143,
VK_FORMAT_BCGH_SFLOAT_BLOCK = 144,
VK_FORMAT_BC7_UNORM_BLOCK = 145,
VK_FORMAT_BC7_SRGB_BLOCK = 146,
VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK = 147,
VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK = 148,
VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK = 149,
VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK = 150,
VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK = 151,
VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK = 152,
VK_FORMAT_EAC_R11_UNORM_BLOCK = 153,
VK_FORMAT_EAC_R11_SNORM_BLOCK = 154,
VK_FORMAT_EAC_R11G11_UNORM_BLOCK = 155,
VK_FORMAT_EAC_R11G11_SNORM_BLOCK = 156,
VK_FORMAT_ASTC_4x4_UNORM_BLOCK = 157,
VK_FORMAT_ASTC_4x4_SRGB_BLOCK = 158,
VK_FORMAT_ASTC_5x4_UNORM_BLOCK = 159,
VK_FORMAT_ASTC_5x4_SRGB_BLOCK = 160,
VK_FORMAT_ASTC_5x5_UNORM_BLOCK = 161,
VK_FORMAT_ASTC_5x5_SRGB_BLOCK = 162,
VK_FORMAT_ASTC_6x5_UNORM_BLOCK = 163,
VK_FORMAT_ASTC_6x5_SRGB_BLOCK = 164,
VK_FORMAT_ASTC_6x6_UNORM_BLOCK = 165,
VK_FORMAT_ASTC_6x6_SRGB_BLOCK = 166,
VK_FORMAT_ASTC_8x5_UNORM_BLOCK = 167,
VK_FORMAT_ASTC_8x5_SRGB_BLOCK = 168,
VK_FORMAT_ASTC_8x6_UNORM_BLOCK = 169,
VK_FORMAT_ASTC_8x6_SRGB_BLOCK = 170,
VK_FORMAT_ASTC_8x8_UNORM_BLOCK = 171,
VK_FORMAT_ASTC_8x8_SRGB_BLOCK = 172,
VK_FORMAT_ASTC_10x5_UNORM_BLOCK = 173,
VK_FORMAT_ASTC_10x5_SRGB_BLOCK = 174,
VK_FORMAT_ASTC_10x6_UNORM_BLOCK = 175,
VK_FORMAT_ASTC_10x6_SRGB_BLOCK = 176,
VK_FORMAT_ASTC_10x8_UNORM_BLOCK = 177,
VK_FORMAT_ASTC_10x8_SRGB_BLOCK = 178,
VK_FORMAT_ASTC_10x10_UNORM_BLOCK = 179,
VK_FORMAT_ASTC_10x10_SRGB_BLOCK = 180,
VK_FORMAT_ASTC_12x10_UNORM_BLOCK = 181,
VK_FORMAT_ASTC_12x10_SRGB_BLOCK = 182,
VK_FORMAT_ASTC_12x12_UNORM_BLOCK = 183,
VK_FORMAT_ASTC_12x12_SRGB_BLOCK = 184,
VK_FORMAT_G8B8R8_422_UNORM = 000156000,
VK_FORMAT_B8G8R8G8_422_UNORM = 000156001,
VK_FORMAT_G8B8_R8_3PLANE_420_UNORM = 000156002,
VK_FORMAT_B8B8R8_2PLANE_420_UNORM = 000156003,
VK_FORMAT_G8B8_R8_3PLANE_422_UNORM = 000156004,
VK_FORMAT_B8B8R8_2PLANE_422_UNORM = 000156005,
VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM = 000156006,
VK_FORMAT_R10X6_UNORM_PACK16 = 000156007,
VK_FORMAT_R10X6G10X6_UNORM_2PACK16 = 000156008,
VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16 = 000156009,
VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16 = 000156010,
VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16 = 000156011,
VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16 = 000156022,
VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16 = 000156023,
VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16 = 1000156025,
// Provided by VK_VERSION_1_1
VK_FORMAT_G12X4_B12X4R12X4_3PLANE_444_UNORM_3PACK16 = 1000156026,
// Provided by VK_VERSION_1_1
VK_FORMAT_G16B16G16R16_422_UNORM = 1000156027,
// Provided by VK_VERSION_1_1
VK_FORMAT_B16G16R16G16_422_UNORM = 1000156028,
// Provided by VK_VERSION_1_1
VK_FORMAT_G16_B16R16_3PLANE_420_UNORM = 1000156029,
// Provided by VK_VERSION_1_1
VK_FORMAT_G16_B16R16_2PLANE_420_UNORM = 1000156030,
// Provided by VK_VERSION_1_1
VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM = 1000156031,
// Provided by VK_VERSION_1_1
VK_FORMAT_G16_B16_R16_2PLANE_422_UNORM = 1000156032,
// Provided by VK_VERSION_1_1
VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM = 1000156033,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK_EXT = 1000066000,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK_EXT = 1000066001,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK_EXT = 1000066002,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK_EXT = 1000066003,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK_EXT = 1000066004,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK_EXT = 1000066005,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK_EXT = 1000066006,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK_EXT = 1000066007,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK_EXT = 1000066008,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK_EXT = 1000066009,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK_EXT = 1000066010,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK_EXT = 1000066011,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK_EXT = 1000066012,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK_EXT = 1000066013,
// Provided by VK_EXT_ycbcr_2plane_444Formats
VK_FORMAT_G8_B8R8_2PLANE_444_UNORM_EXT = 1000330000,
// Provided by VK_EXT_ycbcr_2plane_444Formats
VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16_EXT = 1000330001,
// Provided by VK_EXT_ycbcr_2plane_444Formats
VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_EXT = 1000330002,
• **VK_FORMAT_UNDEFINED** specifies that the format is not specified.

• **VK_FORMAT_R4G4_UNORM_PACK8** specifies a two-component, 8-bit packed unsigned normalized format that has a 4-bit R component in bits 4..7, and a 4-bit G component in bits 0..3.

• **VK_FORMAT_R4G4B4A4_UNORM_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 4-bit R component in bits 12..15, a 4-bit G component in bits 8..11, a 4-bit B component in bits 4..7, and a 4-bit A component in bits 0..3.

• **VK_FORMAT_B4G4R4A4_UNORM_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 4-bit B component in bits 12..15, a 4-bit G component in bits 8..11, a 4-bit R component in bits 4..7, and a 4-bit A component in bits 0..3.

• **VK_FORMAT_A4R4G4B4_UNORM_PACK16_EXT** specifies a four-component, 16-bit packed unsigned normalized format that has a 4-bit A component in bits 12..15, a 4-bit R component in bits 8..11, a 4-bit G component in bits 4..7, and a 4-bit B component in bits 0..3.

• **VK_FORMAT_A4B4G4R4_UNORM_PACK16_EXT** specifies a four-component, 16-bit packed unsigned normalized format that has a 4-bit A component in bits 12..15, a 4-bit B component in bits 8..11, a 4-bit G component in bits 4..7, and a 4-bit R component in bits 0..3.

• **VK_FORMAT_R5G6B5_UNORM_PACK16** specifies a three-component, 16-bit packed unsigned normalized format that has a 5-bit R component in bits 11..15, a 6-bit G component in bits 5..10, and a 5-bit B component in bits 0..4.

• **VK_FORMAT_B5G6R5_UNORM_PACK16** specifies a three-component, 16-bit packed unsigned normalized format that has a 5-bit B component in bits 11..15, a 6-bit G component in bits 5..10, and a 5-bit R component in bits 0..4.

• **VK_FORMAT_R5G5B5A1_UNORM_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 5-bit R component in bits 11..15, a 5-bit G component in bits 6..10, a 5-bit B component in bits 1..5, and a 1-bit A component in bit 0.

• **VK_FORMAT_B5G5R5A1_UNORM_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 5-bit B component in bits 11..15, a 5-bit G component in bits 6..10, a 5-bit R component in bits 1..5, and a 1-bit A component in bit 0.

• **VK_FORMAT_A1R5G5B5_UNORM_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 1-bit A component in bit 15, a 5-bit R component in bits 10..14, a 5-bit G component in bits 5..9, and a 5-bit B component in bits 0..4.

• **VK_FORMAT_R8_UNORM** specifies a one-component, 8-bit unsigned normalized format that has a single 8-bit R component.

• **VK_FORMAT_R8_SNORM** specifies a one-component, 8-bit signed normalized format that has a single 8-bit R component.

• **VK_FORMAT_R8_USCALED** specifies a one-component, 8-bit unsigned scaled integer format that has a
single 8-bit R component.

- **VK_FORMAT_R8_SSCALED** specifies a one-component, 8-bit signed scaled integer format that has a single 8-bit R component.
- **VK_FORMAT_R8_UINT** specifies a one-component, 8-bit unsigned integer format that has a single 8-bit R component.
- **VK_FORMAT_R8_SINT** specifies a one-component, 8-bit signed integer format that has a single 8-bit R component.
- **VK_FORMAT_R8_SR GB** specifies a one-component, 8-bit unsigned normalized format that has a single 8-bit R component stored with sRGB nonlinear encoding.
- **VK_FORMAT_R8G8_UNORM** specifies a two-component, 16-bit unsigned normalized format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.
- **VK_FORMAT_R8G8_SNORM** specifies a two-component, 16-bit signed normalized format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.
- **VK_FORMAT_R8G8_USCALED** specifies a two-component, 16-bit unsigned scaled integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.
- **VK_FORMAT_R8G8_SSCALED** specifies a two-component, 16-bit signed scaled integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.
- **VK_FORMAT_R8G8_UINT** specifies a two-component, 16-bit unsigned integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.
- **VK_FORMAT_R8G8_SINT** specifies a two-component, 16-bit signed integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.
- **VK_FORMAT_R8G8_SRGB** specifies a two-component, 16-bit unsigned normalized format that has an 8-bit R component stored with sRGB nonlinear encoding in byte 0, and an 8-bit G component stored with sRGB nonlinear encoding in byte 1.
- **VK_FORMAT_R8G8B8_UNORM** specifies a three-component, 24-bit unsigned normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
- **VK_FORMAT_R8G8B8_SNORM** specifies a three-component, 24-bit signed normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
- **VK_FORMAT_R8G8B8_USCALED** specifies a three-component, 24-bit unsigned scaled format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
- **VK_FORMAT_R8G8B8_SSCALED** specifies a three-component, 24-bit signed scaled integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
- **VK_FORMAT_R8G8B8_UINT** specifies a three-component, 24-bit unsigned integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
- **VK_FORMAT_R8G8B8_SINT** specifies a three-component, 24-bit signed integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
- **VK_FORMAT_R8G8B8_SRGB** specifies a three-component, 24-bit unsigned normalized format that has an 8-bit R component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, and an 8-bit B component stored with sRGB nonlinear encoding in byte 2.
nonlinear encoding in byte 2.

- **VK_FORMAT_B8G8R8_UNORM** specifies a three-component, 24-bit unsigned normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.
- **VK_FORMAT_B8G8R8_SNORM** specifies a three-component, 24-bit signed normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.
- **VK_FORMAT_B8G8R8_USCALED** specifies a three-component, 24-bit unsigned scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.
- **VK_FORMAT_B8G8R8_SSCALED** specifies a three-component, 24-bit signed scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.
- **VK_FORMAT_B8G8R8_UINT** specifies a three-component, 24-bit unsigned integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.
- **VK_FORMAT_B8G8R8_SINT** specifies a three-component, 24-bit signed integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.
- **VK_FORMAT_B8G8R8_SRGB** specifies a three-component, 24-bit unsigned normalized format that has an 8-bit B component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, and an 8-bit R component stored with sRGB nonlinear encoding in byte 2.
- **VK_FORMAT_R8G8B8A8_UNORM** specifies a four-component, 32-bit unsigned normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.
- **VK_FORMAT_R8G8B8A8_SNORM** specifies a four-component, 32-bit signed normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.
- **VK_FORMAT_R8G8B8A8_USCALED** specifies a four-component, 32-bit unsigned scaled format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.
- **VK_FORMAT_R8G8B8A8_SSCALED** specifies a four-component, 32-bit signed scaled format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.
- **VK_FORMAT_R8G8B8A8_UINT** specifies a four-component, 32-bit unsigned integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.
- **VK_FORMAT_R8G8B8A8_SINT** specifies a four-component, 32-bit signed integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.
- **VK_FORMAT_R8G8B8A8_SRGB** specifies a four-component, 32-bit unsigned normalized format that has an 8-bit R component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, an 8-bit B component stored with sRGB nonlinear encoding in byte 2, and an 8-bit A component in byte 3.
- **VK_FORMAT_B8G8R8A8_UNORM** specifies a four-component, 32-bit unsigned normalized format that
has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_B8G8R8A8_SNORM** specifies a four-component, 32-bit signed normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_B8G8R8A8_USCALED** specifies a four-component, 32-bit unsigned scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_B8G8R8A8_SSCALED** specifies a four-component, 32-bit signed scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_B8G8R8A8_UINT** specifies a four-component, 32-bit unsigned integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_B8G8R8A8_SINT** specifies a four-component, 32-bit signed integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_B8G8R8A8_SRGB** specifies a four-component, 32-bit unsigned normalized format that has an 8-bit B component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, an 8-bit R component stored with sRGB nonlinear encoding in byte 2, and an 8-bit A component in byte 3.

- **VK_FORMAT_A8B8G8R8_UNORM_PACK32** specifies a four-component, 32-bit packed unsigned normalized format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

- **VK_FORMAT_A8B8G8R8_SNORM_PACK32** specifies a four-component, 32-bit packed signed normalized format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

- **VK_FORMAT_A8B8G8R8_USCALED_PACK32** specifies a four-component, 32-bit packed unsigned scaled integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

- **VK_FORMAT_A8B8G8R8_SSCALED_PACK32** specifies a four-component, 32-bit packed signed scaled integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

- **VK_FORMAT_A8B8G8R8_UINT_PACK32** specifies a four-component, 32-bit packed unsigned integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

- **VK_FORMAT_A8B8G8R8_SINT_PACK32** specifies a four-component, 32-bit packed signed integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

- **VK_FORMAT_A8B8G8R8_SRGB_PACK32** specifies a four-component, 32-bit packed unsigned normalized format that has an 8-bit A component in bits 24..31, an 8-bit B component stored with sRGB nonlinear encoding in bits 16..23, an 8-bit G component stored with sRGB nonlinear encoding in bits 8..15, and an 8-bit R component stored with sRGB nonlinear encoding in bits 0..7.
• **VK_FORMAT_A2R10G10B10_UNORM_PACK32** specifies a four-component, 32-bit packed unsigned normalized format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

• **VK_FORMAT_A2R10G10B10_SNORM_PACK32** specifies a four-component, 32-bit packed signed normalized format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

• **VK_FORMAT_A2R10G10B10_USCALED_PACK32** specifies a four-component, 32-bit packed unsigned scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

• **VK_FORMAT_A2R10G10B10_SSCALED_PACK32** specifies a four-component, 32-bit packed signed scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

• **VK_FORMAT_A2R10G10B10_UINT_PACK32** specifies a four-component, 32-bit packed unsigned integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

• **VK_FORMAT_A2R10G10B10_SINT_PACK32** specifies a four-component, 32-bit packed signed integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

• **VK_FORMAT_A2B10G10R10_UNORM_PACK32** specifies a four-component, 32-bit packed unsigned normalized format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

• **VK_FORMAT_A2B10G10R10_SNORM_PACK32** specifies a four-component, 32-bit packed signed normalized format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

• **VK_FORMAT_A2B10G10R10_USCALED_PACK32** specifies a four-component, 32-bit packed unsigned scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

• **VK_FORMAT_A2B10G10R10_SSCALED_PACK32** specifies a four-component, 32-bit packed signed scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

• **VK_FORMAT_A2B10G10R10_UINT_PACK32** specifies a four-component, 32-bit packed unsigned integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

• **VK_FORMAT_A2B10G10R10_SINT_PACK32** specifies a four-component, 32-bit packed signed integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

• **VK_FORMAT_R16_UNORM** specifies a one-component, 16-bit unsigned normalized format that has a single 16-bit R component.

• **VK_FORMAT_R16_SNORM** specifies a one-component, 16-bit signed normalized format that has a single 16-bit R component.

• **VK_FORMAT_R16_USCALED** specifies a one-component, 16-bit unsigned scaled integer format that has a single 16-bit R component.
• **VK_FORMAT_R16_SSCALED** specifies a one-component, 16-bit signed scaled integer format that has a single 16-bit R component.

• **VK_FORMAT_R16_UINT** specifies a one-component, 16-bit unsigned integer format that has a single 16-bit R component.

• **VK_FORMAT_R16_SINT** specifies a one-component, 16-bit signed integer format that has a single 16-bit R component.

• **VK_FORMAT_R16_SFLOAT** specifies a one-component, 16-bit signed floating-point format that has a single 16-bit R component.

• **VK_FORMAT_R16G16_UNORM** specifies a two-component, 32-bit unsigned normalized format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

• **VK_FORMAT_R16G16_SNORM** specifies a two-component, 32-bit signed normalized format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

• **VK_FORMAT_R16G16_USCALED** specifies a two-component, 32-bit unsigned scaled integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

• **VK_FORMAT_R16G16_SSCALED** specifies a two-component, 32-bit signed scaled integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

• **VK_FORMAT_R16G16_UINT** specifies a two-component, 32-bit unsigned integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

• **VK_FORMAT_R16G16_SINT** specifies a two-component, 32-bit signed integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

• **VK_FORMAT_R16G16_SFLOAT** specifies a two-component, 32-bit signed floating-point format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

• **VK_FORMAT_R16G16B16_UNORM** specifies a three-component, 48-bit unsigned normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

• **VK_FORMAT_R16G16B16_SNORM** specifies a three-component, 48-bit signed normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

• **VK_FORMAT_R16G16B16_USCALED** specifies a three-component, 48-bit unsigned scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

• **VK_FORMAT_R16G16B16_SSCALED** specifies a three-component, 48-bit signed scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

• **VK_FORMAT_R16G16B16_UINT** specifies a three-component, 48-bit unsigned integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

• **VK_FORMAT_R16G16B16_SINT** specifies a three-component, 48-bit signed integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

• **VK_FORMAT_R16G16B16_SFLOAT** specifies a three-component, 48-bit signed floating-point format that
has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

- **VK_FORMAT_R16G16B16A16_UNORM** specifies a four-component, 64-bit unsigned normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

- **VK_FORMAT_R16G16B16A16_SNORM** specifies a four-component, 64-bit signed normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

- **VK_FORMAT_R16G16B16A16_USCALED** specifies a four-component, 64-bit unsigned scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

- **VK_FORMAT_R16G16B16A16_SSCALED** specifies a four-component, 64-bit signed scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

- **VK_FORMAT_R16G16B16A16_UINT** specifies a four-component, 64-bit unsigned integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

- **VK_FORMAT_R16G16B16A16_SINT** specifies a four-component, 64-bit signed integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

- **VK_FORMAT_R16G16B16A16_SFLOAT** specifies a four-component, 64-bit signed floating-point format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

- **VK_FORMAT_R32_UINT** specifies a one-component, 32-bit unsigned integer format that has a single 32-bit R component.

- **VK_FORMAT_R32_SINT** specifies a one-component, 32-bit signed integer format that has a single 32-bit R component.

- **VK_FORMAT_R32_SFLOAT** specifies a one-component, 32-bit signed floating-point format that has a single 32-bit R component.

- **VK_FORMAT_R32G32_UINT** specifies a two-component, 64-bit unsigned integer format that has a 32-bit R component in bytes 0..3, and a 32-bit G component in bytes 4..7.

- **VK_FORMAT_R32G32_SINT** specifies a two-component, 64-bit signed integer format that has a 32-bit R component in bytes 0..3, and a 32-bit G component in bytes 4..7.

- **VK_FORMAT_R32G32_SFLOAT** specifies a two-component, 64-bit signed floating-point format that has a 32-bit R component in bytes 0..3, and a 32-bit G component in bytes 4..7.

- **VK_FORMAT_R32G32B32_UINT** specifies a three-component, 96-bit unsigned integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, and a 32-bit B component in bytes 8..11.

- **VK_FORMAT_R32G32B32_SINT** specifies a three-component, 96-bit signed integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, and a 32-bit B component in bytes 8..11.
• **VK_FORMAT_R32G32B32_SFLOAT** specifies a three-component, 96-bit signed floating-point format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, and a 32-bit B component in bytes 8..11.

• **VK_FORMAT_R32G32B32A32_UINT** specifies a four-component, 128-bit unsigned integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, a 32-bit B component in bytes 8..11, and a 32-bit A component in bytes 12..15.

• **VK_FORMAT_R32G32B32A32_SINT** specifies a four-component, 128-bit signed integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, a 32-bit B component in bytes 8..11, and a 32-bit A component in bytes 12..15.

• **VK_FORMAT_R32G32B32A32_SFLOAT** specifies a four-component, 128-bit signed floating-point format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, a 32-bit B component in bytes 8..11, and a 32-bit A component in bytes 12..15.

• **VK_FORMAT_R64_UINT** specifies a one-component, 64-bit unsigned integer format that has a single 64-bit R component.

• **VK_FORMAT_R64_SINT** specifies a one-component, 64-bit signed integer format that has a single 64-bit R component.

• **VK_FORMAT_R64_SFLOAT** specifies a one-component, 64-bit signed floating-point format that has a single 64-bit R component.

• **VK_FORMAT_R64G64_UINT** specifies a two-component, 128-bit unsigned integer format that has a 64-bit R component in bytes 0..7, and a 64-bit G component in bytes 8..15.

• **VK_FORMAT_R64G64_SINT** specifies a two-component, 128-bit signed integer format that has a 64-bit R component in bytes 0..7, and a 64-bit G component in bytes 8..15.

• **VK_FORMAT_R64G64_SFLOAT** specifies a two-component, 128-bit signed floating-point format that has a 64-bit R component in bytes 0..7, and a 64-bit G component in bytes 8..15.

• **VK_FORMAT_R64G64B64_UINT** specifies a three-component, 192-bit unsigned integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, and a 64-bit B component in bytes 16..23.

• **VK_FORMAT_R64G64B64_SINT** specifies a three-component, 192-bit signed integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, and a 64-bit B component in bytes 16..23.

• **VK_FORMAT_R64G64B64_SFLOAT** specifies a three-component, 192-bit signed floating-point format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, and a 64-bit B component in bytes 16..23.

• **VK_FORMAT_R64G64B64A64_UINT** specifies a four-component, 256-bit unsigned integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, a 64-bit B component in bytes 16..23, and a 64-bit A component in bytes 24..31.

• **VK_FORMAT_R64G64B64A64_SINT** specifies a four-component, 256-bit signed integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, a 64-bit B component in bytes 16..23, and a 64-bit A component in bytes 24..31.

• **VK_FORMAT_R64G64B64A64_SFLOAT** specifies a four-component, 256-bit signed floating-point format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, a 64-bit B component in bytes 16..23, and a 64-bit A component in bytes 24..31.
• **VK_FORMAT_B10G11R11_UFLOAT_PACK32** specifies a three-component, 32-bit packed unsigned floating-point format that has a 10-bit B component in bits 22..31, an 11-bit G component in bits 11..21, an 11-bit R component in bits 0..10. See [Unsigned 10-Bit Floating-Point Numbers](#) and [Unsigned 11-Bit Floating-Point Numbers](#).

• **VK_FORMAT_E5B9G9R9_UFLOAT_PACK32** specifies a three-component, 32-bit packed unsigned floating-point format that has a 5-bit shared exponent in bits 27..31, a 9-bit B component mantissa in bits 18..26, a 9-bit G component mantissa in bits 9..17, and a 9-bit R component mantissa in bits 0..8.

• **VK_FORMAT_D16_UNORM** specifies a one-component, 16-bit unsigned normalized format that has a single 16-bit depth component.

• **VK_FORMAT_X8_D24_UNORM_PACK32** specifies a two-component, 32-bit format that has 24 unsigned normalized bits in the depth component and, optionally, 8 bits that are unused.

• **VK_FORMAT_D32_SFLOAT** specifies a one-component, 32-bit signed floating-point format that has 32 bits in the depth component.

• **VK_FORMAT_S8_UINT** specifies a one-component, 8-bit unsigned integer format that has 8 bits in the stencil component.

• **VK_FORMAT_D16_UNORM_S8_UINT** specifies a two-component, 24-bit format that has 16 unsigned normalized bits in the depth component and 8 unsigned integer bits in the stencil component.

• **VK_FORMAT_D24_UNORM_S8_UINT** specifies a two-component, 32-bit packed format that has 8 unsigned integer bits in the stencil component, and 24 unsigned normalized bits in the depth component.

• **VK_FORMAT_D32_SFLOAT_S8_UINT** specifies a two-component format that has 32 signed float bits in the depth component and 8 unsigned integer bits in the stencil component. There are optionally 24 bits that are unused.

• **VK_FORMAT_BC1_RGB_UNORM_BLOCK** specifies a three-component, block-compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data. This format has no alpha and is considered opaque.

• **VK_FORMAT_BC1_RGB_SRGB_BLOCK** specifies a three-component, block-compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data with sRGB nonlinear encoding. This format has no alpha and is considered opaque.

• **VK_FORMAT_BC1_RGBA_UNORM_BLOCK** specifies a four-component, block-compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data, and provides 1 bit of alpha.

• **VK_FORMAT_BC1_RGBA_SRGB_BLOCK** specifies a four-component, block-compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data with sRGB nonlinear encoding, and provides 1 bit of alpha.

• **VK_FORMAT_BC2_UNORM_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values.

• **VK_FORMAT_BC2_SRGB_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values with sRGB nonlinear encoding.
• **VK_FORMAT_BC3_UNORM_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values.

• **VK_FORMAT_BC3_SRGB_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with 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 RGBA texel data with sRGB nonlinear encoding, and provides 1 bit of alpha.

• **VK_FORMAT_ETC2_R8G8B8AB_UNORM_BLOCK** specifies a four-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data.
RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values.

- **VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK** specifies a four-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values with sRGB nonlinear encoding applied.

- **VK_FORMAT_EAC_R11_UNORM_BLOCK** specifies a one-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized red texel data.

- **VK_FORMAT_EAC_R11_SNORM_BLOCK** specifies a one-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of signed normalized red texel data.

- **VK_FORMAT_EAC_R11G11_UNORM_BLOCK** specifies a two-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RG texel data with the first 64 bits encoding red values followed by 64 bits encoding green values.

- **VK_FORMAT_EAC_R11G11_SNORM_BLOCK** specifies a two-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of signed normalized RG texel data with the first 64 bits encoding red values followed by 64 bits encoding green values.

- **VK_FORMAT_ASTC_4x4_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_4x4_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_5x4_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5×4 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_5x4_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5×4 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5×4 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_5x5_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5×5 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_5x5_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5×5 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5×5 rectangle of signed floating-point RGBA texel data.
RGBA texel data.

- **VK_FORMAT_ASTC_6x5_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6×5 rectangle of unsigned normalized RGBA texel data.
- **VK_FORMAT_ASTC_6x5_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6×5 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- **VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6×5 rectangle of signed floating-point RGBA texel data.
- **VK_FORMAT_ASTC_6x6_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6×6 rectangle of unsigned normalized RGBA texel data.
- **VK_FORMAT_ASTC_6x6_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6×6 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- **VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6×6 rectangle of signed floating-point RGBA texel data.
- **VK_FORMAT_ASTC_8x5_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×5 rectangle of unsigned normalized RGBA texel data.
- **VK_FORMAT_ASTC_8x5_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×5 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- **VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×5 rectangle of signed floating-point RGBA texel data.
- **VK_FORMAT_ASTC_8x6_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×6 rectangle of unsigned normalized RGBA texel data.
- **VK_FORMAT_ASTC_8x6_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×6 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- **VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×6 rectangle of signed floating-point RGBA texel data.
- **VK_FORMAT_ASTC_8x8_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×8 rectangle of unsigned normalized RGBA texel data.
- **VK_FORMAT_ASTC_8x8_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×8 rectangle of unsigned normalized RGBA texel data.
texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a $8 \times 8$ rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_10x5_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a $10 \times 5$ rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_10x5_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a $10 \times 5$ rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a $10 \times 5$ rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_10x6_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a $10 \times 6$ rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_10x6_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a $10 \times 6$ rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a $10 \times 6$ rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_10x8_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a $10 \times 8$ rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_10x8_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a $10 \times 8$ rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a $10 \times 8$ rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_10x10_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a $10 \times 10$ rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_10x10_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a $10 \times 10$ rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a $10 \times 10$ rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_12x10_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a $12 \times 10$ rectangle of unsigned normalized RGBA
texel data.

- **VK_FORMAT_ASTC_12x10_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12×10 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12×10 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_12x12_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12×12 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_12x12_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12×12 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12×12 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_G8B8G8R8_422_UNORM** specifies a four-component, 32-bit format containing a pair of G components, an R component, and a B component, collectively encoding a 2×1 rectangle of unsigned normalized RGB texel data. One G value is present at each i coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has an 8-bit G component for the even i coordinate in byte 0, an 8-bit B component in byte 1, an 8-bit G component for the odd i coordinate in byte 2, and an 8-bit R component in byte 3. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a 2×1 compressed texel block.

- **VK_FORMAT_B8G8R8G8_422_UNORM** specifies a four-component, 32-bit format containing a pair of G components, an R component, and a B component, collectively encoding a 2×1 rectangle of unsigned normalized RGB texel data. One G value is present at each i coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has an 8-bit B component in byte 0, an 8-bit G component for the even i coordinate in byte 1, an 8-bit R component in byte 2, and an 8-bit G component for the odd i coordinate in byte 3. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a 2×1 compressed texel block.

- **VK_FORMAT_G8B8G8R8_3PLANE_420_UNORM** specifies an unsigned normalized multi-planar format that has an 8-bit G component in plane 0, an 8-bit B component in plane 1, and an 8-bit R component in plane 2. The horizontal and vertical dimensions of the R and B planes are halved relative to the image dimensions, and each R and B component is shared with the G components for which \( i_G \times 0.5 = i_B = i_R \) and \( j_G \times 0.5 = j_B = j_R \). The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane. 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 \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.

- \texttt{VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM} specifies an unsigned normalized multi-planar format that has an 8-bit G component in plane 0, an 8-bit B component in plane 1, and an 8-bit R component in plane 2. The horizontal dimension of the R and B plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which \( i_G \times 0.5 = i_B = i_R \). The location of each plane when this image is in linear layout can be determined via \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.

- \texttt{VK_FORMAT_G8_B8_R8_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 \texttt{vkGetImageSubresourceLayout}, using \texttt{VK_IMAGE_ASPECT_PLANE_0_BIT} for the G plane, \texttt{VK_IMAGE_ASPECT_PLANE_1_BIT} for the BR plane.

- \texttt{VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM} specifies an unsigned normalized multi-planar format that has an 8-bit G component in plane 0, an 8-bit B component in plane 1, and an 8-bit R component in plane 2. Each plane has the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via \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.

- \texttt{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.

- \texttt{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.

- \texttt{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.

- \texttt{VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16} specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a 2×1 rectangle of unsigned normalized RGB texel data. One G value is present at each i
coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 10-bit G component for the even \(i\) coordinate in the top 10 bits of the word in bytes 0..1, a 10-bit B component in the top 10 bits of the word in bytes 2..3, a 10-bit G component for the odd \(i\) coordinate in the top 10 bits of the word in bytes 4..5, and a 10-bit R component in the top 10 bits of the word in bytes 6..7, with the bottom 6 bits of each word unused. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a \(2\times1\) compressed texel block.

- **VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16** specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a \(2\times1\) rectangle of unsigned normalized RGB texel data. One G value is present at each \(i\) coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 10-bit B component in the top 10 bits of the word in bytes 0..1, a 10-bit G component for the even \(i\) coordinate in the top 10 bits of the word in bytes 2..3, a 10-bit R component in the top 10 bits of the word in bytes 4..5, and a 10-bit G component for the odd \(i\) coordinate in the top 10 bits of the word in bytes 6..7, with the bottom 6 bits of each word unused. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a \(2\times1\) compressed texel block.

- **VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16** specifies an unsigned normalized multi-planar format that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, a 10-bit B component in the top 10 bits of each 16-bit word of plane 1, and a 10-bit R component in the top 10 bits of each 16-bit word of plane 2, with the bottom 6 bits of each word unused. The horizontal and vertical dimensions of the R and B planes are halved relative to the image dimensions, and each R and B component is shared with the G components for which \(i_G = i_R\) and \(j_G = j_R\). 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 RGB texel data. One G value is present at each i coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 12-bit B component in the top 12 bits of the word in bytes 0..1, a 12-bit G component for the even i coordinate in the top 12 bits of the word in bytes 2..3, a 12-bit R component in the top 12 bits of the word in bytes 4..5, and a 12-bit G component for the odd i coordinate in the top 12 bits of the word in bytes 6..7, with the bottom 4 bits of each word unused. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a 2×1 compressed texel block.

VK_FORMAT_G12X4_B12X4R12X4_3PLANE_420_UNORM_3PACK16 specifies an unsigned normalized multiplanar 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 × 0.5| = i_B = i_R and |j_G × 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_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16 specifies an unsigned normalized multiplanar 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 × 0.5| = i_B = i_R and |j_G × 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.
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 \( \text{vkGetImageSubresourceLayout} \), using \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) for the G plane, and \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \) for the BR plane. This format only supports images with a width that is a multiple of two.

- **VK_FORMAT_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 \( \text{vkGetImageSubresourceLayout} \), using \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) for the G plane, \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \) for the B plane, and \( \text{VK_IMAGE_ASPECT_PLANE_2_BIT} \) for the R plane.

- **VK_FORMAT_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 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_B16G16R16_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 \( i_G \times 0.5 = i_B = j_R \) 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 \( \text{vkGetImageSubresourceLayout} \), using \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) for the G plane, and \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \) for the BR plane. This format only supports images with a width and height that is a multiple of two.

- \( \text{VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM} \) 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 \( \lfloor i_G \times 0.5 \rfloor = i_B = j_R \). The location of each plane when this image is in linear layout can be determined via \( \text{vkGetImageSubresourceLayout} \), using \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) for the G plane, \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \) for the B plane, and \( \text{VK_IMAGE_ASPECT_PLANE_2_BIT} \) for the R plane. This format only supports images with a width that is a multiple of two.

- \( \text{VK_FORMAT_G16_B16R16_2PLANE_422_UNORM} \) 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 \( \lfloor i_G \times 0.5 \rfloor = i_B = j_R \). The location of each plane when this image is in linear layout can be determined via \( \text{vkGetImageSubresourceLayout} \), using \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) for the G plane, and \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \) for the BR plane. This format only supports images with a width that is a multiple of two.

- \( \text{VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM} \) 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 \( \text{vkGetImageSubresourceLayout} \), using \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) for the G plane, \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \) for the B plane, and \( \text{VK_IMAGE_ASPECT_PLANE_2_BIT} \) for the R plane.

- \( \text{VK_FORMAT_G8_B8R8_2PLANE_444_UNORM_EXT} \) 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. Both planes have the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via \( \text{vkGetImageSubresourceLayout} \), using \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) for the G plane, \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \) for the B plane, and \( \text{VK_IMAGE_ASPECT_PLANE_2_BIT} \) for the R plane.

- \( \text{VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16_EXT} \) 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, 16-bit BR plane 1 consisting of a 10-bit B component in the top 10 bits of the word in bytes 0..1, and a 10-bit R component in the top 10 bits of the word in bytes 2..3, the bottom 6 bits of each word unused. Both planes have the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via \( \text{vkGetImageSubresourceLayout} \), using \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) for the G plane, and \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \) for the BR plane.

- \( \text{VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_EXT} \) specifies an unsigned normalized multi-planar format
*multi-planar format* that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 12-bit B component in the top 12 bits of the word in bytes 0..1, and a 12-bit R component in the top 12 bits of the word in bytes 2..3, the bottom 4 bits of each word unused. Both planes have the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane.

- `VK_FORMAT_G16_B16R16_2PLANE_444_UNORM_EXT` specifies an unsigned normalized *multi-planar format* that has a 16-bit G component in each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 16-bit B component in the word in bytes 0..1, and a 16-bit R component in the word in bytes 2..3. Both planes have the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane.

### 34.1.1. Compatible formats of planes of multi-planar formats

Individual planes of multi-planar formats are *compatible* with single-plane formats if they occupy the same number of bits per texel block. In the following table, individual planes of a *multi-planar format* are compatible with the format listed against the relevant plane index for that multi-planar format, and any format compatible with the listed single-plane format according to Format Compatibility Classes.

**Table 48. Plane Format Compatibility Table**

<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>VK_FORMAT_R8_UNORM</td>
<td>$w$</td>
<td>$h$</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>VK_FORMAT_R8_UNORM</td>
<td>$w$</td>
<td>$h$</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>VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM</td>
<td>VK_FORMAT_R8_UNORM</td>
<td>$w$</td>
<td>$h$</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>VK_FORMAT_R8_UNORM</td>
<td>$w$</td>
<td>$h$</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>
</tbody>
</table>

1386
<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_444_UNORM</td>
<td>VK_FORMAT_R8_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16</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_B10X6_R10X6_3PLANE_422_UNORM_3PACK16</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16</td>
<td>$w/2$</td>
<td>$h/2$</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>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_R12X4_UNORM_PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R12X4G12X4_UNORM_2PACK16</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16</td>
<td>VK_FORMAT_R12X4G12X4_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>0</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R12X4G12X4_UNORM_2PACK16</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_G12X4_B12X4R12X4_3PLANE_444_UNORM_3PACK16</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G12X4_B12X4R12X4_3PLANE_444_UNORM_3PACK16</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G16_B16R16_2PLANE_420_UNORM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G16_B16R16_2PLANE_422_UNORM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R8_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R8G8_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td></td>
<td><strong>VK_FORMAT_G8_B8R8_2PLANE_444_UNORM_EXT</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R10X6G10X6R10X6_2PLANE_444_UNORM_3PACK16_EXT</td>
<td>$w$</td>
<td>$h$</td>
</tr>
</tbody>
</table>
### 34.1.2. Packed Formats

For the purposes of address alignment when accessing buffer memory containing vertex attribute or texel data, the following formats are considered *packed* - components of the texels or attributes are stored in bitfields packed into one or more 8-, 16-, or 32-bit fundamental data type.

- **Packed into 8-bit data types:**
  - VK_FORMAT_R4G4_UNORM_PACK8

- **Packed into 16-bit data types:**
  - VK_FORMAT_R4G4B4A4_UNORM_PACK16
  - VK_FORMAT_B4G4R4A4_UNORM_PACK16
  - VK_FORMAT_A4R4G4B4_UNORM_PACK16_EXT
  - VK_FORMAT_A4B4G4R4_UNORM_PACK16_EXT
  - 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_R12X4_UNORM_PACK16
  - VK_FORMAT_R10X6G10X6_UNORM_2PACK16
  - VK_FORMAT_R12X4G12X4_UNORM_2PACK16
  - VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16
  - VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16
  - VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16
  - VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16
  - VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16
  - VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16
  - VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16

<table>
<thead>
<tr>
<th>Plane</th>
<th>Compatible format for plane</th>
<th>Width relative to the width $w$ of the plane with the largest dimensions</th>
<th>Height relative to the height $h$ of the plane with the largest dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R12X4G12X4_UNORM_2PACK16</td>
<td>$w$</td>
<td>$h$</td>
</tr>
</tbody>
</table>

**VK_FORMAT_G16_B16R16_2PLANE_444_UNORM_EXT**

<table>
<thead>
<tr>
<th>Plane</th>
<th>Compatible format for plane</th>
<th>Width relative to the width $w$ of the plane with the largest dimensions</th>
<th>Height relative to the height $h$ of the plane with the largest dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R16G16_UNORM</td>
<td>$w$</td>
<td>$h$</td>
</tr>
</tbody>
</table>
• VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16
• VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16
• VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16
• VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16
• VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16
• VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16_EXT
• VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_EXT
• VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16
• VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16
• VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16
• VK_FORMAT_R12X4G12X4B12X4R4G4R4_422_UNORM_4PACK16
• VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16
• VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16

• Packed into 32-bit data types:

• VK_FORMAT_A8R8G8B8_UNORM_PACK32
• VK_FORMAT_A8R8G8B8_SNORM_PACK32
• VK_FORMAT_A8R8G8B8_USCALED_PACK32
• VK_FORMAT_A8R8G8B8_SSCALED_PACK32
• VK_FORMAT_A8R8G8B8_UINT_PACK32
• VK_FORMAT_A8R8G8B8_SINT_PACK32
• VK_FORMAT_A8R8G8B8_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_B10G11R11_UFLOAT_PACK32
• VK_FORMAT_E5B9G9R9_UFLOAT_PACK32
34.1.3. Identification of Formats

A “format” is represented by a single enum value. The name of a format is usually built up by using the following pattern:

```
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 49. Interpretation of Numeric Format

<table>
<thead>
<tr>
<th>Numeric format</th>
<th>SPIR-V Sampled Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNORM</td>
<td>OpTypeFloat</td>
<td>The components are unsigned normalized values in the range [0,1]</td>
</tr>
<tr>
<td>SNORM</td>
<td>OpTypeFloat</td>
<td>The components are signed normalized values in the range [-1,1]</td>
</tr>
<tr>
<td>USCALED</td>
<td>OpTypeFloat</td>
<td>The components are unsigned integer values that get converted to floating-point in the range [0,2^n-1]</td>
</tr>
<tr>
<td>SScaled</td>
<td>OpTypeFloat</td>
<td>The components are signed integer values that get converted to floating-point in the range [-2^n-1,2^n-1-1]</td>
</tr>
<tr>
<td>UINT</td>
<td>OpTypeInt</td>
<td>The components are unsigned integer values in the range [0,2^n-1]</td>
</tr>
<tr>
<td>SINT</td>
<td>OpTypeInt</td>
<td>The components are signed integer values in the range [-2^n-1,2^n-1-1]</td>
</tr>
<tr>
<td>UFLOAT</td>
<td>OpTypeFloat</td>
<td>The components are unsigned floating-point numbers (used by packed, shared exponent, and some compressed formats)</td>
</tr>
<tr>
<td>SFLOAT</td>
<td>OpTypeFloat</td>
<td>The components are signed floating-point numbers</td>
</tr>
<tr>
<td>SRGB</td>
<td>OpTypeFloat</td>
<td>The R, G, and B components are unsigned normalized values that represent values using sRGB nonlinear encoding, while the A component (if one exists) is a regular unsigned normalized value</td>
</tr>
</tbody>
</table>

n is the number of bits in the component.

The suffix _PACKnn indicates that the format is packed into an underlying type with nn bits. The suffix _mPACKnn is a short-hand that indicates that the format has m groups of components (which may or may not be stored in separate planes) that are each packed into an underlying type with nn bits.

The suffix _BLOCK indicates that the format is a block-compressed format, with the representation of multiple pixels encoded interdependently within a region.

Table 50. Interpretation of Compression Scheme

<table>
<thead>
<tr>
<th>Compression scheme</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC</td>
<td>Block Compression. See Block-Compressed Image Formats.</td>
</tr>
<tr>
<td>ETC2</td>
<td>Ericsson Texture Compression. See ETC Compressed Image Formats.</td>
</tr>
<tr>
<td>EAC</td>
<td>ETC2 Alpha Compression. See ETC Compressed Image Formats.</td>
</tr>
<tr>
<td>ASTC</td>
<td>Adaptive Scalable Texture Compression (LDR Profile). See ASTC Compressed Image Formats.</td>
</tr>
</tbody>
</table>

For multi-planar images, the components in separate planes are separated by underscores, and the
number of planes is indicated by the addition of a _2PLANE or _3PLANE suffix. Similarly, the separate aspects of depth-stencil formats are separated by underscores, although these are not considered separate planes. Formats are suffixed by _422 to indicate that planes other than the first are reduced in size by a factor of two horizontally or that the R and B values appear at half the horizontal frequency of the G values, _420 to indicate that planes other than the first are reduced in size by a factor of two both horizontally and vertically, and _444 for consistency to indicate that all three planes of a three-planar image are the same size.

Note
No common format has a single plane containing both R and B components but does not store these components at reduced horizontal resolution.

34.1.4. Representation and Texel Block Size

Color formats must be represented in memory in exactly the form indicated by the format’s name. This means that promoting one format to another with more bits per component and/or additional components must not occur for color formats. Depth/stencil formats have more relaxed requirements as discussed below.

Each format has a texel block size, the number of bytes used to store one texel block (a single addressable element of an uncompressed image, or a single compressed block of a compressed image). The texel block size for each format is shown in the Compatible formats table.

The representation of non-packed formats is that the first component specified in the name of the format is in the lowest memory addresses and the last component specified is in the highest memory addresses. See Byte mappings for non-packed/compressed color formats. The in-memory ordering of bytes within a component is determined by the host endianness.

Table 51. Byte mappings for non-packed/compressed color formats

<table>
<thead>
<tr>
<th>Component</th>
<th>Format</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>VK_FORMAT_R8_</td>
<td></td>
</tr>
<tr>
<td>R G</td>
<td>VK_FORMAT_R8G8_</td>
<td></td>
</tr>
<tr>
<td>R G B</td>
<td>VK_FORMAT_R8G8B8_</td>
<td></td>
</tr>
<tr>
<td>B G R</td>
<td>VK_FORMAT_B8G8R8_</td>
<td></td>
</tr>
<tr>
<td>R G B A</td>
<td>VK_FORMAT_R8G8B8A8_</td>
<td></td>
</tr>
<tr>
<td>B G R A</td>
<td>VK_FORMAT_B8G8R8A8_</td>
<td></td>
</tr>
<tr>
<td>G_0 B G_1 R</td>
<td>VK_FORMAT_B8G8R8_422_UNORM</td>
<td></td>
</tr>
<tr>
<td>B G_0 G_1</td>
<td>VK_FORMAT_B8G8R8_422_UNORM</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>VK_FORMAT_R16_</td>
<td></td>
</tr>
<tr>
<td>R G</td>
<td>VK_FORMAT_R16G16_</td>
<td></td>
</tr>
<tr>
<td>R G B</td>
<td>VK_FORMAT_R16G16B16_</td>
<td></td>
</tr>
<tr>
<td>R G B A</td>
<td>VK_FORMAT_R16G16B16A16_</td>
<td></td>
</tr>
</tbody>
</table>
Packed formats store multiple components within one underlying type. The bit representation is that the first component specified in the name of the format is in the most-significant bits and the last component specified is in the least-significant bits of the underlying type. The in-memory ordering of bytes comprising the underlying type is determined by the host endianness.

Table 52. Bit mappings for packed 8-bit formats

<table>
<thead>
<tr>
<th>Bit</th>
<th>VK_FORMAT_R4G4_UNORM_PACK8</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>R</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
</tr>
</tbody>
</table>

Table 53. Bit mappings for packed 16-bit formats

<table>
<thead>
<tr>
<th>Bit</th>
<th>VK_FORMAT_R4G4B4A4_UNORM_PACK16</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>R</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>VK_FORMAT_B4G4R4A4_UNORM_PACK16</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>VK_FORMAT_A4R4G4B4_UNORM_PACK16_EXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>
Table 54. Bit mappings for packed 32-bit formats
34.1.5. Depth/Stencil Formats

Depth/stencil formats are considered opaque and need not be stored in the exact number of bits per texel or component ordering indicated by the format enum. However, implementations must not substitute a different depth or stencil precision than is described in the format (e.g. D16 must not be implemented as D24 or D32).

34.1.6. Format Compatibility Classes

Uncompressed color formats are compatible with each other if they occupy the same number of bits per texel block. Compressed color formats are compatible with each other if the only difference between them is the numerical type of the uncompressed pixels (e.g. signed vs. unsigned, or SRGB vs. UNORM encoding). Each depth/stencil format is only compatible with itself. In the following table, all the formats in the same row are compatible.

Table 55. Compatible Formats

<table>
<thead>
<tr>
<th>Class, Texel Block Size, # Texels/Block</th>
<th>Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit</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>
</tbody>
</table>

**Bit**

<table>
<thead>
<tr>
<th>E</th>
<th>B</th>
<th>G</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Bit**

<table>
<thead>
<tr>
<th>X</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>22</td>
</tr>
<tr>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>19</td>
<td>18</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Class, Texel Block Size, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>16-bit</td>
<td>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, VK_FORMAT_R10X6_UNORM_PACK16, VK_FORMAT_R12X4_UNORM_PACK16, VK_FORMAT_A4R4G4B4_UNORM_PACK16_EXT, VK_FORMAT_A4B4G4R4_UNORM_PACK16_EXT</td>
</tr>
<tr>
<td>24-bit</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, # Texels/Block</td>
<td></td>
</tr>
<tr>
<td>--------------------------------------</td>
<td></td>
</tr>
<tr>
<td>32-bit</td>
<td></td>
</tr>
<tr>
<td>Block size 4 byte</td>
<td></td>
</tr>
<tr>
<td>1x1x1 block extent</td>
<td></td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
</tbody>
</table>

### Formats

- `VK_FORMAT_R8G8B8A8_UNORM`
- `VK_FORMAT_R8G8B8A8_SNORM`
- `VK_FORMAT_R8G8B8A8_USCALED`
- `VK_FORMAT_R8G8B8A8_SSCALED`
- `VK_FORMAT_R8G8B8A8_UINT`
- `VK_FORMAT_R8G8B8A8_SINT`
- `VK_FORMAT_B8G8R8A8_UNORM`
- `VK_FORMAT_B8G8R8A8_SNORM`
- `VK_FORMAT_B8G8R8A8_USCALED`
- `VK_FORMAT_B8G8R8A8_SSCALED`
- `VK_FORMAT_B8G8R8A8_UINT`
- `VK_FORMAT_B8G8R8A8_SINT`
- `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_A2R10G10B10_UNORM_PACK32`
- `VK_FORMAT_A2R10G10B10_SNORM_PACK32`
- `VK_FORMAT_A2R10G10B10_USCALED_PACK32`
- `VK_FORMAT_A2R10G10B10_SSCALED_PACK32`
- `VK_FORMAT_A2R10G10B10_UINT_PACK32`
- `VK_FORMAT_A2R10G10B10_SINT_PACK32`
- `VK_FORMAT_A2B10G10R10_UNORM_PACK32`
- `VK_FORMAT_A2B10G10R10_SNORM_PACK32`
- `VK_FORMAT_A2B10G10R10_USCALED_PACK32`
- `VK_FORMAT_A2B10G10R10_SSCALED_PACK32`
- `VK_FORMAT_A2B10G10R10_UINT_PACK32`
- `VK_FORMAT_A2B10G10R10_SINT_PACK32`
- `VK_FORMAT_R16G16_UNORM`
- `VK_FORMAT_R16G16_SNORM`
- `VK_FORMAT_R16G16_USCALED`
- `VK_FORMAT_R16G16_SSCALED`
- `VK_FORMAT_R16G16_UINT`
- `VK_FORMAT_R16G16_SINT`
- `VK_FORMAT_R16G16_SFLOAT`
- `VK_FORMAT_R32_UINT`
- `VK_FORMAT_R32_SINT`
- `VK_FORMAT_R32_SFLOAT`
- `VK_FORMAT_B10G11R11_UFLOAT_PACK32`
- `VK_FORMAT_E5B9G9R9_UFLOAT_PACK32`
- `VK_FORMAT_R10X6G10X6_UNORM_2PACK16`
- `VK_FORMAT_R12X4G12X4_UNORM_2PACK16`
<table>
<thead>
<tr>
<th>Class, Texel Block Size, # Texels/Block</th>
<th>Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>96-bit</td>
<td>VK_FORMAT_R32G32B32_UINT, VK_FORMAT_R32G32B32_SINT, VK_FORMAT_R32G32B32_SFLOAT</td>
</tr>
<tr>
<td>128-bit</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</td>
<td>VK_FORMAT_R64G64B64_UINT, VK_FORMAT_R64G64B64_SINT, VK_FORMAT_R64G64B64_SFLOAT</td>
</tr>
<tr>
<td>256-bit</td>
<td>VK_FORMAT_R64G64B64A64_UINT, VK_FORMAT_R64G64B64A64_SINT, VK_FORMAT_R64G64B64A64_SFLOAT</td>
</tr>
<tr>
<td>D16</td>
<td>VK_FORMAT_D16_UNORM</td>
</tr>
<tr>
<td>Class, Texel Block Size, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>D24</td>
<td>VK_FORMAT_X8_D24_UNORM_PACK32</td>
</tr>
<tr>
<td>Block size 4 byte</td>
<td></td>
</tr>
<tr>
<td>1x1x1 block extent</td>
<td></td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>D32</td>
<td>VK_FORMAT_D32_SFLOAT</td>
</tr>
<tr>
<td>Block size 4 byte</td>
<td></td>
</tr>
<tr>
<td>1x1x1 block extent</td>
<td></td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>S8</td>
<td>VK_FORMAT_S8_UINT</td>
</tr>
<tr>
<td>Block size 1 byte</td>
<td></td>
</tr>
<tr>
<td>1x1x1 block extent</td>
<td></td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>D16S8</td>
<td>VK_FORMAT_D16_UNORM_S8_UINT</td>
</tr>
<tr>
<td>Block size 3 byte</td>
<td></td>
</tr>
<tr>
<td>1x1x1 block extent</td>
<td></td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>D24S8</td>
<td>VK_FORMAT_D24_UNORM_S8_UINT</td>
</tr>
<tr>
<td>Block size 4 byte</td>
<td></td>
</tr>
<tr>
<td>1x1x1 block extent</td>
<td></td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>D32S8</td>
<td>VK_FORMAT_D32_SFLOAT_S8_UINT</td>
</tr>
<tr>
<td>Block size 5 byte</td>
<td></td>
</tr>
<tr>
<td>1x1x1 block extent</td>
<td></td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>BC1_RGB</td>
<td>VK_FORMAT_BC1_RGB_UNORM_BLOCK, VK_FORMAT_BC1_RGB_SRGB_BLOCK</td>
</tr>
<tr>
<td>Block size 8 byte</td>
<td></td>
</tr>
<tr>
<td>4x4x1 block extent</td>
<td></td>
</tr>
<tr>
<td>16 texel/block</td>
<td></td>
</tr>
<tr>
<td>BC1_RGBA</td>
<td>VK_FORMAT_BC1_RGBA_UNORM_BLOCK, VK_FORMAT_BC1_RGBA_SRGB_BLOCK</td>
</tr>
<tr>
<td>Block size 8 byte</td>
<td></td>
</tr>
<tr>
<td>4x4x1 block extent</td>
<td></td>
</tr>
<tr>
<td>16 texel/block</td>
<td></td>
</tr>
<tr>
<td>BC2</td>
<td>VK_FORMAT_BC2_UNORM_BLOCK, VK_FORMAT_BC2_SRGB_BLOCK</td>
</tr>
<tr>
<td>Block size 16 byte</td>
<td></td>
</tr>
<tr>
<td>4x4x1 block extent</td>
<td></td>
</tr>
<tr>
<td>16 texel/block</td>
<td></td>
</tr>
<tr>
<td>BC3</td>
<td>VK_FORMAT_BC3_UNORM_BLOCK, VK_FORMAT_BC3_SRGB_BLOCK</td>
</tr>
<tr>
<td>Block size 16 byte</td>
<td></td>
</tr>
<tr>
<td>4x4x1 block extent</td>
<td></td>
</tr>
<tr>
<td>16 texel/block</td>
<td></td>
</tr>
<tr>
<td>Class, Texel Block Size, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>----------------------------------</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 8 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>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, VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK_EXT</td>
</tr>
<tr>
<td>Class, Texel Block Size, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| ASTC_5x4  
Block size 16 byte  
5x4x1 block extent  
20 texel/block | VK_FORMAT_ASTC_5x4_UNORM_BLOCK,  
VK_FORMAT_ASTC_5x4_SRGB_BLOCK,  
VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK_EXT |
| ASTC_5x5  
Block size 16 byte  
5x5x1 block extent  
25 texel/block | VK_FORMAT_ASTC_5x5_UNORM_BLOCK,  
VK_FORMAT_ASTC_5x5_SRGB_BLOCK,  
VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK_EXT |
| ASTC_6x5  
Block size 16 byte  
6x5x1 block extent  
30 texel/block | VK_FORMAT_ASTC_6x5_UNORM_BLOCK,  
VK_FORMAT_ASTC_6x5_SRGB_BLOCK,  
VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK_EXT |
| ASTC_6x6  
Block size 16 byte  
6x6x1 block extent  
36 texel/block | VK_FORMAT_ASTC_6x6_UNORM_BLOCK,  
VK_FORMAT_ASTC_6x6_SRGB_BLOCK,  
VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK_EXT |
| ASTC_8x5  
Block size 16 byte  
8x5x1 block extent  
40 texel/block | VK_FORMAT_ASTC_8x5_UNORM_BLOCK,  
VK_FORMAT_ASTC_8x5_SRGB_BLOCK,  
VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK_EXT |
| ASTC_8x6  
Block size 16 byte  
8x6x1 block extent  
48 texel/block | VK_FORMAT_ASTC_8x6_UNORM_BLOCK,  
VK_FORMAT_ASTC_8x6_SRGB_BLOCK,  
VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK_EXT |
| ASTC_8x8  
Block size 16 byte  
8x8x1 block extent  
64 texel/block | VK_FORMAT_ASTC_8x8_UNORM_BLOCK,  
VK_FORMAT_ASTC_8x8_SRGB_BLOCK,  
VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK_EXT |
| ASTC_10x5  
Block size 16 byte  
10x5x1 block extent  
50 texel/block | VK_FORMAT_ASTC_10x5_UNORM_BLOCK,  
VK_FORMAT_ASTC_10x5_SRGB_BLOCK,  
VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK_EXT |
| ASTC_10x6  
Block size 16 byte  
10x6x1 block extent  
60 texel/block | VK_FORMAT_ASTC_10x6_UNORM_BLOCK,  
VK_FORMAT_ASTC_10x6_SRGB_BLOCK,  
VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK_EXT |
| ASTC_10x8  
Block size 16 byte  
10x8x1 block extent  
80 texel/block | VK_FORMAT_ASTC_10x8_UNORM_BLOCK,  
VK_FORMAT_ASTC_10x8_SRGB_BLOCK,  
VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK_EXT |
<table>
<thead>
<tr>
<th>Class, Texel Block Size, # Texels/Block</th>
<th>Formats</th>
</tr>
</thead>
<tbody>
<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, VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK_EXT</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, VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK_EXT</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, VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK_EXT</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>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>Class, Texel Block Size, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>64-bit R10G10B10A10 Block size 8 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16</td>
</tr>
<tr>
<td>64-bit G10B10G10R10 Block size 8 byte 2x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16</td>
</tr>
<tr>
<td>64-bit B10G10R10G10 Block size 8 byte 2x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16</td>
</tr>
<tr>
<td>10-bit 3-plane 420 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16</td>
</tr>
<tr>
<td>10-bit 2-plane 420 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16</td>
</tr>
<tr>
<td>10-bit 3-plane 422 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16</td>
</tr>
<tr>
<td>10-bit 2-plane 422 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16</td>
</tr>
<tr>
<td>10-bit 3-plane 444 Block size 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16</td>
</tr>
<tr>
<td>64-bit R12G12B12A12 Block size 8 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_R12X4G12X4B12X4A12X4_UNORM_4PACK16</td>
</tr>
<tr>
<td>64-bit G12B12G12R12 Block size 8 byte 2x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16</td>
</tr>
<tr>
<td>Class, Texel Block Size, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>64-bit B12G12R12G12 8 byte 2x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16</td>
</tr>
<tr>
<td>12-bit 3-plane 420 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16</td>
</tr>
<tr>
<td>12-bit 2-plane 420 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16</td>
</tr>
<tr>
<td>12-bit 3-plane 422 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16</td>
</tr>
<tr>
<td>12-bit 2-plane 422 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16</td>
</tr>
<tr>
<td>12-bit 3-plane 444 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16</td>
</tr>
<tr>
<td>64-bit G16B16G16R16 8 byte 2x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G16B16G16R16_422_UNORM</td>
</tr>
<tr>
<td>64-bit B16G16R16G16 8 byte 2x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_B16G16R16G16_422_UNORM</td>
</tr>
<tr>
<td>16-bit 3-plane 420 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM</td>
</tr>
<tr>
<td>16-bit 2-plane 420 6 byte 1x1x1 block extent 1 texel/block</td>
<td>VK_FORMAT_G16_B16R16_2PLANE_420_UNORM</td>
</tr>
<tr>
<td>Class, Texel Block Size, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>16-bit 3-plane 422</td>
<td>VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM</td>
</tr>
<tr>
<td>Block size 6 byte</td>
<td></td>
</tr>
<tr>
<td>1x1x1 block extent</td>
<td></td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>16-bit 2-plane 422</td>
<td>VK_FORMAT_G16_B16R16_2PLANE_422_UNORM</td>
</tr>
<tr>
<td>Block size 6 byte</td>
<td></td>
</tr>
<tr>
<td>1x1x1 block extent</td>
<td></td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>16-bit 3-plane 444</td>
<td>VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM</td>
</tr>
<tr>
<td>Block size 6 byte</td>
<td></td>
</tr>
<tr>
<td>1x1x1 block extent</td>
<td></td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>PVRTC1_2BPP</td>
<td>VK_FORMAT_PVRTC1_2BPP_UNORM_BLOCK_IMG,</td>
</tr>
<tr>
<td>Block size 8 byte</td>
<td></td>
</tr>
<tr>
<td>8x4x1 block extent</td>
<td></td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>PVRTC1_4BPP</td>
<td>VK_FORMAT_PVRTC1_4BPP_UNORM_BLOCK_IMG,</td>
</tr>
<tr>
<td>Block size 8 byte</td>
<td></td>
</tr>
<tr>
<td>4x4x1 block extent</td>
<td></td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>PVRTC2_2BPP</td>
<td>VK_FORMAT_PVRTC2_2BPP_UNORM_BLOCK_IMG,</td>
</tr>
<tr>
<td>Block size 8 byte</td>
<td></td>
</tr>
<tr>
<td>8x4x1 block extent</td>
<td></td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>PVRTC2_4BPP</td>
<td>VK_FORMAT_PVRTC2_4BPP_UNORM_BLOCK_IMG,</td>
</tr>
<tr>
<td>Block size 8 byte</td>
<td></td>
</tr>
<tr>
<td>4x4x1 block extent</td>
<td></td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>8-bit 2-plane 444</td>
<td>VK_FORMAT_G8_B8R8_2PLANE_444_UNORM_EXT</td>
</tr>
<tr>
<td>Block size 3 byte</td>
<td></td>
</tr>
<tr>
<td>1x1x1 block extent</td>
<td></td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>10-bit 2-plane 444</td>
<td>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16_EXT</td>
</tr>
<tr>
<td>Block size 6 byte</td>
<td></td>
</tr>
<tr>
<td>1x1x1 block extent</td>
<td></td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
<tr>
<td>12-bit 2-plane 444</td>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_EXT</td>
</tr>
<tr>
<td>Block size 6 byte</td>
<td></td>
</tr>
<tr>
<td>1x1x1 block extent</td>
<td></td>
</tr>
<tr>
<td>1 texel/block</td>
<td></td>
</tr>
</tbody>
</table>
### 34.2. Format Properties

To query supported format features which are properties of the physical device, call:

```c
// Provided by VK_VERSION_1_0
void vkGetPhysicalDeviceFormatProperties(
    VkPhysicalDevice physicalDevice,
    VkFormat format,
    VkFormatProperties* pFormatProperties);
```

- `physicalDevice` is the physical device from which to query the format properties.
- `format` is the format whose properties are queried.
- `pFormatProperties` is a pointer to a `VkFormatProperties` structure in which physical device properties for `format` are returned.

#### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceFormatProperties-physicalDevice-parameter `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceFormatProperties-format-parameter `format` must be a valid `VkFormat` value
- VUID-vkGetPhysicalDeviceFormatProperties-pFormatProperties-parameter `pFormatProperties` must be a valid pointer to a `VkFormatProperties` structure

The `VkFormatProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkFormatProperties {
    VkFormatFeatureFlags linearTilingFeatures;
    VkFormatFeatureFlags optimalTilingFeatures;
    VkFormatFeatureFlags bufferFeatures;
} VkFormatProperties;
```

- `linearTilingFeatures` is a bitmask of `VkFormatFeatureFlagBits` specifying features supported by images created with a `tiling` parameter of `VK_IMAGE_TILING_LINEAR`.
- `optimalTilingFeatures` is a bitmask of `VkFormatFeatureFlagBits` specifying features supported
by images created with a `tiling` parameter of `VK_IMAGE_TILING_OPTIMAL`.

- `bufferFeatures` is a bitmask of `VkFormatFeatureFlagBits` specifying features supported by buffers.

**Note**

If no format feature flags are supported, the format itself is not supported, and images of that format cannot be created.

If `format` is a block-compressed format, then `bufferFeatures` must not support any features for the format.

If `format` is not a multi-plane format then `linearTilingFeatures` and `optimalTilingFeatures` must not contain `VK_FORMAT_FEATURE_DISJOINT_BIT`.

Bits which can be set in the `VkFormatProperties` features `linearTilingFeatures`, `optimalTilingFeatures`, `VkDrmFormatModifierPropertiesEXT::drmFormatModifierTilingFeatures`, 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,

    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,

    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,

    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,
}
```
These values may be set in `linearTilingFeatures`, `optimalTilingFeatures`, and `VkDrmFormatModifierPropertiesEXT::drmFormatModifierTilingFeatures`, specifying that the features are supported by images or image views or sampler Y’C_bC_r conversion objects created with the queried `vkGetPhysicalDeviceFormatProperties::format`:

- **VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT** specifies that an image view can be sampled from.
- **VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT** specifies that an image view can be used as a storage image.
- **VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT** specifies that an image view can be used as a storage image that supports atomic operations.
- **VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT** specifies that an image view can be used as a framebuffer color attachment and as an input attachment.
- **VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT** specifies that an image view can be used as a framebuffer color attachment that supports blending and as an input attachment.
- **VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT** specifies that an image view can be used as a framebuffer depth/stencil attachment and as an input attachment.
- **VK_FORMAT_FEATURE_BLIT_SRC_BIT** specifies that an image can be used as `srcImage` for the `vkCmdBlitImage2KHR` and `vkCmdBlitImage` commands.
- **VK_FORMAT_FEATURE_BLIT_DST_BIT** specifies that an image can be used as `dstImage` for the `vkCmdBlitImage2KHR` and `vkCmdBlitImage` commands.
- **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT** specifies that if **VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT** is also set, an image view can be used with a sampler that has either of `magFilter` or `minFilter` set to `VK_FILTER_LINEAR`, or `mipmapMode` set to `VK_SAMPLER_MIPMAP_MODE_LINEAR`. If **VK_FORMAT_FEATURE_BLIT_SRC_BIT** is also set, an image can be used as the `srcImage` to `vkCmdBlitImage2KHR` and `vkCmdBlitImage` with a filter of `VK_FILTER_LINEAR`. This bit must only be exposed for formats that also support the **VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT** or **VK_FORMAT_FEATURE_BLIT_SRC_BIT**.
If the format being queried is a depth/stencil format, this bit only specifies that the depth aspect (not the stencil aspect) of an image of this format supports linear filtering, and that linear filtering of the depth aspect is supported whether depth compare is enabled in the sampler or not. Where depth comparison is supported it may be linear filtered whether this bit is present or not, but where this bit is not present the filtered value may be computed in an implementation-dependent manner which differs from the normal rules of linear filtering. The resulting value must be in the range [0,1] and should be proportional to, or a weighted average of, the number of comparison passes or failures.

- **VK_FORMAT_FEATURE_TRANSFER_SRC_BIT** specifies that an image can be used as a source image for copy commands.

- **VK_FORMAT_FEATURE_TRANSFER_DST_BIT** specifies that an image can be used as a destination image for copy commands and clear commands.

- **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT** specifies that a 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_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT** specifies that a VkImage can be used with a sampler that has either of magFilter or minFilter set to VK_FILTER_CUBIC_EXT, or be the source image for a blit with filter set to VK_FILTER_CUBIC_EXT. This bit must only be exposed for formats that also support the VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT. If the format being queried is a depth/stencil format, this only specifies that the depth aspect is cubic filterable.

- **VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT** specifies that an application can define a sampler Y’CbCr conversion using this format as a source, and that an image of this format can be used with a VkSamplerYcbcrConversionCreateInfo with ChromaOffset and/or YChromaOffset of VK_CHROMA_LOCATION_MIDPOINT. Otherwise both xChromaOffset and yChromaOffset must be VK_CHROMA_LOCATION_COSITED_EVEN. If a format does not incorporate chroma downsampling (it is not a “422” or “420” format) but the implementation supports sampler Y’CbCr conversion for this format, the implementation must set VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT.

- **VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT** specifies that an application can define a sampler Y’CbCr conversion using this format as a source, and that an image of this format can be used with a VkSamplerYcbcrConversionCreateInfo with ChromaOffset and/or YChromaOffset of VK_CHROMA_LOCATION_COSITED_EVEN. Otherwise both xChromaOffset and yChromaOffset must be VK_CHROMA_LOCATION_MIDPOINT. If neither VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT nor VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT is set, the application must not define a sampler Y’CbCr conversion using this format as a source.

- **VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT** specifies that an application can define a sampler Y’CbCr 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**
IT 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.

- VK_FORMAT_FEATURE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR specifies that an image view can be used as a fragment shading rate attachment. An implementation must not set this feature for formats with numeric type other than *UINT, or set it as a buffer feature.

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).

```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 the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **formatProperties** is a VkFormatProperties structure describing features supported by the requested format.

### Valid Usage (Implicit)

- VUID-VkFormatProperties2-sType-sType  
  **sType** must be VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_2

- VUID-VkFormatProperties2-pNext-pNext  
  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 VkDrmFormatModifierPropertiesList2EXT or VkDrmFormatModifierPropertiesListEXT

- VUID-VkFormatProperties2-sType-unique  
  The **sType** value of each struct in the **pNext** chain must be unique

To obtain the list of Linux DRM format modifiers compatible with a VkFormat, add a VkDrmFormatModifierPropertiesListEXT structure to the **pNext** chain of VkFormatProperties2.

The VkDrmFormatModifierPropertiesListEXT structure is defined as:

```c
// Provided by VK_EXT_image_drm_format_modifier
```
typedef struct VkDrmFormatModifierPropertiesListEXT {
    VkStructureType sType;
    void* pNext;
    uint32_t drmFormatModifierCount;
    VkDrmFormatModifierPropertiesEXT* pDrmFormatModifierProperties;
} VkDrmFormatModifierPropertiesListEXT;

• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **drmFormatModifierCount** is an inout parameter related to the number of modifiers compatible with the format, as described below.
• **pDrmFormatModifierProperties** is either NULL or a pointer to an array of VkDrmFormatModifierPropertiesEXT structures.

If pDrmFormatModifierProperties is NULL, then the function returns in drmFormatModifierCount the number of modifiers compatible with the queried format. Otherwise, the application must set drmFormatModifierCount to the length of the array pDrmFormatModifierProperties; the function will write at most drmFormatModifierCount elements to the array, and will return in drmFormatModifierCount the number of elements written.

Among the elements in array pDrmFormatModifierProperties, each returned drmFormatModifier must be unique.

**Valid Usage (Implicit)**

• VUID-VkDrmFormatModifierPropertiesListEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_DRM_FORMAT_MODIFIER_PROPERTIES_LIST_EXT

The VkDrmFormatModifierPropertiesEXT structure describes properties of a VkFormat when that format is combined with a Linux DRM format modifier. These properties, like those of VkFormatProperties2, are independent of any particular image.

The VkDrmFormatModifierPropertiesEXT structure is defined as:

```c
// Provided by VK_EXT_image_drm_format_modifier
typedef struct VkDrmFormatModifierPropertiesEXT {
    uint64_t drmFormatModifier;
    uint32_t drmFormatModifierPlaneCount;
    VkFormatFeatureFlags drmFormatModifierTilingFeatures;
} VkDrmFormatModifierPropertiesEXT;
```

• **drmFormatModifier** is a Linux DRM format modifier.
• **drmFormatModifierPlaneCount** is the number of memory planes in any image created with format and drmFormatModifier. An image's memory plane count is distinct from its format plane count, as explained below.
• **drmFormatModifierTilingFeatures** is a bitmask of `VkFormatFeatureFlagBits` that are supported by any image created with `format` and `drmFormatModifier`.

The returned **drmFormatModifierTilingFeatures** must contain at least one bit.

The implementation must not return `DRM_FORMAT_MOD_INVALID` in `drmFormatModifier`.

An image's **memory planecount** (as returned by `drmFormatModifierPlaneCount`) is distinct from its **format planecount** (in the sense of multi-planar Y'CbCr formats). In `VkImageAspectFlags`, each `VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT` represents a **memory plane** and each `VK_IMAGE_ASPECT_PLANE_i_BIT` a **format plane**.

An image's set of **format planes** is an ordered partition of the image's **content** into separable groups of format components. The ordered partition is encoded in the name of each `VkFormat`. For example, `VK_FORMAT_G8_B8R8_2PLANE_420_UNORM` contains two **format planes**; the first plane contains the green component and the second plane contains the blue component and red component. If the format name does not contain `PLANE`, then the format contains a single plane; for example, `VK_FORMAT_R8G8B8A8_UNORM`. Some commands, such as `vkCmdCopyBufferToImage`, do not operate on all format components in the image, but instead operate only on the **format planes** explicitly chosen by the application and operate on each **format plane** independently.

An image's set of **memory planes** is an ordered partition of the image's **memory** rather than the image's **content**. Each **memory plane** is a contiguous range of memory. The union of an image's **memory planes** is not necessarily contiguous.

If an image is **linear**, then the partition is the same for **memory planes** and for **format planes**. Therefore, if the returned `drmFormatModifier` is `DRM_FORMAT_MOD_LINEAR`, then `drmFormatModifierPlaneCount` must equal the **format planecount**, and `drmFormatModifierTilingFeatures` must be identical to the `VkFormatProperties2::linearTilingFeatures` returned in the same pNext chain.

If an image is **non-linear**, then the partition of the image's **memory** into **memory planes** is implementation-specific and may be unrelated to the partition of the image's **content** into **format planes**. For example, consider an image whose **format** is `VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM`, tiling is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, whose `drmFormatModifier` is not `DRM_FORMAT_MOD_LINEAR`, and flags lacks `VK_IMAGE_CREATE_DISJOINT_BIT`. The image has 3 **format planes**, and commands such as `vkCmdCopyBufferToImage` act on each **format plane** independently as if the data of each **format plane** were separable from the data of the other planes. In a straightforward implementation, the implementation may store the image's content in 3 adjacent **memory planes** where each **memory plane** corresponds exactly to a **format plane**. However, the implementation may also store the image's content in a single **memory plane** where all format components are combined using an implementation-private block-compressed format; or the implementation may store the image's content in a collection of 7 adjacent **memory planes** using an implementation-private sharding technique. Because the image is non-linear and non-disjoint, the implementation has much freedom when choosing the image's placement in memory.

The **memory planecount** applies to function parameters and structures only when the API specifies an explicit requirement on `drmFormatModifierPlaneCount`. In all other cases, the **memory planecount** is ignored.
34.2.1. Potential Format Features

Some valid usage conditions depend on the format features supported by an 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, VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT, or VK_IMAGE_TILING_LINEAR.

34.3. Required Format Support

Implementations must support at least the following set of features on the listed formats. For images, these features must be supported for every VkImageType (including arrayed and cube variants) unless otherwise noted. These features are supported on existing formats without needing to advertise an extension or needing to explicitly enable them. Support for additional functionality beyond the requirements listed here is queried using the vkGetPhysicalDeviceFormatProperties command.

Note

Unless otherwise excluded below, the required formats are supported for all VkImageCreateFlags values as long as those flag values are otherwise allowed.

The following tables show which feature bits must be supported for each format. Formats that are required to support VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT must also support VK_FORMAT_FEATURE_TRANSFER_SRC_BIT and VK_FORMAT_FEATURE_TRANSFER_DST_BIT.

Table 56. 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 57. 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>Feature Bits</td>
</tr>
<tr>
<td>-----------------------------------</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 58. Feature bits in `bufferFeatures` |
<table>
<thead>
<tr>
<th></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 59. Mandatory format support: sub-byte components

<table>
<thead>
<tr>
<th>Format Features</th>
<th>Format Features</th>
<th>Format Features</th>
<th>Format Features</th>
<th>Format Features</th>
<th>Format Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_ATOMIC_BIT</td>
</tr>
</tbody>
</table>

Format features marked † must be supported for optimalTilingFeatures if the VkPhysicalDevice supports the VkPhysicalDevice4444FormatsFeaturesEXT::formatA4R4G4B4 feature.

Format features marked ‡ must be supported for optimalTilingFeatures if the VkPhysicalDevice supports the VkPhysicalDevice4444FormatsFeaturesEXT::formatA4B4G4R4 feature.
Table 60. Mandatory format support: 1-3 byte-sized components

<table>
<thead>
<tr>
<th>Feature</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_R8_UNORM</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</td>
<td>VK_FORMAT_R8_SNORM</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</td>
<td>VK_FORMAT_R8_USCALED</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT</td>
<td>VK_FORMAT_R8_SSCALED</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
<td>VK_FORMAT_R8_UINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</td>
<td>VK_FORMAT_R8_SINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_DST_BIT</td>
<td>VK_FORMAT_R8_SRGB</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLEND_DST_BIT</td>
<td>VK_FORMAT_R8G8_UNORM</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</td>
<td>VK_FORMAT_R8G8_SNORM</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_DST_BIT</td>
<td>VK_FORMAT_R8G8_USCALED</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLEND_DST_BIT</td>
<td>VK_FORMAT_R8G8_SSCALED</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLEND_DST_BIT</td>
<td>VK_FORMAT_R8G8_UINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLEND_DST_BIT</td>
<td>VK_FORMAT_R8G8_SINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLEND_DST_BIT</td>
<td>VK_FORMAT_R8G8_SRGB</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLEND_DST_BIT</td>
<td>VK_FORMAT_R8G8B8_UNORM</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLEND_DST_BIT</td>
<td>VK_FORMAT_R8G8B8_SNORM</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLEND_DST_BIT</td>
<td>VK_FORMAT_R8G8B8_USCALED</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLEND_DST_BIT</td>
<td>VK_FORMAT_R8G8B8_SSCALED</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLEND_DST_BIT</td>
<td>VK_FORMAT_R8G8B8_UINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLEND_DST_BIT</td>
<td>VK_FORMAT_R8G8B8_SINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLEND_DST_BIT</td>
<td>VK_FORMAT_R8G8B8_SRGB</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLEND_DST_BIT</td>
<td>VK_FORMAT_B8G8R8_UNORM</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLEND_DST_BIT</td>
<td>VK_FORMAT_B8G8R8_SNORM</td>
</tr>
</tbody>
</table>
Format features marked with ‡ must be supported for optimalTilingFeatures if the VkPhysicalDevice supports the shaderStorageImageExtendedFormats feature.
### Table 61. Mandatory format support: 4 byte-sized components

<table>
<thead>
<tr>
<th>Format</th>
<th>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</th>
<th>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</th>
<th>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</th>
<th>VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT</th>
<th>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</th>
<th>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</th>
<th>VK_FORMAT_FEATURE_BLIT_DST_BIT</th>
<th>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</th>
<th>VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</th>
<th>VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT</th>
<th>VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</th>
<th>VK_FORMAT_FEATURE_BLIT_SRC_BIT</th>
<th>VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_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>
</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>
</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>
</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>
</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>
</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>
</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>
</tr>
<tr>
<td>VK_FORMAT_A8B8G8R8B8_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>
</tr>
<tr>
<td>VK_FORMAT_A8B8G8R8B8_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>
</tr>
<tr>
<td>VK_FORMAT_A8B8G8R8B8_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>
</tr>
<tr>
<td>VK_FORMAT_A8B8G8R8B8_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>
</tr>
<tr>
<td>VK_FORMAT_A8B8G8R8B8_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>
</tr>
<tr>
<td>VK_FORMAT_A8B8G8R8B8_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>
</tr>
<tr>
<td>VK_FORMAT_A8B8G8R8B8_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>
</tr>
</tbody>
</table>
Table 62. Mandatory format support: 10- and 12-bit components

<table>
<thead>
<tr>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</td>
<td>VK_FORMAT_FEATURE_BLIT_DST_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</td>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT</td>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</td>
<td>VK_FORMAT_FEATURE_BLIT_SRC_BIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT</td>
</tr>
</tbody>
</table>

Format

- VK_FORMAT_A2R10G10B10_UNORM_PACK32
- VK_FORMAT_A2R10G10B10_SNORM_PACK32
- VK_FORMAT_A2R10G10B10_USCALED_PACK32
- VK_FORMAT_A2R10G10B10_SSCALED_PACK32
- VK_FORMAT_A2R10G10B10_UINT_PACK32
- VK_FORMAT_A2R10G10B10_SINT_PACK32
- VK_FORMAT_A2B10G10R10_UNORM_PACK32
- VK_FORMAT_A2B10G10R10_SNORM_PACK32
- VK_FORMAT_A2B10G10R10_USCALED_PACK32
- VK_FORMAT_A2B10G10R10_SSCALED_PACK32
- VK_FORMAT_A2B10G10R10_UINT_PACK32
- VK_FORMAT_A2B10G10R10_SINT_PACK32
- VK_FORMAT_R10X6_UNORM_PACK16
- VK_FORMAT_R10X6G10X6_UNORM_2PACK16
- VK_FORMAT_R12X4_UNORM_PACK16
- VK_FORMAT_R12X4G12X4_UNORM_2PACK16

Format features marked with ‡ must be supported for optimalTilingFeatures if the VkPhysicalDevice supports the shaderStorageImageExtendedFormats feature.
Table 63. Mandatory format support: 16-bit components

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
<td>‡</td>
</tr>
<tr>
<td>Format Features</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16A16_USCALED</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16A16_SSCALED</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16A16_UINT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16A16_SINT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16A16_SFLOAT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format features marked with ‡ **must** be supported for `optimalTilingFeatures` if the `VkPhysicalDevice` supports the `shaderStorageImageExtendedFormats` feature.
Table 64. 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>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VK_FORMAT_R32_SINT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VK_FORMAT_R32_SFLOAT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32_UINT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32_SINT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32_SFLOAT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32B32_UINT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32B32_SINT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32B32_SFLOAT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32B32A32_UINT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32B32A32_SINT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32B32A32_SFLOAT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

If the shaderImageFloat32Atomics or the shaderImageFloat32AtomicAdd feature is supported, VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT and VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT must be advertised in optimalTilingFeatures for VK_FORMAT_R32_SFLOAT.
### Table 65. Mandatory format support: 64-bit/uneven components

<table>
<thead>
<tr>
<th>Format features</th>
<th>(\text{VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT} )</th>
<th>(\text{VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT} )</th>
<th>(\text{VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT} )</th>
<th>(\text{VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</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>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_DST_BIT</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>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</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>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</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>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

| Format features marked with † | must be supported for optimalTilingFeatures if the VkPhysicalDevice supports the shaderStorageImageExtendedFormats feature. |

If the shaderImageInt64Atomics feature is supported, VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT and VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT must be advertised in optimalTilingFeatures for both VK_FORMAT_R64_UINT and VK_FORMAT_R64_SINT.
## Table 66. Mandatory format support: depth/stencil with VkImageType VK_IMAGE_TYPE_2D

<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_D16_UNORM</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_X8_D24_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>
</tr>
<tr>
<td>VK_FORMAT_D32_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_S8_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_D16_UNORM_S8_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_D24_UNORM_S8_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_D32_SFLOAT_S8_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>
</tbody>
</table>

**bufferFeatures must not support any features for these formats.**

**feature must** be supported for at least one of **VK_FORMAT_X8_D24_UNORM_PACK32** and **VK_FORMAT_D32_SFLOAT**, and **must** be supported for at least one of **VK_FORMAT_D24_UNORM_S8_UINT** and **VK_FORMAT_D32_SFLOAT_S8_UINT**.
Table 67. Mandatory format support: BC compressed formats with VkImageType VK_IMAGE_TYPE_2D and VK_IMAGE_TYPE_3D

<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_TEXTURE_ATTACHMENT_BLEND_BIT</th>
<th>VK_FORMAT_FEATURE BLIT_DST_BIT</th>
<th>VK_FORMAT_FEATURE_TEXTURE_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_FILTER_LINEAR_BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_BC1_RGB_UNORM_BLOCK</td>
<td>✦ ✦ ✦</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGB_SRGB_BLOCK</td>
<td>✦ ✦ ✦</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGBA_UNORM_BLOCK</td>
<td>✦ ✦ ✦</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGBA_SRGB_BLOCK</td>
<td>✦ ✦ ✦</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC2_UNORM_BLOCK</td>
<td>✦ ✦ ✦</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC2_SRGB_BLOCK</td>
<td>✦ ✦ ✦</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC3_UNORM_BLOCK</td>
<td>✦ ✦ ✦</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC3_SNORM_BLOCK</td>
<td>✦ ✦ ✦</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC4_UNORM_BLOCK</td>
<td>✦ ✦ ✦</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC4_SNORM_BLOCK</td>
<td>✦ ✦ ✦</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC5_UNORM_BLOCK</td>
<td>✦ ✦ ✦</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC5_SNORM_BLOCK</td>
<td>✦ ✦ ✦</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC6H_UFLOAT_BLOCK</td>
<td>✦ ✦ ✦</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC6H_SFLOAT_BLOCK</td>
<td>✦ ✦ ✦</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC7_UNORM_BLOCK</td>
<td>✦ ✦ ✦</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC7_SRGB_BLOCK</td>
<td>✦ ✦ ✦</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</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: ETC2 and EAC compressed formats with VkImageType VK_IMAGE_TYPE_2D, or Mandatory format support: ASTC LDR compressed formats with VkImageType VK_IMAGE_TYPE_2D.
Table 68. Mandatory format support: ETC2 and EAC compressed formats with \texttt{VkImageType}

\texttt{VK\_IMAGE\_TYPE\_2D}

<table>
<thead>
<tr>
<th>Format</th>
<th>\texttt{VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT}</th>
<th>\texttt{VK_FORMAT_FEATURE_BLIT_SRC_BIT}</th>
<th>\texttt{VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK}</td>
<td>† † †</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK}</td>
<td>† † †</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK}</td>
<td>† † †</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK}</td>
<td>† † †</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK}</td>
<td>† † †</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK}</td>
<td>† † †</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_EAC_R11_UNORM_BLOCK}</td>
<td>† † †</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_EAC_R11_SNORM_BLOCK}</td>
<td>† † †</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_EAC_R11G11_UNORM_BLOCK}</td>
<td>† † †</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_EAC_R11G11_SNORM_BLOCK}</td>
<td>† † †</td>
<td></td>
<td></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: BC compressed formats with \texttt{VkImageType \texttt{VK\_IMAGE\_TYPE\_2D}} and \texttt{VK\_IMAGE\_TYPE\_3D}, or Mandatory format support: ASTC LDR compressed formats with \texttt{VkImageType \texttt{VK\_IMAGE\_TYPE\_2D}}.
<table>
<thead>
<tr>
<th>Format</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_ASTC_4x4_UNORM_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_4x4_SRGB_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x4_UNORM_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x4_SRGB_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x5_UNORM_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x5_SRGB_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x5_UNORM_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x5_SRGB_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x6_UNORM_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x6_SRGB_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x5_UNORM_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x5_SRGB_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x6_UNORM_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x6_SRGB_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x8_UNORM_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x8_SRGB_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x5_UNORM_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x5_SRGB_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x6_UNORM_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x6_SRGB_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x8_UNORM_BLOCK</td>
<td>†††</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 69. Mandatory format support: ASTC LDR compressed formats with VkImageType VK_IMAGE_TYPE_2D
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 
on `optimalTilingFeatures` for all the formats in at least one of: this table, Mandatory format support: 
BC compressed formats with `VkImageType VK_IMAGE_TYPE_2D and VK_IMAGE_TYPE_3D`, or Mandatory 
format support: ETC2 and EAC compressed formats with `VkImageType VK_IMAGE_TYPE_2D`.

If cubic filtering is supported, `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT` must be 
supported for the following image view types:

- `VK_IMAGE_VIEW_TYPE_2D`
- `VK_IMAGE_VIEW_TYPE_2D_ARRAY`

for the following formats:

- `VK_FORMAT_R4G4_UNORM_PACK8`
- `VK_FORMAT_R4G4B4A4_UNORM_PACK16`
- `VK_FORMAT_B4G4R4A4_UNORM_PACK16`
- `VK_FORMAT_R5G6B5_UNORM_PACK16`
- `VK_FORMAT_B5G6R5_UNORM_PACK16`
- `VK_FORMAT_R5G5B5A1_UNORM_PACK16`
- `VK_FORMAT_B5G5R5A1_UNORM_PACK16`
- `VK_FORMAT_A1R5G5B5_UNORM_PACK16`
- `VK_FORMAT_R8_UNORM`
- `VK_FORMAT_R8_SNORM`
- `VK_FORMAT_R8_SRGB`
- `VK_FORMAT_R8G8_UNORM`
- `VK_FORMAT_R8G8_SNORM`
- `VK_FORMAT_R8G8_SRGB`
- `VK_FORMAT_R8G8B8_UNORM`
- `VK_FORMAT_R8G8B8_SNORM`
- `VK_FORMAT_R8G8B8_SRGB`
- `VK_FORMAT_B8G8R8_UNORM`
• VK_FORMAT_B8G8R8_SNORM
• VK_FORMAT_B8G8R8_SRGB
• VK_FORMAT_R8G8B8A8_UNORM
• VK_FORMAT_R8G8B8A8_SNORM
• VK_FORMAT_R8G8B8A8_SRGB
• VK_FORMAT_B8G8R8A8_UNORM
• VK_FORMAT_B8G8R8A8_SNORM
• VK_FORMAT_B8G8R8A8_SRGB
• VK_FORMAT_A8B8G8R8_UNORM_PACK32
• VK_FORMAT_A8B8G8R8_SNORM_PACK32
• VK_FORMAT_A8B8G8R8_SRGB_PACK32

If ETC compressed formats are supported, **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT** must be supported for the following image view types:

• VK_IMAGE_VIEW_TYPE_2D
• VK_IMAGE_VIEW_TYPE_2D_ARRAY

for the following additional formats:

• VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK
• VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK
• VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK
• VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK
• VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK
• VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK

If cubic filtering is supported for any other formats, the following image view types **must** be supported for those formats:

• VK_IMAGE_VIEW_TYPE_2D
• VK_IMAGE_VIEW_TYPE_2D_ARRAY

To be used with **VkImageView** with subresourceRange.aspectMask equal to **VK_IMAGE_ASPECT_COLOR_BIT**, **sampler Y’C_bC_r** conversion must be enabled for the following formats:
<table>
<thead>
<tr>
<th>Format</th>
<th>Planes</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_G8B8G8R8_422_UNORM</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8G8_422_UNORM</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G8_B8R8_2PLANE_420_UNORM</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G8_B8R8_2PLANE_422_UNORM</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G8_B8_3PLANE_444_UNORM</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_R12X4G12X4B12X4A12X4_UNORM_4PACK16</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16</td>
<td>3</td>
</tr>
<tr>
<td>Format</td>
<td>Count</td>
</tr>
<tr>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td>VK_FORMAT_G16B16G16R16_422_UNORM</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_B16G16R16G16_422_UNORM</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G16_B16R16_2PLANE_420_UNORM</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G16_B16R16_2PLANE_422_UNORM</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G8_B8R8_2PLANE_444_UNORM_EXT</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16_EX</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_EX</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G16_B16R16_2PLANE_444_UNORM_EXT</td>
<td>2</td>
</tr>
</tbody>
</table>

Format features marked † must be supported for optimalTilingFeatures with VkImageType VK_IMAGE_TYPE_2D if the VkPhysicalDevice supports the VkPhysicalDeviceSamplerYcbcrConversionFeatures feature.

Implementations are not required to support the VK_IMAGE_CREATE_SPARSE_BINDING_BIT, VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT, or VK_IMAGE_CREATE_SPARSE_ALIASED_BIT VkImageCreateFlags for the above formats that require sampler Y′CgC_r conversion. To determine whether the implementation supports sparse image creation flags with these formats use vkGetPhysicalDeviceImageFormatProperties or vkGetPhysicalDeviceImageFormatProperties2.

**VK_FORMAT_FEATURE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR** must be supported for the following formats if the attachmentFragmentShadingRate feature is supported:

- VK_FORMAT_R8_UINT

### 34.3.1. Formats without shader storage format

The device-level features for using a storage image with an image format of Unknown, shaderStorageImageReadWithoutFormat and shaderStorageImageWriteWithoutFormat, only apply to the following formats:

- VK_FORMAT_R8G8B8A8_UNORM
- VK_FORMAT_R8G8B8A8_SNORM
- VK_FORMAT_R8G8B8A8_UINT
- VK_FORMAT_R8G8B8A8_SINT
- VK_FORMAT_R32_UINT
- VK_FORMAT_R32_SINT
- VK_FORMAT_R32_SFLOAT
- VK_FORMAT_R32G32_UINT
- VK_FORMAT_R32G32_SINT
• VK_FORMAT_R32G32_SFLOAT
• VK_FORMAT_R32G32B32A32_UINT
• VK_FORMAT_R32G32B32A32_SINT
• VK_FORMAT_R32G32B32A32_SFLOAT
• VK_FORMAT_R16G16B16A16_UINT
• VK_FORMAT_R16G16B16A16_SINT
• VK_FORMAT_R16G16B16A16_SFLOAT
• VK_FORMAT_R16G16_SFLOAT
• VK_FORMAT_B10G11R11_UFLOAT_PACK32
• VK_FORMAT_R16_SFLOAT
• VK_FORMAT_R16G16B16A16_UNORM
• VK_FORMAT_A2B10G10R10_UNORM_PACK32
• VK_FORMAT_R16G16_UNORM
• VK_FORMAT_R8G8_UNORM
• VK_FORMAT_R16_UNORM
• VK_FORMAT_R8_UNORM
• VK_FORMAT_R16G16B16A16_SNORM
• VK_FORMAT_R16G16_SNORM
• VK_FORMAT_R8G8_SNORM
• VK_FORMAT_R16_SNORM
• VK_FORMAT_R8_SNORM
• VK_FORMAT_R16G16_SINT
• VK_FORMAT_R8G8_SINT
• VK_FORMAT_R16_SINT
• VK_FORMAT_R8_SINT
• VK_FORMAT_A2B10G10R10_UINT_PACK32
• VK_FORMAT_R16G16_UINT
• VK_FORMAT_R16G8_UINT
• VK_FORMAT_R16_UINT
• VK_FORMAT_R8_UINT

**Note**

This list of formats is the union of required storage formats from Required Format Support section and formats listed in shaderStorageImageExtendedFormats.
Chapter 35. Additional Capabilities

This chapter describes additional capabilities beyond the minimum capabilities described in the Limits and Formats chapters, including:

- Additional Image Capabilities
- Additional Buffer Capabilities
- Optional Semaphore Capabilities
- Optional Fence Capabilities
- Timestamp Calibration Capabilities

35.1. Additional Image Capabilities

Additional image capabilities, such as larger dimensions or additional sample counts for certain image types, or additional capabilities for linear tiling format images, are described in this section.

To query additional capabilities specific to image types, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkGetPhysicalDeviceImageFormatProperties(
    VkPhysicalDevice physicalDevice,
    VkFormat format,
    VkImageType type,
    VkImageTiling tiling,
    VkImageUsageFlags usage,
    VkImageCreateFlags flags,
    VkImageFormatProperties* pImageFormatProperties);
```

- `physicalDevice` is the physical device from which to query the image capabilities.
- `format` is a `VkFormat` value specifying the image format, corresponding to `VkImageCreateInfo::format`.
- `type` is a `VkImageType` value specifying the image type, corresponding to `VkImageCreateInfo::imageType`.
- `tiling` is a `VkImageTiling` value specifying the image tiling, corresponding to `VkImageCreateInfo::tiling`.
- `usage` is a bitmask of `VkImageUsageFlagBits` specifying the intended usage of the image, corresponding to `VkImageCreateInfo::usage`.
- `flags` is a bitmask of `VkImageCreateFlagBits` specifying additional parameters of the image, corresponding to `VkImageCreateInfo::flags`.
- `pImageFormatProperties` is a pointer to a `VkImageFormatProperties` structure in which capabilities are returned.

The `format`, `type`, `tiling`, `usage`, and `flags` parameters correspond to parameters that would be
consumed by `vkCreateImage` (as members of `VkImageCreateInfo`).

If `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**

- VUID-vkGetPhysicalDeviceImageFormatProperties-tiling-02248
  
  tiling must not be `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`. (Use `vkGetPhysicalDeviceImageFormatProperties2` instead)

**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

  
  pImageFormatProperties must be a valid pointer to a `VkImageFormatProperties` structure
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY
• VK_ERROR_FORMAT_NOT_SUPPORTED

The VkImageFormatProperties structure is defined as:

```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
  ◦ `VkPhysicalDeviceImageFormatInfo2::tiling` was VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT
  ◦ 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'C_bC_r 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'C_bC_r conversion

• If tiling is VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT, then **maxArrayLayers must** not be 0.

• **sampleCounts** is a bitmask of VkSampleCountFlagBits specifying all the supported sample counts.
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

  `pImageFormatProperties` must be a valid pointer to a `VkImageFormatProperties2` structure

### Return Codes

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_FORMAT_NOT_SUPPORTED`

The `VkPhysicalDeviceImageFormatInfo2` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceImageFormatInfo2 {
    VkStructureType sType;
    const void* pNext;
    VkFormat format;
    VkImageType type;
    VkImageTiling tiling;
    VkImageUsageFlags usage;
    VkImageCreateFlags flags;
} VkPhysicalDeviceImageFormatInfo2;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure. The `pNext` chain of `VkPhysicalDeviceImageFormatInfo2` is used to provide additional image parameters to `vkGetPhysicalDeviceImageFormatProperties2`.
- **format** is a `VkFormat` value indicating the image format, corresponding to `VkImageCreateInfo::format`.
- **type** is a `VkImageType` value indicating the image type, corresponding to `VkImageCreateInfo::imageType`.
- **tiling** is a `VkImageTiling` value indicating the image tiling, corresponding to `VkImageCreateInfo::tiling`.
- **usage** is a bitmask of `VkImageUsageFlagBits` indicating the intended usage of the image, corresponding to `VkImageCreateInfo::usage`.
• flags is a bitmask of VkImageCreateFlagBits indicating additional parameters of the image, corresponding to VkImageCreateInfo::flags.

The members of VkPhysicalDeviceImageFormatInfo2 correspond to the arguments to vkGetPhysicalDeviceImageFormatProperties, with sType and pNext added for extensibility.

---

**Valid Usage**

- **VUID-VkPhysicalDeviceImageFormatInfo2-tiling-02249**
  
  tiling must be VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT if and only if the pNext chain includes VkPhysicalDeviceImageDrmFormatModifierInfoEXT.

- **VUID-VkPhysicalDeviceImageFormatInfo2-tiling-02313**
  
  If tiling is VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT and flags contains VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT, then the pNext chain must include a VkImageFormatListCreateInfo structure with non-zero viewFormatCount.

---

**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, VkPhysicalDeviceExternalImageFormatInfo, VkPhysicalDeviceImageDrmFormatModifierInfoEXT, or VkPhysicalDeviceImageViewImageFormatInfoEXT.

- **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:
typedef struct VkImageFormatProperties2 {
    VkStructureType     sType;
    void*                pNext;
    VkImageFormatProperties imageFormatProperties;
} VkImageFormatProperties2;

• **sType** is the type of this structure.
• **pNext** is ** NULL** or a pointer to a structure extending this structure. The **pNext** chain of VkImageFormatProperties2 is used to allow the specification of additional capabilities to be returned from `vkGetPhysicalDeviceImageFormatProperties2`.
• **imageFormatProperties** is a VkImageFormatProperties structure in which capabilities are returned.

If the combination of parameters to `vkGetPhysicalDeviceImageFormatProperties2` is not supported by the implementation for use in `vkCreateImage`, then all members of `imageFormatProperties` will be filled with zero.

Note
Filling `imageFormatProperties` with zero for unsupported formats is an exception to the usual rule that output structures have undefined contents on error. This exception was unintentional, but is preserved for backwards compatibility. This exception only applies to `imageFormatProperties`, not `sType`, `pNext`, or any structures chained from `pNext`.

Valid Usage (Implicit)
• VUID-VkImageFormatProperties2-sType-sType
  *sType must be VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2*

• VUID-VkImageFormatProperties2-pNext-pNext
  Each `pNext` member of any structure (including this one) in the `pNext` chain must be either ** NULL** or a pointer to a valid instance of `VkExternalImageFormatProperties`, `VkFilterCubicImageViewImageFormatPropertiesEXT`, 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:
typedef struct VkPhysicalDeviceExternalImageFormatInfo {
    VkStructureType sType;
    const void* pNext;
    VkExternalMemoryHandleTypeFlagBits handleType;
} VkPhysicalDeviceExternalImageFormatInfo;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **handleType** is a VkExternalMemoryHandleTypeFlagBits value specifying the memory handle type that will be used with the memory associated with the image.

If **handleType** is 0, `vkGetPhysicalDeviceImageFormatProperties2` will behave as if `VkPhysicalDeviceExternalImageFormatInfo` was not present, and `VkExternalImageFormatProperties` will be ignored.

If **handleType** is not compatible with the `format`, `type`, `tiling`, `usage`, and `flags` specified in `VkPhysicalDeviceImageFormatInfo2`, then `vkGetPhysicalDeviceImageFormatProperties2` returns `VK_ERROR_FORMAT_NOT_SUPPORTED`.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceExternalImageFormatInfo-sType-sType
  - **sType** must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_IMAGE_FORMAT_INFO`

- VUID-VkPhysicalDeviceExternalImageFormatInfo-handleType-parameter
  - If **handleType** is not 0, **handleType** must be a valid `VkExternalMemoryHandleTypeFlagBits` value

Possible values of `VkPhysicalDeviceExternalImageFormatInfo::handleType`, specifying an external memory handle type, are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkExternalMemoryHandleTypeFlagBits {
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT = 0x00000001,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT = 0x00000002,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT = 0x00000004,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT = 0x00000008,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT = 0x00000010,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT = 0x00000020,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT = 0x00000040,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT = 0x00000200,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT = 0x00000080,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT = 0x00000100,
} 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.

• **VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT** specifies a host pointer returned by a host memory allocation command. It does not own a reference to the underlying memory resource, and will therefore become invalid if the host memory is freed.

• **VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT** specifies a host pointer to host mapped foreign memory. It does not own a reference to the underlying memory resource, and will therefore become invalid if the foreign memory is unmapped or otherwise becomes no longer available.

• **VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT** is a file descriptor for a Linux dma_buf. It owns a reference to the underlying memory resource represented by its Vulkan memory object.

• **VK_EXTERNAL_MEMORY_HANDLE_TYPE_SCI_BUF_BIT_NV** specifies a volatile memory object (NvSciBufObj) that is backed by a buffer and shareable across various hardware engines including the CPU,
and software (intra-process and inter-process) and hardware (system memory) operating domains.
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 71. 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>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_SCI_BUF_BIT_NV</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
</tbody>
</table>

**Note**

The above table does not restrict the drivers and devices with which VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT and VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT may be shared, as these handle types inherently mean memory that does not come from the same device, as they import memory from the host or a foreign device, respectively.

**Note**

Even though the above table does not restrict the drivers and devices with which VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT may be shared, query mechanisms exist in the Vulkan API that prevent the import of incompatible dma-bufs (such as vkGetMemoryFdPropertiesKHR) and that prevent incompatible usage of dma-bufs (such as VkPhysicalDeviceExternalBufferInfo and VkPhysicalDeviceExternalImageFormatInfo).
typedef VkFlags VkExternalMemoryHandleTypeFlags;

VkExternalMemoryHandleTypeFlags is a bitmask type for setting a mask of zero or more VkExternalMemoryHandleTypeFlagBits.

The VkExternalImageFormatProperties structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkExternalImageFormatProperties {
    VkStructureType sType;
    void* pNext;
    VkExternalMemoryProperties externalMemoryProperties;
} VkExternalImageFormatProperties;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `externalMemoryProperties` is a VkExternalMemoryProperties structure specifying various capabilities of the external handle type when used with the specified image creation parameters.

### Valid Usage (Implicit)

- VUID-VkExternalImageFormatProperties-sType-sType
  - `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**

Implementations must not report **VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT** for images or buffers with external handle type **VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT**, or **VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT**.

```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 query the image capabilities that are compatible with a Linux DRM format modifier, set VkPhysicalDeviceImageFormatInfo2::tiling to **VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT** and add a VkPhysicalDeviceImageDrmFormatModifierInfoEXT structure to the pNext chain of VkPhysicalDeviceImageFormatInfo2.
The `VkPhysicalDeviceImageDrmFormatModifierInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_image_drm_format_modifier
typedef struct VkPhysicalDeviceImageDrmFormatModifierInfoEXT {
    VkStructureType sType;
    const void* pNext;
    uint64_t drmFormatModifier;
    VkSharingMode sharingMode;
    uint32_t queueFamilyIndexCount;
    const uint32_t* pQueueFamilyIndices;
} VkPhysicalDeviceImageDrmFormatModifierInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `drmFormatModifier` is the image's Linux DRM format modifier, corresponding to `VkImageDrmFormatModifierExplicitCreateInfoEXT::modifier` or to `VkImageDrmFormatModifierListCreateInfoEXT::pModifiers`.
- `sharingMode` specifies how the image will be accessed by multiple queue families.
- `queueFamilyIndexCount` is the number of entries in the `pQueueFamilyIndices` array.
- `pQueueFamilyIndices` is a pointer to an array of queue families that will access the image. It is ignored if `sharingMode` is not `VK_SHARING_MODE_CONCURRENT`.

If the `drmFormatModifier` is incompatible with the parameters specified in `VkPhysicalDeviceImageFormatInfo2` and its `pNext` chain, then `vkGetPhysicalDeviceImageFormatProperties2` returns `VK_ERROR_FORMAT_NOT_SUPPORTED`. The implementation must support the query of any `drmFormatModifier`, including unknown and invalid modifier values.

### Valid Usage

- VUID-VkPhysicalDeviceImageDrmFormatModifierInfoEXT-sharingMode-02314
  If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, then `pQueueFamilyIndices` must be a valid pointer to an array of `queueFamilyIndexCount uint32_t` values
- VUID-VkPhysicalDeviceImageDrmFormatModifierInfoEXT-sharingMode-02315
  If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, then `queueFamilyIndexCount` must be greater than 1
- VUID-VkPhysicalDeviceImageDrmFormatModifierInfoEXT-sharingMode-02316
  If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, each element of `pQueueFamilyIndices` must be unique and must be less than the `pQueueFamilyPropertyCount` returned by `vkGetPhysicalDeviceQueueFamilyProperties2` for the `physicalDevice` that was used to create device.
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceImageDrmFormatModifierInfoEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_DRM_FORMAT_MODIFIER_INFO_EXT

- VUID-VkPhysicalDeviceImageDrmFormatModifierInfoEXT-sharingMode-parameter
  sharingMode must be a valid VkSharingMode value

To determine the number of combined image samplers required to support a multi-planar format, add VkSamplerYcbcrConversionImageFormatProperties to the pNext chain of the VkImageFormatProperties2 structure in a call to vkGetPhysicalDeviceImageFormatProperties2.

The VkSamplerYcbcrConversionImageFormatProperties structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkSamplerYcbcrConversionImageFormatProperties {
    VkStructureType sType;
    void* pNext;
    uint32_t combinedImageSamplerDescriptorCount;
} VkSamplerYcbcrConversionImageFormatProperties;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `combinedImageSamplerDescriptorCount` is the number of combined image sampler descriptors that the implementation uses to access the format.

Valid Usage (Implicit)

- VUID-VkSamplerYcbcrConversionImageFormatProperties-sType-sType
  sType must be VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_IMAGE_FORMAT_PROPERTIES

`combinedImageSamplerDescriptorCount` is a number between 1 and the number of planes in the format. A descriptor set layout binding with immutable Y’C₆₇C₇ conversion samplers will have a maximum `combinedImageSamplerDescriptorCount` which is the maximum across all formats supported by its samplers of the `combinedImageSamplerDescriptorCount` for each format. Descriptor sets with that layout will internally use that maximum `combinedImageSamplerDescriptorCount` descriptors for each descriptor in the binding. This expanded number of descriptors will be consumed from the descriptor pool when a descriptor set is allocated, and counts towards the maxDescriptorSetSamplers, maxDescriptorSetSampledImages, maxPerStageDescriptorSamplers, and maxPerStageDescriptorSampledImages limits.

**Note**

All descriptors in a binding use the same maximum `combinedImageSamplerDescriptorCount` descriptors to allow implementations to use a uniform stride for dynamic indexing of the descriptors in the binding.
For example, consider a descriptor set layout binding with two descriptors and immutable samplers for multi-planar formats that have `VkSamplerYcbcrConversionImageFormatProperties::combinedImageSamplerDescriptorCount` values of 2 and 3 respectively. There are two descriptors in the binding and the maximum `combinedImageSamplerDescriptorCount` is 3, so descriptor sets with this layout consume 6 descriptors from the descriptor pool. To create a descriptor pool that allows allocating four descriptor sets with this layout, `descriptorCount` must be at least 24.

To determine if cubic filtering can be used with a given image format and a given image view type add a `VkPhysicalDeviceImageViewImageFormatInfoEXT` structure to the `pNext` chain of the `VkPhysicalDeviceImageViewImageFormatInfo2` structure, and a `VkFilterCubicImageViewImageFormatPropertiesEXT` structure to the `pNext` chain of the `VkImageFormatProperties2` structure.

The `VkPhysicalDeviceImageViewImageFormatInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_filter_cubic
typedef struct VkPhysicalDeviceImageViewImageFormatInfoEXT {
    VkStructureType sType;
    void* pNext;
    VkImageViewType imageViewType;
} VkPhysicalDeviceImageViewImageFormatInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `imageViewType` is a `VkImageViewType` value specifying the type of the image view.

Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceImageViewImageFormatInfoEXT-sType-sType`  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_VIEW_IMAGE_FORMAT_INFO_EXT`

- `VUID-VkPhysicalDeviceImageViewImageFormatInfoEXT-imageViewType-parameter`  
  `imageViewType` must be a valid `VkImageViewType` value

The `VkFilterCubicImageViewImageFormatPropertiesEXT` structure is defined as:

```c
// Provided by VK_EXT_filter_cubic
typedef struct VkFilterCubicImageViewImageFormatPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 filterCubic;
    VkBool32 filterCubicMinmax;
} VkFilterCubicImageViewImageFormatPropertiesEXT;
```
• **sType** is the type of this structure.

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

• **filterCubic** tells if image format, image type and image view type can be used with cubic filtering. This field is set by the implementation. User-specified value is ignored.

• **filterCubicMinmax** tells if image format, image type and image view type can be used with cubic filtering and minmax filtering. This field is set by the implementation. User-specified value is ignored.

---

### Valid Usage (Implicit)

- **VUID-VkFilterCubicImageViewImageFormatPropertiesEXT-sType-sType**
  - *sType must be VK_STRUCTURE_TYPE_FILTER_CUBIC_IMAGE_VIEW_IMAGE_FORMAT_PROPERTIES_EXT*

### Valid Usage

- **VUID-VkFilterCubicImageViewImageFormatPropertiesEXT-pNext-02627**
  - If the **pNext** chain of the **VkImageFormatProperties2** structure includes a **VkFilterCubicImageViewImageFormatPropertiesEXT** structure, the **pNext** chain of the **VkPhysicalDeviceImageFormatInfo2** structure must include a **VkPhysicalDeviceImageViewImageFormatInfoEXT** structure with an **imageViewType** that is compatible with **imageType**

---

### 35.1.1. Supported Sample Counts

**vkGetPhysicalDeviceImageFormatProperties** returns a bitmask of **VkSampleCountFlagBits** in **sampleCounts** specifying the supported sample counts for the image parameters.

**sampleCounts** will be set to **VK_SAMPLE_COUNT_1_BIT** if at least one of the following conditions is true:

- **tiling** is **VK_IMAGE_TILING_LINEAR**
- **type** is not **VK_IMAGE_TYPE_2D**
- **flags** contains **VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT**
- Neither the **VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT** flag nor the **VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT** flag in **VkFormatProperties::optimalTilingFeatures** returned by **vkGetPhysicalDeviceFormatProperties** is set
- **VkPhysicalDeviceExternalImageFormatInfo::handleType** is an external handle type for which multisampled image support is not required.
- **format** is one of the formats that require a sampler **Y’CgCr** conversion
- **usage** contains **VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR**

Otherwise, the bits set in **sampleCounts** will be the sample counts supported for the specified values of **usage** and **format**. For each bit set in **usage**, the supported sample counts relate to the limits in **VkPhysicalDeviceLimits** as follows:
• If `usage` includes `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` and `format` is a floating- or fixed-point color format, a superset of `VkPhysicalDeviceLimits::framebufferColorSampleCounts`

• If `usage` includes `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` and `format` is an integer format, a superset of `VkPhysicalDeviceVulkan12Properties::framebufferIntegerColorSampleCounts`

• If `usage` includes `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, and `format` includes a depth aspect, a superset of `VkPhysicalDeviceLimits::framebufferDepthSampleCounts`

• If `usage` includes `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, and `format` includes a stencil aspect, a superset of ` VkPhysicalDeviceLimits::framebufferStencilSampleCounts`

• If `usage` includes `VK_IMAGE_USAGE_SAMPLED_BIT`, and `format` includes a color aspect, a superset of `VkPhysicalDeviceLimits::sampledImageColorSampleCounts`

• If `usage` includes `VK_IMAGE_USAGE_SAMPLED_BIT`, and `format` includes a depth aspect, a superset of `VkPhysicalDeviceLimits::sampledImageDepthSampleCounts`

• If `usage` includes `VK_IMAGE_USAGE_SAMPLED_BIT`, and `format` includes an integer format, a superset of `VkPhysicalDeviceLimits::sampledImageIntegerSampleCounts`

• If `usage` includes `VK_IMAGE_USAGE_STORAGE_BIT`, a superset of `VkPhysicalDeviceLimits::storageImageSampleCounts`

If multiple bits are set in `usage`, `sampleCounts` will be the intersection of the per-usage values described above.

If none of the bits described above are set in `usage`, then there is no corresponding limit in `VkPhysicalDeviceLimits`. In this case, `sampleCounts` must include at least `VK_SAMPLE_COUNT_1_BIT`.

### 35.1.2. Allowed Extent Values Based On Image Type

Implementations may support extent values larger than the required minimum/maximum values for certain types of images. `VkImageFormatProperties::maxExtent` for each type is subject to the constraints below.

**Note**

Implementations must support images with dimensions up to the required minimum/maximum values for all types of images. It follows that the query for additional capabilities must return extent values that are at least as large as the required values.

For **VK_IMAGE_TYPE_1D**:

• `maxExtent.width` ≥ `VkPhysicalDeviceLimits::maxImageDimension1D`

• `maxExtent.height` = 1

• `maxExtent.depth` = 1

For **VK_IMAGE_TYPE_2D** when `flags` does not contain `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`:

• `maxExtent.width` ≥ `VkPhysicalDeviceLimits::maxImageDimension2D`

• `maxExtent.height` ≥ `VkPhysicalDeviceLimits::maxImageDimension2D`
• maxExtent.depth = 1

For VK_IMAGE_TYPE_2D when flags contains VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT:

• maxExtent.width ≥ VkPhysicalDeviceLimits::maxImageDimensionCube
• maxExtent.height ≥ VkPhysicalDeviceLimits::maxImageDimensionCube
• maxExtent.depth = 1

For VK_IMAGE_TYPE_3D:

• maxExtent.width ≥ VkPhysicalDeviceLimits::maxImageDimension3D
• maxExtent.height ≥ VkPhysicalDeviceLimits::maxImageDimension3D
• maxExtent.depth ≥ VkPhysicalDeviceLimits::maxImageDimension3D

### 35.2. Additional Buffer Capabilities

To query the external handle types supported by buffers, call:

```cpp
// 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:
typedef struct VkPhysicalDeviceExternalBufferInfo {
    VkStructureType sType;
    const void* pNext;
    VkBufferCreateFlags flags;
    VkBufferUsageFlags usage;
    VkExternalMemoryHandleTypeFlagBits handleType;
} VkPhysicalDeviceExternalBufferInfo;

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **flags** is a bitmask of `VkBufferCreateFlagBits` describing additional parameters of the buffer, corresponding to `VkBufferCreateInfo::flags`.
- **usage** is a bitmask of `VkBufferUsageFlagBits` describing the intended usage of the buffer, corresponding to `VkBufferCreateInfo::usage`.
- **handleType** is a `VkExternalMemoryHandleTypeFlagBits` value specifying the memory handle that will be used with the memory associated with the buffer.

**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-requiredbitmask
  `usage` must not be `0`

- VUID-VkPhysicalDeviceExternalBufferInfo-handleType-parameter
  `handleType` must be a valid `VkExternalMemoryHandleTypeFlagBits` value

The `VkExternalBufferProperties` structure is defined as:

typedef struct VkExternalBufferProperties {
    VkStructureType sType;
    void* pNext;
    VkExternalMemoryProperties externalMemoryProperties;
} VkExternalBufferProperties;
• `sType` is the type of this structure.

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

• `externalMemoryProperties` is a `VkExternalMemoryProperties` structure specifying various capabilities of the external handle type when used with the specified buffer creation parameters.

---

### Valid Usage (Implicit)

- VUID-VkExternalBufferProperties-sType-sType
  - `sType` must be `VK_STRUCTURE_TYPE_EXTERNAL_BUFFER_PROPERTIES`

- VUID-VkExternalBufferProperties-pNext-pNext
  - `pNext` must be NULL

---

### 35.3. Optional Semaphore Capabilities

Semaphores may support import and export of their payload to external handles. To query the external handle types supported by semaphores, call:

```c
// Provided by VK_VERSION_1_1
void vkGetPhysicalDeviceExternalSemaphoreProperties(
  VkPhysicalDevice physicalDevice,
  const VkPhysicalDeviceExternalSemaphoreInfo* pExternalSemaphoreInfo,
  VkExternalSemaphoreProperties* pExternalSemaphoreProperties);
```

- `physicalDevice` is the physical device from which to query the semaphore capabilities.

- `pExternalSemaphoreInfo` is a pointer to a `VkPhysicalDeviceExternalSemaphoreInfo` structure describing the parameters that would be consumed by `vkCreateSemaphore`.

- `pExternalSemaphoreProperties` is a pointer to a `VkExternalSemaphoreProperties` structure in which capabilities are returned.

---

### Valid Usage (Implicit)

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

- VUID-vkGetPhysicalDeviceExternalSemaphoreProperties-pExternalSemaphoreInfo-parameter
  - `pExternalSemaphoreInfo` must be a valid pointer to a valid `VkPhysicalDeviceExternalSemaphoreInfo` structure

- VUID-vkGetPhysicalDeviceExternalSemaphoreProperties-pExternalSemaphoreProperties-parameter
  - `pExternalSemaphoreProperties` must be a valid pointer to a `VkExternalSemaphoreProperties` structure
The `VkPhysicalDeviceExternalSemaphoreInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceExternalSemaphoreInfo {
    VkStructureType sType;
    const void* pNext;
    VkExternalSemaphoreHandleTypeFlagBits handleType;
} VkPhysicalDeviceExternalSemaphoreInfo;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **handleType** is a `VkExternalSemaphoreHandleTypeFlagBits` value specifying the external semaphore handle type for which capabilities will be returned.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceExternalSemaphoreInfo-sType-sType
  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
  • **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,
    // Provided by VK_NV_external_sci_sync
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SCI_SYNC_OBJ_BIT_NV = 0x00000020,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_FENCE_BIT = 0x00000000,
} VkExternalSemaphoreHandleTypeFlagBits;
```

- **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT** specifies a POSIX file descriptor handle that has only limited valid usage outside of Vulkan and other compatible APIs. It **must** be compatible with the POSIX system calls `dup`, `dup2`, `close`, and the non-standard system call `dup3`. Additionally,
it must be transportable over a socket using an SCM_RIGHTS control message. It owns a reference to the underlying synchronization primitive represented by its Vulkan semaphore object.

- **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT** specifies an NT handle that has only limited valid usage outside of Vulkan and other compatible APIs. It must be compatible with the functions DuplicateHandle, CloseHandle, CompareObjectHandles, GetHandleInformation, and SetHandleInformation. It owns a reference to the underlying synchronization primitive represented by its Vulkan semaphore object.

- **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT** specifies a global share handle that has only limited valid usage outside of Vulkan and other compatible APIs. It is not compatible with any native APIs. It does not own a reference to the underlying synchronization primitive represented by its Vulkan semaphore object, and will therefore become invalid when all Vulkan semaphore objects associated with it are destroyed.

- **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT** specifies an NT handle returned by ID3D12Device::CreateSharedHandle referring to a Direct3D 12 fence, or ID3D11Device5::CreateFence referring to a Direct3D 11 fence. It owns a reference to the underlying synchronization primitive associated with the Direct3D fence.

- **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_FENCE_BIT** is an alias of **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT** with the same meaning. It is provided for convenience and code clarity when interacting with D3D11 fences.

- **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT** specifies a POSIX file descriptor handle to a Linux Sync File or Android Fence object. It can be used with any native API accepting a valid sync file or fence as input. It owns a reference to the underlying synchronization primitive associated with the file descriptor. Implementations which support importing this handle type must accept any type of sync or fence FD supported by the native system they are running on.

- **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SCI_SYNC_OBJ_BIT_NV** specifies a synchronization object (NvSciSyncObj) shareable across various hardware engines including the CPU and software (intra-process and inter-process) operating domains and perform signal and wait operations.

---

**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 72. 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_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_ZIRCON_EVENT_BIT_FUCHSIA</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SCI_SYNC_OBJ_BIT_NV</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
</tbody>
</table>

// Provided by VK_VERSION_1_1
typedef VkFlags VkExternalSemaphoreHandleTypeFlags;

VkExternalSemaphoreHandleTypeFlags is a bitmask type for setting a mask of zero or more VkExternalSemaphoreHandleTypeFlagBits.

The VkExternalSemaphoreProperties structure is defined as:

// Provided by VK_VERSION_1_1
typedef struct VkExternalSemaphoreProperties {
    VkStructureType sType;
    void* pNext;
    VkExternalSemaphoreHandleTypeFlags exportFromImportedHandleTypes;
    VkExternalSemaphoreHandleTypeFlags compatibleHandleTypes;
    VkExternalSemaphoreFeatureFlags externalSemaphoreFeatures;
} VkExternalSemaphoreProperties;

- sType is the type of this structure
- pNext is NULL or a pointer to a structure extending this structure.
- exportFromImportedHandleTypes is a bitmask of VkExternalSemaphoreHandleTypeFlagBits specifying which types of imported handle handleType can be exported from.
- compatibleHandleTypes is a bitmask of VkExternalSemaphoreHandleTypeFlagBits specifying handle types which can be specified at the same time as handleType when creating a semaphore.
- externalSemaphoreFeatures is a bitmask of VkExternalSemaphoreFeatureFlagBits describing the
features of handleType.

If handleType is not supported by the implementation, then VkExternalSemaphoreProperties::externalSemaphoreFeatures will be set to zero.

### Valid Usage (Implicit)

- **VUID-VkExternalSemaphoreProperties-sType-sType**  
  sType must be `VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES`
- **VUID-VkExternalSemaphoreProperties-pNext-pNext**  
  pNext must be NULL

Possible values of VkExternalSemaphoreProperties::externalSemaphoreFeatures, specifying the features of an external semaphore handle type, are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkExternalSemaphoreFeatureFlagBits {
    VK_EXTERNAL_SEMAPHORE_FEATURE_EXPORTABLE_BIT = 0x00000001,
    VK_EXTERNAL_SEMAPHORE_FEATURE_IMPORTABLE_BIT = 0x00000002,
} VkExternalSemaphoreFeatureFlagBits;
```

- **VK_EXTERNAL_SEMAPHORE_FEATURE_EXPORTABLE_BIT** specifies that handles of this type can be exported from Vulkan semaphore objects.
- **VK_EXTERNAL_SEMAPHORE_FEATURE_IMPORTABLE_BIT** specifies that handles of this type can be imported as Vulkan semaphore objects.

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

VkExternalSemaphoreFeatureFlags is a bitmask type for setting a mask of zero or more VkExternalSemaphoreFeatureFlagBits.

### 35.4. Optional Fence Capabilities

Fences may support import and export of their payload to external handles. To query the external handle types supported by fences, call:

```c
// Provided by VK_VERSION_1_1
void vkGetPhysicalDeviceExternalFenceProperties(
    VkPhysicalDevice physicalDevice,
    const VkPhysicalDeviceExternalFenceInfo* pExternalFenceInfo,
    VkExternalFenceProperties* pExternalFenceProperties);
```

- **physicalDevice** is the physical device from which to query the fence capabilities.
• `pExternalFenceInfo` is a pointer to a `VkPhysicalDeviceExternalFenceInfo` structure describing the parameters that would be consumed by `vkCreateFence`.

• `pExternalFenceProperties` is a pointer to a `VkExternalFenceProperties` structure in which capabilities are returned.

### Valid Usage (Implicit)

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

- **VUID-vkGetPhysicalDeviceExternalFenceProperties-pExternalFenceInfo-parameter**
  
  `pExternalFenceInfo` **must** be a valid pointer to a valid `VkPhysicalDeviceExternalFenceInfo` structure

- **VUID-vkGetPhysicalDeviceExternalFenceProperties-pExternalFenceProperties-parameter**
  
  `pExternalFenceProperties` **must** be a valid pointer to a `VkExternalFenceProperties` structure

The `VkPhysicalDeviceExternalFenceInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceExternalFenceInfo {
    VkStructureType sType;
    const void* pNext;
    VkExternalFenceHandleTypeFlagBits handleType;
} VkPhysicalDeviceExternalFenceInfo;
```

- `sType` is the type of this structure.
- `pNext` is **NULL** or a pointer to a structure extending this structure.
- `handleType` is a `VkExternalFenceHandleTypeFlagBits` value specifying an external fence handle type for which capabilities will be returned.

**Note**

Handles of type `VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT` generated by the implementation may represent either Linux Sync Files or Android Fences at the implementation's discretion. Applications **should** only use operations defined for both types of file descriptors, unless they know via means external to Vulkan the type of the file descriptor, or are prepared to deal with the system-defined operation failures resulting from using the wrong type.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceExternalFenceInfo-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_FENCE_INFO`

- **VUID-VkPhysicalDeviceExternalFenceInfo-pNext-pNext**
pNext must be NULL

- VUID-VkPhysicalDeviceExternalFenceInfo-handleType-parameter
  handleType must be a valid VkExternalFenceHandleTypeFlagBits value

Bits which may be set in VkPhysicalDeviceExternalFenceInfo::handleType, and in the exportFromImportedHandleTypes and compatibleHandleTypes members of VkExternalFenceProperties, to indicate external fence handle types, are:

```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,
    // Provided by VK_NV_external_sci_sync, VK_NV_external_sci_sync2
    VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_OBJ_BIT_NV = 0x00000010,
    // Provided by VK_NV_external_sci_sync, VK_NV_external_sci_sync2
    VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_FENCE_BIT_NV = 0x00000020,
} 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.

- **VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_OBJ_BIT_NV** specifies a synchronization object (NvSciSyncObj) shareable across various hardware engines including the CPU and software (intra-process and inter-process) operating domains and perform signal and wait operations.

- **VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_FENCE_BIT_NV** specifies a struct of NvSciSyncFence that is a snapshot of a synchronization object's underlying primitive and represents its possible state.
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 73. 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>
<tr>
<td>VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_OBJ_BIT_NV</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_FENCE_BIT_NV</td>
<td>Must match</td>
<td>Must match</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`

- VUID-VkExternalFenceProperties-pNext-pNext
  
  `pNext` **must be** `NULL`

Bits which **may** be set in `VkExternalFenceProperties::externalFenceFeatures`, indicating features of a fence external handle type, are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkExternalFenceFeatureFlagBits {
    VK_EXTERNAL_FENCE_FEATURE_EXPORTABLE_BIT = 0x00000001,
    VK_EXTERNAL_FENCE_FEATURE_IMPORTABLE_BIT = 0x00000002,
} VkExternalFenceFeatureFlagBits;
```

- `VK_EXTERNAL_FENCE_FEATURE_EXPORTABLE_BIT` specifies handles of this type **can** be exported from Vulkan fence objects.
- `VK_EXTERNAL_FENCE_FEATURE_IMPORTABLE_BIT` specifies handles of this type **can** be imported to Vulkan fence objects.

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

`VkExternalFenceFeatureFlags` is a bitmask type for setting a mask of zero or more `VkExternalFenceFeatureFlagBits`.

### 35.5. Timestamp Calibration Capabilities

To query the set of time domains for which a physical device supports timestamp calibration, call:

```c
// Provided by VK_EXT_calibrated_timestamps
VkResult vkGetPhysicalDeviceCalibrateableTimeDomainsEXT(
    VkPhysicalDevice physicalDevice,  
    uint32_t* pTimeDomainCount,  
    VkTimeDomainEXT* pTimeDomains);
```

- `physicalDevice` is the physical device from which to query the set of calibrateable time domains.
- `pTimeDomainCount` is a pointer to an integer related to the number of calibrateable time domains available or queried, as described below.
- `pTimeDomains` is either `NULL` or a pointer to an array of `VkTimeDomainEXT` values, indicating the supported calibrateable time domains.
If `pTimeDomains` is `NULL`, then the number of calibrateable time domains supported for the given `physicalDevice` is returned in `pTimeDomainCount`. Otherwise, `pTimeDomainCount` must point to a variable set by the user to the number of elements in the `pTimeDomains` array, and on return the variable is overwritten with the number of values actually written to `pTimeDomains`. If the value of `pTimeDomainCount` is less than the number of calibrateable time domains supported, at most `pTimeDomainCount` values will be written to `pTimeDomains`, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available time domains were returned.

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

### Valid Usage (Implicit)
- VUID-vkGetPhysicalDeviceCalibrateableTimeDomainsEXT-physicalDevice-parameter `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceCalibrateableTimeDomainsEXT-pTimeDomainCount-parameter `pTimeDomainCount` must be a valid pointer to a `uint32_t` value
- VUID-vkGetPhysicalDeviceCalibrateableTimeDomainsEXT-pTimeDomains-parameter If the value referenced by `pTimeDomainCount` is not 0, and `pTimeDomains` is not `NULL`, `pTimeDomains` must be a valid pointer to an array of `pTimeDomainCount` `VkTimeDomainEXT` values

### Return Codes

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

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

### 35.6. Object Refresh Capabilities

To query the set of object types that require periodic refreshing, call:

```c
// Provided by VK_KHR_object_refresh
VkResult vkGetPhysicalDeviceRefreshableObjectTypesKHR(
    VkPhysicalDevice physicalDevice,
    uint32_t* pRefreshableObjectTypeCount,
    VkObjectType* pRefreshableObjectTypes);
```

- `physicalDevice` is the physical device from which to query the set of refreshable object types.
• `pRefreshableObjectTypes` is a pointer to an integer related to the number of refreshable object types available or queried, as described below.

• `pRefreshableObjectTypes` is either `NULL` or a pointer to an array of `VkObjectType` values, indicating the supported refreshable object types.

If `pRefreshableObjectTypes` is `NULL`, then the number of refreshable object types supported for the given `physicalDevice` is returned in `pRefreshableObjectTypeCount`. Otherwise, `pRefreshableObjectTypeCount` must point to a variable set by the user to the number of elements in the `pRefreshableObjectTypes` array, and on return the variable is overwritten with the number of object types actually written to `pRefreshableObjectTypes`. If the value of `pRefreshableObjectTypeCount` is less than the number of refreshable object types supported, at most `pRefreshableObjectTypeCount` object types will be written, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available object types were returned.

### Valid Usage (Implicit)

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

- VUID-vkGetPhysicalDeviceRefreshableObjectTypesKHR-pRefreshableObjectTypeCount-parameter
  - `pRefreshableObjectTypeCount` must be a valid pointer to a `uint32_t` value

- VUID-vkGetPhysicalDeviceRefreshableObjectTypesKHR-pRefreshableObjectTypes-parameter
  - If the value referenced by `pRefreshableObjectTypeCount` is not 0, and `pRefreshableObjectTypes` is not `NULL`, `pRefreshableObjectTypes` must be a valid pointer to an array of `pRefreshableObjectTypeCount` `VkObjectType` values

### Return Codes

**Success**

- `VK_SUCCESS`
- `VK_INCOMPLETE`
Chapter 36. Debugging

To aid developers in tracking down errors in the application’s use of Vulkan, particularly in combination with an external debugger or profiler, debugging extensions may be available.

The VkObjectType enumeration defines values, each of which corresponds to a specific Vulkan handle type. These values can be used to associate debug information with a particular type of object through one or more extensions.

```c
// Provided by VK_VERSION_1_0
typedef enum VkObjectType {
    VK_OBJECT_TYPE_UNKNOWN = 0,
    VK_OBJECT_TYPE_INSTANCE = 1,
    VK_OBJECT_TYPE_PHYSICAL_DEVICE = 2,
    VK_OBJECT_TYPE_DEVICE = 3,
    VK_OBJECT_TYPE_QUEUE = 4,
    VK_OBJECT_TYPE_SEMAPHORE = 5,
    VK_OBJECT_TYPE_COMMAND_BUFFER = 6,
    VK_OBJECT_TYPE_FENCE = 7,
    VK_OBJECT_TYPE_DEVICE_MEMORY = 8,
    VK_OBJECT_TYPE_BUFFER = 9,
    VK_OBJECT_TYPE_IMAGE = 10,
    VK_OBJECT_TYPE_EVENT = 11,
    VK_OBJECT_TYPE_QUERY_POOL = 12,
    VK_OBJECT_TYPE_BUFFER_VIEW = 13,
    VK_OBJECT_TYPE_IMAGE_VIEW = 14,
    VK_OBJECT_TYPE_SHADER_MODULE = 15,
    VK_OBJECT_TYPE_PIPELINE_CACHE = 16,
    VK_OBJECT_TYPE_PIPELINE_LAYOUT = 17,
    VK_OBJECT_TYPE_RENDER_PASS = 18,
    VK_OBJECT_TYPE_PIPELINE = 19,
    VK_OBJECT_TYPE_DESCRIPTOR_SET_LAYOUT = 20,
    VK_OBJECT_TYPE_SAMPLER = 21,
    VK_OBJECT_TYPE_DESCRIPTOR_POOL = 22,
    VK_OBJECT_TYPE_DESCRIPTOR_SET = 23,
    VK_OBJECT_TYPE_FRAMEBUFFER = 24,
    VK_OBJECT_TYPE_COMMAND_POOL = 25,
    // Provided by VK_VERSION_1_1
    VK_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION = 1000156000,
    // Provided by VK_KHR_surface
    VK_OBJECT_TYPE_SURFACE_KHR = 1000000000,
    // Provided by VK_KHR_swapchain
    VK_OBJECT_TYPE_SWAPCHAIN_KHR = 1000001000,
    // Provided by VK_KHR_display
    VK_OBJECT_TYPE_DISPLAY_KHR = 1000002000,
    // Provided by VK_KHR_display
    VK_OBJECT_TYPE_DISPLAY_MODE_KHR = 1000002001,
    // Provided by VK_EXT_debug_utils
    VK_OBJECT_TYPE_DEBUG_UTILS_MESSENGER_EXT = 1000128000,
    // Provided by VK_NV_external_sci_sync2

1466
```
VK_OBJECT_TYPE_SEMAPHORE_SCI_SYNC_POOL_NV = 1000489000,
} 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>
<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>
<tr>
<td>VK_OBJECT_TYPE_SURFACE_KHR</td>
<td>VkSurfaceKHR</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_SWAPCHAIN_KHR</td>
<td>VkSwapchainKHR</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DISPLAY_KHR</td>
<td>VkDisplayKHR</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DISPLAY_MODE_KHR</td>
<td>VkDisplayModeKHR</td>
</tr>
</tbody>
</table>
36.1. Debug Utilities

Vulkan provides flexible debugging utilities for debugging an application.

The *Object Debug Annotation* section describes how to associate either a name or binary data with a specific Vulkan object.

The *Queue Labels* section describes how to annotate and group the work submitted to a queue.

The *Command Buffer Labels* section describes how to associate logical elements of the scene with commands in a *VkCommandBuffer*.

The *Debug Messengers* section describes how to create debug messenger objects associated with an application supplied callback to capture debug messages from a variety of Vulkan components.

36.1.1. Object Debug Annotation

It can be useful for an application to provide its own content relative to a specific Vulkan object. The following commands allow application developers to associate user-defined information with Vulkan objects.

**Object Naming**

An object can be provided a user-defined name by calling *vkSetDebugUtilsObjectNameEXT* as defined below.

```c
// Provided by VK_EXT_debug_utils
VkResult vkSetDebugUtilsObjectNameEXT(
    VkDevice device,
    const VkDebugUtilsObjectNameInfoEXT* pNameInfo);
```

- *device* is the device that created the object.
- *pNameInfo* is a pointer to a *VkDebugUtilsObjectNameInfoEXT* structure specifying parameters of the name to set on the object.

**Valid Usage**

- VUID-vkSetDebugUtilsObjectNameEXT-pNameInfo-02587
  pNameInfo->objectType *must* not be VK_OBJECT_TYPE_UNKNOWN
- VUID-vkSetDebugUtilsObjectNameEXT-pNameInfo-02588
The `VkDebugUtilsObjectNameInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_debug_utils
typedef struct VkDebugUtilsObjectNameInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkObjectType objectType;
    uint64_t objectHandle;
    const char* pObjectName;
} VkDebugUtilsObjectNameInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `objectType` is a `VkObjectType` specifying the type of the object to be named.
- `objectHandle` is the object to be named.
- `pObjectName` is either `NULL` or a null-terminated UTF-8 string specifying the name to apply to `objectHandle`.

Applications may change the name associated with an object simply by calling `vkSetDebugUtilsObjectNameEXT` again with a new string. If `pObjectName` is either `NULL` or an empty
string, then any previously set name is removed.

### Valid Usage

- **VUID-VkDebugUtilsObjectNameInfoEXT-objectType-02589**
  
  If `objectType` is `VK_OBJECT_TYPE_UNKNOWN`, `objectHandle` **must** not be `VK_NULL_HANDLE`.

- **VUID-VkDebugUtilsObjectNameInfoEXT-objectType-02590**
  
  If `objectType` is not `VK_OBJECT_TYPE_UNKNOWN`, `objectHandle` **must** be `VK_NULL_HANDLE` or a valid Vulkan handle of the type associated with `objectType` as defined in the `VkObjectType` and Vulkan Handle Relationship table.

### Valid Usage (Implicit)

- **VUID-VkDebugUtilsObjectNameInfoEXT-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_NAME_INFO_EXT`.

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

- **VUID-VkDebugUtilsObjectNameInfoEXT-objectType-parameter**
  
  `objectType` **must** be a valid `VkObjectType` value.

- **VUID-VkDebugUtilsObjectNameInfoEXT-pObjectName-parameter**
  
  If `pObjectName` is not `NULL`, `pObjectName` **must** be a null-terminated UTF-8 string.

### Object Data Association

In addition to setting a name for an object, debugging and validation layers **may** have uses for additional binary data on a per-object basis that have no other place in the Vulkan API.

For example, a `VkShaderModule` could have additional debugging data attached to it to aid in offline shader tracing.

Additional data can be attached to an object by calling `vkSetDebugUtilsObjectTagEXT` as defined below.

```c
// Provided by VK_EXT_debug_utils
VkResult vkSetDebugUtilsObjectTagEXT(
    VkDevice device,
    const VkDebugUtilsObjectTagInfoEXT* pTagInfo);
```

- `device` is the device that created the object.
- `pTagInfo` is a pointer to a `VkDebugUtilsObjectTagInfoEXT` structure specifying parameters of the tag to attach to the object.
Valid Usage (Implicit)

- VUID-vkSetDebugUtilsObjectTagEXT-device-parameter
  
  \textit{device} must be a valid \texttt{VkDevice} handle

- VUID-vkSetDebugUtilsObjectTagEXT-pTagInfo-parameter
  
  \textit{pTagInfo} must be a valid pointer to a valid \texttt{VkDebugUtilsObjectTagInfoEXT} structure

Host Synchronization

- Host access to \textit{pTagInfo->objectHandle} must be externally synchronized

Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The \texttt{VkDebugUtilsObjectTagInfoEXT} structure is defined as:

```c
// Provided by VK_EXT_debug_utils
typedef struct VkDebugUtilsObjectTagInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkObjectType objectType;
    uint64_t objectHandle;
    uint64_t tagName;
    size_t tagSize;
    const void* pTag;
} VkDebugUtilsObjectTagInfoEXT;
```

- \texttt{sType} is the type of this structure.
- \texttt{pNext} is \texttt{NULL} or a pointer to a structure extending this structure.
- \texttt{objectType} is a \texttt{VkObjectType} specifying the type of the object to be named.
- \texttt{objectHandle} is the object to be tagged.
- \texttt{tagName} is a numerical identifier of the tag.
- \texttt{tagSize} is the number of bytes of data to attach to the object.
- \texttt{pTag} is a pointer to an array of \texttt{tagSize} bytes containing the data to be associated with the object.
The `tagName` parameter gives a name or identifier to the type of data being tagged. This can be used by debugging layers to easily filter for only data that can be used by that implementation.

### Valid Usage

- VUID-VkDebugUtilsObjectTagInfoEXT-objectType-01908
  
  `objectType` **must** not be `VK_OBJECT_TYPE_UNKNOWN`

- VUID-VkDebugUtilsObjectTagInfoEXT-objectHandle-01910
  
  `objectHandle` **must** be a valid Vulkan handle of the type associated with `objectType` as defined in the `VkObjectType` and Vulkan Handle Relationship table.

### Valid Usage (Implicit)

- VUID-VkDebugUtilsObjectTagInfoEXT-sType-sType
  
  `sType` **must** be `VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_TAG_INFO_EXT`

- VUID-VkDebugUtilsObjectTagInfoEXT-pNext-pNext
  
  `pNext` **must** be `NULL`

- VUID-VkDebugUtilsObjectTagInfoEXT-objectType-parameter
  
  `objectType` **must** be a valid `VkObjectType` value

- VUID-VkDebugUtilsObjectTagInfoEXT-pTag-parameter
  
  `pTag` **must** be a valid pointer to an array of `tagSize` bytes

- VUID-VkDebugUtilsObjectTagInfoEXT-tagSize-arraylength
  
  `tagSize` **must** be greater than `0`

### 36.1.2. Queue Labels

All Vulkan work must be submitted using queues. It is possible for an application to use multiple queues, each containing multiple command buffers, when performing work. It can be useful to identify which queue, or even where in a queue, something has occurred.

To begin identifying a region using a debug label inside a queue, you may use the `vkQueueBeginDebugUtilsLabelEXT` command.

Then, when the region of interest has passed, you may end the label region using `vkQueueEndDebugUtilsLabelEXT`.

Additionally, a single debug label may be inserted at any time using `vkQueueInsertDebugUtilsLabelEXT`.

A queue debug label region is opened by calling:

```c
// Provided by VK_EXT_debug_utils
void vkQueueBeginDebugUtilsLabelEXT(
    VkQueue queue,
```
const VkDebugUtilsLabelEXT* pLabelInfo);

- queue is the queue in which to start a debug label region.
- pLabelInfo is a pointer to a VkDebugUtilsLabelEXT structure specifying parameters of the label region to open.

**Valid Usage (Implicit)**

- VUID-vkQueueBeginDebugUtilsLabelEXT-queue-parameter queue must be a valid VkQueue handle
- VUID-vkQueueBeginDebugUtilsLabelEXT-pLabelInfo-parameter pLabelInfo must be a valid pointer to a valid VkDebugUtilsLabelEXT structure

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>Any</td>
</tr>
</tbody>
</table>

The VkDebugUtilsLabelEXT structure is defined as:

```c
// Provided by VK_EXT_debug_utils
typedef struct VkDebugUtilsLabelEXT {
    VkStructureType sType;
    const void* pNext;
    const char* pLabelName;
    float color[4];
} VkDebugUtilsLabelEXT;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- pLabelName is a pointer to a null-terminated UTF-8 string containing the name of the label.
- color is an optional RGBA color value that can be associated with the label. A particular implementation may choose to ignore this color value. The values contain RGBA values in order, in the range 0.0 to 1.0. If all elements in color are set to 0.0 then it is ignored.

**Valid Usage (Implicit)**

- VUID-VkDebugUtilsLabelEXT-sType-sType sType must be VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT
- VUID-VkDebugUtilsLabelEXT-pNext-pNext pNext must be NULL
A queue debug label region is closed by calling:

```c
// Provided by VK_EXT_debug_utils
void vkQueueEndDebugUtilsLabelEXT(VkQueue queue);
```

- `queue` is the queue in which a debug label region should be closed.

The calls to `vkQueueBeginDebugUtilsLabelEXT` and `vkQueueEndDebugUtilsLabelEXT` must be matched and balanced.

**Valid Usage**

- VUID-vkQueueEndDebugUtilsLabelEXT-None-01911
  There must be an outstanding `vkQueueBeginDebugUtilsLabelEXT` command prior to the `vkQueueEndDebugUtilsLabelEXT` on the queue

**Valid Usage (Implicit)**

- VUID-vkQueueEndDebugUtilsLabelEXT-queue-parameter
  `queue` must be a valid `VkQueue` handle

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>Any</td>
</tr>
</tbody>
</table>

A single label can be inserted into a queue by calling:

```c
// Provided by VK_EXT_debug_utils
void vkQueueInsertDebugUtilsLabelEXT(VkQueue queue,
                                      const VkDebugUtilsLabelEXT* pLabelInfo);
```

- `queue` is the queue into which a debug label will be inserted.
- `pLabelInfo` is a pointer to a `VkDebugUtilsLabelEXT` structure specifying parameters of the label to insert.
36.1.3. Command Buffer Labels

Typical Vulkan applications will submit many command buffers in each frame, with each command buffer containing a large number of individual commands. Being able to logically annotate regions of command buffers that belong together as well as hierarchically subdivide the frame is important to a developer's ability to navigate the commands viewed holistically.

To identify the beginning of a debug label region in a command buffer, `vkCmdBeginDebugUtilsLabelEXT` can be used as defined below.

To indicate the end of a debug label region in a command buffer, `vkCmdEndDebugUtilsLabelEXT` can be used.

To insert a single command buffer debug label inside of a command buffer, `vkCmdInsertDebugUtilsLabelEXT` can be used as defined below.

A command buffer debug label region can be opened by calling:

```c
// Provided by VK_EXT_debug_utils
void vkCmdBeginDebugUtilsLabelEXT(
    VkCommandBuffer commandBuffer,
    const VkDebugUtilsLabelEXT* pLabelInfo);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `pLabelInfo` is a pointer to a `VkDebugUtilsLabelEXT` structure specifying parameters of the label region to open.

```c
// Provided by VK_EXT_debug_utils
void vkCmdInsertDebugUtilsLabelEXT(
    VkCommandBuffer commandBuffer,
    const VkDebugUtilsLabelEXT* pLabelInfo);
```

- `commandBuffer` must be a valid `VkCommandBuffer` handle.
- `pLabelInfo` must be a valid pointer to a valid `VkDebugUtilsLabelEXT` structure.
• VUID-vkCmdBeginDebugUtilsLabelEXT-pLabelInfo-parameter
  pLabelInfo must be a valid pointer to a valid VkDebugUtilsLabelEXT structure

• VUID-vkCmdBeginDebugUtilsLabelEXT-commandBuffer-recording
  commandBuffer must be in the recording state

• VUID-vkCmdBeginDebugUtilsLabelEXT-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations

### Host Synchronization

• Host access to commandBuffer must be externally synchronized

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

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
</tr>
</tbody>
</table>

A command buffer label region can be closed by calling:

```c
// Provided by VK_EXT_debug_utils
void vkCmdEndDebugUtilsLabelEXT(
    VkCommandBuffer commandBuffer);
```

• commandBuffer is the command buffer into which the command is recorded.

An application may open a debug label region in one command buffer and close it in another, or otherwise split debug label regions across multiple command buffers or multiple queue submissions. When viewed from the linear series of submissions to a single queue, the calls to vkCmdBeginDebugUtilsLabelEXT and vkCmdEndDebugUtilsLabelEXT must be matched and balanced.

There can be problems reporting command buffer debug labels during the recording process because command buffers may be recorded out of sequence with the resulting execution order. Since the recording order may be different, a solitary command buffer may have an inconsistent view of the debug label regions by itself. Therefore, if an issue occurs during the recording of a command buffer, and the environment requires returning debug labels, the implementation may return only those labels it is aware of. This is true even if the implementation is aware of only the debug labels within the command buffer being actively recorded.
Valid Usage

- VUID-vkCmdEndDebugUtilsLabelEXT-commandBuffer-01912
  There must be an outstanding `vkCmdBeginDebugUtilsLabelEXT` command prior to the `vkCmdEndDebugUtilsLabelEXT` on the queue that `commandBuffer` is submitted to

- VUID-vkCmdEndDebugUtilsLabelEXT-commandBuffer-01913
  If `commandBuffer` is a secondary command buffer, there must be an outstanding `vkCmdBeginDebugUtilsLabelEXT` command recorded to `commandBuffer` that has not previously been ended by a call to `vkCmdEndDebugUtilsLabelEXT`

Valid Usage (Implicit)

- VUID-vkCmdEndDebugUtilsLabelEXT-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdEndDebugUtilsLabelEXT-commandBuffer-recording
  `commandBuffer` must be in the recording state

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

Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
</tr>
</tbody>
</table>

A single debug label can be inserted into a command buffer by calling:

```c
// Provided by VK_EXT_debug_utils
void vkCmdInsertDebugUtilsLabelEXT(
    VkCommandBuffer           commandBuffer,
    const VkDebugUtilsLabelEXT* pLabelInfo);
```

- `commandBuffer` is the command buffer into which the command is recorded.
• pInfo is a pointer to a VkDebugUtilsLabelEXT structure specifying parameters of the label to insert.

Valid Usage (Implicit)

- VUID-vkCmdInsertDebugUtilsLabelEXT-commandBuffer-parameter commandBuffer must be a valid VkCommandBuffer handle
- VUID-vkCmdInsertDebugUtilsLabelEXT-pLabelInfo-parameter pLabelInfo must be a valid pointer to a valid VkDebugUtilsLabelEXT structure
- VUID-vkCmdInsertDebugUtilsLabelEXT-commandBuffer-recording commandBuffer must be in the recording state
- VUID-vkCmdInsertDebugUtilsLabelEXT-commandBuffer-cmdpool The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations

Host Synchronization

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

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
</tr>
</tbody>
</table>

36.1.4. Debug Messengers

Vulkan allows an application to register multiple callbacks with any Vulkan component wishing to report debug information. Some callbacks may log the information to a file, others may cause a debug break point or other application defined behavior. A primary producer of callback messages are the validation layers. An application can register callbacks even when no validation layers are enabled, but they will only be called for the Vulkan loader and, if implemented, other layer and driver events.

A VkDebugUtilsMessengerEXT is a messenger object which handles passing along debug messages to a provided debug callback.

// Provided by VK_EXT_debug_utils
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDebugUtilsMessengerEXT)
The debug messenger will provide detailed feedback on the application’s use of Vulkan when events of interest occur. When an event of interest does occur, the debug messenger will submit a debug message to the debug callback that was provided during its creation. Additionally, the debug messenger is responsible with filtering out debug messages that the callback is not interested in and will only provide desired debug messages.

A debug messenger triggers a debug callback with a debug message when an event of interest occurs. To create a debug messenger which will trigger a debug callback, call:

```c
// Provided by VK_EXT_debug_utils
VkResult vkCreateDebugUtilsMessengerEXT(
    VkInstance instance, const VkDebugUtilsMessengerCreateInfoEXT* pCreateInfo,
    const VkAllocationCallbacks* pAllocator, VkDebugUtilsMessengerEXT* pMessenger);
```

- `instance` is the instance the messenger will be used with.
- `pCreateInfo` is a pointer to a `VkDebugUtilsMessengerCreateInfoEXT` structure containing the callback pointer, as well as defining conditions under which this messenger will trigger the callback.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pMessenger` is a pointer to a `VkDebugUtilsMessengerEXT` handle in which the created object is returned.

### Valid Usage (Implicit)

- VUID-vkCreateDebugUtilsMessengerEXT-instance-parameter
  - `instance` must be a valid `VkInstance` handle
- VUID-vkCreateDebugUtilsMessengerEXT-pCreateInfo-parameter
  - `pCreateInfo` must be a valid pointer to a valid `VkDebugUtilsMessengerCreateInfoEXT` structure
- VUID-vkCreateDebugUtilsMessengerEXT-pAllocator-null
  - `pAllocator` must be `NULL`
- VUID-vkCreateDebugUtilsMessengerEXT-pMessenger-parameter
  - `pMessenger` must be a valid pointer to a `VkDebugUtilsMessengerEXT` handle

### Return Codes

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
The application **must** ensure that `vkCreateDebugUtilsMessengerEXT` is not executed in parallel with any Vulkan command that is also called with `instance` or child of `instance` as the dispatchable argument.

The definition of `VkDebugUtilsMessengerCreateInfoEXT` is:

```c
// Provided by VK_EXT_debug_utils
typedef struct VkDebugUtilsMessengerCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkDebugUtilsMessengerCreateFlagsEXT flags;
    VkDebugUtilsMessageSeverityFlagsEXT messageSeverity;
    VkDebugUtilsMessageTypeFlagsEXT messageType;
    PFN_vkDebugUtilsMessengerCallbackEXT pfnUserCallback;
    void* pUserData;
} VkDebugUtilsMessengerCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is `0` and is reserved for future use.
- `messageSeverity` is a bitmask of `VkDebugUtilsMessageSeverityFlagBitsEXT` specifying which severity of event(s) will cause this callback to be called.
- `messageType` is a bitmask of `VkDebugUtilsMessageTypeFlagBitsEXT` specifying which type of event(s) will cause this callback to be called.
- `pfnUserCallback` is the application callback function to call.
- `pUserData` is user data to be passed to the callback.

For each `VkDebugUtilsMessengerEXT` that is created the `VkDebugUtilsMessengerCreateInfoEXT::messageSeverity` and `VkDebugUtilsMessengerCreateInfoEXT::messageType` determine when that `VkDebugUtilsMessengerCreateInfoEXT::pfnUserCallback` is called. The process to determine if the user’s `pfnUserCallback` is triggered when an event occurs is as follows:

1. The implementation will perform a bitwise AND of the event’s `VkDebugUtilsMessageSeverityFlagBitsEXT` with the `messageSeverity` provided during creation of the `VkDebugUtilsMessengerEXT` object.
   a. If the value is 0, the message is skipped.
2. The implementation will perform bitwise AND of the event’s `VkDebugUtilsMessageTypeFlagBitsEXT` with the `messageType` provided during the creation of the `VkDebugUtilsMessengerEXT` object.
   a. If the value is 0, the message is skipped.
3. The callback will trigger a debug message for the current event

The callback will come directly from the component that detected the event, unless some other layer intercepts the calls for its own purposes (filter them in a different way, log to a system error
An application can receive multiple callbacks if multiple `VkDebugUtilsMessengerEXT` objects are created. A callback will always be executed in the same thread as the originating Vulkan call.

A callback can be called from multiple threads simultaneously (if the application is making Vulkan calls from multiple threads).

**Valid Usage**

- VUID-VkDebugUtilsMessengerCreateInfoEXT-pfnUserCallback-01914
  
  The `pfnUserCallback` must be a valid `PFN_vkDebugUtilsMessengerCallbackEXT`.

**Valid Usage (Implicit)**

- VUID-VkDebugUtilsMessengerCreateInfoEXT-sType-sType
  
  The `sType` must be `VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT`.

- VUID-VkDebugUtilsMessengerCreateInfoEXT-flags-zerobitmask
  
  The `flags` must be `0`.

- VUID-VkDebugUtilsMessengerCreateInfoEXT-messageSeverity-parameter
  
  The `messageSeverity` must be a valid combination of `VkDebugUtilsMessageSeverityFlagBitsEXT` values.

- VUID-VkDebugUtilsMessengerCreateInfoEXT-messageSeverity-requiredbitmask
  
  The `messageSeverity` must not be `0`.

- VUID-VkDebugUtilsMessengerCreateInfoEXT-messageType-parameter
  
  The `messageType` must be a valid combination of `VkDebugUtilsMessageTypeFlagBitsEXT` values.

- VUID-VkDebugUtilsMessengerCreateInfoEXT-messageType-requiredbitmask
  
  The `messageType` must not be `0`.

- VUID-VkDebugUtilsMessengerCreateInfoEXT-pfnUserCallback-parameter
  
  The `pfnUserCallback` must be a valid `PFN_vkDebugUtilsMessengerCallbackEXT` value.

// Provided by VK_EXT_debug_utils

typedef VkFlags VkDebugUtilsMessengerCreateFlagsEXT;

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

Bits which can be set in `VkDebugUtilsMessengerCreateInfoEXT::messageSeverity`, specifying event severities which cause a debug messenger to call the callback, are:

// Provided by VK_EXT_debug_utils

typedef enum VkDebugUtilsMessageSeverityFlagBitsEXT {
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT = 0x00000001,
};
• **VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT** specifies the most verbose output indicating all diagnostic messages from the Vulkan loader, layers, and drivers should be captured.

• **VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT** specifies an informational message such as resource details that may be handy when debugging an application.

• **VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT** specifies use of Vulkan that *may* expose an app bug. Such cases may not be immediately harmful, such as a fragment shader outputting to a location with no attachment. Other cases *may* point to behavior that is almost certainly bad when unintended such as using an image whose memory has not been filled. In general if you see a warning but you know that the behavior is intended/desired, then simply ignore the warning.

• **VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT** specifies that the application has violated a valid usage condition of the specification.

*Note*
The values of **VkDebugUtilsMessageSeverityFlagBitsEXT** are sorted based on severity. The higher the flag value, the more severe the message. This allows for simple boolean operation comparisons when looking at **VkDebugUtilsMessageSeverityFlagBitsEXT** values.

For example:

```c
if (messageSeverity >= VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT) {
    // Do something for warnings and errors
}
```

In addition, space has been left between the enums to allow for later addition of new severities in between the existing values.

// Provided by VK_EXT_debug_utils
typedef VkFlags VkDebugUtilsMessageSeverityFlagsEXT;

**VkDebugUtilsMessageSeverityFlagsEXT** is a bitmask type for setting a mask of zero or more **VkDebugUtilsMessageSeverityFlagBitsEXT**.

Bits which *can* be set in **VkDebugUtilsMessengerCreateInfoEXT::messageType**, specifying event types which cause a debug messenger to call the callback, are:

// Provided by VK_EXT_debug_utils
typedef enum VkDebugUtilsMessageTypeFlagBitsEXT {


VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT = 0x00000001,
VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT = 0x00000002,
VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT = 0x00000004,
}

VkDebugUtilsMessageTypeFlagBitsEXT;

- **VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT** specifies that some general event has occurred. This is typically a non-specification, non-performance event.
- **VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT** specifies that something has occurred during validation against the Vulkan specification that may indicate invalid behavior.
- **VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT** specifies a potentially non-optimal use of Vulkan, e.g. using `vkCmdClearColorImage` when setting `VkAttachmentDescription::loadOp` to `VK_ATTACHMENT_LOAD_OP_CLEAR` would have worked.

```
// Provided by VK_EXT_debug_utils
typedef VkFlags VkDebugUtilsMessageTypeFlagsEXT;
```

`VkDebugUtilsMessageTypeFlagsEXT` is a bitmask type for setting a mask of zero or more `VkDebugUtilsMessageTypeFlagBitsEXT`.

The prototype for the `VkDebugUtilsMessengerCreateInfoEXT::pfnUserCallback` function implemented by the application is:

```
// Provided by VK_EXT_debug_utils
typedef VkBool32 (VKAPI_PTR *PFN_vkDebugUtilsMessengerCallbackEXT)(
    VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,
    VkDebugUtilsMessageTypeFlagsEXT messageTypes,
    const VkDebugUtilsMessengerCallbackDataEXT* pCallbackData,
    void* pUserData);
```

- **messageSeverity** specifies the `VkDebugUtilsMessageSeverityFlagBitsEXT` that triggered this callback.
- **messageTypes** is a bitmask of `VkDebugUtilsMessageTypeFlagBitsEXT` specifying which type of event(s) triggered this callback.
- **pCallbackData** contains all the callback related data in the `VkDebugUtilsMessengerCallbackDataEXT` structure.
- **pUserData** is the user data provided when the `VkDebugUtilsMessengerEXT` was created.

The callback returns a `VkBool32`, which is interpreted in a layer-specified manner. The application **should** always return `VK_FALSE`. The `VK_TRUE` value is reserved for use in layer development.

**Valid Usage**

- VUID-PFN_vkDebugUtilsMessengerCallbackEXT-None-04769
  The callback **must** not make calls to any Vulkan commands
The definition of `VkDebugUtilsMessengerCallbackDataEXT` is:

```c
// Provided by VK_EXT_debug_utils
typedef struct VkDebugUtilsMessengerCallbackDataEXT {
    VkStructureType sType;
    const void* pNext;
    VkDebugUtilsMessengerCallbackDataFlagsEXT flags;
    const char* pMessageIdName;
    int32_t messageIdNumber;
    const char* pMessage;
    uint32_t queueLabelCount;
    const VkDebugUtilsLabelEXT* pQueueLabels;
    uint32_t cmdBufLabelCount;
    const VkDebugUtilsLabelEXT* pCmdBufLabels;
    uint32_t objectCount;
    const VkDebugUtilsObjectNameInfoEXT* pObjects;
} VkDebugUtilsMessengerCallbackDataEXT;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is 0 and is reserved for future use.
- **pMessageIdName** is a null-terminated string that identifies the particular message ID that is associated with the provided message. If the message corresponds to a validation layer message, then this string may contain the portion of the Vulkan specification that is believed to have been violated.
- **messageIdNumber** is the ID number of the triggering message. If the message corresponds to a validation layer message, then this number is related to the internal number associated with the message being triggered.
- **pMessage** is a null-terminated string detailing the trigger conditions.
- **queueLabelCount** is a count of items contained in the **pQueueLabels** array.
- **pQueueLabels** is NULL or a pointer to an array of `VkDebugUtilsLabelEXT` active in the current `VkQueue` at the time the callback was triggered. Refer to Queue Labels for more information.
- **cmdBufLabelCount** is a count of items contained in the **pCmdBufLabels** array.
- **pCmdBufLabels** is NULL or a pointer to an array of `VkDebugUtilsLabelEXT` active in the current `VkCommandBuffer` at the time the callback was triggered. Refer to Command Buffer Labels for more information.
- **objectCount** is a count of items contained in the **pObjects** array.
- **pObjects** is a pointer to an array of `VkDebugUtilsObjectNameInfoEXT` objects related to the detected issue. The array is roughly in order or importance, but the 0th element is always guaranteed to be the most important object for this message.

**Note**

This structure should only be considered valid during the lifetime of the triggered
Since adding queue and command buffer labels behaves like pushing and popping onto a stack, the order of both `pQueueLabels` and `pCmdBufLabels` is based on the order the labels were defined. The result is that the first label in either `pQueueLabels` or `pCmdBufLabels` will be the first defined (and therefore the oldest) while the last label in each list will be the most recent.

Note

`pQueueLabels` will only be non-NULL if one of the objects in `pObjects` can be related directly to a defined `VkQueue` which has had one or more labels associated with it.

Likewise, `pCmdBufLabels` will only be non-NULL if one of the objects in `pObjects` can be related directly to a defined `VkCommandBuffer` which has had one or more labels associated with it. Additionally, while command buffer labels allow for beginning and ending across different command buffers, the debug messaging framework cannot guarantee that labels in `pCmdBufLabels` will contain those defined outside of the associated command buffer. This is partially due to the fact that the association of one command buffer with another may not have been defined at the time the debug message is triggered.

Valid Usage (Implicit)

- VUID-VkDebugUtilsMessengerCallbackDataEXT-sType-sType
  sType **must** be `VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CALLBACK_DATA_EXT`

- VUID-VkDebugUtilsMessengerCallbackDataEXT-pNext-pNext
  pNext **must** be `NULL`

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

- VUID-VkDebugUtilsMessengerCallbackDataEXT-pMessageIdName-parameter
  If `pMessageIdName` is not `NULL`, `pMessageIdName` **must** be a null-terminated UTF-8 string

- VUID-VkDebugUtilsMessengerCallbackDataEXT-pMessage-parameter
  pMessage **must** be a null-terminated UTF-8 string

- VUID-VkDebugUtilsMessengerCallbackDataEXT-pQueueLabels-parameter
  If `queueLabelCount` is not `0`, `pQueueLabels` **must** be a valid pointer to an array of `queueLabelCount` valid `VkDebugUtilsLabelEXT` structures

- VUID-VkDebugUtilsMessengerCallbackDataEXT-pCmdBufLabels-parameter
  If `cmdBufLabelCount` is not `0`, `pCmdBufLabels` **must** be a valid pointer to an array of `cmdBufLabelCount` valid `VkDebugUtilsLabelEXT` structures

- VUID-VkDebugUtilsMessengerCallbackDataEXT-pObjects-parameter
  If `objectCount` is not `0`, `pObjects` **must** be a valid pointer to an array of `objectCount` valid `VkDebugUtilsObjectNameInfoEXT` structures

// Provided by VK_EXT_debug_utils
typedef VkFlags VkDebugUtilsMessengerCallbackDataFlagsEXT;

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

There may be times that a user wishes to intentionally submit a debug message. To do this, call:

```c
// Provided by VK_EXT_debug_utils
void vkSubmitDebugUtilsMessageEXT(
    VkInstance instance,
    VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,
    VkDebugUtilsMessageTypeFlagsEXT messageTypes,
    const VkDebugUtilsMessengerCallbackDataEXT* pCallbackData);
```

- `instance` is the debug stream's `VkInstance`.
- `messageSeverity` is a `VkDebugUtilsMessageSeverityFlagBitsEXT` value specifying the severity of this event/message.
- `messageTypes` is a bitmask of `VkDebugUtilsMessageTypeFlagBitsEXT` specifying which type of event(s) to identify with this message.
- `pCallbackData` contains all the callback related data in the `VkDebugUtilsMessengerCallbackDataEXT` structure.

The call will propagate through the layers and generate callback(s) as indicated by the message's flags. The parameters are passed on to the callback in addition to the `pUserData` value that was defined at the time the messenger was registered.

### Valid Usage

- VUID-vkSubmitDebugUtilsMessageEXT-objectType-02591
  The `objectType` member of each element of `pCallbackData->pObjects` must not be `VK_OBJECT_TYPE_UNKNOWN`

### Valid Usage (Implicit)

- VUID-vkSubmitDebugUtilsMessageEXT-instance-parameter
  `instance` must be a valid `VkInstance` handle
- VUID-vkSubmitDebugUtilsMessageEXT-messageSeverity-parameter
  `messageSeverity` must be a valid `VkDebugUtilsMessageSeverityFlagBitsEXT` value
- VUID-vkSubmitDebugUtilsMessageEXT-messageTypes-parameter
  `messageTypes` must be a valid combination of `VkDebugUtilsMessageTypeFlagBitsEXT` values
- VUID-vkSubmitDebugUtilsMessageEXT-messageTypes-requiredbitsetmask
  `messageTypes` must not be 0
To destroy a `VkDebugUtilsMessengerEXT` object, call:

```c
// Provided by VK_EXT_debug_utils
void vkDestroyDebugUtilsMessengerEXT(
    VkInstance instance, /* instance is the instance where the callback was created. */
    VkDebugUtilsMessengerEXT messenger, /* messenger is the VkDebugUtilsMessengerEXT object to destroy. messenger is an externally synchronized object and must not be used on more than one thread at a time. This means that vkDestroyDebugUtilsMessengerEXT must not be called when a callback is active. */
    const VkAllocationCallbacks* pAllocator); /* pAllocator controls host memory allocation as described in the Memory Allocation chapter. */
```

### Valid Usage (Implicit)
- **VUID-vkDestroyDebugUtilsMessengerEXT-instance-parameter**
  
  `instance must be a valid VkInstance handle`

- **VUID-vkDestroyDebugUtilsMessengerEXT-messenger-parameter**
  
  If `messenger` is not `VK_NULL_HANDLE`, `messenger must be a valid VkDebugUtilsMessengerEXT handle`

- **VUID-vkDestroyDebugUtilsMessengerEXT-pAllocator-null**

  `pAllocator must be NULL`

- **VUID-vkDestroyDebugUtilsMessengerEXT-messenger-parent**

  If `messenger` is a valid handle, it `must` have been created, allocated, or retrieved from `instance`

### Host Synchronization
- Host access to `messenger` `must` be externally synchronized

The application `must` ensure that `vkDestroyDebugUtilsMessengerEXT` is not executed in parallel with any Vulkan command that is also called with `instance` or child of `instance` as the dispatchable argument.

## 36.2. 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.

### 36.2.1. 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 the type of 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

- VUID-VkFaultData-pNext-05019
  `pNext` must be `NULL` or a valid pointer to an implementation-specific structure

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:

```
// 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:

```
// 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.
36.2.2. 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 \textit{faultQueryBehavior}.

For each filled \texttt{pFaults} entry, if \texttt{pNext} is not \texttt{NULL}, the implementation will fill in any implementation-specific structures applicable to that fault that are included in the \texttt{pNext} chain.

\begin{quote}
\textbf{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}.
\end{quote}

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

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
\textbf{Valid Usage} \par
\hspace{1cm} \hspace{1cm} \hspace{1cm}
\textbullet \hspace{1cm} \texttt{VUID-vkGetFaultData-pFaultCount-05020}
\texttt{pFaultCount must} be less than or equal to \texttt{maxQueryFaultCount} \\
\hline
\textbf{Valid Usage (Implicit)} \par
\hspace{1cm} \hspace{1cm} \hspace{1cm}
\textbullet \hspace{1cm} \texttt{VUID-vkGetFaultData-device-parameter}
\texttt{device must} be a valid \texttt{VkDevice} handle \\
\textbullet \hspace{1cm} \texttt{VUID-vkGetFaultData-faultQueryBehavior-parameter}
\texttt{faultQueryBehavior must} be a valid \texttt{VkFaultQueryBehavior} value \\
\textbullet \hspace{1cm} \texttt{VUID-vkGetFaultData-pUnrecordedFaults-parameter}
\texttt{pUnrecordedFaults must} be a valid pointer to a \texttt{VkBool32} value \\
\textbullet \hspace{1cm} \texttt{VUID-vkGetFaultData-pFaultCount-parameter}
\texttt{pFaultCount must} be a valid pointer to a \texttt{uint32_t} value \\
\textbullet \hspace{1cm} \texttt{VUID-vkGetFaultData-pFaults-parameter}
If \texttt{pFaults} is not \texttt{NULL}, \texttt{pFaults must} be a valid pointer to an array of \texttt{pFaultCount VkFaultData} structures \\
\textbullet \hspace{1cm} \texttt{VUID-vkGetFaultData-pFaultCount-arraylength}
If \texttt{pFaults} is not \texttt{NULL}, the value referenced by \texttt{pFaultCount must} be greater than 0 \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
\textbf{Return Codes} \par
\hspace{1cm} \hspace{1cm} \hspace{1cm}
\textbf{Success} \par
\hspace{1cm} \hspace{1cm} \hspace{1cm}
\hspace{1cm} \hspace{1cm} \hspace{1cm}
\textbullet \hspace{1cm} \texttt{VK_SUCCESS} \\
\textbullet \hspace{1cm} \texttt{VK_INCOMPLETE} \\
\hline
\end{tabular}
\end{table}
### Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

Possible values that **can** be set in `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 = 0,
} 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.

### 36.2.3. 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 VKSC_VERSION_1_0
typedef struct VkFaultCallbackInfo {
    VkStructureType sType;
    void* pNext;
    uint32_t faultCount;
    VkFaultData* pFaults;
    PFN_vkFaultCallbackFunction pfnFaultCallback;
} VkFaultCallbackInfo;
```

- **sType** is the type of 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.
Prior to Vulkan SC 1.0.11, the application was required to provide the \texttt{pFaults} array for fault callback data. This proved to be unwieldly 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 \texttt{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

- \texttt{VUID-VkFaultCallbackInfo-faultCount-05138}
  \texttt{faultCount} must either be 0, or equal to \texttt{VkPhysicalDeviceVulkanSC10Properties::maxCallbackFaultCount}

### Valid Usage (Implicit)

- \texttt{VUID-VkFaultCallbackInfo-sType-sType}
  \texttt{sType} must be \texttt{VK_STRUCTURE_TYPE_FAULT_CALLBACK_INFO}

- \texttt{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{faultCountVkFaultData} structures

- \texttt{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);
```
• *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 `PFN_vkFaultCallbackFunction` since the last call to this callback.

• *faultCount* will contain the number of reported faults in the array pointed to by *pFaults*.

• *pFaults* will point to an array of *faultCount* `VkFaultData` structures containing the fault information.

An implementation must only make calls to `pfnFaultCallback` during the execution of an API command. An implementation must only make calls into the application-provided fault callback 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 75. List of SPIR-V Capabilities and corresponding Vulkan features, extensions, or core version

<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>
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></td>
<td>VkPhysicalDeviceShaderImageAtomicInt64FeaturesEXT::shaderImageInt64Atomics</td>
</tr>
<tr>
<td>AtomicFloat32AddEXT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceShaderAtomicFloatFeaturesEXT::shaderBufferFloat32AtomicAdd</td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceShaderAtomicFloatFeaturesEXT::shaderSharedFloat32AtomicAdd</td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceShaderAtomicFloatFeaturesEXT::shaderImageFloat32AtomicAdd</td>
</tr>
<tr>
<td>AtomicFloat64AddEXT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceShaderAtomicFloatFeaturesEXT::shaderBufferFloat64AtomicAdd</td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceShaderAtomicFloatFeaturesEXT::shaderSharedFloat64AtomicAdd</td>
</tr>
<tr>
<td>Int64ImageEXT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceShaderImageAtomicInt64FeaturesEXT::shaderImageInt64Atomic</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>
</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>StorageImageMultisample</td>
<td>Vulkan feature that enables multisampling</td>
</tr>
<tr>
<td>UniformBufferArrayDynamicIndexing</td>
<td>Vulkan feature that enables dynamic index entries</td>
</tr>
<tr>
<td>SampledImageArrayDynamicIndexing</td>
<td>Vulkan feature that enables dynamic index entries</td>
</tr>
<tr>
<td>StorageBufferArrayDynamicIndexing</td>
<td>Vulkan feature that enables dynamic index entries</td>
</tr>
<tr>
<td>StorageImageArrayDynamicIndexing</td>
<td>Vulkan feature that enables dynamic index entries</td>
</tr>
<tr>
<td>ClipDistance</td>
<td>Vulkan feature that enables clipping</td>
</tr>
<tr>
<td>CullDistance</td>
<td>Vulkan feature that enables culling</td>
</tr>
<tr>
<td>ImageCubeArray</td>
<td>Vulkan feature that enables array usage</td>
</tr>
<tr>
<td>SampleRateShading</td>
<td>Vulkan feature that enables sample rate shading</td>
</tr>
<tr>
<td>SparseResidency</td>
<td>Vulkan feature that enables sparse residency</td>
</tr>
<tr>
<td>MinLod</td>
<td>Vulkan feature that enables min lod</td>
</tr>
<tr>
<td>SampledCubeArray</td>
<td>Vulkan feature that enables array usage</td>
</tr>
<tr>
<td>ImageMSArray</td>
<td>Vulkan feature that enables storage image multisample</td>
</tr>
<tr>
<td>StorageImageExtendedFormats</td>
<td>Vulkan feature that enables storage image multisample</td>
</tr>
<tr>
<td>InterpolationFunction</td>
<td>Vulkan feature that enables sample rate shading</td>
</tr>
<tr>
<td>StorageImageReadWithoutFormat</td>
<td>Vulkan feature that enables storage image multisample</td>
</tr>
<tr>
<td>StorageImageWriteWithoutFormat</td>
<td>Vulkan feature that enables storage image multisample</td>
</tr>
<tr>
<td>MultiViewport</td>
<td>Vulkan feature that enables multi viewport</td>
</tr>
<tr>
<td>SPIR-V</td>
<td>OpCapability</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Vulkan feature, extension, or core version</td>
<td></td>
</tr>
</tbody>
</table>

**DrawParameters**

- VkPhysicalDeviceVulkan11Features::shaderDrawParameters
- VkPhysicalDeviceShaderDrawParametersFeatures::shaderDrawParameters

**MultiView**

- VkPhysicalDeviceVulkan11Features::multiview

**DeviceGroup**

- VK_VERSION_1_1

**VariablePointersStorageBuffer**

- VkPhysicalDeviceVulkan11Features::variablePointersStorageBuffer

**VariablePointers**

- VkPhysicalDeviceVulkan11Features::variablePointers

**ShaderClockKHR**

- VK_KHR_shader_clock

**StencilExportEXT**

- VK_EXT_shader_stencil_export

**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
<table>
<thead>
<tr>
<th>SPIR-V Capability</th>
<th>Vulkan feature, extension, or core version</th>
</tr>
</thead>
<tbody>
<tr>
<td>GroupNonUniformShuffleRelative</td>
<td>VK_SUBGROUP_FEATURE_SHUFFLE_RELATIVE_BIT</td>
</tr>
<tr>
<td>GroupNonUniformClustered</td>
<td>VK_SUBGROUP_FEATURE_CLUSTERED_BIT</td>
</tr>
<tr>
<td>GroupNonUniformQuad</td>
<td>VK_SUBGROUP_FEATURE_QUAD_BIT</td>
</tr>
<tr>
<td>SampleMaskPostDepthCoverage</td>
<td>VK_EXT_post_depth_coverage</td>
</tr>
<tr>
<td>ShaderNonUniform</td>
<td>VK_VERSION_1_2</td>
</tr>
<tr>
<td>RuntimeDescriptorArray</td>
<td>VkPhysicalDeviceVulkan12Features::runtimeDescriptorArray</td>
</tr>
<tr>
<td>InputAttachmentArrayDynamicIndexing</td>
<td>VkPhysicalDeviceVulkan12Features::shaderInputAttachmentArrayDynamicIndexing</td>
</tr>
<tr>
<td>UniformTexelBufferArrayDynamicIndexing</td>
<td>VkPhysicalDeviceVulkan12Features::shaderUniformTexelBufferArrayDynamicIndexing</td>
</tr>
<tr>
<td>StorageTexelBufferArrayDynamicIndexing</td>
<td>VkPhysicalDeviceVulkan12Features::shaderStorageTexelBufferArrayDynamicIndexing</td>
</tr>
<tr>
<td>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>SPIR-V OpCapability</td>
<td>Vulkan feature, extension, or core version</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------------------------</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>DenormFlushToZero</td>
<td>VkPhysicalDeviceVulkan12Properties::shaderDenormFlushToZeroFloat16</td>
</tr>
<tr>
<td>SignedZeroInfNanPreserve</td>
<td>VkPhysicalDeviceVulkan12Properties::shaderSignedZeroInfNanPreserveFloat16</td>
</tr>
<tr>
<td>RoundingModeRTE</td>
<td>VkPhysicalDeviceVulkan12Properties::shaderRoundingModeRTEFloat16</td>
</tr>
<tr>
<td>RoundingModeRTZ</td>
<td>VkPhysicalDeviceVulkan12Properties::shaderRoundingModeRTZFloat16</td>
</tr>
<tr>
<td>PhysicalStorageBufferAddresses</td>
<td>VkPhysicalDeviceVulkan12Features::bufferDeviceAddress</td>
</tr>
<tr>
<td>FragmentShaderSampleInterlockEXT</td>
<td>VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT::fragmentShaderSampleInterlock</td>
</tr>
<tr>
<td>FragmentShaderPixelInterlockEXT</td>
<td>VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT::fragmentShaderPixelInterlock</td>
</tr>
<tr>
<td>FragmentShaderShadingRateInterlockEXT</td>
<td>VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT::fragmentShaderShadingRateInterlock</td>
</tr>
<tr>
<td>DemoteToHelperInvocationEXT</td>
<td>VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT::shaderDemoteToHelperInvocation</td>
</tr>
</tbody>
</table>
SPIR-V `OpCapability`

Vulkan feature, extension, or core version

FragmentShadingRateKHR

`VkPhysicalDeviceFragmentShadingRateFeaturesKHR::pipelineFragmentShadingRate`

`VkPhysicalDeviceFragmentShadingRateFeaturesKHR::primitiveFragmentShadingRate`

`VkPhysicalDeviceFragmentShadingRateFeaturesKHR::attachmentFragmentShadingRate`

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 76. List of SPIR-V Extensions and corresponding Vulkan extensions or core version**

<table>
<thead>
<tr>
<th>SPIR-V <code>OpExtension</code></th>
<th>Vulkan extension or core version</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SPV_KHR_variable_pointers</code></td>
<td><code>VK_VERSION_1_1</code></td>
</tr>
<tr>
<td><code>SPV_KHR_shader_draw_parameters</code></td>
<td><code>VK_VERSION_1_1</code></td>
</tr>
<tr>
<td><code>SPV_KHR_8bit_storage</code></td>
<td><code>VK_VERSION_1_2</code></td>
</tr>
<tr>
<td><code>SPV_KHR_16bit_storage</code></td>
<td><code>VK_VERSION_1_1</code></td>
</tr>
<tr>
<td><code>SPV_KHR_shader_clock</code></td>
<td><code>VK_KHR_shader_clock</code></td>
</tr>
<tr>
<td><code>SPV_KHR_float_controls</code></td>
<td><code>VK_VERSION_1_2</code></td>
</tr>
</tbody>
</table>
SPIR-V OpExtension

Vulkan extension or core version

<table>
<thead>
<tr>
<th>Extension</th>
<th>Vulkan Core Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPV_KHR_storage_buffer_storage_class</td>
<td>VK_VERSION_1_1</td>
</tr>
<tr>
<td>SPV_KHR_post_depth_coverage</td>
<td>VK_EXT_post_depth_coverage</td>
</tr>
<tr>
<td>SPV_EXT_shader_stencil_export</td>
<td>VK_EXT_shader_stencil_export</td>
</tr>
<tr>
<td>SPV_EXT_shader_viewport_index_layer</td>
<td>VK_VERSION_1_2</td>
</tr>
<tr>
<td>SPV_EXT_descriptor_indexing</td>
<td>VK_VERSION_1_2</td>
</tr>
<tr>
<td>SPV_KHR_vulkan_memory_model</td>
<td>VK_VERSION_1_2</td>
</tr>
<tr>
<td>SPV_KHR_physical_storage_buffer</td>
<td>VK_VERSION_1_2</td>
</tr>
<tr>
<td>SPV_EXT_fragment_shader_interlock</td>
<td>VK_EXT_fragment_shader_interlock</td>
</tr>
<tr>
<td>SPV_EXT_demote_to_helper_invocation</td>
<td>VK_EXT_shader_demote_to_helper_invocation</td>
</tr>
<tr>
<td>SPV_KHR_fragment_shading_rate</td>
<td>VK_KHR_fragment_shading_rate</td>
</tr>
<tr>
<td>SPV_EXT_shader_image_int64</td>
<td>VK_EXT_shader_image_atomic_int64</td>
</tr>
<tr>
<td>SPV_KHR_terminate_invocation</td>
<td>VK_KHR_shader_terminate_invocation</td>
</tr>
<tr>
<td>SPV_KHR_multiview</td>
<td>VK_VERSION_1_1</td>
</tr>
<tr>
<td>SPV_EXT_shader_atomic_float_add</td>
<td>VK_EXT_shader_atomic_float_add</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 models

- **VUID-StandaloneSpirv-None-04638**
  Scope for memory **must** be limited to Device, QueueFamily, Workgroup, ShaderCallKHR, Subgroup, or Invocation

- **VUID-StandaloneSpirv-None-04639**
  If the Scope for memory is Workgroup, then it **must** only be used in the task, mesh, or compute execution models

- **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 models

- **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-None-04643**
  Storage Class **must** be limited to UniformConstant, Input, Uniform, Output, Workgroup, Private, Function, PushConstant, Image, StorageBuffer, RayPayloadKHR, IncomingRayPayloadKHR, HitAttributeKHR, CallableDataKHR, IncomingCallableDataKHR, ShaderRecordBufferKHR, or PhysicalStorageBuffer

- **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 models

- **VUID-StandaloneSpirv-None-04645**
  If the Storage Class is Workgroup, then it **must** only be used in the task, mesh, or compute execution models

- **VUID-StandaloneSpirv-OpAtomicStore-04730**
  OpAtomicStore **must** not use Acquire, AcquireRelease, or SequentiallyConsistent memory semantics
OpAtomicLoad must not use Release, AcquireRelease, or SequentiallyConsistent memory semantics

OpMemoryBarrier must use one of Acquire, Release, AcquireRelease, or SequentiallyConsistent memory semantics

OpMemoryBarrier must include at least one storage class

If the semantics for OpControlBarrier includes one of Acquire, Release, AcquireRelease, or SequentiallyConsistent memory semantics, then it must include at least one storage class

Any OpVariable with an Initializer operand must have Output, Private, Function, or Workgroup as its Storage Class operand

Any OpVariable with an Initializer operand and Workgroup as its Storage Class operand must use OpConstantNull as the initializer

Scope for OpReadClockKHR must be limited to Subgroup or Device

The OriginLowerLeft execution mode must not be used; fragment entry points must declare OriginUpperLeft

The PixelCenterInteger execution mode must not be used (pixels are always centered at half-integer coordinates)

Any variable in the UniformConstant storage class must be typed as either OpTypeImage, OpTypeSampler, OpTypeSampledImage, OpTypeAccelerationStructureKHR, or an array of one of these types

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)

OpTypeImage must have a “Sampled” operand of 1 (sampled image) or 2 (storage image)

The converted bit width, signedness, and numeric type of the Image Format operand of an OpTypeImage must match the Sampled Type, as defined in Image Format and Type Matching

If an OpImageTexelPointer is used in an atomic operation, the image type of the image parameter to OpImageTexelPointer must have an image format of R64i, R64ui, R32f, R32i, or R32ui.
OpImageQuerySizeLod, OpImageQueryLod, and OpImageQueryLevels must only consume an “Image” operand whose type has its “Sampled” operand set to 1.

An OpTypeImage with a “Dim” operand of SubpassData must have an “Arrayed” operand of 0 (non-arrayed) and a “Sampled” operand of 2 (storage image).

The (u,v) coordinates used for a SubpassData must be the <id> of a constant vector (0,0), or if a layer coordinate is used, must be a vector that was formed with constant 0 for the u and v components.

Objects of types OpTypeImage, OpTypeSampler, OpTypeSampledImage, and arrays of these types must not be stored to or modified.

Any image operation must use at most one of the Offset, ConstOffset, and ConstOffsets image operands.

Image operand Offset must only be used with OpImage*Gather instructions.

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 must not consume an image whose Dim is 3D.

Objects of types OpTypeAccelerationStructureKHR and arrays of this type must not be stored to or modified.

The value of the “Hit Kind” operand of OpReportIntersectionKHR must be in the range [0,127].

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 not be used with BuiltIn.

The Location decorations must be used on user-defined variables.

The Location or Component decorations must not be used with BuiltIn.
The **Location** decorations **must** be used on an **OpVariable** with a structure type that is not a block.

- VUID-StandaloneSpirv-Location-04918
  The **Location** decorations **must** not be used on the members of **OpVariable** with a structure type that is decorated with **Location**.

- VUID-StandaloneSpirv-Location-04919
  The **Location** decorations **must** be used on each member of **OpVariable** with a structure that is a block not decorated with **Location**.

- VUID-StandaloneSpirv-Component-04920
  The **Component** decoration value **must** not be greater than 3.

- VUID-StandaloneSpirv-Component-04921
  If the **Component** decoration is used on an **OpVariable** that has a **OpTypeVector** type with a **Component Type** with a **Width** that is less than or equal to 32, the sum of its **Component Count** and the **Component** decoration value **must** be less than 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 4.

- VUID-StandaloneSpirv-Component-04923
  The **Component** decorations value **must** not be 1 or 3 for scalar or two-component 64-bit data types.

- VUID-StandaloneSpirv-Component-04924
  The **Component** decorations **must** not used with any type that is not a scalar or vector.

- VUID-StandaloneSpirv-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 **Output** storage class in a fragment shader.

- VUID-StandaloneSpirv-Flat-06202
  The **Flat**, **NoPerspective**, **Sample**, and **Centroid** decorations **must** not be used on variables with the **Input** storage class in a vertex shader.

- VUID-StandaloneSpirv-Flat-04744
  Any variable with integer or double-precision floating-point type and with **Input** storage class in a fragment shader, **must** be decorated **Flat**.

- VUID-StandaloneSpirv-ViewportRelativeNV-04672
  The **ViewportRelativeNV** decoration **must** only be used on a variable decorated with **Layer** in the vertex, tessellation evaluation, or geometry shader stages.

- VUID-StandaloneSpirv-ViewportRelativeNV-04673
  The **ViewportRelativeNV** decoration **must** not be used unless a variable decorated with one
of ViewportIndex or ViewportMaskNV is also statically used by the same OpEntryPoint

- VUID-StandaloneSpirv-ViewportMaskNV-04674
  The ViewportMaskNV and ViewportIndex decorations must not both be statically used by one or more OpEntryPoint’s that form the pre-rasterization shader stages of a graphics pipeline

- VUID-StandaloneSpirv-FPRoundingMode-04675
  Rounding modes other than round-to-nearest-even and round-towards-zero must not be used for the FPRoundingMode decoration

- VUID-StandaloneSpirv-FPRoundingMode-04676
  The FPRoundingMode decoration must only be used for a width-only conversion instruction whose only uses are Object operands of OpStore instructions storing through a pointer to a 16-bit floating-point object in the StorageBuffer, PhysicalStorageBuffer, Uniform, or Output storage class

- VUID-StandaloneSpirv-Invariant-04677
  Variables decorated with Invariant and variables with structure types that have any members decorated with Invariant must be in the Output or Input storage class, Invariant used on an Input storage class variable or structure member has no effect

- VUID-StandaloneSpirv-VulkanMemoryModel-04678
  If the VulkanMemoryModel capability is not declared, the Volatile decoration must be used on any variable declaration that includes one of the SMIDNV, WarpIDNV, SubgroupSize, SubgroupLocalInvocationId, SubgroupEqMask, SubgroupGeMask, SubgroupGtMask, SubgroupLeMask, or SubgroupLtMask BuiltIn decorations when used in the ray generation, closest hit, miss, intersection, or callable shaders, or with the RayTmaxKHR Builtin decoration when used in an intersection shader

- VUID-StandaloneSpirv-VulkanMemoryModel-04679
  If the VulkanMemoryModel capability is declared, the OpLoad instruction must use the Volatile memory semantics when it accesses into any variable that includes one of the SMIDNV, WarpIDNV, SubgroupSize, SubgroupLocalInvocationId, SubgroupEqMask, SubgroupGeMask, SubgroupGtMask, SubgroupLeMask, or SubgroupLtMask BuiltIn decorations when used in the ray generation, closest hit, miss, intersection, or callable shaders, or with the RayTmaxKHR Builtin decoration when used in an intersection shader

- VUID-StandaloneSpirv-OpTypeRuntimeArray-04680
  OpTypeRuntimeArray must only be used for the last member of an OpTypeStruct that is in the StorageBuffer or PhysicalStorageBuffer storage class decorated as Block, or that is in the Uniform storage class decorated as BufferBlock

- 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 T2 of a variable V, unless either: a) T is equal to T2, b) V is declared in the Function, or Private storage classes, c) V is a non-Block variable in the Workgroup storage class, or d) V is an interface variable with an additional level of arrayness, as described in interface matching, and T is the member type of the array type T2

- 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
For each compute shader entry point, either a LocalSize or LocalSizeId execution mode, or an object decorated with the WorkgroupSize decoration must be specified.

For compute shaders using the DerivativeGroupQuadsNV execution mode, the first two dimensions of the local workgroup size must be a multiple of two.

For compute shaders using the DerivativeGroupLinearNV execution mode, the product of the dimensions of the local workgroup size must be a multiple of four.

If OpGroupNonUniformBallotBitCount is used, the group operation must be limited to Reduce, InclusiveScan, or ExclusiveScan.

The Pointer operand of all atomic instructions must have a Storage Class limited to Uniform, Workgroup, Image, StorageBuffer, or PhysicalStorageBuffer.

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.

The size of any output block containing any member decorated with Offset that is a 64-bit type must be a multiple of 8.

The first member of an output block 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

- VUID-StandaloneSpirv-XfbBuffer-04696
  For any two variables or block members in the output interface of the entry point being compiled with the same `XfbBuffer` value, the ranges determined by the `Offset` decoration and the size of the type must not overlap

- VUID-StandaloneSpirv-XfbBuffer-04697
  All block members in the output interface of the entry point being compiled that are in the same block and have a declared or inherited `XfbBuffer` decoration must specify the same `XfbBuffer` value

- VUID-StandaloneSpirv-RayPayloadKHR-04698
  `RayPayloadKHR` storage class must only be used in ray generation, closest hit or miss shaders

- VUID-StandaloneSpirv-IncomingRayPayloadKHR-04699
  `IncomingRayPayloadKHR` storage class must only be used in closest hit, any-hit, or miss shaders

- VUID-StandaloneSpirv-IncomingRayPayloadKHR-04700
  There must be at most one variable with the `IncomingRayPayloadKHR` storage class in the input interface of an entry point

- VUID-StandaloneSpirv-HitAttributeKHR-04701
  `HitAttributeKHR` storage class must only be used in intersection, any-hit, or closest hit shaders

- VUID-StandaloneSpirv-HitAttributeKHR-04702
  There must be at most one variable with the `HitAttributeKHR` storage class in the input interface of an entry point

- VUID-StandaloneSpirv-HitAttributeKHR-04703
  A variable with `HitAttributeKHR` storage class must only be written to in an intersection shader

- VUID-StandaloneSpirv-CallableDataKHR-04704
  `CallableDataKHR` storage class must only be used in ray generation, closest hit, miss, and callable shaders

- VUID-StandaloneSpirv-IncomingCallableDataKHR-04705
  `IncomingCallableDataKHR` storage class must only be used in callable shaders

- VUID-StandaloneSpirv-IncomingCallableDataKHR-04706
  There must be at most one variable with the `IncomingCallableDataKHR` storage class in the input interface of an entry point

- VUID-StandaloneSpirv-Base-04707
  The `Base` operand of `OpPtrAccessChain` must point to one of the following: `Workgroup`, if `VariablePointers` is enabled; `StorageBuffer`, if `VariablePointers` or `VariablePointersStorageBuffer` is enabled; `PhysicalStorageBuffer`, if the `PhysicalStorageBuffer64` addressing model is enabled

- VUID-StandaloneSpirv-PhysicalStorageBuffer64-04708
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.

- **VUID-StandaloneSpirv-PhysicalStorageBuffer64-04709**
  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.

- **VUID-StandaloneSpirv-PhysicalStorageBuffer64-04710**
  If the PhysicalStorageBuffer64 addressing model is enabled, OpConvertUToPtr and OpConvertPtrToU must use an integer type whose Width is 64.

- **VUID-StandaloneSpirv-OpTypeForwardPointer-04711**
  OpTypeForwardPointer must have a storage class of PhysicalStorageBuffer.

- **VUID-StandaloneSpirv-None-04745**
  All variables with a storage class of PushConstant declared as an array must only be accessed by dynamically uniform indices.

- **VUID-StandaloneSpirv-Result-04780**
  The Result Type operand of any OpImageRead or OpImageSparseRead instruction must be a vector of four components.

- **VUID-StandaloneSpirv-Base-04781**
  The Base operand of any OpBitCount, OpBitReverse, OpBitFieldInsert, OpBitFieldSExtract, or OpBitFieldUExtract instruction must be a 32-bit integer scalar or a vector of 32-bit integers.

- **VUID-StandaloneSpirv-DescriptorSet-06491**
  If a variable is decorated by DescriptorSet or Binding, the storage class must correspond to an entry in Shader Resource and Storage Class Correspondence.

### 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-shaderSubgroupClock-06267**
  If shaderSubgroupClock is not enabled, the Subgroup scope must not be used for OpReadClockKHR.

- **VUID-RuntimeSpirv-shaderDeviceClock-06268**

1510
If `shaderDeviceClock` is not enabled, the `Device` scope **must** not be used for `OpReadClockKHR`.

- **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-Location-06272**  
  The sum of `Location` and the number of locations the variable it decorates consumes **must** be less than or equal to the value for the matching `Execution Model` defined in Shader Input and Output Locations.

- **VUID-RuntimeSpirv-Fragment-06427**  
  When blending is enabled and one of the dual source blend modes is in use, the maximum number of output attachments written to in the `Fragment Execution Model` **must** be less than or equal to `maxFragmentDualSrcAttachments`.

- **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-OpTypeRuntimeArray-06273**  
  `OpTypeRuntimeArray` **must** only be used for an array of variables with storage class `Uniform`, `StorageBuffer`, or `UniformConstant`, or for the outermost dimension of an array of arrays of such variables if the `runtimeDescriptorArray` feature is enabled.

- **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-None-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-None-06280**  
  `shaderBufferFloat32Atomics`, or `shaderBufferFloat32AtomicAdd`, or
shaderBufferFloat64Atomics, or shaderBufferFloat64AtomicAdd must be enabled for floating-point atomic operations to be supported on a Pointer with a Storage Class of StorageBuffer.

- VUID-RuntimeSpirv-None-06281
  shaderSharedFloat32Atomics, or shaderSharedFloat32AtomicAdd, or shaderSharedFloat64Atomics, or shaderSharedFloat64AtomicAdd must be enabled for floating-point atomic operations to be supported on a Pointer with a Storage Class of Workgroup.

- VUID-RuntimeSpirv-None-06282
  shaderImageFloat32Atomics or shaderImageFloat32AtomicAdd must be enabled for 32-bit floating-point atomic operations to be supported on a Pointer with a Storage Class of Image.

- VUID-RuntimeSpirv-None-06283
  sparseImageFloat32Atomics or sparseImageFloat32AtomicAdd must be enabled for 32-bit floating-point atomic operations to be supported on sparse images.

- VUID-RuntimeSpirv-None-06288
  shaderImageInt64Atomics must be enabled for 64-bit integer atomic operations to be supported on a Pointer with a Storage Class of Image.

- VUID-RuntimeSpirv-None-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-None-06290
  If denormBehaviorIndependence is VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_NONE, then the entry point must use the same denormals execution mode for all floating-point types.

- VUID-RuntimeSpirv-None-06291
  If roundingModeIndependence is VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_32_BIT_ONLY, then the entry point must use the same rounding execution mode for both 16-bit and 64-bit floating-point types.

- VUID-RuntimeSpirv-None-06292
  If roundingModeIndependence is VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_NONE, then the entry point must use the same rounding execution mode for all floating-point types.

- VUID-RuntimeSpirv-None-06293
  If shaderSignedZeroInfNanPreserveFloat16 is VK_FALSE, then SignedZeroInfNanPreserve for 16-bit floating-point type must not be used.

- VUID-RuntimeSpirv-None-06294
  If shaderSignedZeroInfNanPreserveFloat32 is VK_FALSE, then SignedZeroInfNanPreserve for 32-bit floating-point type must not be used.

- VUID-RuntimeSpirv-None-06295
  If shaderSignedZeroInfNanPreserveFloat64 is VK_FALSE, then SignedZeroInfNanPreserve for 64-bit floating-point type must not be used.

- VUID-RuntimeSpirv-None-06296
  If shaderDenormPreserveFloat16 is VK_FALSE, then DenormPreserve for 16-bit floating-point
type must not be used.

- VUID-RuntimeSpirv-shaderDenormPreserveFloat32-06297
  If shaderDenormPreserveFloat32 is VK_FALSE, then DenormPreserve for 32-bit floating-point type must not be used.

- VUID-RuntimeSpirv-shaderDenormPreserveFloat64-06298
  If shaderDenormPreserveFloat64 is VK_FALSE, then DenormPreserve for 64-bit floating-point type must not be used.

- VUID-RuntimeSpirv-shaderDenormFlushToZeroFloat16-06299
  If shaderDenormFlushToZeroFloat16 is VK_FALSE, then DenormFlushToZero for 16-bit floating-point type must not be used.

- VUID-RuntimeSpirv-shaderDenormFlushToZeroFloat32-06300
  If shaderDenormFlushToZeroFloat32 is VK_FALSE, then DenormFlushToZero for 32-bit floating-point type must not be used.

- VUID-RuntimeSpirv-shaderDenormFlushToZeroFloat64-06301
  If shaderDenormFlushToZeroFloat64 is VK_FALSE, then DenormFlushToZero for 64-bit floating-point type must not be used.

- VUID-RuntimeSpirv-shaderRoundingModeRTEFloat16-06302
  If shaderRoundingModeRTEFloat16 is VK_FALSE, then RoundingModeRTE for 16-bit floating-point type must not be used.

- VUID-RuntimeSpirv-shaderRoundingModeRTEFloat32-06303
  If shaderRoundingModeRTEFloat32 is VK_FALSE, then RoundingModeRTE for 32-bit floating-point type must not be used.

- VUID-RuntimeSpirv-shaderRoundingModeRTEFloat64-06304
  If shaderRoundingModeRTEFloat64 is VK_FALSE, then RoundingModeRTE for 64-bit floating-point type must not be used.

- VUID-RuntimeSpirv-shaderRoundingModeRTZFloat16-06305
  If shaderRoundingModeRTZFloat16 is VK_FALSE, then RoundingModeRTZ for 16-bit floating-point type must not be used.

- VUID-RuntimeSpirv-shaderRoundingModeRTZFloat32-06306
  If shaderRoundingModeRTZFloat32 is VK_FALSE, then RoundingModeRTZ for 32-bit floating-point type must not be used.

- VUID-RuntimeSpirv-shaderRoundingModeRTZFloat64-06307
  If shaderRoundingModeRTZFloat64 is VK_FALSE, then RoundingModeRTZ for 64-bit floating-point type must not be used.

- VUID-RuntimeSpirv-PhysicalStorageBuffer64-06314
  If the PhysicalStorageBuffer64 addressing model is enabled any load or store through a physical pointer type must be aligned to a multiple of the size of the largest scalar type in the pointed-to type.

- VUID-RuntimeSpirv-PhysicalStorageBuffer64-06315
  If the PhysicalStorageBuffer64 addressing model is enabled the pointer value of a memory access instruction must be at least as aligned as specified by the Aligned memory access operand.
• **VUID-RuntimeSpirv-DescriptorSet-06323**
  DescriptorSet and Binding decorations **must** obey the constraints on storage class, type, and descriptor type described in DescriptorSet and Binding Assignment.

• **VUID-RuntimeSpirv-None-06335**
  shaderBufferFloat32Atomics, or shaderBufferFloat32AtomicAdd, or shaderSharedFloat32Atomics, or shaderSharedFloat32AtomicAdd, or shaderImageFloat32Atomics, or shaderImageFloat32AtomicAdd **must** be enabled for 32-bit floating point atomic operations.

• **VUID-RuntimeSpirv-None-06336**
  shaderBufferFloat64Atomics, or shaderBufferFloat64AtomicAdd, or shaderSharedFloat64Atomics, or shaderSharedFloat64AtomicAdd **must** be enabled for 64-bit floating point atomic operations.

• **VUID-RuntimeSpirv-NonWritable-06340**
  If fragmentStoresAndAtomics is not enabled, then all storage image, storage texel buffer, and storage buffer variables in the fragment stage **must** be decorated with the NonWritable decoration.

• **VUID-RuntimeSpirv-NonWritable-06341**
  If vertexPipelineStoresAndAtomics is not enabled, then all storage image, storage texel buffer, and storage buffer variables in the vertex, tessellation, and geometry stages **must** be decorated with the NonWritable decoration.

• **VUID-RuntimeSpirv-OpAtomic-05091**
  If shaderAtomicInstructions is not enabled, the SPIR-V Atomic Instructions listed in 3.37.18 (OpAtomic*) **must** not be used [SCID-1].

• **VUID-RuntimeSpirv-None-06342**
  If subgroupQuadOperationsInAllStages is VK_FALSE, then quad subgroup operations **must** not be used except for in fragment and compute stages.

• **VUID-RuntimeSpirv-None-06343**
  Group operations with subgroup scope **must** not be used if the shader stage is not in subgroupSupportedStages.

• **VUID-RuntimeSpirv-Offset-06344**
  The first element of the Offset operand of InterpolateAtOffset **must** be greater than or equal to:

  \[
  \text{frag}_\text{width} \times \min\text{InterpolationOffset}
  \]

  where \( \text{frag}_\text{width} \) 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:

  \[
  \text{frag}_\text{width} \times (\max\text{InterpolationOffset} + \text{ULP}) - \text{ULP}
  \]
where \( \text{fragwidth} \) is the width of the current fragment in pixels and \( ULP = \frac{1}{2^{\text{subPixelInterpolationOffsetBits}}} \).

- **VUID-RuntimeSpirv-Offset-06346**

  The second element of the \( \text{Offset} \) operand of \( \text{InterpolateAtOffset} \) must be greater than or equal to:

  \[
  \text{fragheight} \times \min\text{InterpolationOffset}
  \]

  where \( \text{fragheight} \) is the height of the current fragment in pixels.

- **VUID-RuntimeSpirv-Offset-06347**

  The second element of the \( \text{Offset} \) operand of \( \text{InterpolateAtOffset} \) must be less than or equal to:

  \[
  \text{fragheight} \times (\max\text{InterpolationOffset} + ULP) - ULP
  \]

  where \( \text{fragheight} \) is the height of the current fragment in pixels and \( ULP = \frac{1}{2^{\text{subPixelInterpolationOffsetBits}}} \).

- **VUID-RuntimeSpirv-x-06429**

  The \( x \) size in \( \text{LocalSize} \) or \( \text{LocalSizeId} \) must be less than or equal to \( \text{VkPhysicalDeviceLimits::maxComputeWorkGroupSize}[0] \)

- **VUID-RuntimeSpirv-y-06430**

  The \( y \) size in \( \text{LocalSize} \) or \( \text{LocalSizeId} \) must be less than or equal to \( \text{VkPhysicalDeviceLimits::maxComputeWorkGroupSize}[1] \)

- **VUID-RuntimeSpirv-z-06431**

  The \( z \) size in \( \text{LocalSize} \) or \( \text{LocalSizeId} \) must be less than or equal to \( \text{VkPhysicalDeviceLimits::maxComputeWorkGroupSize}[2] \)

- **VUID-RuntimeSpirv-x-06432**

  The product of \( x \) size, \( y \) size, and \( z \) size in \( \text{LocalSize} \) or \( \text{LocalSizeId} \) must be less than or equal to \( \text{VkPhysicalDeviceLimits::maxComputeWorkGroupInvocations} \)

- **VUID-RuntimeSpirv-LocalSizeId-06433**

  The execution mode \( \text{LocalSizeId} \) must not be used

- **VUID-RuntimeSpirv-OpVariable-06373**

  Any \( \text{OpVariable} \) with \( \text{Workgroup} \) as its \( \text{Storage Class} \) must not have an \( \text{Initializer} \) operand

- **VUID-RuntimeSpirv-OpImage-06376**

  If an \( \text{OpImage\_Gather} \) operation has an image operand of \( \text{Offset} \), \( \text{ConstOffset} \), or \( \text{ConstOffsets} \) the offset value must be greater than or equal to \( \text{minTexelGatherOffset} \)

- **VUID-RuntimeSpirv-OpImage-06377**

  If an \( \text{OpImage\_Gather} \) operation has an image operand of \( \text{Offset} \), \( \text{ConstOffset} \), or \( \text{ConstOffsets} \) the offset value must be less than or equal to \( \text{maxTexelGatherOffset} \)

- **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

**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.

OpMatrixTimesVector, OpMatrixTimesMatrix, OpOuterProduct, OpDot; and the following extended instructions for GLSL: Round, RoundEven, Trunc, FAbs, Floor, Ceil, Fract, Radians, Degrees, Sin, Cos, Tan, Asin, Acos, Atan, Sinh, Cosh, Tanh, Asinh, Acosh, Atanh, Atan2, Pow, Exp, Log, Exp2, Log2, Sqrt, InverseSqrt, Determinant, MatrixInverse, Modf, ModfStruct, FMin, FMax, FClamp, FMix, Step, SmoothStep, Fma, UnpackHalf2x16, UnpackDouble2x32, Length, Distance, Cross, Normalize, FaceForward, Reflect, Refract, NMin, NMax, NC clamp. Other SPIR-V instructions (except those excluded above) may also flush denormalized values.


The precision of double-precision instructions is at least that of single precision.

The precision of operations is defined either in terms of rounding, as an error bound in ULP, or as inherited from a formula as follows.

**Correctly Rounded**

Operations described as “correctly rounded” will return the infinitely precise result, x, rounded so as to be representable in floating-point. The rounding mode is not specified, unless the entry point is declared with the RoundingModeRTE or the RoundingModeRTZ execution mode. These execution modes affect only correctly rounded SPIR-V instructions. These execution modes do not affect OpQuantizeToF16. If the rounding mode is not specified then this rounding is implementation specific, subject to the following rules. If x is exactly representable then x will be returned. Otherwise, either the floating-point value closest to x 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 \([x - n \times ulp(x), x + n \times ulp(x)]\). The function \( ulp(x) \) is defined as follows:

If there exist non-equal floating-point numbers \( a \) and \( b \) such that \( a \leq x \leq b \) then \( ulp(x) \) is the minimum possible distance between such numbers, \( ulp(x) = \min_b a, b - a \). If such numbers do not exist then \( ulp(x) \) is defined to be the difference between the two finite floating-point numbers nearest to x.

Where the range of allowed return values includes any value of magnitude larger than that of the largest representable finite floating-point number, operations may, additionally, return either an infinity of the appropriate sign or the finite number with the largest magnitude of the appropriate sign. If the infinitely precise result of the operation is not mathematically defined then the value
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 \( E = \max(|x - F_{\text{mid}}|, |x - F_{\text{max}}|) \). 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 77. 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>x</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>
The `OpFRem` and `OpFMod` instructions use cheap approximations of remainder, and the error can be large due to the discontinuity in `trunc()` and `floor()`. This can produce mathematically unexpected results in some cases, such as `FMod(x, x)` computing `x` rather than `0`, and can also cause the result to have a different sign than the infinitely precise result.

**Table 78. 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><code>fma()</code></td>
<td>Inherited from <code>OpFMul</code> followed by <code>OpFAdd</code>.</td>
<td></td>
</tr>
<tr>
<td><code>exp(x), exp2(x)</code></td>
<td>3 + 2 ×</td>
<td>x</td>
</tr>
<tr>
<td><code>log(), log2()</code></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><code>pow(x, y)</code></td>
<td>Inherited from <code>exp2(y × log2(x))</code>.</td>
<td></td>
</tr>
<tr>
<td><code>sqrt()</code></td>
<td>Inherited from <code>exp2(y × log2(x))</code>.</td>
<td></td>
</tr>
<tr>
<td><code>inversesqrt()</code></td>
<td>Inherited from <code>1.0 / inversesqrt()</code>.</td>
<td></td>
</tr>
<tr>
<td><code>radians(x)</code></td>
<td>Inherited from <code>x × π/180</code>.</td>
<td></td>
</tr>
<tr>
<td><code>degrees(x)</code></td>
<td>Inherited from <code>x × π/180</code>.</td>
<td></td>
</tr>
<tr>
<td><code>sin()</code></td>
<td>Absolute error ≤ 2⁻¹¹ inside the range [−π, π].</td>
<td>Absolute error ≤ 2⁻⁷ inside the range [−π, π].</td>
</tr>
<tr>
<td><code>cos()</code></td>
<td>Absolute error ≤ 2⁻¹¹ inside the range [−π, π].</td>
<td>Absolute error ≤ 2⁻⁷ inside the range [−π, π].</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 <code>atan2(x, sqrt(1.0 - x × x))</code>.</td>
<td></td>
</tr>
<tr>
<td><code>acos(x)</code></td>
<td>Inherited from <code>atan2(1.0 - x × x, x)</code>.</td>
<td></td>
</tr>
<tr>
<td><code>atan(), atan2()</code></td>
<td>4096 ULP</td>
<td>5 ULP.</td>
</tr>
<tr>
<td><code>sinh(x)</code></td>
<td>Inherited from <code>(exp(x) - exp(−x)) × 0.5</code>.</td>
<td></td>
</tr>
<tr>
<td><code>cosh(x)</code></td>
<td>Inherited from <code>(exp(x) + exp(−x)) × 0.5</code>.</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 <code>log(x + sqrt(x × x +1.0))</code>.</td>
<td></td>
</tr>
<tr>
<td><code>acosh(x)</code></td>
<td>Inherited from <code>log(x + sqrt(x × x +1.0))</code>.</td>
<td></td>
</tr>
<tr>
<td><code>atanh(x)</code></td>
<td>Inherited from (\frac{1.0 + x}{1.0 - x}) × 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 <code>sqrt(dot(x, x))</code>.</td>
<td></td>
</tr>
</tbody>
</table>
### Instruction | Single precision, unless decorated with RelaxedPrecision | Half precision
---|---|---
\( \text{distance}(x, y) \) | Inherited from \( \text{length}(x - y) \). | 
\( \text{cross}() \) | Inherited from \( \text{OpFSub(OpFMul, OpFMul)} \). | 
\( \text{normalize}(x) \) | Inherited from \( \frac{x}{\text{length}(x)} \). | 
\( \text{faceforward}(N, I, N\text{Ref}) \) | Inherited from \( \text{dot}(N\text{Ref}, I) < 0.0 \ ? \ N : -N \). | 
\( \text{reflect}(x, y) \) | Inherited from \( x - 2.0 \times \text{dot}(y, x) \times y \). | 
\( \text{refract}(I, N, \eta) \) | Inherited from \( k < 0.0 \ ? \ 0.0 \ ? \ (eta \times \text{dot}(N, I) + \sqrt{k}) \times N \), where \( k = 1 - \eta \times \eta \times (1.0 - \text{dot}(N, I) \times \text{dot}(N, I)) \). | 
\( \text{round} \) | Correctly rounded. | 
\( \text{roundEven} \) | Correctly rounded. | 
\( \text{trunc} \) | Correctly rounded. | 
\( \text{fabs} \) | Correctly rounded. | 
\( \text{fsign} \) | Correctly rounded. | 
\( \text{floor} \) | Correctly rounded. | 
\( \text{ceil} \) | Correctly rounded. | 
\( \text{fract} \) | Correctly rounded. | 
\( \text{modf} \) | Correctly rounded. | 
\( \text{fmin} \) | Correctly rounded. | 
\( \text{fmax} \) | Correctly rounded. | 
\( \text{fclamp} \) | Correctly rounded. | 
\( \text{fmix}(x, y, a) \) | Inherited from \( x \times (1.0 - a) + y \times a \). | 
\( \text{step} \) | Correctly rounded. | 
\( \text{smoothStep}(\text{edge0}, \text{edge1}, x) \) | Inherited from \( t \times t \times (3.0 - 2.0 \times t) \), where \( t = \text{clamp}(\frac{x - \text{edge0}}{\text{edge1} - \text{edge0}}, 0.0, 1.0) \). | 
\( \text{nmin} \) | Correctly rounded. | 
\( \text{nmax} \) | Correctly rounded. | 
\( \text{nclamp} \) | Correctly rounded. | 

GLSL.std.450 extended instructions specifically defined in terms of the above instructions inherit the above errors. GLSL.std.450 extended instructions not listed above and not defined in terms of the above have undefined precision.

For the \( \text{OpSRem} \) and \( \text{OpSMod} \) instructions, if either operand is negative the result is undefined.

Note
While the \( \text{OpSRem} \) and \( \text{OpSMod} \) instructions are supported by the Vulkan...
environment, they require non-negative values and thus do not enable additional functionality beyond what \texttt{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 \texttt{Image Operands} and the underlying image's \texttt{Sampled Type} as follows:

1. If the instruction's \texttt{Image Operands} contains the \texttt{SignExtend} operand then the access is signed.
2. If the instruction's \texttt{Image Operands} contains the \texttt{ZeroExtend} operand then the access is unsigned.
3. Otherwise, the image accesses signedness matches that of the \texttt{Sampled Type} of the \texttt{OpTypeImage} being accessed.

**Image Format and Type Matching**

When specifying the \texttt{Image Format} of an \texttt{OpTypeImage}, the converted bit width and type, as shown in the table below, **must** match the \texttt{Sampled Type}. The signedness **must** match the signedness of any access to the image.

\begin{table}
<table>
<thead>
<tr>
<th>Image Format</th>
<th>Type</th>
<th>Bit Width</th>
<th>Signedness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown</td>
<td>Any</td>
<td>Any</td>
<td>Any</td>
</tr>
</tbody>
</table>
\end{table}

*Note*

Formatted accesses are always converted from a shader readable type to the resource's format or vice versa via \texttt{Format Conversion} for reads and \texttt{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 \texttt{Image Format}, the \texttt{Sampled Type} **must** be the type described in the \texttt{Type} column of the below table, with its \texttt{Literal Width} set to that in the \texttt{Bit Width} column. Every access that is made to the image **must** have a signedness equal to that in the \texttt{Signedness} column (where applicable).
<table>
<thead>
<tr>
<th>Image Format</th>
<th>Type</th>
<th>Bit Width</th>
<th>Signedness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rgba32f</td>
<td>OpTypeFloat</td>
<td>32</td>
<td>N/A</td>
</tr>
<tr>
<td>Rg32f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R32f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba16f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg16f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba16Snorm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg16Snorm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16Snorm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgb10A2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R11fG11fB10f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba8Snorm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg8Snorm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8Snorm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Compatibility Between SPIR-V Image Formats And Vulkan Formats

SPIR-V Image Format values are compatible with VkFormat values as defined below:

**Table 79. 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>Rgba32f</td>
<td>VK_FORMAT_R32G32B32A32_SFLOAT</td>
</tr>
<tr>
<td>Rgba16f</td>
<td>VK_FORMAT_R16G16B16A16_SFLOAT</td>
</tr>
<tr>
<td>R32f</td>
<td>VK_FORMAT_R32_SFLOAT</td>
</tr>
<tr>
<td>Rgba8</td>
<td>VK_FORMAT_R8G8B8A8_UNORM</td>
</tr>
<tr>
<td>Rgba8Snorm</td>
<td>VK_FORMAT_R8G8B8A8_SNORM</td>
</tr>
<tr>
<td>Rg32f</td>
<td>VK_FORMAT_R32G32_SFLOAT</td>
</tr>
<tr>
<td>Rg16f</td>
<td>VK_FORMAT_R16G16_SFLOAT</td>
</tr>
<tr>
<td>R64i</td>
<td>VK_FORMAT_R16G16B16A16_SFLOAT</td>
</tr>
<tr>
<td>R64ui</td>
<td>VK_FORMAT_R16G16B16A16_SFLOAT</td>
</tr>
<tr>
<td>R11fG11fB10f</td>
<td>VK_FORMAT_B10G11R11_UFLOAT_PACK32</td>
</tr>
<tr>
<td>SPIR-V Image Format</td>
<td>Compatible Vulkan Format</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>R16f</td>
<td>VK_FORMAT_R16_SFLOAT</td>
</tr>
<tr>
<td>Rgba16</td>
<td>VK_FORMAT_R16G16B16A16_UNORM</td>
</tr>
<tr>
<td>Rgb10A2</td>
<td>VK_FORMAT_A2B10G10R10_UNORM_PACK32</td>
</tr>
<tr>
<td>Rg16</td>
<td>VK_FORMAT_R16G16_UNORM</td>
</tr>
<tr>
<td>Rg8</td>
<td>VK_FORMAT_R8G8_UNORM</td>
</tr>
<tr>
<td>R16</td>
<td>VK_FORMAT_R16_UNORM</td>
</tr>
<tr>
<td>R8</td>
<td>VK_FORMAT_R8_UNORM</td>
</tr>
<tr>
<td>Rgba16Snorm</td>
<td>VK_FORMAT_R16G16B16A16_SNORM</td>
</tr>
<tr>
<td>Rg16Snorm</td>
<td>VK_FORMAT_R16G16_SNORM</td>
</tr>
<tr>
<td>Rg8Snorm</td>
<td>VK_FORMAT_R8G8_SNORM</td>
</tr>
<tr>
<td>R16Snorm</td>
<td>VK_FORMAT_R16_SNORM</td>
</tr>
<tr>
<td>R8Snorm</td>
<td>VK_FORMAT_R8_SNORM</td>
</tr>
<tr>
<td>Rgba32i</td>
<td>VK_FORMAT_R32G32B32A32_SINT</td>
</tr>
<tr>
<td>Rgba16i</td>
<td>VK_FORMAT_R16G16B16A16_SINT</td>
</tr>
<tr>
<td>Rgba8i</td>
<td>VK_FORMAT_R8G8B8A8_SINT</td>
</tr>
<tr>
<td>R32i</td>
<td>VK_FORMAT_R32_SINT</td>
</tr>
<tr>
<td>Rg32i</td>
<td>VK_FORMAT_R32G32_SINT</td>
</tr>
<tr>
<td>Rg16i</td>
<td>VK_FORMAT_R16G16_SINT</td>
</tr>
<tr>
<td>Rg8i</td>
<td>VK_FORMAT_R8G8_SINT</td>
</tr>
<tr>
<td>R16i</td>
<td>VK_FORMAT_R16_SINT</td>
</tr>
<tr>
<td>R8i</td>
<td>VK_FORMAT_R8_SINT</td>
</tr>
<tr>
<td>Rgba32ui</td>
<td>VK_FORMAT_R32G32B32A32_UINT</td>
</tr>
<tr>
<td>Rgba16ui</td>
<td>VK_FORMAT_R16G16B16A16_UINT</td>
</tr>
<tr>
<td>Rgba8ui</td>
<td>VK_FORMAT_R8G8B8A8_UINT</td>
</tr>
<tr>
<td>R32ui</td>
<td>VK_FORMAT_R32_UINT</td>
</tr>
<tr>
<td>Rgb10a2ui</td>
<td>VK_FORMAT_A2B10G10R10_UINT_PACK32</td>
</tr>
<tr>
<td>Rg32ui</td>
<td>VK_FORMAT_R32G32_UINT</td>
</tr>
<tr>
<td>Rg16ui</td>
<td>VK_FORMAT_R16G16_UINT</td>
</tr>
<tr>
<td>Rg8ui</td>
<td>VK_FORMAT_R8G8_UINT</td>
</tr>
<tr>
<td>R16ui</td>
<td>VK_FORMAT_R16_UINT</td>
</tr>
<tr>
<td>R8ui</td>
<td>VK_FORMAT_R8_UINT</td>
</tr>
<tr>
<td>R64i</td>
<td>VK_FORMAT_R64_SINT</td>
</tr>
<tr>
<td>R64ui</td>
<td>VK_FORMAT_R64_UINT</td>
</tr>
</tbody>
</table>
Appendix B: Memory Model

Agent

*Operation* is a general term for any task that is executed on the system.

*Note*

An operation is by definition something that is executed. Thus if an instruction is skipped due to control flow, it does not constitute an operation.

Each operation is executed by a particular *agent*. Possible agents include each shader invocation, each host thread, and each fixed-function stage of the pipeline.

Memory Location

A *memory location* identifies unique storage for 8 bits of data. Memory operations access a *set of memory locations* consisting of one or more memory locations at a time, e.g. an operation accessing a 32-bit integer in memory would read/write a set of four memory locations. Memory operations that access whole aggregates *may* access any padding bytes between elements or members, but no padding bytes at the end of the aggregate. Two sets of memory locations *overlap* if the intersection of their sets of memory locations is non-empty. A memory operation *must* not affect memory at a memory location not within its set of memory locations.

Memory locations for buffers and images are explicitly allocated in *VkDeviceMemory* objects, and are implicitly allocated for SPIR-V variables in each shader invocation.

Allocation

The values stored in newly allocated memory locations are determined by a SPIR-V variable’s initializer, if present, or else are undefined. At the time an allocation is created there have been no memory operations to any of its memory locations. The initialization is not considered to be a memory operation.

*Note*

For tessellation control shader output variables, a consequence of initialization not being considered a memory operation is that some implementations may need to insert a barrier between the initialization of the output variables and any reads of those variables.

Memory Operation

For an operation *A* and memory location *M*:

- *A reads* *M* if and only if the data stored in *M* is an input to *A*.
- *A writes* *M* if and only if the data output from *A* is stored to *M*.
• A accesses \( M \) if and only if it either reads or writes (or both) \( M \).

\[ \text{Note} \]
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 \( \text{vkGetBufferDeviceAddress} \) 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://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.html#DynamicInstance) as a way of referring to a particular execution of a static instruction. Program-order is an ordering on dynamic instances of instructions executed by a single shader invocation:

• (Basic block): If instructions A and B are in the same basic block, and A is listed in the module before B, then the \( n \)'th dynamic instance of A is program-ordered before the \( n \)'th dynamic instance of B.

• (Branch): The dynamic instance of a branch or switch instruction is program-ordered before the dynamic instance of the OpLabel instruction to which it transfers control.

• (Call entry): The dynamic instance of an OpFunctionCall instruction is program-ordered before the dynamic instances of the OpFunctionParameter instructions and the body of the called function.

• (Call exit): The dynamic instance of the instruction following an OpFunctionCall instruction is program-ordered after the dynamic instance of the return instruction executed by the called function.

• (Transitive Closure): If dynamic instance A of any instruction is program-ordered before dynamic instance B of any instruction and B is program-ordered before dynamic instance C of any instruction then A is program-ordered before C.

• (Complete definition): No other dynamic instances are program-ordered.
For instructions executed on the host, the source language defines the program-order relation (e.g. as “sequenced-before”).

Scope

Atomic and barrier instructions include scopes which identify sets of shader invocations that must obey the requested ordering and atomicity rules of the operation, as defined below.

The various scopes are described in detail in the Shaders chapter.

Atomic Operation

An atomic operation on the device is any SPIR-V operation whose name begins with OpAtomic. An atomic operation on the host is any operation performed with an std::atomic typed object.

Each atomic operation has a memory scope and a semantics. Informally, the scope determines which other agents it is atomic with respect to, and the semantics constrains its ordering against other memory accesses. Device atomic operations have explicit scopes and semantics. Each host atomic operation implicitly uses the CrossDevice scope, and uses a memory semantics equivalent to a C++ std::memory_order value of relaxed, acquire, release, acq_rel, or seq_cst.

Two atomic operations A and B are potentially-mutually-ordered if and only if all of the following are true:

• They access the same set of memory locations.
• They use the same reference.
• A is in the instance of B’s memory scope.
• B is in the instance of A’s memory scope.
• A and B are not the same operation (irreflexive).

Two atomic operations A and B are mutually-ordered if and only if they are potentially-mutually-ordered and any of the following are true:

• A and B are both device operations.
• A and B are both host operations.
• A is a device operation, B is a host operation, and the implementation supports concurrent host- and device-atomics.

Note

If two atomic operations are not mutually-ordered, and if their sets of memory locations overlap, then each must be synchronized against the other as if they were non-atomic operations.
Scoped Modification Order

For a given atomic write A, all atomic writes that are mutually-ordered with A occur in an order known as A's scoped modification order. A's scoped modification order relates no other operations.

**Note**
Invocations outside the instance of A's memory scope may observe the values at A's set of memory locations becoming visible to it in an order that disagrees with the scoped modification order.

**Note**
It is valid to have non-atomic operations or atomics in a different scope instance to the same set of memory locations, as long as they are synchronized against each other as if they were non-atomic (if they are not, it is treated as a data race). That means this definition of A's scoped modification order could include atomic operations that occur much later, after intervening non-atomics. That is a bit non-intuitive, but it helps to keep this definition simple and non-circular.

Memory Semantics

Non-atomic memory operations, by default, may be observed by one agent in a different order than they were written by another agent.

Atomics and some synchronization operations include memory semantics, which are flags that constrain the order in which other memory accesses (including non-atomic memory accesses and availability and visibility operations) performed by the same agent can be observed by other agents, or can observe accesses by other agents.

Device instructions that include semantics are OpAtomic*, OpControlBarrier, OpMemoryBarrier, and OpMemoryNamedBarrier. Host instructions that include semantics are some std::atomic methods and memory fences.

SPIR-V supports the following memory semantics:

- Relaxed: No constraints on order of other memory accesses.
- Acquire: A memory read with this semantic performs an acquire operation. A memory barrier with this semantic is an acquire barrier.
- Release: A memory write with this semantic performs a release operation. A memory barrier with this semantic is a release barrier.
- AcquireRelease: A memory read-modify-write operation with this semantic performs both an acquire operation and a release operation, and inherits the limitations on ordering from both of those operations. A memory barrier with this semantic is both a release and acquire barrier.

**Note**
SPIR-V does not support “consume” semantics on the device.
The memory semantics operand also includes *storage class semantics* which indicate which storage classes are constrained by the synchronization. SPIR-V storage class semantics include:

- UniformMemory
- WorkgroupMemory
- ImageMemory
- OutputMemory

Each SPIR-V memory operation accesses a single storage class. Semantics in synchronization operations can include a combination of storage classes.

The UniformMemory storage class semantic applies to accesses to memory in the PhysicalStorageBuffer, Uniform and StorageBuffer storage classes. The WorkgroupMemory storage class semantic applies to accesses to memory in the Workgroup storage class. The ImageMemory storage class semantic applies to accesses to memory in the Image storage class. The OutputMemory storage class semantic applies to accesses to memory in the Output storage class.

Note

Informally, these constraints limit how memory operations can be reordered, and these limits apply not only to the order of accesses as performed in the agent that executes the instruction, but also to the order the effects of writes become visible to all other agents within the same instance of the instruction’s memory scope.

Note

Release and acquire operations in different threads can act as synchronization operations, to guarantee that writes that happened before the release are visible after the acquire. (This is not a formal definition, just an Informative forward reference.)

Note

The OutputMemory storage class semantic is only useful in tessellation control shaders, which is the only execution model where output variables are shared between invocations.

The memory semantics operand can also include availability and visibility flags, which apply availability and visibility operations as described in *availability and visibility*. The availability/visibility flags are:

- MakeAvailable: Semantics must be Release or AcquireRelease. Performs an availability operation before the release operation or barrier.
- MakeVisible: Semantics must be Acquire or AcquireRelease. Performs a visibility operation after the acquire operation or barrier.

The specifics of these operations are defined in *Availability and Visibility Semantics*.

Host atomic operations may support a different list of memory semantics and synchronization
operations, depending on the host architecture and source language.

**Release Sequence**

After an atomic operation $A$ performs a release operation on a set of memory locations $M$, the *release sequence headed by $A$* is the longest continuous subsequence of $A$’s scoped modification order that consists of:

- the atomic operation $A$ as its first element
- atomic read-modify-write operations on $M$ by any agent

**Note**
The atomics in the last bullet must be mutually-ordered with $A$ by virtue of being in $A$’s scoped modification order.

**Note**
This intentionally omits “atomic writes to $M$ performed by the same agent that performed $A$”, which is present in the corresponding C++ definition.

**Synchronizes-With**

*Synchronizes-with* is a relation between operations, where each operation is either an atomic operation or a memory barrier (aka fence on the host).

If $A$ and $B$ are atomic operations, then $A$ synchronizes-with $B$ if and only if all of the following are true:

- $A$ performs a release operation
- $B$ performs an acquire operation
- $A$ and $B$ are mutually-ordered
- $B$ reads a value written by $A$ or by an operation in the release sequence headed by $A$

*OpControlBarrier, OpMemoryBarrier, and OpMemoryNamedBarrier* are *memory barrier* instructions in SPIR-V.

If $A$ is a release barrier and $B$ is an atomic operation that performs an acquire operation, then $A$ synchronizes-with $B$ if and only if all of the following are true:

- there exists an atomic write $X$ (with any memory semantics)
- $A$ is program-ordered before $X$
- $X$ and $B$ are mutually-ordered
- $B$ reads a value written by $X$ or by an operation in the release sequence headed by $X$
  - If $X$ is relaxed, it is still considered to head a hypothetical release sequence for this rule
- $A$ and $B$ are in the instance of each other’s memory scopes
• X’s storage class is in A’s semantics.

If A is an atomic operation that performs a release operation and B is an acquire barrier, then A synchronizes-with B if and only if all of the following are true:

• there exists an atomic read X (with any memory semantics)
• X is program-ordered before B
• X and A are mutually-ordered
• X reads a value written by A or by an operation in the release sequence headed by A
• A and B are in the instance of each other’s memory scopes
• X’s storage class is in B’s semantics.

If A is a release barrier and B is an acquire barrier, then A synchronizes-with B if all of the following are true:

• there exists an atomic write X (with any memory semantics)
• A is program-ordered before X
• there exists an atomic read Y (with any memory semantics)
• Y is program-ordered before B
• X and Y are mutually-ordered
• Y reads the value written by X or by an operation in the release sequence headed by X
  ◦ If X is relaxed, it is still considered to head a hypothetical release sequence for this rule
• A and B are in the instance of each other’s memory scopes
• X’s and Y’s storage class is in A’s and B’s semantics.
  ◦ NOTE: X and Y must have the same storage class, because they are mutually ordered.

If A is a release barrier, B is an acquire barrier, and C is a control barrier (where A can equal C, and B can equal C), then A synchronizes-with B if all of the following are true:

• A is program-ordered before (or equals) C
• C is program-ordered before (or equals) B
• A and B are in the instance of each other’s memory scopes
• A and B are in the instance of C’s execution scope

Note
This is similar to the barrier-barrier synchronization above, but with a control barrier filling the role of the relaxed atomics.

Let F be an ordering of fragment shader invocations, such that invocation $F_1$ is ordered before invocation $F_2$ if and only if $F_1$ and $F_2$ overlap as described in Fragment Shader Interlock and $F_1$ executes the interlocked code before $F_2$.

If A is an OpEndInvocationInterlockEXT instruction and B is an OpBeginInvocationInterlockEXT instruction.
instruction, then A synchronizes-with B if the agent that executes A is ordered before the agent that executes B in F. A and B are both considered to have FragmentInterlock memory scope and semantics of UniformMemory and ImageMemory, and A is considered to have Release semantics and B is considered to have Acquire semantics.

Note

OpBeginInvocationInterlockEXT and OpBeginInvocationInterlockEXT do not perform implicit availability or visibility operations. Usually, shaders using fragment shader interlock will declare the relevant resources as coherent to get implicit per-instruction availability and visibility operations.

No other release and acquire barriers synchronize-with each other.

**System-Synchronizes-With**

System-synchronizes-with is a relation between arbitrary operations on the device or host. Certain operations system-synchronize-with each other, which informally means the first operation occurs before the second and that the synchronization is performed without using application-visible memory accesses.

If there is an execution dependency between two operations A and B, then the operation in the first synchronization scope system-synchronizes-with the operation in the second synchronization scope.

Note

This covers all Vulkan synchronization primitives, including device operations executing before a synchronization primitive is signaled, wait operations happening before subsequent device operations, signal operations happening before host operations that wait on them, and host operations happening before vkQueueSubmit. The list is spread throughout the synchronization chapter, and is not repeated here.

System-synchronizes-with implicitly includes all storage class semantics and has CrossDevice scope.

If A system-synchronizes-with B, we also say A is system-synchronized-before B and B is system-synchronized-after A.

**Private vs. Non-Private**

By default, non-atomic memory operations are treated as private, meaning such a memory operation is not intended to be used for communication with other agents. Memory operations with the NonPrivatePointer/NonPrivateTexel bit set are treated as non-private, and are intended to be used for communication with other agents.

More precisely, for private memory operations to be Location-Ordered between distinct agents requires using system-synchronizes-with rather than shader-based synchronization. Non-private memory operations still obey program-order.
Atomic operations are always considered non-private.

**Inter-Thread-Happens-Before**

Let SC be a non-empty set of storage class semantics. Then (using template syntax) operation A *inter-thread-happens-before<SC>* operation B if and only if any of the following is true:

- A system-synchronizes-with B
- A synchronizes-with B, and both A and B have all of SC in their semantics
- A is an operation on memory in a storage class in SC or that has all of SC in its semantics, B is a release barrier or release atomic with all of SC in its semantics, and A is program-ordered before B
- A is an acquire barrier or acquire atomic with all of SC in its semantics, B is an operation on memory in a storage class in SC or that has all of SC in its semantics, and A is program-ordered before B
- A and B are both host operations and A inter-thread-happens-before B as defined in the host language specification
- A inter-thread-happens-before<SC> some X and X inter-thread-happens-before<SC> B

**Happens-Before**

Operation A *happens-before* operation B if and only if any of the following is true:

- A is program-ordered before B
- A inter-thread-happens-before<SC> B for some set of storage classes SC

*Happens-after* is defined similarly.

**Note**

Unlike C++, happens-before is not always sufficient for a write to be visible to a read. Additional availability and visibility operations may be required for writes to be visible-to other memory accesses.

**Note**

Happens-before is not transitive, but each of program-order and inter-thread-happens-before<SC> are transitive. These can be thought of as covering the “single-threaded” case and the “multi-threaded” case, and it is not necessary (and not valid) to form chains between the two.

**Availability and Visibility**

*Availability* and *visibility* are states of a write operation, which (informally) track how far the write has permeated the system, i.e. which agents and references are able to observe the write. Availability state is per *memory domain*. Visibility state is per (agent,reference) pair. Availability

---

1533
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.
- **fragment interlock instance** - accessible by fragment shader agents that overlap, corresponding to the FragmentInterlock scope.
- **workgroup instance** - accessible by shader agents in the same workgroup, corresponding to the Workgroup scope.
- **subgroup instance** - accessible by shader agents in the same subgroup, corresponding to the Subgroup scope.

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, overlapping fragment shader invocation, in any shader invocation, or anywhere on the device, or host. The shader, queue family instance, fragment interlock instance, workgroup instance, and subgroup instance domains are only used for shader-based availability/visibility operations, in other cases writes can be made available from/visible to the shader via the device domain.

*Availability operations, visibility operations, and memory domain operations* alter the state of the write operations that happen-before them, and which are included in their source scope to be available or visible to their destination scope.

- For an availability operation, the source scope is a set of (agent,reference,memory location) tuples, and the destination scope is a set of memory domains.
- For a memory domain operation, the source scope is a memory domain and the destination scope is a memory domain.
- For a visibility operation, the source scope is a set of memory domains and the destination scope is a set of (agent,reference,memory location) tuples.

How the scopes are determined depends on the specific operation. Availability and memory domain operations expand the set of memory domains to which the write is available. Visibility
operations expand the set of (agent,reference,memory location) tuples to which the write is visible.

Recall that availability and visibility states are per-memory location, and let W be a write operation to one or more locations performed by agent A via reference R. Let L be one of the locations written. (W,L) (the write W to L), is initially not available to any memory domain and only visible to (A,R,L). An availability operation AV that happens-after W and that includes (A,R,L) in its source scope makes (W,L) available to the memory domains in its destination scope.

A memory domain operation DOM that happens-after AV and for which (W,L) is available in the source scope makes (W,L) available in the destination memory domain.

A visibility operation VIS that happens-after AV (or DOM) and for which (W,L) is available in any domain in the source scope makes (W,L) visible to all (agent,reference,L) tuples included in its destination scope.

If write $W_2$ happens-after W, and their sets of memory locations overlap, then W will not be available/visible to all agents/references for those memory locations that overlap (and future AV/DOM/VIS ops cannot revive W’s write to those locations).

Availability, memory domain, and visibility operations are treated like other non-atomic memory accesses for the purpose of memory semantics, meaning they can be ordered by release-acquire sequences or memory barriers.

An availability chain is a sequence of availability operations 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 \texttt{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 \texttt{VK_ACCESS_HOST_READ_BIT} or \texttt{VK_ACCESS_HOST_WRITE_BIT}, then the dependency includes a memory domain operation from device domain to host domain.

\texttt{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.

\texttt{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).

\texttt{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:

• \texttt{Device} scope uses the shader domain

• \texttt{QueueFamily} scope uses the queue family instance domain

• \texttt{FragmentInterlock} scope uses the fragment interlock instance domain

• \texttt{Workgroup} scope uses the workgroup instance domain

• \texttt{Subgroup} uses the subgroup instance domain

• \texttt{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 fragment interlock instance or workgroup instance or subgroup instance domain, they will also see writes that were made visible via the device domain, i.e. those writes previously performed by non-shader agents and made visible via API commands.

**Note**

It is expected that all invocations in a subgroup execute on the same processor with the same path to memory, and thus availability and visibility operations with subgroup scope can be expected to be “free”.

**Location-Ordered**

Let X and Y be memory accesses to overlapping sets of memory locations M, where X 🏷️ Y. Let \((A_X, R_X)\) be the agent and reference used for X, and \((A_Y, R_Y)\) be the agent and reference used for Y. For now, let “→” denote happens-before and “\(→_{rcpo}\)” denote the reflexive closure of program-ordered before.

If \(D_1\) and \(D_2\) are different memory domains, then let \(\text{DOM}(D_1, D_2)\) 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
• \( X \) is a read, both \( X \) and \( Y \) are non-private, and \( X \rightarrow Y \)

• \( X \) is a read, and \( X \) (transitively) system-synchronizes with \( Y \)

• If \( R_x = R_y \) and \( A_x \) and \( A_y \) access a common memory domain \( D \) (e.g. are in the same workgroup instance if \( D \) is the workgroup instance domain), and both \( X \) and \( Y \) are non-private:
  ◦ \( X \) is a write, \( Y \) is a write, \( AVC(A_x,R_x,D,L) \) is an availability chain making \((X,L)\) available to domain \( D \), and \( X \rightarrow_r\text{AVC}(A_x,R_x,D,L) \rightarrow Y \)
  ◦ \( X \) is a write, \( Y \) is a read, \( AVC(A_x,R_x,D,L) \) is an availability chain making \((X,L)\) available to domain \( D \), \( VISC(A_y,R_y,D,L) \) is a visibility chain making writes to \( L \) available in domain \( D \) visible to \( Y \), and \( X \rightarrow_r\text{AVC}(A_x,R_x,D,L) \rightarrow VISC(A_y,R_y,D,L) \rightarrow_r Y \)
  ◦ If \( \text{VkPhysicalDeviceVulkanMemoryModelFeatures}::\text{vulkanMemoryModelAvailabilityVisibilityChains} = \text{VK_FALSE} \), then \( AVC \) and \( VISC \) must each only have a single element in the chain, in each sub-bullet above.

• Let \( D_x \) and \( D_y \) each be either the device domain or the host domain, depending on whether \( A_x \) and \( A_y \) execute on the device or host:
  ◦ \( X \) is a write and \( Y \) is a write, and \( X \rightarrow AV(A_x,R_x,D_x,L) \rightarrow \text{DOM}(D_x,D_y) \rightarrow Y \)
  ◦ \( X \) is a write and \( Y \) is a read, and \( X \rightarrow AV(A_x,R_x,D_x,L) \rightarrow \text{DOM}(D_x,D_y) \rightarrow \text{VIS}(A_y,R_y,D_y,L) \rightarrow Y \)

\[ \text{Note} \]

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 \( X \neq Y \), and at least one of \( X \) and \( Y \) is a write, and \( X \) and \( Y \) are not mutually-ordered atomic operations. If there does not exist a location-ordered relation between \( X \) and \( Y \) for each location in \( M \), then there is a data race.

Applications must ensure that no data races occur during the execution of their application.

\[ \text{Note} \]

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 \( M_o \) be a non-empty subset of \( M \). Then \( X \) is visible-to \( Y \) for memory locations \( M_o \) if and only if all of the following are true:
• X is location-ordered before Y for each location L in M₂.
• There does not exist another write Z to any location L in M₂ such that X is location-ordered before Z for location L and Z is location-ordered before Y for location L.

If X is visible-to Y, then Y reads the value written by X for locations M₂.

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 (M₂) 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, scope modification order, and reads-from).
• If A is a write and B is a read, then B must take its value from A or a write later than A in A’s scoped modification order (no cycles between location-order, scoped modification order, and...
• 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 80. 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.

**Table 81. Mapping of Vulkan ETC formats to descriptions**

<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 82. 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>
ASTC textures containing HDR block encodings **should** be passed to the API using an ASTC SFLOAT texture format.

Note
---

An HDR block in a texture passed using a LDR UNORM format will return the appropriate ASTC error color if the implementation supports only the ASTC LDR profile, but may result in either the error color or a decompressed HDR color if the implementation supports HDR decoding.

**ASTC decode mode**

If the `VK_EXT_astc_decode_mode` extension is enabled, the decode mode is determined as follows:

Table 83. Mapping of Vulkan ASTC decoding format to ASTC decoding modes

<table>
<thead>
<tr>
<th>VkFormat</th>
<th>Decoding mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_R16G16B16A16_SFLOAT</td>
<td>decode_float16</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8A8_UNORM</td>
<td>decode_unorm8</td>
</tr>
<tr>
<td>VK_FORMAT_E5B9G9R9_UFLOAT_PACK32</td>
<td>decode_rgbe5</td>
</tr>
</tbody>
</table>

Otherwise, the ASTC decode mode is `decode_float16`.

---
Note that an implementation **may** use HDR mode when linear LDR mode is requested unless the decode mode is `decode_unorm8`. 
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_separate_stencil_usage
- 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**

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
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
- vkGetBufferOpaqueCaptureAddress
- vkGetDeviceMemoryOpaqueCaptureAddress
- vkGetSemaphoreCounterValue
- vkResetQueryPool
- vkSignalSemaphore
- vkWaitSemaphores

New Structures

- VkAttachmentDescription2
- VkAttachmentReference2
- VkBufferDeviceAddressInfo
- VkConformanceVersion
- VkDeviceMemoryOpaqueCaptureAddressInfo
- VkFramebufferAttachmentImageInfo
- VkRenderPassCreateInfo2
- VkSemaphoreSignalInfo
- VkSemaphoreWaitInfo
- VkSubpassBeginInfo
- VkSubpassDependency2
- VkSubpassDescription2
- `VkSubpassEndInfo`
- **Extending `VkAttachmentDescription2`**:
  - `VkAttachmentDescriptionStencilLayout`
- **Extending `VkAttachmentReference2`**:
  - `VkAttachmentReferenceStencilLayout`
- **Extending `VkBufferCreateInfo`**:
  - `VkBufferOpaqueCaptureAddressCreateInfo`
- **Extending `VkDescriptorSetAllocateInfo`**:
  - `VkDescriptorSetVariableDescriptorCountAllocateInfo`
- **Extending `VkDescriptorSetLayoutCreateInfo`**:
  - `VkDescriptorSetLayoutBindingFlagsCreateInfo`
- **Extending `VkDescriptorSetLayoutSupport`**:
  - `VkDescriptorSetVariableDescriptorCountLayoutSupport`
- **Extending `VkFramebufferCreateInfo`**:
  - `VkFramebufferAttachmentsCreateInfo`
- **Extending `VkImageCreateInfo`, `VkPhysicalDeviceImageFormatInfo2`**:
  - `VkImageStencilUsageCreateInfo`
- **Extending `VkImageCreateInfo`, `VkSwapchainCreateInfoKHR`, `VkPhysicalDeviceImageFormatInfo2`**:
  - `VkImageFormatListCreateInfo`
- **Extending `VkMemoryAllocateInfo`**:
  - `VkMemoryOpaqueCaptureAddressAllocateInfo`
- **Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`**:
  - `VkPhysicalDevice8BitStorageFeatures`
  - `VkPhysicalDeviceBufferDeviceAddressFeatures`
  - `VkPhysicalDeviceDescriptorIndexingFeatures`
  - `VkPhysicalDeviceHostQueryResetFeatures`
  - `VkPhysicalDeviceImagelessFramebufferFeatures`
  - `VkPhysicalDeviceScalarBlockLayoutFeatures`
  - `VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures`
  - `VkPhysicalDeviceShaderAtomicInt64Features`
  - `VkPhysicalDeviceShaderFloat16Int8Features`
  - `VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures`
  - `VkPhysicalDeviceTimelineSemaphoreFeatures`
  - `VkPhysicalDeviceUniformBufferStandardLayoutFeatures`
• `VkPhysicalDeviceVulkan11Features`
• `VkPhysicalDeviceVulkan12Features`
• `VkPhysicalDeviceVulkanMemoryModelFeatures`

• **Extending** `VkPhysicalDeviceProperties2`:
  • `VkPhysicalDeviceDepthStencilResolveProperties`
  • `VkPhysicalDeviceDescriptorIndexingProperties`
  • `VkPhysicalDeviceDriverProperties`
  • `VkPhysicalDeviceFloatControlsProperties`
  • `VkPhysicalDeviceSamplerFilterMinmaxProperties`
  • `VkPhysicalDeviceTimelineSemaphoreProperties`
  • `VkPhysicalDeviceVulkan11Properties`
  • `VkPhysicalDeviceVulkan12Properties`

• **Extending** `VkRenderPassBeginInfo`:
  • `VkRenderPassAttachmentBeginInfo`

• **Extending** `VkSamplerCreateInfo`:
  • `VkSamplerReductionModeCreateInfo`

• **Extending** `VkSemaphoreCreateInfo`, `VkPhysicalDeviceExternalSemaphoreInfo`:
  • `VkSemaphoreTypeCreateInfo`

• **Extending** `VkSubmitInfo`, `VkBindSparseInfo`:
  • `VkTimelineSemaphoreSubmitInfo`

• **Extending** `VkSubpassDescription2`:
  • `VkSubpassDescriptionDepthStencilResolve`

**New Enums**

• `VkDescriptorBindingFlagBits`
• `VkDriverId`
• `VkResolveModeFlagBits`
• `VkSamplerReductionMode`
• `VkSemaphoreType`
• `VkSemaphoreWaitFlagBits`
• `VkShaderFloatControlsIndependence`

**New Bitmasks**

• `VkDescriptorBindingFlags`
• `VkResolveModeFlags`
• VkSemaphoreWaitFlags

New Enum Constants

• VK_MAX_DRIVER_INFO_SIZE
• VK_MAX_DRIVER_NAME_SIZE
• Extending VkBufferCreateFlagBits:
  ◦ VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT
• Extending VkBufferUsageFlagBits:
  ◦ VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT
• Extending VkDescriptorPoolCreateFlagBits:
  ◦ VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT
• Extending VkDescriptorSetLayoutCreateFlagBits:
  ◦ VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT
• Extending VkFormatFeatureFlagBits:
  ◦ VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT
• Extending VkFramebufferCreateFlagBits:
  ◦ VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT
• Extending VkImageLayout:
  ◦ VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL
  ◦ VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL
  ◦ VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL
  ◦ VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL
• Extending VkMemoryAllocateFlagBits:
  ◦ VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT
  ◦ VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT
• Extending VkResult:
  ◦ VK_ERROR_FRAGMENTATION
  ◦ VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS
• Extending VkSamplerAddressMode:
  ◦ VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_2
  ◦ VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_STENCIL_LAYOUT
  ◦ VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_2
  ◦ VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_STENCIL_LAYOUT
VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO
VK_STRUCTURE_TYPE_BUFFER_OPAQUE_CAPTURE_ADDRESS_CREATE_INFO
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_BINDING_FLAGS_CREATE_INFO
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_ALLOCATE_INFO
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_LAYOUT_SUPPORT
VK_STRUCTURE_TYPE_DEVICE_MEMORY_OPAQUE_CAPTURE_ADDRESS_INFO
VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENTS_CREATE_INFO
VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENT_IMAGE_INFO
VK_STRUCTURE_TYPE_IMAGE_FORMAT_LIST_CREATE_INFO
VK_STRUCTURE_TYPE_IMAGE_STENCIL_USAGE_CREATE_INFO
VK_STRUCTURE_TYPE_MEMORY_OPAQUE_CAPTURE_ADDRESS_ALLOCATE_INFO
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_8BIT_STORAGE_FEATURES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEPTH_STENCIL_RESOLVE_PROPERTIES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_FEATURES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_PROPERTIES
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DRIVER_PROPERTIES
VK_STRUCTURE_TYPE_PHYSICALDEVICE_FLOAT_CONTROLS_PROPERTIES
VK_STRUCTURE_TYPE_PHYSICALDEVICE_HOST_QUERY_RESET_FEATURES
VK_STRUCTURE_TYPE_PHYSICALDEVICE_IMAGELESS_FRAMEBUFFER_FEATURES
VK_STRUCTURE_TYPE_PHYSICALDEVICE_SAMPLER_FILTER_MINMAX_PROPERTIES
VK_STRUCTURE_TYPE_PHYSICALDEVICE_SCALAR_BLOCK_LAYOUT_FEATURES
VK_STRUCTURE_TYPE_PHYSICALDEVICE_SEPARATE_DEPTH_STENCIL_LAYOUTS_FEATURES
VK_STRUCTURE_TYPE_PHYSICALDEVICE_SHADER_ATOMIC_INT64_FEATURES
VK_STRUCTURE_TYPE_PHYSICALDEVICE_SHADER_FLOAT16_INT8_FEATURES
VK_STRUCTURE_TYPE_PHYSICALDEVICE_SHADER_SUBGROUP_EXTENDEDTYPES_FEATURES
VK_STRUCTURE_TYPE_PHYSICALDEVICE_TIMELINE_SEMAPHORE_FEATURES
VK_STRUCTURE_TYPE_PHYSICALDEVICE_TIMELINE_SEMAPHORE_PROPERTIES
VK_STRUCTURE_TYPE_PHYSICALDEVICE_UNIFORM_BUFFER_STANDARD_LAYOUT_FEATURES
VK_STRUCTURE_TYPE_PHYSICALDEVICE_VULKAN_1_1_FEATURES
VK_STRUCTURE_TYPE_PHYSICALDEVICE_VULKAN_1_1_PROPERTIES
VK_STRUCTURE_TYPE_PHYSICALDEVICE_VULKAN_1_2_FEATURES
VK_STRUCTURE_TYPE_PHYSICALDEVICE_VULKAN_1_2_PROPERTIES
VK_STRUCTURE_TYPE_PHYSICALDEVICE_VULKAN_MEMORY_MODEL_FEATURES
VK_STRUCTURE_TYPE_RENDER_PASS_ATTACHMENT_BEGIN_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
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_samplerycbcr_conversion**

If the VK_KHR_samplerycbcr_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
• vkUpdateDescriptorSetWithTemplate

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, VkRenderingInfoKHR:
  - 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_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_UNORM_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`:**

---

1564
- `VK_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE`
- `VK_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION`

- **Extending `VkPipelineCreateFlagBits`:**
  - `VK_PIPELINE_CREATE_DISPATCH_BASE`
  - `VK_PIPELINE_CREATE_DISPATCH_BASE_BIT`
  - `VK_PIPELINE_CREATE_VIEW_INDEX_FROM_DEVICE_INDEX_BIT`

- **Extending `VkQueueFlagBits`:**
  - `VK_QUEUE_PROTECTED_BIT`

- **Extending `VkResult`:**
  - `VK_ERROR_INVALID_EXTERNAL_HANDLE`
  - `VK_ERROR_OUT_OF_POOL_MEMORY`

- **Extending `VkStructureType`:**
  - `VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORYDEVICE_GROUP_INFO`
  - `VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO`
  - `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORYDEVICE_GROUP_INFO`
  - `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_INFO`
  - `VK_STRUCTURE_TYPE_BIND_IMAGE_PLANE_MEMORY_INFO`
  - `VK_STRUCTURE_TYPE_BUFFER_MEMORY_REQUIREMENTS_INFO_2`
  - `VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_SUPPORT`
  - `VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_BIND_SPARSE_INFO`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_COMMAND_BUFFERBEGIN_INFO`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUPDEVICE_CREATE_INFO`
  - `VK_STRUCTURE_TYPE_DEVICEGROUP_RENDER_PASSBEGIN_INFO`
  - `VK_STRUCTURE_TYPE_DEVICEGROUP_SUBMIT_INFO`
  - `VK_STRUCTURE_TYPEDEVICE_QUEUE_INFO_2`
  - `VK_STRUCTURE_TYPEEXPORT_FENCE_CREATE_INFO`
  - `VK_STRUCTURE_TYPEEXPORT_MEMORY_ALLOCATE_INFO`
  - `VK_STRUCTURE_TYPEEXPORT_SEMAPHORE_CREATE_INFO`
  - `VK_STRUCTURE_TYPEEXTERNAL_BUFFER_PROPERTIES`
  - `VK_STRUCTURE_TYPEEXTERNAL_FENCE_PROPERTIES`
  - `VK_STRUCTURE_TYPEEXTERNAL_IMAGE_FORMAT_PROPERTIES`
  - `VK_STRUCTURE_TYPEEXTERNAL_MEMORY_BUFFER_CREATE_INFO`
  - `VK_STRUCTURE_TYPEEXTERNAL_MEMORYIMAGE_CREATE_INFO`
  - `VK_STRUCTURE_TYPEEXTERNAL_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

1566
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
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
- 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
- vkCmdSetViewport
- vkCmdSetViewports
- 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
• VkShaderModuleCreateInfo
• VkSparseBufferMemoryBindInfo
• VkSparseImageFormatProperties
• VkSparseImageMemoryBind
• VkSparseImageMemoryBindInfo
• VkSparseImageMemoryRequirements
• VkSparseImageOpaqueMemoryBindInfo
• VkSparseMemoryBind
• VkSpecializationInfo
• VkSpecializationMapEntry
• VkStencilOpState
• VkSubmitInfo
• VkSubpassDependency
• VkSubpassDescription
• VkSubresourceLayout
• VkVertexInputAttributeDescription
• VkVertexInputBindingDescription
• VkViewport
• VkWriteDescriptorSet

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
- VkInternalAllocationType
- VkLogicOp
- VkMemoryHeapFlagBits
- VkMemoryPropertyFlagBits
- VkObjectType
-VkPhysicalDeviceType
- VkPipelineBindPoint
- VkPipelineCacheHeaderVersion
- VkPipelineCreateFlagBits
- VkPipelineShaderStageCreateFlagBits
- VkPipelineStageFlagBits
- VkPolygonMode
- VkPrimitiveTopology
- VkQueryControlFlagBits
- VkQueryPipelineStatisticFlagBits
- VkQueryResultFlagBits
- VkQueryType
- VkQueueFlagBits
- VkRenderPassCreateFlagBits
- VkResult
• VkSampleCountFlagBits
• VkSamplerAddressMode
• VkSamplerCreateFlagBits
• VkSamplerMipmapMode
• VkShaderStageFlagBits
• VkSharingMode
• VkSparseImageFormatFlagBits
• VkSparseMemoryBindFlagBits
• VkStencilFaceFlagBits
• VkStencilOp
• VkStructureType
• VkSubpassContents
• VkSubpassDescriptionFlagBits
• VkSystemAllocationScope
• VkVendorId
• VkVertexInputRate

New Bitmasks

• VkAccessFlags
• VkAttachmentDescriptionFlags
• VkBufferCreateFlags
• VkBufferUsageFlags
• VkBufferViewCreateFlags
• VkColorComponentFlags
• VkCommandBufferResetFlags
• VkCommandBufferUsageFlags
• VkCommandPoolCreateFlags
• VkCommandPoolResetFlags
• VkCullModeFlags
• VkDependencyFlags
• VkDescriptorPoolCreateFlags
• VkDescriptorPoolResetFlags
• VkDescriptorSetLayoutCreateFlags
• VkDeviceCreateFlags
• VkDeviceQueueCreateFlags
- VkEventCreateFlags
- VkFenceCreateFlags
- VkFormatFeatureFlags
- VkFramebufferCreateFlags
- VkImageAspectFlags
- VkImageCreateFlags
- VkImageUsageFlags
- VkImageViewCreateFlags
- VkInstanceCreateFlags
- VkMemoryHeapFlags
- VkMemoryMapFlags
- VkMemoryPropertyFlags
- VkPipelineCacheCreateFlags
- VkPipelineColorBlendStateCreateFlags
- VkPipelineCreateFlags
- VkPipelineDepthStencilStateCreateFlags
- VkPipelineDynamicStateCreateFlags
- VkPipelineInputAssemblyStateCreateFlags
- VkPipelineLayoutCreateFlags
- VkPipelineMultisampleStateCreateFlags
- VkPipelineRasterizationStateCreateFlags
- VkPipelineShaderStageCreateFlags
- VkPipelineStageFlags
- VkPipelineTessellationStateCreateFlags
- VkPipelineViewportStateCreateFlags
- VkQueryControlFlags
- VkQueryPipelineStatisticFlags
- VkQueryPoolCreateFlags
- VkQueryResultFlags
- VkQueueFlags
- VkRenderPassCreateFlags
- VkSampleCountFlags
- VkSamplerCreateFlags
- VkSemaphoreCreateFlags
- VkSemaphoreCreateFlags
• VkShaderModuleCreateFlags
• VkShaderStageFlags
• VkSparseImageFormatFlags
• VkSparseMemoryBindFlags
• VkStencilFaceFlags
• VkSubpassDescriptionFlags

New Headers

• vk_platform

New Enum Constants

• VK_ATTACHMENT_UNUSED
• VK_FALSE
• VK_LOD_CLAMP_NONE
• VK_QUEUE_FAMILY_IGNORED
• VK_REMAINING_ARRAY_LAYERS
• VK_REMAINING_MIP_LEVELS
• VK_SUBPASS_EXTERNAL
• VK_TRUE
• 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://www.khronos.org/registry/vulkansc/

and allows generating versions of the Specification incorporating different extensions.

The Vulkan Documentation and Extensions document specifies the processes by which extensions and layers are created. Authors creating extensions and layers must follow the mandatory procedures in that document.

The remainder of this appendix documents a set of extensions chosen when this document was built. Versions of the Specification published in the Registry include:

- Core API + mandatory extensions required of all Vulkan implementations.
- Core API + all registered and published 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.

List of Current Extensions

- VK_KHR_copy_commands2
- VK_KHR_display
- VK_KHR_display_swapchain
- VK_KHR_external_fence_fd
- VK_KHR_external_memory_fd
- VK_KHR_external_semaphore_fd
- VK_KHR_fragment_shading_rate
- VK_KHR_get_display_properties2
- VK_KHR_get_surface_capabilities2
- VK_KHR_incremental_present
- VK_KHR_object_refresh
- VK_KHR_performance_query
- VK_KHR_shader_clock
- VK_KHR_shader_terminate_invocation
- VK_KHR_shared_presentable_image
• VK_KHR_surface
• VK_KHR_swapchain
• VK_KHR_swapchain_mutable_format
• VK_KHR_synchronization2
• VK_EXT_4444_formats
• VK_EXT_application_parameters
• VK_EXT_astc_decode_mode
• VK_EXT_blend_operation_advanced
• VK_EXT_calibrated_timestamps
• VK_EXT_color_write_enable
• VK_EXT_conservative_rasterization
• VK_EXT_custom_border_color
• VK_EXT_debug_utils
• VK_EXT_depth_clip_enable
• VK_EXT_depth_range_unrestricted
• VK_EXT_direct_mode_display
• VK_EXT_discard_rectangles
• VK_EXT_display_control
• VK_EXT_display_surface_counter
• VK_EXT_extended_dynamic_state
• VK_EXT_extended_dynamic_state2
• VK_EXT_external_memory_dma_buf
• VK_EXT_external_memory_host
• VK_EXT_filter_cubic
• VK_EXT_fragment_shader_interlock
• VK_EXT_global_priority
• VK_EXT_hdr_metadata
• VK_EXT_headless_surface
• VK_EXT_image_drm_format_modifier
• VK_EXT_image_robustness
• VK_EXT_index_type_uint8
• VK_EXT_line_rasterization
• VK_EXT_memory_budget
• VK_EXT_pci_bus_info
• VK_EXT_post_depth_coverage
• VK_EXT_queue_family_foreign
• VK_EXT_robustness2
• VK_EXT_sample_locations
• VK_EXT_shader_atomic_float
• VK_EXT_shader_demote_to_helper_invocation
• VK_EXT_shader_image_atomic_int64
• VK_EXT_shader_stencil_export
• VK_EXT_subgroup_size_control
• VK_EXT_swapchain_colorspace
• VK_EXT_texel_buffer_alignment
• VK_EXT_texture_compression_astc_hdr
• VK_EXT_validation_features
• VK_EXT_vertex_attribute_divisor
• VK_EXT_vertex_input_dynamic_state
• VK_EXT_yccr_2plane_444_formats
• VK_EXT_yccr_image_arrays
• VK_NV_external_memory_sci_buf
• VK_NV_external_sci_sync2
• VK_NV_private_vendor_info
VK_KHR_copy_commands2

Name String
VK_KHR_copy_commands2

Extension Type
Device extension

Registered Extension Number
338

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0

Contact
• Jeff Leger @jackohound

Other Extension Metadata

Last Modified Date
2020-07-06

Interactions and External Dependencies
• None

Contributors
• Jeff Leger, Qualcomm
• Tobias Hector, AMD
• Jan-Harald Fredriksen, ARM
• Tom Olson, ARM

Description

This extension provides extensible versions of the Vulkan buffer and image copy commands. The new commands are functionally identical to the core commands, except that their copy parameters are specified using extensible structures that can be used to pass extension-specific information.

The following extensible copy commands are introduced with this extension: vkCmdCopyBuffer2KHR, vkCmdCopyImage2KHR, vkCmdCopyBufferToImage2KHR, vkCmdCopyImageToBuffer2KHR, vkCmdBlitImage2KHR, and vkCmdResolveImage2KHR. Each command contains an *Info2KHR structure parameter that includes sType/pNext members. Lower level structures describing each region to be copied are also extended with sType/pNext members.
New Commands

- vkCmdBlitImage2KHR
- vkCmdCopyBuffer2KHR
- vkCmdCopyBufferToImage2KHR
- vkCmdCopyImage2KHR
- vkCmdCopyImageToBuffer2KHR
- vkCmdResolveImage2KHR

New Structures

- VkBlitImageInfo2KHR
- VkBufferCopy2KHR
- VkBufferImageCopy2KHR
- VkCopyBufferInfo2KHR
- VkCopyBufferToImageInfo2KHR
- VkCopyImageInfo2KHR
- VkCopyImageToBufferInfo2KHR
- VkImageBlit2KHR
- VkImageCopy2KHR
- VkImageResolve2KHR
- VkResolveImageInfo2KHR

New Enum Constants

- VK_KHR_COPY_COMMANDS_2_EXTENSION_NAME
- VK_KHR_COPY_COMMANDS_2_SPEC_VERSION

Extending VkStructureType:

- VK_STRUCTURE_TYPE_BLIT_IMAGE_INFO_2_KHR
- VK_STRUCTURE_TYPE_BUFFER_COPY_2_KHR
- VK_STRUCTURE_TYPE_BUFFER_IMAGE_COPY_2_KHR
- VK_STRUCTURE_TYPE_COPY_BUFFER_INFO_2_KHR
- VK_STRUCTURE_TYPE_COPY_BUFFER_TO_IMAGE_INFO_2_KHR
- VK_STRUCTURE_TYPE_COPY_IMAGE_INFO_2_KHR
- VK_STRUCTURE_TYPE_COPY_IMAGE_TO_BUFFER_INFO_2_KHR
- VK_STRUCTURE_TYPE_IMAGE_BLIT_2_KHR
- VK_STRUCTURE_TYPE_IMAGE_COPY_2_KHR
- VK_STRUCTURE_TYPE_IMAGE_RESOLVE_2_KHR
VK_STRUCTURE_TYPE_RESOLVE_IMAGE_INFO_2_KHR

Version History

• Revision 1, 2020-07-06 (Jeff Leger)
  ○ Internal revisions

VK_KHR_display

Name String

VK_KHR_display

Extension Type

Instance extension

Registered Extension Number

3

Revision

23

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_surface

Contact

• James Jones cubanismo
• Norbert Nopper FslNopper

Other Extension Metadata

Last Modified Date

2017-03-13

IP Status

No known IP claims.

Contributors

• James Jones, NVIDIA
• Norbert Nopper, Freescale
• Jeff Vigil, Qualcomm
• Daniel Rakos, AMD

Description

This extension provides the API to enumerate displays and available modes on a given device.
New Object Types

- VkDisplayKHR
- VkDisplayModeKHR

New Commands

- vkCreateDisplayModeKHR
- vkCreateDisplayPlaneSurfaceKHR
- vkGetDisplayModePropertiesKHR
- vkGetDisplayPlaneCapabilitiesKHR
- vkGetDisplayPlaneSupportedDisplaysKHR
- vkGetPhysicalDeviceDisplayPlanePropertiesKHR
- vkGetPhysicalDeviceDisplayPropertiesKHR

New Structures

- VkDisplayModeCreateInfoKHR
- VkDisplayModeParametersKHR
- VkDisplayModePropertiesKHR
- VkDisplayPlaneCapabilitiesKHR
- VkDisplayPlanePropertiesKHR
- VkDisplayPropertiesKHR
- VkDisplaySurfaceCreateInfoKHR

New Enums

- VkDisplayPlaneAlphaFlagBitsKHR

New Bitmasks

- VkDisplayModeCreateFlagsKHR
- VkDisplayPlaneAlphaFlagsKHR
- VkDisplaySurfaceCreateFlagsKHR
- VkSurfaceTransformFlagsKHR

New Enum Constants

- VK_KHR_DISPLAY_EXTENSION_NAME
- VK_KHR_DISPLAY_SPEC_VERSION

Extending VkObjectType:
Issues

1) Which properties of a mode should be fixed in the mode information vs. settable in some other function when setting the mode? E.g., do we need to double the size of the mode pool to include both stereo and non-stereo modes? YUV and RGB scanout even if they both take RGB input images? BGR vs. RGB input? etc.

**PROPOSED RESOLUTION:** Many modern displays support at most a handful of resolutions and timings natively. Other “modes” are expected to be supported using scaling hardware on the display engine or GPU. Other properties, such as rotation and mirroring should not require duplicating hardware modes just to express all combinations. Further, these properties may be implemented on a per-display or per-overlay granularity.

To avoid the exponential growth of modes as mutable properties are added, as was the case with EGLConfig/WGL pixel formats/GLXFBConfig, this specification should separate out hardware properties and configurable state into separate objects. Modes and overlay planes will express capabilities of the hardware, while a separate structure will allow applications to configure scaling, rotation, mirroring, color keys, LUT values, alpha masks, etc. for a given swapchain independent of the mode in use. Constraints on these settings will be established by properties of the immutable objects.

Note the resolution of this issue may affect issue 5 as well.

2) What properties of a display itself are useful?

**PROPOSED RESOLUTION:** This issue is too broad. It was meant to prompt general discussion, but resolving this issue amounts to completing this specification. All interesting properties should be included. The issue will remain as a placeholder since removing it would make it hard to parse existing discussion notes that refer to issues by number.

3) How are multiple overlay planes within a display or mode enumerated?

**PROPOSED RESOLUTION:** They are referred to by an index. Each display will report the number of overlay planes it contains.

4) Should swapchains be created relative to a mode or a display?

**PROPOSED RESOLUTION:** When using this extension, swapchains are created relative to a mode and a plane. The mode implies the display object the swapchain will present to. If the specified mode is not the display's current mode, the new mode will be applied when the first image is presented to the swapchain, and the default operating system mode, if any, will be restored when the swapchain is destroyed.
5) Should users query generic ranges from displays and construct their own modes explicitly using those constraints rather than querying a fixed set of modes (Most monitors only have one real “mode” these days, even though many support relatively arbitrary scaling, either on the monitor side or in the GPU display engine, making “modes” something of a relic/compatibility construct).

**PROPOSED RESOLUTION:** Expose both. Display information structures will expose a set of predefined modes, as well as any attributes necessary to construct a customized mode.

6) Is it fine if we return the display and display mode handles in the structure used to query their properties?

**PROPOSED RESOLUTION:** Yes.

7) Is there a possibility that not all displays of a device work with all of the present queues of a device? If yes, how do we determine which displays work with which present queues?

**PROPOSED RESOLUTION:** No known hardware has such limitations, but determining such limitations is supported automatically using the existing `VK_KHR_surface` and `VK_KHR_swapchain` query mechanisms.

8) Should all presentation need to be done relative to an overlay plane, or can a display mode + display be used alone to target an output?

**PROPOSED RESOLUTION:** Require specifying a plane explicitly.

9) Should displays have an associated window system display, such as an `HDC` or `Display*`?

**PROPOSED RESOLUTION:** No. Displays are independent of any windowing system in use on the system. Further, neither `HDC` nor `Display*` refer to a physical display object.

10) Are displays queried from a physical GPU or from a device instance?

**PROPOSED RESOLUTION:** Developers prefer to query modes directly from the physical GPU so they can use display information as an input to their device selection algorithms prior to device creation. This avoids the need to create placeholder device instances to enumerate displays.

This preference must be weighed against the extra initialization that must be done by driver vendors prior to device instance creation to support this usage.

11) Should displays and/or modes be dispatchable objects? If functions are to take displays, overlays, or modes as their first parameter, they must be dispatchable objects as defined in Khronos bug 13529. If they are not added to the list of dispatchable objects, functions operating on them must take some higher-level object as their first parameter. There is no performance case against making them dispatchable objects, but they would be the first extension objects to be dispatchable.

**PROPOSED RESOLUTION:** Do not make displays or modes dispatchable. They will dispatch based on their associated physical device.

12) Should hardware cursor capabilities be exposed?
PROPOSED RESOLUTION: Defer. This could be a separate extension on top of the base WSI specs.

if they are one physical display device to an end user, but may internally be implemented as two side-by-side displays using the same display engine (and sometimes cabling) resources as two physically separate display devices.

RESOLVED: Tiled displays will appear as a single display object in this API.

14) Should the raw EDID data be included in the display information?

RESOLVED: No. A future extension could be added which reports the EDID if necessary. This may be complicated by the outcome of issue 13.

15) Should min and max scaling factor capabilities of overlays be exposed?

RESOLVED: Yes. This is exposed indirectly by allowing applications to query the min/max position and extent of the source and destination regions from which image contents are fetched by the display engine when using a particular mode and overlay pair.

16) Should devices be able to expose planes that can be moved between displays? If so, how?

RESOLVED: Yes. Applications can determine which displays a given plane supports using vkGetDisplayPlaneSupportedDisplaysKHR.

17) Should there be a way to destroy display modes? If so, does it support destroying “built in” modes?

RESOLVED: Not in this extension. A future extension could add this functionality.

18) What should the lifetime of display and built-in display mode objects be?

RESOLVED: The lifetime of the instance. These objects cannot be destroyed. A future extension may be added to expose a way to destroy these objects and/or support display hotplug.

19) Should persistent mode for smart panels be enabled/disabled at swapchain creation time, or on a per-present basis.

RESOLVED: On a per-present basis.

Version History

- Revision 1, 2015-02-24 (James Jones)
  - Initial draft
- Revision 2, 2015-03-12 (Norbert Nopper)
  - Added overlay enumeration for a display.
- Revision 3, 2015-03-17 (Norbert Nopper)
  - Fixed typos and namings as discussed in Bugzilla.
  - Reordered and grouped functions.
• Added functions to query count of display, mode and overlay.
• Added native display handle, which may be needed on some platforms to create a native Window.

Revision 4, 2015-03-18 (Norbert Nopper)
• Removed primary and virtualPostion members (see comment of James Jones in Bugzilla).
• Added native overlay handle to information structure.
• Replaced , with ; in struct.

Revision 6, 2015-03-18 (Daniel Rakos)
• Added WSI extension suffix to all items.
• Made the whole API more “Vulkanish”.
• Replaced all functions with a single vkGetDisplayInfoKHR function to better match the rest of the API.
• Made the display, display mode, and overlay objects be first class objects, not subclasses of VkBaseObject as they do not support the common functions anyways.
• Renamed *Info structures to *Properties.
• Removed overlayIndex field from VkOverlayProperties as there is an implicit index already as a result of moving to a “Vulkanish” API.
• Displays are not get through device, but through physical GPU to match the rest of the Vulkan API. Also this is something ISVs explicitly requested.
• Added issue (6) and (7).

Revision 7, 2015-03-25 (James Jones)
• Added an issues section
• Added rotation and mirroring flags

Revision 8, 2015-03-25 (James Jones)
• Combined the duplicate issues sections introduced in last change.
• Added proposed resolutions to several issues.

Revision 9, 2015-04-01 (Daniel Rakos)
• Rebased extension against Vulkan 0.82.0

Revision 10, 2015-04-01 (James Jones)
• Added issues (10) and (11).
• Added more straw-man issue resolutions, and cleaned up the proposed resolution for issue (4).
• Updated the rotation and mirroring enums to have proper bitmask semantics.

Revision 11, 2015-04-15 (James Jones)
• Added proposed resolution for issues (1) and (2).
• Added issues (12), (13), (14), and (15)
- Removed pNativeHandle field from overlay structure.
- Fixed small compilation errors in example code.

Revision 12, 2015-07-29 (James Jones)
- Rewrote the guts of the extension against the latest WSI swapchain specifications and the latest Vulkan API.
- Address overlay planes by their index rather than an object handle and refer to them as “planes” rather than “overlays” to make it slightly clearer that even a display with no “overlays” still has at least one base “plane” that images can be displayed on.
- Updated most of the issues.
- Added an “extension type” section to the specification header.
- Re-used the VK_EXT_KHR_surface surface transform enumerations rather than redefining them here.
- Updated the example code to use the new semantics.

Revision 13, 2015-08-21 (Ian Elliott)
- Renamed this extension and all of its enumerations, types, functions, etc. This makes it compliant with the proposed standard for Vulkan extensions.
- Switched from “revision” to “version”, including use of the VK_MAKE_VERSION macro in the header file.

Revision 14, 2015-09-01 (James Jones)
- Restore single-field revision number.

Revision 15, 2015-09-08 (James Jones)
- Added alpha flags enum.
- Added premultiplied alpha support.

Revision 16, 2015-09-08 (James Jones)
- Added description section to the spec.
- Added issues 16 - 18.

Revision 17, 2015-10-02 (James Jones)
- Planes are now a property of the entire device rather than individual displays. This allows planes to be moved between multiple displays on devices that support it.
- Added a function to create a VkSurfaceKHR object describing a display plane and mode to align with the new per-platform surface creation conventions.
- Removed detailed mode timing data. It was agreed that the mode extents and refresh rate are sufficient for current use cases. Other information could be added back in as an extension if it is needed in the future.
- Added support for smart/persistent/buffered display devices.

Revision 18, 2015-10-26 (Ian Elliott)
- Renamed from VK_EXT_KHR_display to VK_KHR_display.
• Revision 19, 2015-11-02 (James Jones)
  ◦ Updated example code to match revision 17 changes.

• Revision 20, 2015-11-03 (Daniel Rakos)
  ◦ Added allocation callbacks to creation functions.

• Revision 21, 2015-11-10 (Jesse Hall)
  ◦ Added VK_DISPLAY_PLANE_ALPHA_OPAQUE_BIT_KHR, and use
    VkDisplayPlaneAlphaFlagBitsKHR for VkDisplayPlanePropertiesKHR::alphaMode instead of
    VkDisplayPlaneAlphaFlagsKHR, since it only represents one mode.
  ◦ Added reserved flags bitmask to VkDisplayPlanePropertiesKHR.
  ◦ Use VkSurfaceTransformFlagBitsKHR instead of obsolete VkSurfaceTransformKHR.
  ◦ Renamed vkGetDisplayPlaneSupportedDisplaysKHR parameters for clarity.

• Revision 22, 2015-12-18 (James Jones)
  ◦ Added missing “planeIndex” parameter to vkGetDisplayPlaneSupportedDisplaysKHR()

• Revision 23, 2017-03-13 (James Jones)
  ◦ Closed all remaining issues. The specification and implementations have been shipping with
    the proposed resolutions for some time now.
  ◦ Removed the sample code and noted it has been integrated into the official Vulkan SDK cube
    demo.

**VK_KHR_display_swapchain**

**Name String**

VK_KHR_display_swapchain

**Extension Type**

Device extension

**Registered Extension Number**

4

**Revision**

10

**Extension and Version Dependencies**

• Requires Vulkan 1.0

• Requires VK_KHR_swapchain

• Requires VK_KHR_display

**Contact**

• James Jones 📩 cubanismo
Other Extension Metadata

Last Modified Date
2017-03-13

IP Status
No known IP claims.

Contributors
- James Jones, NVIDIA
- Jeff Vigil, Qualcomm
- Jesse Hall, Google

Description
This extension provides an API to create a swapchain directly on a device’s display without any underlying window system.

New Commands
- vkCreateSharedSwapchainsKHR

New Structures
- Extending VkPresentInfoKHR:
  - VkDisplayPresentInfoKHR

New Enum Constants
- VK_KHR_DISPLAY_SWAPCHAIN_EXTENSION_NAME
- VK_KHR_DISPLAY_SWAPCHAIN_SPEC_VERSION
- Extending VkResult:
  - VK_ERROR_INCOMPATIBLE_DISPLAY_KHR
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_DISPLAY_PRESENT_INFO_KHR

Issues
1) Should swapchains sharing images each hold a reference to the images, or should it be up to the application to destroy the swapchains and images in an order that avoids the need for reference counting?

RESOLVED: Take a reference. The lifetime of presentable images is already complex enough.

2) Should the srcRect and dstRect parameters be specified as part of the presentation command, or at swapchain creation time?
RESOLVED: As part of the presentation command. This allows moving and scaling the image on the screen without the need to respecify the mode or create a new swapchain and presentable images.

3) Should srcRect and dstRect be specified as rects, or separate offset/extent values?

RESOLVED: As rects. Specifying them separately might make it easier for hardware to expose support for one but not the other, but in such cases applications must just take care to obey the reported capabilities and not use non-zero offsets or extents that require scaling, as appropriate.

4) How can applications create multiple swapchains that use the same images?

RESOLVED: By calling vkCreateSharedSwapchainsKHR.

An earlier resolution used vkCreateSwapchainKHR, chaining multiple VkSwapchainCreateInfoKHR structures through pNext. In order to allow each swapchain to also allow other extension structs, a level of indirection was used: VkSwapchainCreateInfoKHR::pNext pointed to a different structure, which had both sType and pNext members for additional extensions, and also had a pointer to the next VkSwapchainCreateInfoKHR structure. The number of swapchains to be created could only be found by walking this linked list of alternating structures, and the pSwapchains out parameter was reinterpreted to be an array of VkSwapchainKHR handles.

Another option considered was a method to specify a “shared” swapchain when creating a new swapchain, such that groups of swapchains using the same images could be built up one at a time. This was deemed unusable because drivers need to know all of the displays an image will be used on when determining which internal formats and layouts to use for that image.

Version History

- Revision 1, 2015-07-29 (James Jones)
  - Initial draft
- Revision 2, 2015-08-21 (Ian Elliott)
  - Renamed this extension and all of its enumerations, types, functions, etc. This makes it compliant with the proposed standard for Vulkan extensions.
  - Switched from “revision” to “version”, including use of the VK_MAKE_VERSION macro in the header file.
- Revision 3, 2015-09-01 (James Jones)
  - Restore single-field revision number.
- Revision 4, 2015-09-08 (James Jones)
  - Allow creating multiple swap chains that share the same images using a single call to vkCreateSwapchainKHR().
- Revision 5, 2015-09-10 (Alon Or-bach)
  - Removed underscores from SWAP_CHAIN in two enums.
- Revision 6, 2015-10-02 (James Jones)
  - Added support for smart panels/buffered displays.
• Revision 7, 2015-10-26 (Ian Elliott)
  ◦ Renamed from VK_EXT_KHR_display_swapchain to VK_KHR_display_swapchain.

• Revision 8, 2015-11-03 (Daniel Rakos)
  ◦ Updated sample code based on the changes to VK_KHR_swapchain.

• Revision 9, 2015-11-10 (Jesse Hall)
  ◦ Replaced VkDisplaySwapchainCreateInfoKHR with vkCreateSharedSwapchainsKHR, changing resolution of issue #4.

• Revision 10, 2017-03-13 (James Jones)
  ◦ Closed all remaining issues. The specification and implementations have been shipping with the proposed resolutions for some time now.
  ◦ Removed the sample code and noted it has been integrated into the official Vulkan SDK cube demo.

**VK_KHR_external_fence_fd**

*Name String*

VK_KHR_external_fence_fd

*Extension Type*

Device extension

*Registered Extension Number*

116

*Revision*

1

*Extension and Version Dependencies*

• Requires Vulkan 1.0
  • Requires VK_KHR_external_fence

*Contact*

• Jesse Hall [critsec](https://twitter.com/critsec)

*Other Extension Metadata*

*Last Modified Date*

2017-05-08

*IP Status*

No known IP claims.

*Contributors*

• Jesse Hall, Google
Description

An application using external memory may wish to synchronize access to that memory using fences. This extension enables an application to export fence payload to and import fence payload from POSIX file descriptors.

New Commands

- `vkGetFenceFdKHR`
- `vkImportFenceFdKHR`

New Structures

- `VkFenceGetFdInfoKHR`
- `VkImportFenceFdInfoKHR`

New Enum Constants

- `VK_KHR_EXTERNAL_FENCE_FD_EXTENSION_NAME`
- `VK_KHR_EXTERNAL_FENCE_FD_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_FENCE_GET_FD_INFO_KHR`
  - `VK_STRUCTURE_TYPE_IMPORT_FENCE_FD_INFO_KHR`

Issues

This extension borrows concepts, semantics, and language from `VK_KHR_external_semaphore_fd`. That extension's issues apply equally to this extension.

Version History

- Revision 1, 2017-05-08 (Jesse Hall)
  - Initial revision

`VK_KHR_external_memory_fd`

Name String

`VK_KHR_external_memory_fd`
Extension Type
Device extension

Registered Extension Number
75

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_external_memory

Contact
• James Jones cubanismo

Other Extension Metadata

Last Modified Date
2016-10-21

IP Status
No known IP claims.

Contributors
• James Jones, NVIDIA
• Jeff Juliano, NVIDIA

Description
An application may wish to reference device memory in multiple Vulkan logical devices or instances, in multiple processes, and/or in multiple APIs. This extension enables an application to export POSIX file descriptor handles from Vulkan memory objects and to import Vulkan memory objects from POSIX file descriptor handles exported from other Vulkan memory objects or from similar resources in other APIs.

New Commands
• vkGetMemoryFdKHR
• vkGetMemoryFdPropertiesKHR

New Structures
• VkMemoryFdPropertiesKHR
• VkMemoryGetFdInfoKHR
• Extending VkMemoryAllocateInfo:
New Enum Constants

- VK_KHR_EXTERNAL_MEMORY_FD_EXTENSION_NAME
- VK_KHR_EXTERNAL_MEMORY_FD_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_IMPORT_MEMORY_FD_INFO_KHR
  - VK_STRUCTURE_TYPE_MEMORY_FD_PROPERTIES_KHR
  - VK_STRUCTURE_TYPE_MEMORY_GET_FD_INFO_KHR

Issues

1) Does the application need to close the file descriptor returned by vkGetMemoryFdKHR?

**RESOLVED:** Yes, unless it is passed back in to a driver instance to import the memory. A successful get call transfers ownership of the file descriptor to the application, and a successful import transfers it back to the driver. Destroying the original memory object will not close the file descriptor or remove its reference to the underlying memory resource associated with it.

2) Do drivers ever need to expose multiple file descriptors per memory object?

**RESOLVED:** No. This would indicate there are actually multiple memory objects, rather than a single memory object.

3) How should the valid size and memory type for POSIX file descriptor memory handles created outside of Vulkan be specified?

**RESOLVED:** The valid memory types are queried directly from the external handle. The size will be specified by future extensions that introduce such external memory handle types.

Version History

- Revision 1, 2016-10-21 (James Jones)
  - Initial revision

**VK_KHR_external_semaphore_fd**

**Name String**

VK_KHR_external_semaphore_fd

**Extension Type**

Device extension

**Registered Extension Number**

80
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_external_semaphore`

Contact

- James Jones `@cubanismo`

Other Extension Metadata

Last Modified Date

2016-10-21

IP Status

No known IP claims.

Contributors

- Jesse Hall, Google
- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Carsten Rohde, NVIDIA

Description

An application using external memory may wish to synchronize access to that memory using semaphores. This extension enables an application to export semaphore payload to and import semaphore payload from POSIX file descriptors.

New Commands

- `vkGetSemaphoreFdKHR`
- `vkImportSemaphoreFdKHR`

New Structures

- `VkImportSemaphoreFdInfoKHR`
- `VkSemaphoreGetFdInfoKHR`

New Enum Constants

- `VK_KHR_EXTERNAL_SEMAPHORE_FD_EXTENSION_NAME`
- `VK_KHR_EXTERNAL_SEMAPHORE_FD_SPEC_VERSION`
- Extending `VkStructureType:`
Issues

1) Does the application need to close the file descriptor returned by `vkGetSemaphoreFdKHR`?

**RESOLVED:** Yes, unless it is passed back in to a driver instance to import the semaphore. A successful get call transfers ownership of the file descriptor to the application, and a successful import transfers it back to the driver. Destroying the original semaphore object will not close the file descriptor or remove its reference to the underlying semaphore resource associated with it.

Version History

- Revision 1, 2016-10-21 (Jesse Hall)
  - Initial revision

**VK_KHR_fragment_shading_rate**

Name String

`VK_KHR_fragment_shading_rate`

Extension Type

Device extension

Registered Extension Number

227

Revision

2

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_create_renderpass2`
- Requires `VK_KHR_get_physical_device_properties2`

Contact

- Tobias Hector @tobski

Other Extension Metadata

Last Modified Date

2021-09-30

Interactions and External Dependencies

- This extension requires `SPV_KHR_fragment_shading_rate`.
- This extension provides API support for `GL_EXT_fragment_shading_rate`
Contributors

• Tobias Hector, AMD
• Guennadi Riguer, AMD
• Matthaeus Chajdas, AMD
• Pat Brown, Nvidia
• Matthew Netsch, Qualcomm
• Slawomir Grajewski, Intel
• Jan-Harald Fredriksen, Arm
• Jeff Bolz, Nvidia
• Contributors to the VK_NV_shading_rate_image specification
• Contributors to the VK_EXT_fragment_density_map specification

Description

This extension adds the ability to change the rate at which fragments are shaded. Rather than the usual single fragment invocation for each pixel covered by a primitive, multiple pixels can be shaded by a single fragment shader invocation.

Up to three methods are available to the application to change the fragment shading rate:

• **Pipeline Fragment Shading Rate**, which allows the specification of a rate per-draw.
• **Primitive Fragment Shading Rate**, which allows the specification of a rate per primitive, specified during shading.
• **Attachment Fragment Shading Rate**, which allows the specification of a rate per-region of the framebuffer, specified in a specialized image attachment.

Additionally, these rates can all be specified and combined in order to adjust the overall detail in the image at each point.

This functionality can be used to focus shading efforts where higher levels of detail are needed in some parts of a scene compared to others. This can be particularly useful in high resolution rendering, or for XR contexts.

This extension also adds support for the `SPV_KHR_fragment_shading_rate` extension which enables setting the `primitive fragment shading rate`, and allows querying the final shading rate from a fragment shader.

New Commands

• `vkCmdSetFragmentShadingRateKHR`
• `vkGetPhysicalDeviceFragmentShadingRatesKHR`
New Structures

- **VkPhysicalDeviceFragmentShadingRateKHR**
- Extending **VkGraphicsPipelineCreateInfo**:
  - **VkPipelineFragmentShadingRateStateCreateInfoKHR**
- Extending **VkPhysicalDeviceFeatures2**, **VkDeviceCreateInfo**:
  - **VkPhysicalDeviceFragmentShadingRateFeaturesKHR**
- Extending **VkPhysicalDeviceProperties2**:
  - **VkPhysicalDeviceFragmentShadingRatePropertiesKHR**
- Extending **VkSubpassDescription2**:
  - **VkFragmentShadingRateAttachmentInfoKHR**

New Enums

- **VkFragmentShadingRateCombinerOpKHR**

New Enum Constants

- **VK_KHR_FRAGMENT_SHADING_RATE_EXTENSION_NAME**
- **VK_KHR_FRAGMENT_SHADING_RATE_SPEC_VERSION**
- Extending **VkAccessFlagBits**:
  - **VK_ACCESS_FRAGMENT_SHADING_RATE_ATTACHMENT_READ_BIT_KHR**
- Extending **VkDynamicState**:
  - **VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR**
- Extending **VkFormatFeatureFlagBits**:
  - **VK_FORMAT_FEATURE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR**
- Extending **VkImageLayout**:
  - **VK_IMAGE_LAYOUT_FRAGMENT_SHADING_RATE_ATTACHMENT_OPTIMAL_KHR**
- Extending **VkImageUsageFlagBits**:
  - **VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR**
- Extending **VkPipelineStageFlagBits**:
  - **VK_PIPELINE_STAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR**
- Extending **VkStructureType**:
  - **VK_STRUCTURE_TYPE_FRAGMENT_SHADING_RATE_ATTACHMENT_INFO_KHR**
  - **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADING_RATE_FEATURES_KHR**
  - **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADING_RATE_KHR**
  - **VK_STRUCTURE_TYPE_PHYSICAL DEVICE_FRAGMENT_SHADING_RATE_PROPERTIES_KHR**
  - **VK_STRUCTURE_TYPE_PIPELINE_FRAGMENT_SHADING_RATE_STATE_CREATE_INFO_KHR**
Version History

• Revision 1, 2020-05-06 (Tobias Hector)
  ◦ Initial revision
• Revision 2, 2021-09-30 (Jon Leech)
  ◦ Add interaction with VK_KHR_format_feature_flags2 to vk.xml

VK_KHR_get_display_properties2

Name String
VK_KHR_get_display_properties2

Extension Type
Instance extension

Registered Extension Number
122

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_display

Contact
• James Jones cubanismo

Other Extension Metadata

Last Modified Date
2017-02-21

IP Status
No known IP claims.

Contributors
• Ian Elliott, Google
• James Jones, NVIDIA

Description

This extension provides new entry points to query device display properties and capabilities in a way that can be easily extended by other extensions, without introducing any further entry points. This extension can be considered the VK_KHR_display equivalent of the VK_KHR_get_physical_device_properties2 extension.
New Commands

- vkGetDisplayModeProperties2KHR
- vkGetDisplayPlaneCapabilities2KHR
- vkGetPhysicalDeviceDisplayPlaneProperties2KHR
- vkGetPhysicalDeviceDisplayProperties2KHR

New Structures

- VkDisplayModeProperties2KHR
- VkDisplayPlaneCapabilities2KHR
- VkDisplayPlaneInfo2KHR
- VkDisplayPlaneProperties2KHR
- VkDisplayProperties2KHR

New Enum Constants

- VK_KHR_GET_DISPLAY_PROPERTIES_2_EXTENSION_NAME
- VK_KHR_GET_DISPLAY_PROPERTIES_2_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_DISPLAY_MODE_PROPERTIES_2_KHR
  - VK_STRUCTURE_TYPE_DISPLAY_PLANE_CAPABILITIES_2_KHR
  - VK_STRUCTURE_TYPE_DISPLAY_PLANE_INFO_2_KHR
  - VK_STRUCTURE_TYPE_DISPLAY_PLANE_PROPERTIES_2_KHR
  - VK_STRUCTURE_TYPE_DISPLAY_PROPERTIES_2_KHR

Issues

1) What should this extension be named?

RESOLVED: VK_KHR_get_display_properties2. Other alternatives:

- VK_KHR_display2
- One extension, combined with VK_KHR_surface_capabilities2.

2) Should extensible input structs be added for these new functions?

RESOLVED:

- vkGetPhysicalDeviceDisplayProperties2KHR: No. The only current input is a VkPhysicalDevice. Other inputs would not make sense.
- vkGetPhysicalDeviceDisplayPlaneProperties2KHR: No. The only current input is a VkPhysicalDevice. Other inputs would not make sense.
vkGetDisplayModeProperties2KHR: No. The only current inputs are a VkPhysicalDevice and a VkDisplayModeKHR. Other inputs would not make sense.

3) Should additional display query functions be extended?

RESOLVED:

vkGetDisplayPlaneSupportedDisplaysKHR: No. Extensions should instead extend vkGetDisplayPlaneCapabilitiesKHR().

Version History

• Revision 1, 2017-02-21 (James Jones)
  ◦ Initial draft.

VK_KHR_get_surface_capabilities2

Name String
VK_KHR_get_surface_capabilities2

Extension Type
Instance extension

Registered Extension Number
120

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_surface

Contact
• James Jones cubanismo

Other Extension Metadata

Last Modified Date
2017-02-27

IP Status
No known IP claims.

Contributors
• Ian Elliott, Google
• James Jones, NVIDIA
Description

This extension provides new entry points to query device surface capabilities in a way that can be easily extended by other extensions, without introducing any further entry points. This extension can be considered the VK_KHR_surface equivalent of the VK_KHR_get_physical_device_properties2 extension.

New Commands

- vkGetPhysicalDeviceSurfaceCapabilities2KHR
- vkGetPhysicalDeviceSurfaceFormats2KHR

New Structures

- VkPhysicalDeviceSurfaceInfo2KHR
- VkSurfaceCapabilities2KHR
- VkSurfaceFormat2KHR

New Enum Constants

- VK_KHR_GET_SURFACE_CAPABILITIES_2_EXTENSION_NAME
- VK_KHR_GET_SURFACE_CAPABILITIES_2_SPEC_VERSION

Extending VkStructureType:

- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SURFACE_INFO_2_KHR
- VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_KHR
- VK_STRUCTURE_TYPE_SURFACE_FORMAT_2_KHR

Issues

1) What should this extension be named?

RESOLVED: VK_KHR_get_surface_capabilities2. Other alternatives:

- VK_KHR_surface2

- One extension, combining a separate display-specific query extension.

2) Should additional WSI query functions be extended?

RESOLVED:

- vkGetPhysicalDeviceSurfaceCapabilitiesKHR: Yes. The need for this motivated the extension.
- vkGetPhysicalDeviceSurfaceSupportKHR: No. Currently only has boolean output. Extensions should instead extend vkGetPhysicalDeviceSurfaceCapabilities2KHR.
• `vkGetPhysicalDeviceSurfaceFormatsKHR`: Yes.

• `vkGetPhysicalDeviceSurfacePresentModesKHR`: No. Recent discussion concluded this introduced too much variability for applications to deal with. Extensions should instead extend `vkGetPhysicalDeviceSurfaceCapabilities2KHR`.

• `vkGetPhysicalDeviceXlibPresentationSupportKHR`: Not in this extension.

• `vkGetPhysicalDeviceXcbPresentationSupportKHR`: Not in this extension.

• `vkGetPhysicalDeviceWaylandPresentationSupportKHR`: Not in this extension.

• `vkGetPhysicalDeviceWin32PresentationSupportKHR`: Not in this extension.

**Version History**

• Revision 1, 2017-02-27 (James Jones)
  ◦ Initial draft.

**VK_KHR_incremental_present**

**Name String**

`VK_KHR_incremental_present`

**Extension Type**

Device extension

**Registered Extension Number**

85

**Revision**

2

**Extension and Version Dependencies**

• Requires Vulkan 1.0
  
• Requires `VK_KHR_swapchain`

**Contact**

• Ian Elliott @ianelliottus

**Other Extension Metadata**

**Last Modified Date**

2016-11-02

**IP Status**

No known IP claims.

**Contributors**

• Ian Elliott, Google
Description

This device extension extends \texttt{vkQueuePresentKHR} from the \texttt{VK_KHR_swapchain} extension, allowing an application to specify a list of rectangular, modified regions of each image to present. This should be used in situations where an application is only changing a small portion of the presentable images within a swapchain, since it enables the presentation engine to avoid wasting time presenting parts of the surface that have not changed.

This extension is leveraged from the \texttt{EGL_KHR_swapbuffers_with_damage} extension.

New Structures

- \texttt{VkPresentRegionKHR}
- \texttt{VkRectLayerKHR}
- Extending \texttt{VkPresentInfoKHR}:
  - \texttt{VkPresentRegionsKHR}

New Enum Constants

- \texttt{VK_KHR_INCREMENTAL_PRESENT_EXTENSION_NAME}
- \texttt{VK_KHR_INCREMENTAL_PRESENT_SPEC_VERSION}
- Extending \texttt{VkStructureType}:
  - \texttt{VK_STRUCTURE_TYPE_PRESENT_REGIONS_KHR}

Issues

1) How should we handle steroescopic-3D swapchains? We need to add a layer for each rectangle. One approach is to create another struct containing the \texttt{VkRect2D} plus layer, and have \texttt{VkPresentRegionsKHR} point to an array of that struct. Another approach is to have two parallel arrays, \texttt{pRectangles} and \texttt{pLayers}, where \texttt{pRectangles[i]} and \texttt{pLayers[i]} must be used together. Which approach should we use, and if the array of a new structure, what should that be called?

\textbf{RESOLVED}: Create a new structure, which is a \texttt{VkRect2D} plus a layer, and will be called \texttt{VkRectLayerKHR}.
2) Where is the origin of the `VkRectLayerKHR`?

**RESOLVED**: The upper left corner of the presentable image(s) of the swapchain, per the definition of framebuffer coordinates.

3) Does the rectangular region, `VkRectLayerKHR`, specify pixels of the swapchain's image(s), or of the surface?

**RESOLVED**: Of the image(s). Some presentation engines may scale the pixels of a swapchain's image(s) to the size of the surface. The size of the swapchain's image(s) will be consistent, where the size of the surface may vary over time.

4) What if all of the rectangles for a given swapchain contain a width and/or height of zero?

**RESOLVED**: The application is indicating that no pixels changed since the last present. The presentation engine may use such a hint and not update any pixels for the swapchain. However, all other semantics of `vkQueuePresentKHR` must still be honored, including waiting for semaphores to signal.

5) When the swapchain is created with `VkSwapchainCreateInfoKHR::preTransform` set to a value other than `VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR`, should the rectangular region, `VkRectLayerKHR`, be transformed to align with the `preTransform`?

**RESOLVED**: No. The rectangular region in `VkRectLayerKHR` should not be transformed. As such, it may not align with the extents of the swapchain's image(s). It is the responsibility of the presentation engine to transform the rectangular region. This matches the behavior of the Android presentation engine, which set the precedent.

**Version History**

- Revision 1, 2016-11-02 (Ian Elliott)
  - Internal revisions
- Revision 2, 2021-03-18 (Ian Elliott)
  - Clarified alignment of rectangles for presentation engines that support transformed swapchains.

**VK_KHR_object_refresh**

**Name String**

`VK_KHR_object_refresh`

**Extension Type**

Device extension

**Registered Extension Number**

309
Extension and Version Dependencies

- Requires Vulkan 1.0

Contact

- Aidan Fabius afabius

Other Extension Metadata

Last Modified Date

2020-01-14

IP Status

No known IP claims.

Contributors

- Aidan Fabius, Core Avionics
- Mark Bellamy, ARM

Description

Many safety critical environments are required to contend with single event upsets (SEUs). These occur when a bit in a physical device’s memory or register is inadvertently flipped. It is typical for host memory to include automatic error detection (EDC) or correction (ECC) on platforms where this a concern. However, device-accessible memory may not have these protections. In that case, the data must be periodically refreshed.

Unextended Vulkan provides a variety of methods to mitigate SEUs. Image and buffer objects can be bound to SEU-safe memory, and many object types can be refreshed explicitly by the application by reloading or regenerating the object’s data. However, implementations may store internal object-specific data in non-SEU-safe memory, and unextended Vulkan provides no clear method to determine which object types this applies to or how to refresh that data.

This extension adds a mechanism to query which object types store implementation-internal data in device regions susceptible to SEUs, and to explicitly refresh that implementation-internal data.

New Commands

- vkCmdRefreshObjectsKHR
- vkGetPhysicalDeviceRefreshableObjectTypesKHR

New Structures

- VkRefreshObjectKHR
- VkRefreshObjectListKHR
New Enums
• VkRefreshObjectFlagBitsKHR

New Bitmasks
• VkRefreshObjectFlagsKHR

New Enum Constants
• VK_KHR_OBJECT_REFRESH_EXTENSION_NAME
• VK_KHR_OBJECT_REFRESH_SPEC_VERSION
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_REFRESH_OBJECT_LIST_KHR

Issues
1) Should this extension refresh object data, or validate whether or not the data has been corrupted?

**RESOLVED** This extension should refresh data, not validate it. This reduces application error-handling complexity, and invalid data would have to be refreshed anyway.

2) Should object refreshes be done using the host or with command buffers?

**RESOLVED** Object refreshes should be done with command buffers. This reduces the synchronization complexity.

3) Refresh operations will need a pipeline barrier so that subsequent commands will see the results of the refresh. What access flags and pipeline stage should apply to refresh operations? Should they use new flags and stages, or re-use an existing one?

**RESOLVED** Object refreshes are considered to be a transfer operation for the purposes of pipeline barriers.

4) Should this extension add a feature bit?

**RESOLVED** A feature bit is not necessary. In the case of this extension being promoted to core, implementations that do not support or require refreshing of any object types will return 0 for the count parameter of vkGetPhysicalDeviceRefreshableObjectTypesKHR.

Examples
None.

Version History
• Revision 1, 2020-01-14
**VK_KHR_performance_query**

**Name String**

VK_KHR_performance_query

**Extension Type**

Device extension

**Registered Extension Number**

117

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.1

**Special Use**

- Developer tools

**Contact**

- Alon Or-bach @alonorbach

**Other Extension Metadata**

**Last Modified Date**

2019-10-08

**IP Status**

No known IP claims.

**Contributors**

- Jesse Barker, Unity Technologies
- Kenneth Benzie, Codeplay
- Jan-Harald Fredriksen, ARM
- Jeff Leger, Qualcomm
- Jesse Hall, Google
- Tobias Hector, AMD
- Neil Henning, Codeplay
- Baldur Karlsson
- Lionel Landwerlin, Intel
- Peter Lohrmann, AMD
- Alon Or-bach, Samsung
- Daniel Rakos, AMD
Description

The **VK_KHR_performance_query** extension adds a mechanism to allow querying of performance counters for use in applications and by profiling tools.

Each queue family **may** expose counters that **can** be enabled on a queue of that family. We extend **VkQueryType** to add a new query type for performance queries, and chain a structure on **VkQueryPoolCreateInfo** to specify the performance queries to enable.

New Commands

- `vkAcquireProfilingLockKHR`
- `vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR`
- `vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR`
- `vkReleaseProfilingLockKHR`

New Structures

- `VkAcquireProfilingLockInfoKHR`
- `VkPerformanceCounterDescriptionKHR`
- `VkPerformanceCounterKHR`
- **Extending** `VkPhysicalDeviceFeatures2, VkDeviceCreateInfo`:
  - `VkPhysicalDevicePerformanceQueryFeaturesKHR`
- **Extending** `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDevicePerformanceQueryPropertiesKHR`
- **Extending** `VkQueryPoolCreateInfo`:
  - `VkQueryPoolPerformanceCreateInfoKHR`
- **Extending** `VkSubmitInfo, VkSubmitInfo2KHR`:
  - `VkPerformanceQuerySubmitInfoKHR`

If Vulkan SC 1.0 is supported:

- **Extending** `VkDeviceObjectReservationCreateInfo`:
  - `VkPerformanceQueryReservationInfoKHR`

New Unions

- `VkPerformanceCounterResultKHR`
New Enums

- VkAcquireProfilingLockFlagBitsKHR
- VkPerformanceCounterDescriptionFlagBitsKHR
- VkPerformanceCounterScopeKHR
- VkPerformanceCounterStorageKHR
- VkPerformanceCounterUnitKHR

New Bitmasks

- VkAcquireProfilingLockFlagsKHR
- VkPerformanceCounterDescriptionFlagsKHR

New Enum Constants

- VK_KHR_PERFORMANCE_QUERY_EXTENSION_NAME
- VK_KHR_PERFORMANCE_QUERY_SPEC_VERSION

Extending VkQueryType:

- VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR

Extending VkStructureType:

- VK_STRUCTURE_TYPE_ACQUIRE_PROFILING_LOCK_INFO_KHR
- VK_STRUCTURE_TYPE_PERFORMANCE_COUNTER_DESCRIPTION_KHR
- VK_STRUCTURE_TYPE_PERFORMANCE_COUNTER_KHR
- VK_STRUCTURE_TYPE_PERFORMANCE_QUERY_SUBMIT_INFO_KHR
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PERFORMANCE_QUERY_FEATURES_KHR
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PERFORMANCE_QUERY_PROPERTIES_KHR
- VK_STRUCTURE_TYPE_QUERY_POOL_PERFORMANCE_CREATE_INFO_KHR

If Vulkan SC 1.0 is supported:

Extending VkStructureType:

- VK_STRUCTURE_TYPE_PERFORMANCE_QUERY_RESERVATION_INFO_KHR

Issues

1) Should this extension include a mechanism to begin a query in command buffer A and end the query in command buffer B?

**RESOLVED** No - queries are tied to command buffer creation and thus have to be encapsulated within a single command buffer.

2) Should this extension include a mechanism to begin and end queries globally on the queue, not using the existing command buffer commands?
RESOLVED No - for the same reasoning as the resolution of 1).

3) Should this extension expose counters that require multiple passes?

RESOLVED Yes - users should re-submit a command buffer with the same commands in it multiple times, specifying the pass to count as the query parameter in VkPerformanceQuerySubmitInfoKHR.

4) How to handle counters across parallel workloads?

RESOLVED In the spirit of Vulkan, a counter description flag VK_PERFORMANCE_COUNTER_DESCRIPTION_CONCURRENTLY_IMPACTED_BIT_KHR denotes that the accuracy of a counter result is affected by parallel workloads.

5) How to handle secondary command buffers?

RESOLVED Secondary command buffers inherit any counter pass index specified in the parent primary command buffer. Note: this is no longer an issue after change from issue 10 resolution.

6) What commands does the profiling lock have to be held for?

RESOLVED For any command buffer that is being queried with a performance query pool, the profiling lock must be held while that command buffer is in the recording, executable, or pending state.

7) Should we support vkCmdCopyQueryPoolResults?

RESOLVED Yes.

8) Should we allow performance queries to interact with multiview?

RESOLVED Yes, but the performance queries must be performed once for each pass per view.

9) Should a queryCount > 1 be usable for performance queries?

RESOLVED Yes. Some vendors will have costly performance counter query pool creation, and would rather if a certain set of counters were to be used multiple times that a queryCount > 1 can be used to amortize the instantiation cost.

10) Should we introduce an indirect mechanism to set the counter pass index?

RESOLVED Specify the counter pass index at submit time instead, to avoid requiring re-recording of command buffers when multiple counter passes are needed.

Examples
The following example shows how to find what performance counters a queue family supports, setup a query pool to record these performance counters, how to add the query pool to the command buffer to record information, and how to get the results from the query pool.

```
// A previously created physical device
VkPhysicalDevice physicalDevice;
```
// One of the queue families our device supports
uint32_t queueFamilyIndex;

uint32_t counterCount;

// Get the count of counters supported
vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR(
    physicalDevice,
    queueFamilyIndex,
    &counterCount,
    NULL,
    NULL);

VkPerformanceCounterKHR* counters =
    malloc(sizeof(VkPerformanceCounterKHR) * counterCount);
VkPerformanceCounterDescriptionKHR* counterDescriptions =
    malloc(sizeof(VkPerformanceCounterDescriptionKHR) * counterCount);

// Get the counters supported
vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR(
    physicalDevice,
    queueFamilyIndex,
    &counterCount,
    counters,
    counterDescriptions);

// Try to enable the first 8 counters
uint32_t enabledCounters[8];

const uint32_t enabledCounterCount = min(counterCount, 8));

for (uint32_t i = 0; i < enabledCounterCount; i++) {
    enabledCounters[i] = i;
}

// A previously created device that had the performanceCounterQueryPools feature
// set to VK_TRUE
VkDevice device;

VkQueryPoolPerformanceCreateInfoKHR performanceQueryCreateInfo = {
    VK_STRUCTURE_TYPE_QUERY_POOL_PERFORMANCE_CREATE_INFO_KHR,
    NULL,

    // Specify the queue family that this performance query is performed on
    queueFamilyIndex,

    // The number of counters to enable
    enabledCounterCount,

    // The array of indices of counters to enable
// Get the number of passes our counters will require.
uint32_t numPasses;

vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR(
    physicalDevice,
   .performanceQueryCreateInfo,
   .numPasses);

VkQueryPoolCreateInfo queryPoolCreateInfo = {
    VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO,
   .performanceQueryCreateInfo,
    0,
    // Using our new query type here
    VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR,
    1,
    0
};

VkQueryPool queryPool;

VkResult result = vkCreateQueryPool(
    device,
    queryPoolCreateInfo,
    NULL,
    &queryPool);

assert(VK_SUCCESS == result);

// A queue from queueFamilyIndex
VkQueue queue;

// A command buffer we want to record counters on
VkCommandBuffer commandBuffer;

VkCommandBufferBeginInfo commandBufferBeginInfo = {
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,
    NULL,
    0,
    NULL
};

VkAcquireProfilingLockInfoKHR lockInfo = {
    VK_STRUCTURE_TYPE_ACQUIRE_PROFILING_LOCK_INFO_KHR,
    NULL,
0,
UINT64_MAX // Wait forever for the lock
};

// Acquire the profiling lock before we record command buffers
// that will use performance queries
result = vkAcquireProfilingLockKHR(device, &lockInfo);
assert(VK_SUCCESS == result);
result = vkBeginCommandBuffer(commandBuffer, &commandBufferBeginInfo);
assert(VK_SUCCESS == result);

vkCmdResetQueryPool(commandBuffer, queryPool, 0, 1);

vkCmdBeginQuery(commandBuffer, queryPool, 0, 0);

// Perform the commands you want to get performance information on
// ...

// Perform a barrier to ensure all previous commands were complete before
// ending the query
vkCmdPipelineBarrier(commandBuffer,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
    0,
    0,
    NULL,
    0,
    NULL,
    0,
    NULL);

vkCmdEndQuery(commandBuffer, queryPool, 0);
result = vkEndCommandBuffer(commandBuffer);
assert(VK_SUCCESS == result);
for (uint32_t counterPass = 0; counterPass < numPasses; counterPass++) {

VkPerformanceQuerySubmitInfoKHR performanceQuerySubmitInfo = {
    VK_STRUCTURE_TYPE_PERFORMANCE_QUERY_SUBMIT_INFO_KHR,
    NULL,
    counterPass
};

// Submit the command buffer and wait for its completion
// ...
}

// Release the profiling lock after the command buffer is no longer in the
// pending state.
vkReleaseProfilingLockKHR(device);

result = vkResetCommandBuffer(commandBuffer, 0);
assert(VK_SUCCESS == result);

// Create an array to hold the results of all counters
VkPerformanceCounterResultKHR* recordedCounters = malloc(
    sizeof(VkPerformanceCounterResultKHR) * enabledCounterCount);

result = vkGetQueryPoolResults(
    device,
    queryPool,
    0,
    1,
    sizeof(VkPerformanceCounterResultKHR) * enabledCounterCount,
    recordedCounters,
    sizeof(VkPerformanceCounterResultKHR),
    NULL);

// recordedCounters is filled with our counters, we will look at one for posterity
switch (counters[0].storage) {
    case VK_PERFORMANCE_COUNTER_STORAGE_INT32:
        // use recordCounters[0].int32 to get at the counter result!
        break;
    case VK_PERFORMANCE_COUNTER_STORAGE_INT64:
        // use recordCounters[0].int64 to get at the counter result!
        break;
    case VK_PERFORMANCE_COUNTER_STORAGE_UINT32:
        // use recordCounters[0].uint32 to get at the counter result!
        break;
    case VK_PERFORMANCE_COUNTER_STORAGE_UINT64:
        // use recordCounters[0].uint64 to get at the counter result!
        break;
    case VK_PERFORMANCE_COUNTER_STORAGE_FLOAT32:
        break;
}
// use recordCounters[0].float32 to get at the counter result!
break;
case VK_PERFORMANCE_COUNTER_STORAGE_FLOAT64:
    // use recordCounters[0].float64 to get at the counter result!
    break;
}

Version History

• Revision 1, 2019-10-08

VK_KHR_shader_clock

Name String

    VK_KHR_shader_clock

Extension Type

    Device extension

Registered Extension Number

    182

Revision

    1

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Contact

• Aaron Hagan (@ahagan)

Other Extension Metadata

Last Modified Date

    2019-4-25

IP Status

    No known IP claims.

Interactions and External Dependencies

• This extension requires SPV_KHR_shader_clock.
• This extension provides API support for ARB_shader_clock and EXT_shader_realtime_clock

Contributors

• Aaron Hagan, AMD
• Daniel Koch, NVIDIA
Description

This extension advertises the SPIR-V ShaderClockKHR capability for Vulkan, which allows a shader to query a real-time or monotonically incrementing counter at the subgroup level or across the device level. The two valid SPIR-V scopes for OpReadClockKHR are Subgroup and Device.

When using GLSL source-based shading languages, the clockRealtime*EXT() timing functions map to the OpReadClockKHR instruction with a scope of Device, and the clock*ARB() timing functions map to the OpReadClockKHR instruction with a scope of Subgroup.

New Structures

- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceShaderClockFeaturesKHR

New Enum Constants

- VK_KHR_SHADER_CLOCK_EXTENSION_NAME
- VK_KHR_SHADER_CLOCK_SPEC_VERSION

- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_CLOCK_FEATURES_KHR

New SPIR-V Capabilities

- ShaderClockKHR

Version History

- Revision 1, 2019-4-25 (Aaron Hagan)
  - Initial revision

VK_KHR_shader_terminate_invocation

Name String

- VK_KHR_shader_terminate_invocation

Extension Type

- Device extension

Registered Extension Number

- 216

Revision

- 1

Extension and Version Dependencies

- Requires Vulkan 1.0
• Requires `VK_KHR_get_physical_device_properties2`

Contact
• Jesse Hall @critsec

Other Extension Metadata

Last Modified Date
2020-08-11

IP Status
No known IP claims.

Interactions and External Dependencies
• Requires the `SPV_KHR_terminate_invocation` SPIR-V extension.

Contributors
• Alan Baker, Google
• Jeff Bolz, NVIDIA
• Jesse Hall, Google
• Ralph Potter, Samsung
• Tom Olson, Arm

Description
This extension adds Vulkan support for the `SPV_KHR_terminate_invocation` SPIR-V extension. That SPIR-V extension provides a new instruction, `OpTerminateInvocation`, which causes a shader invocation to immediately terminate and sets the coverage of shaded samples to 0; only previously executed instructions will have observable effects. The `OpTerminateInvocation` instruction, along with the `OpDemoteToHelperInvocation` instruction from the `VK_EXT_shader_demote_to_helper_invocation` extension, together replace the `OpKill` instruction, which could behave like either of these instructions. `OpTerminateInvocation` provides the behavior required by the GLSL `discard` statement, and should be used when available by GLSL compilers and applications that need the GLSL `discard` behavior.

New Structures
• Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  ◦ `VkPhysicalDeviceShaderTerminateInvocationFeaturesKHR`

New Enum Constants
• `VK_KHR_SHADER_TERMINATE_INVOCATION_EXTENSION_NAME`
• `VK_KHR_SHADER_TERMINATE_INVOCATION_SPEC_VERSION`

Extending `VkStructureType`:
• `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_TERMINATE_INVOCATION_FEATURES_KHR`
Version History

- Revision 1, 2020-08-11 (Jesse Hall)

VK_KHR_shared_presentable_image

Name String
VK_KHR_shared_presentable_image

Extension Type
Device extension

Registered Extension Number
112

Revision
1

Extension and Version Dependencies
- Requires Vulkan 1.0
- Requires VK_KHR_swapchain
- Requires VK_KHR_get_physical_device_properties2
- Requires VK_KHR_get_surface_capabilities2

Contact
- Alon Or-bach @alonorbach

Other Extension Metadata

Last Modified Date
2017-03-20

IP Status
No known IP claims.

Contributors
- Alon Or-bach, Samsung Electronics
- Ian Elliott, Google
- Jesse Hall, Google
- Pablo Ceballos, Google
- Chris Forbes, Google
- Jeff Juliano, NVIDIA
- James Jones, NVIDIA
- Daniel Rakos, AMD
Description

This extension extends `VK_KHR_swapchain` to enable creation of a shared presentable image. This allows the application to use the image while the presentation engine is accessing it, in order to reduce the latency between rendering and presentation.

New Commands

- `vkGetSwapchainStatusKHR`

New Structures

- Extending `VkSurfaceCapabilities2KHR`:
  - `VkSharedPresentSurfaceCapabilitiesKHR`

New Enum Constants

- `VK_KHR_SHARED_PRESENTABLE_IMAGE_EXTENSION_NAME`
- `VK_KHR_SHARED_PRESENTABLE_IMAGE_SPEC_VERSION`
- Extending `VkImageLayout`:
  - `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`
- Extending `VkPresentModeKHR`:
  - `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`
  - `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_SHARED_PRESENT_SURFACE_CAPABILITIES_KHR`

Issues

1) Should we allow a Vulkan WSI swapchain to toggle between normal usage and shared presentation usage?

**RESOLVED**: No. WSI swapchains are typically recreated with new properties instead of having their properties changed. This can also save resources, assuming that fewer images are needed for shared presentation, and assuming that most VR applications do not need to switch between normal and shared usage.

2) Should we have a query for determining how the presentation engine refresh is triggered?
RESOLVED: Yes. This is done via which presentation modes a surface supports.

3) Should the object representing a shared presentable image be an extension of a VkSwapchainKHR or a separate object?

RESOLVED: Extension of a swapchain due to overlap in creation properties and to allow common functionality between shared and normal presentable images and swapchains.

4) What should we call the extension and the new structures it creates?

RESOLVED: Shared presentable image / shared present.

5) Should the minImageCount and presentMode values of the VkSwapchainCreateInfoKHR be ignored, or required to be compatible values?

RESOLVED: minImageCount must be set to 1, and presentMode should be set to either VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR or VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR.

6) What should the layout of the shared presentable image be?

RESOLVED: After acquiring the shared presentable image, the application must transition it to the VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR layout prior to it being used. After this initial transition, any image usage that was requested during swapchain creation can be performed on the image without layout transitions being performed.

7) Do we need a new API for the trigger to refresh new content?

RESOLVED: vkQueuePresentKHR to act as API to trigger a refresh, as will allow combination with other compatible extensions to vkQueuePresentKHR.

8) How should an application detect a VK_ERROR_OUT_OF_DATE_KHR error on a swapchain using the VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR present mode?

RESOLVED: Introduce vkGetSwapchainStatusKHR to allow applications to query the status of a swapchain using a shared presentation mode.

9) What should subsequent calls to vkQueuePresentKHR for VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR swapchains be defined to do?

RESOLVED: State that implementations may use it as a hint for updated content.

10) Can the ownership of a shared presentable image be transferred to a different queue?

RESOLVED: No. It is not possible to transfer ownership of a shared presentable image obtained from a swapchain created using VK_SHARING_MODE_EXCLUSIVE after it has been presented.

11) How should vkQueueSubmit behave if a command buffer uses an image from a VK_ERROR_OUT_OF_DATE_KHR swapchain?

RESOLVED: vkQueueSubmit is expected to return the VK_ERROR_DEVICE_LOST error.

12) Can Vulkan provide any guarantee on the order of rendering, to enable beam chasing?
RESOLVED: This could be achieved via use of render passes to ensure strip rendering.

Version History

- Revision 1, 2017-03-20 (Alon Or-bach)
  - Internal revisions

VK_KHR_surface

Name String

VK_KHR_surface

Extension Type

Instance extension

Registered Extension Number

1

Revision

25

Extension and Version Dependencies

- Requires Vulkan 1.0

Contact

- James Jones @cubanismo
- Ian Elliott @ianelliottus

Other Extension Metadata

Last Modified Date

2016-08-25

IP Status

No known IP claims.

Contributors

- Patrick Doane, Blizzard
- Ian Elliott, LunarG
- Jesse Hall, Google
- James Jones, NVIDIA
- David Mao, AMD
- Norbert Nopper, Freescale
- Alon Or-bach, Samsung
- Daniel Rakos, AMD
The `VK_KHR_surface` extension is an instance extension. It introduces `VkSurfaceKHR` objects, which abstract native platform surface or window objects for use with Vulkan. It also provides a way to determine whether a queue family in a physical device supports presenting to particular surface.

Separate extensions for each platform provide the mechanisms for creating `VkSurfaceKHR` objects, but once created they may be used in this and other platform-independent extensions, in particular the `VK_KHR_swapchain` extension.

**New Object Types**

- `VkSurfaceKHR`

**New Commands**

- `vkDestroySurfaceKHR`
- `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`
- `vkGetPhysicalDeviceSurfaceFormatsKHR`
- `vkGetPhysicalDeviceSurfacePresentModesKHR`
- `vkGetPhysicalDeviceSurfaceSupportKHR`

**New Structures**

- `VkSurfaceCapabilitiesKHR`
- `VkSurfaceFormatKHR`

**New Enums**

- `VkColorSpaceKHR`
- `VkCompositeAlphaFlagBitsKHR`
- `VkPresentModeKHR`
- `VkSurfaceTransformFlagBitsKHR`

**New Bitmasks**

- `VkCompositeAlphaFlagsKHR`
New Enum Constants

- VK_KHR_SURFACE_EXTENSION_NAME
- VK_KHR_SURFACE_SPEC_VERSION

Extending VkObjectType:
- VK_OBJECT_TYPE_SURFACE_KHR

Extending VkResult:
- VK_ERROR_NATIVE_WINDOW_IN_USE_KHR
- VK_ERROR_SURFACE_LOST_KHR

Issues

1) Should this extension include a method to query whether a physical device supports presenting to a specific window or native surface on a given platform?

**RESOLVED:** Yes. Without this, applications would need to create a device instance to determine whether a particular window can be presented to. Knowing that a device supports presentation to a platform in general is not sufficient, as a single machine might support multiple seats, or instances of the platform that each use different underlying physical devices. Additionally, on some platforms, such as the X Window System, different drivers and devices might be used for different windows depending on which section of the desktop they exist on.

2) Should the `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`, `vkGetPhysicalDeviceSurfaceFormatsKHR`, and `vkGetPhysicalDeviceSurfacePresentModesKHR` functions be in this extension and operate on physical devices, rather than being in `VK_KHR_swapchain` (i.e. device extension) and being dependent on `VkDevice`?

**RESOLVED:** Yes. While it might be useful to depend on `VkDevice` (and therefore on enabled extensions and features) for the queries, Vulkan was released only with the `VkPhysicalDevice` versions. Many cases can be resolved by a Valid Usage statement, and/or by a separate `pNext` chain version of the query struct specific to a given extension or parameters, via extensible versions of the queries: `vkGetPhysicalDeviceSurfaceCapabilities2KHR`, and `vkGetPhysicalDeviceSurfaceFormats2KHR`.

3) Should Vulkan support Xlib or XCB as the API for accessing the X Window System platform?

**RESOLVED:** Both. XCB is a more modern and efficient API, but Xlib usage is deeply ingrained in many applications and likely will remain in use for the foreseeable future. Not all drivers necessarily need to support both, but including both as options in the core specification will probably encourage support, which should in turn ease adoption of the Vulkan API in older codebases. Additionally, the performance improvements possible with XCB likely will not have a measurable impact on the performance of Vulkan presentation and other minimal window system interactions defined here.

4) Should the GBM platform be included in the list of platform enums?

**RESOLVED:** Deferred, and will be addressed with a platform-specific extension to be written in the
Version History

- Revision 1, 2015-05-20 (James Jones)
  ○ Initial draft, based on LunarG KHR spec, other KHR specs, patches attached to bugs.

- Revision 2, 2015-05-22 (Ian Elliott)
  ○ Created initial Description section.
  ○ Removed query for whether a platform requires the use of a queue for presentation, since it was decided that presentation will always be modeled as being part of the queue.
  ○ Fixed typos and other minor mistakes.

- Revision 3, 2015-05-26 (Ian Elliott)
  ○ Improved the Description section.

- Revision 4, 2015-05-27 (James Jones)
  ○ Fixed compilation errors in example code.

- Revision 5, 2015-06-01 (James Jones)
  ○ Added issues 1 and 2 and made related spec updates.

- Revision 6, 2015-06-01 (James Jones)
  ○ Merged the platform type mappings table previously removed from VK_KHR_swapchain with the platform description table in this spec.
  ○ Added issues 3 and 4 documenting choices made when building the initial list of native platforms supported.

- Revision 7, 2015-06-11 (Ian Elliott)
  ○ Updated table 1 per input from the KHR TSG.
  ○ Updated issue 4 (GBM) per discussion with Daniel Stone. He will create a platform-specific extension sometime in the future.

- Revision 8, 2015-06-17 (James Jones)
  ○ Updated enum-extending values using new convention.
  ○ Fixed the value of VK_SURFACE_PLATFORM_INFO_TYPE_SUPPORTED_KHR.

- Revision 9, 2015-06-17 (James Jones)
  ○ Rebased on Vulkan API version 126.

- Revision 10, 2015-06-18 (James Jones)
  ○ Marked issues 2 and 3 resolved.

- Revision 11, 2015-06-23 (Ian Elliott)
  ○ Examples now show use of function pointers for extension functions.
  ○ Eliminated extraneous whitespace.

- Revision 12, 2015-07-07 (Daniel Rakos)
Added error section describing when each error is expected to be reported.

Replaced the term “queue node index” with “queue family index” in the spec as that is the agreed term to be used in the latest version of the core header and spec.

Replaced bool32_t with VkBool32.

• Revision 13, 2015-08-06 (Daniel Rakos)
  Updated spec against latest core API header version.

• Revision 14, 2015-08-20 (Ian Elliott)
  Renamed this extension and all of its enumerations, types, functions, etc. This makes it compliant with the proposed standard for Vulkan extensions.
  Switched from “revision” to “version”, including use of the VK_MAKE_VERSION macro in the header file.
  Did miscellaneous cleanup, etc.

• Revision 15, 2015-08-20 (Ian Elliott—porting a 2015-07-29 change from James Jones)
  Moved the surface transform enums here from VK_WSI_swapchain so they could be reused by VK_WSI_display.

• Revision 16, 2015-09-01 (James Jones)
  Restore single-field revision number.

• Revision 17, 2015-09-01 (James Jones)
  Fix example code compilation errors.

• Revision 18, 2015-09-26 (Jesse Hall)
  Replaced VkSurfaceDescriptionKHR with the VkSurfaceKHR object, which is created via layered extensions. Added VkDestroySurfaceKHR.

• Revision 19, 2015-09-28 (Jesse Hall)
  Renamed from VK_EXT_KHR_swapchain to VK_EXT_KHR_surface.

• Revision 20, 2015-09-30 (Jeff Vigil)
  Add error result VK_ERROR_SURFACE_LOST_KHR.

• Revision 21, 2015-10-15 (Daniel Rakos)
  Updated the resolution of issue #2 and include the surface capability queries in this extension.
  Renamed SurfaceProperties to SurfaceCapabilities as it better reflects that the values returned are the capabilities of the surface on a particular device.
  Other minor cleanup and consistency changes.

• Revision 22, 2015-10-26 (Ian Elliott)
  Renamed from VK_EXT_KHR_surface to VK_KHR_surface.

• Revision 23, 2015-11-03 (Daniel Rakos)
  Added allocation callbacks to vkDestroySurfaceKHR.

• Revision 24, 2015-11-10 (Jesse Hall)
• Removed VkSurfaceTransformKHR. Use VkSurfaceTransformFlagBitsKHR instead.
• Rename VkSurfaceCapabilitiesKHR member maxImageArraySize to maxImageArrayLayers.

• Revision 25, 2016-01-14 (James Jones)
  • Moved VK_ERROR_NATIVE_WINDOW_IN_USE_KHR from the VK_KHR_android_surface to the VK_KHR_surface extension.

• 2016-08-23 (Ian Elliott)
  • Update the example code, to not have so many characters per line, and to split out a new example to show how to obtain function pointers.

• 2016-08-25 (Ian Elliott)
  • A note was added at the beginning of the example code, stating that it will be removed from future versions of the appendix.

**VK_KHR_swapchain**

**Name String**

VK_KHR_swapchain

**Extension Type**

Device extension

**Registered Extension Number**

2

**Revision**

70

**Extension and Version Dependencies**

• Requires Vulkan 1.0
• Requires VK_KHR_surface

**Contact**

• James Jones 📩cubanismo
• Ian Elliott 📩ianelliottus

**Other Extension Metadata**

**Last Modified Date**

2017-10-06

**IP Status**

No known IP claims.

**Interactions and External Dependencies**

• Interacts with Vulkan 1.1
**Contributors**

- Patrick Doane, Blizzard
- Ian Elliott, LunarG
- Jesse Hall, Google
- Mathias Heyer, NVIDIA
- James Jones, NVIDIA
- David Mao, AMD
- Norbert Nopper, Freescale
- Alon Or-bach, Samsung
- Daniel Rakos, AMD
- Graham Sellers, AMD
- Jeff Vigil, Qualcomm
- Chia-I Wu, LunarG
- Jason Ekstrand, Intel
- Matthaeus G. Chajdas, AMD
- Ray Smith, ARM

**Description**

The `VK_KHR_swapchain` extension is the device-level companion to the `VK_KHR_surface` extension. It introduces `VkSwapchainKHR` objects, which provide the ability to present rendering results to a surface.

**New Object Types**

- `VkSwapchainKHR`

**New Commands**

- `vkAcquireNextImageKHR`
- `vkCreateSwapchainKHR`
- `vkDestroySwapchainKHR`
- `vkGetSwapchainImagesKHR`
- `vkQueuePresentKHR`

If Version 1.1 is supported:

- `vkAcquireNextImage2KHR`
- `vkGetDeviceGroupPresentCapabilitiesKHR`
- `vkGetDeviceGroupSurfacePresentModesKHR`
• vkGetPhysicalDevicePresentRectanglesKHR

**New Structures**

• VkPresentInfoKHR
  • VkSwapchainCreateInfoKHR

If **Version 1.1** is supported:

• VkAcquireNextImageInfoKHR
  • VkDeviceGroupPresentCapabilitiesKHR
  
- Extending VkBindImageMemoryInfo:
  • VkBindImageMemorySwapchainInfoKHR
  
- Extending VkImageCreateInfo:
  • VkImageSwapchainCreateInfoKHR
  
- Extending VkPresentInfoKHR:
  • VkDeviceGroupPresentInfoKHR
  
- Extending VkSwapchainCreateInfoKHR:
  • VkDeviceGroupSwapchainCreateInfoKHR

**New Enums**

• VkSwapchainCreateFlagBitsKHR

If **Version 1.1** is supported:

• VkDeviceGroupPresentModeFlagBitsKHR

**New Bitmasks**

• VkSwapchainCreateFlagsKHR

If **Version 1.1** is supported:

• VkDeviceGroupPresentModeFlagsKHR

**New Enum Constants**

• VK_KHR_SWAPCHAIN_EXTENSION_NAME
  • VK_KHR_SWAPCHAIN_SPEC_VERSION
  
- Extending VkImageLayout:
  • VK_IMAGE_LAYOUT_PRESENT_SRC_KHR
  
- Extending VkObjectType:
  • VK_OBJECT_TYPE_SWAPCHAIN_KHR
• Extending 

  • Extending 

   ◦ VK_ERROR_OUT_OF_DATE_KHR
    ◦ VK_SUBOPTIMAL_KHR

  • Extending 

   ◦ VK_STRUCTURE_TYPE_PRESENT_INFO_KHR
    ◦ VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR

If Version 1.1 is supported:

• Extending 

  ◦ VK_STRUCTURE_TYPE_ACQUIRE_NEXT_IMAGE_INFO_KHR
  ◦ VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_SWAPCHAIN_INFO_KHR
  ◦ VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_CAPABILITIES_KHR
  ◦ VK_STRUCTURE_TYPEDEVICE_GROUP_PRESENT_INFO_KHR
  ◦ VK_STRUCTURE_TYPE_DEVICE_GROUP_SWAPCHAIN_CREATE_INFO_KHR
  ◦ VK_STRUCTURE_TYPE_IMAGE_SWAPCHAIN_CREATE_INFO_KHR

• Extending 

  ◦ VK_SWAPCHAIN_CREATE_PROTECTED_BIT_KHR
  ◦ VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR

Issues

1) Does this extension allow the application to specify the memory backing of the presentable images?

**RESOLVED:** No. Unlike standard images, the implementation will allocate the memory backing of the presentable image.

2) What operations are allowed on presentable images?

**RESOLVED:** This is determined by the image usage flags specified when creating the presentable image's swapchain.

3) Does this extension support MSAA presentable images?

**RESOLVED:** No. Presentable images are always single-sampled. Multi-sampled rendering must use regular images. To present the rendering results the application must manually resolve the multi-sampled image to a single-sampled presentable image prior to presentation.

4) Does this extension support stereo/multi-view presentable images?

**RESOLVED:** Yes. The number of views associated with a presentable image is determined by the imageArrayLayers specified when creating a swapchain. All presentable images in a given swapchain use the same array size.
5) Are the layers of stereo presentable images half-sized?

**RESOLVED:** No. The image extents always match those requested by the application.

6) Do the “present” and “acquire next image” commands operate on a queue? If not, do they need to include explicit semaphore objects to interlock them with queue operations?

**RESOLVED:** The present command operates on a queue. The image ownership operation it represents happens in order with other operations on the queue, so no explicit semaphore object is required to synchronize its actions.

Applications may want to acquire the next image in separate threads from those in which they manage their queue, or in multiple threads. To make such usage easier, the acquire next image command takes a semaphore to signal as a method of explicit synchronization. The application must later queue a wait for this semaphore before queuing execution of any commands using the image.

7) Does `vkAcquireNextImageKHR` block if no images are available?

**RESOLVED:** The command takes a timeout parameter. Special values for the timeout are 0, which makes the call a non-blocking operation, and `UINT64_MAX`, which blocks indefinitely. Values in between will block for up to the specified time. The call will return when an image becomes available or an error occurs. It may, but is not required to, return before the specified timeout expires if the swapchain becomes out of date.

8) Can multiple presents be queued using one `vkQueuePresentKHR` call?

**RESOLVED:** Yes. `VkPresentInfoKHR` contains a list of swapchains and corresponding image indices that will be presented. When supported, all presentations queued with a single `vkQueuePresentKHR` call will be applied atomically as one operation. The same swapchain must not appear in the list more than once. Later extensions may provide applications stronger guarantees of atomicity for such present operations, and/or allow them to query whether atomic presentation of a particular group of swapchains is possible.

9) How do the presentation and acquire next image functions notify the application the targeted surface has changed?

**RESOLVED:** Two new result codes are introduced for this purpose:

- **`VK_SUBOPTIMAL_KHR`** - Presentation will still succeed, subject to the window resize behavior, but the swapchain is no longer configured optimally for the surface it targets. Applications should query updated surface information and recreate their swapchain at the next convenient opportunity.

- **`VK_ERROR_OUT_OF_DATE_KHR`** - Failure. The swapchain is no longer compatible with the surface it targets. The application must query updated surface information and recreate the swapchain before presentation will succeed.

These can be returned by both `vkAcquireNextImageKHR` and `vkQueuePresentKHR`.

10) Does the `vkAcquireNextImageKHR` command return a semaphore to the application via an
output parameter, or accept a semaphore to signal from the application as an object handle parameter?

**RESOLVED:** Accept a semaphore to signal as an object handle. This avoids the need to specify whether the application must destroy the semaphore or whether it is owned by the swapchain, and if the latter, what its lifetime is and whether it can be reused for other operations once it is received from `vkAcquireNextImageKHR`.

11) What types of swapchain queuing behavior should be exposed? Options include swap interval specification, mailbox/most recent vs. FIFO queue management, targeting specific vertical blank intervals or absolute times for a given present operation, and probably others. For some of these, whether they are specified at swapchain creation time or as per-present parameters needs to be decided as well.

**RESOLVED:** The base swapchain extension will expose 3 possible behaviors (of which, FIFO will always be supported):

- **Immediate present:** Does not wait for vertical blanking period to update the current image, likely resulting in visible tearing. No internal queue is used. Present requests are applied immediately.

- **Mailbox queue:** Waits for the next vertical blanking period to update the current image. No tearing should be observed. An internal single-entry queue is used to hold pending presentation requests. If the queue is full when a new presentation request is received, the new request replaces the existing entry, and any images associated with the prior entry become available for reuse by the application.

- **FIFO queue:** Waits for the next vertical blanking period to update the current image. No tearing should be observed. An internal queue containing `numSwapchainImages - 1` entries is used to hold pending presentation requests. New requests are appended to the end of the queue, and one request is removed from the beginning of the queue and processed during each vertical blanking period in which the queue is non-empty.

Not all surfaces will support all of these modes, so the modes supported will be returned using a surface information query. All surfaces must support the FIFO queue mode. Applications must choose one of these modes up front when creating a swapchain. Switching modes can be accomplished by recreating the swapchain.

12) Can `VK_PRESENT_MODE_MAILBOX_KHR` provide non-blocking guarantees for `vkAcquireNextImageKHR`? If so, what is the proper criteria?

**RESOLVED:** Yes. The difficulty is not immediately obvious here. Naively, if at least 3 images are requested, mailbox mode should always have an image available for the application if the application does not own any images when the call to `vkAcquireNextImageKHR` was made. However, some presentation engines may have more than one “current” image, and would still need to block in some cases. The right requirement appears to be that if the application allocates the surface's minimum number of images + 1 then it is guaranteed non-blocking behavior when it does not currently own any images.

13) Is there a way to create and initialize a new swapchain for a surface that has generated a `VK_SUBOPTIMAL_KHR` return code while still using the old swapchain?
RESOLVED: Not as part of this specification. This could be useful to allow the application to create an “optimal” replacement swapchain and rebuild all its command buffers using it in a background thread at a low priority while continuing to use the “suboptimal” swapchain in the main thread. It could probably use the same “atomic replace” semantics proposed for recreating direct-to-device swapchains without incurring a mode switch. However, after discussion, it was determined some platforms probably could not support concurrent swapchains for the same surface though, so this will be left out of the base KHR extensions. A future extension could add this for platforms where it is supported.

14) Should there be a special value for VkSurfaceCapabilitiesKHR::maxImageCount to indicate there are no practical limits on the number of images in a swapchain?

RESOLVED: Yes. There will often be cases where there is no practical limit to the number of images in a swapchain other than the amount of available resources (i.e., memory) in the system. Trying to derive a hard limit from things like memory size is prone to failure. It is better in such cases to leave it to applications to figure such soft limits out via trial/failure iterations.

15) Should there be a special value for VkSurfaceCapabilitiesKHR::currentExtent to indicate the size of the platform surface is undefined?

RESOLVED: Yes. On some platforms (Wayland, for example), the surface size is defined by the images presented to it rather than the other way around.

16) Should there be a special value for VkSurfaceCapabilitiesKHR::maxImageExtent to indicate there is no practical limit on the surface size?

RESOLVED: No. It seems unlikely such a system would exist. 0 could be used to indicate the platform places no limits on the extents beyond those imposed by Vulkan for normal images, but this query could just as easily return those same limits, so a special “unlimited” value does not seem useful for this field.

17) How should surface rotation and mirroring be exposed to applications? How do they specify rotation and mirroring transforms applied prior to presentation?

RESOLVED: Applications can query both the supported and current transforms of a surface. Both are specified relative to the device’s “natural” display rotation and direction. The supported transforms indicate which orientations the presentation engine accepts images in. For example, a presentation engine that does not support transforming surfaces as part of presentation, and which is presenting to a surface that is displayed with a 90-degree rotation, would return only one supported transform bit: VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR. Applications must transform their rendering by the transform they specify when creating the swapchain in preTransform field.

18) Can surfaces ever not support VK_MIRROR_NONE? Can they support vertical and horizontal mirroring simultaneously? Relatedly, should VK_MIRROR_NONE[_BIT] be zero, or bit one, and should applications be allowed to specify multiple pre and current mirror transform bits, or exactly one?

RESOLVED: Since some platforms may not support presenting with a transform other than the native window’s current transform, and prerotation/mirroring are specified relative to the device’s natural rotation and direction, rather than relative to the surface’s current rotation and direction, it is necessary to express lack of support for no mirroring. To allow this, the MIRROR_NONE enum must
occupy a bit in the flags. Since \texttt{MIRROR\_NONE} must be a bit in the bitmask rather than a bitmask with no values set, allowing more than one bit to be set in the bitmask would make it possible to describe undefined transforms such as \texttt{VK\_MIRROR\_NONE\_BIT | VK\_MIRROR\_HORIZONTAL\_BIT}, or a transform that includes both “no mirroring” and “horizontal mirroring” simultaneously. Therefore, it is desirable to allow specifying all supported mirroring transforms using only one bit. The question then becomes, should there be a \texttt{VK\_MIRROR\_HORIZONTAL\_AND\_VERTICAL\_BIT} to represent a simultaneous horizontal and vertical mirror transform? However, such a transform is equivalent to a 180 degree rotation, so presentation engines and applications that wish to support or use such a transform can express it through rotation instead. Therefore, 3 exclusive bits are sufficient to express all needed mirroring transforms.

19) Should support for sRGB be required?

\textbf{RESOLVED}: In the advent of UHD and HDR display devices, proper color space information is vital to the display pipeline represented by the swapchain. The app can discover the supported format/color-space pairs and select a pair most suited to its rendering needs. Currently only the sRGB color space is supported, future extensions may provide support for more color spaces. See issues 23 and 24.

20) Is there a mechanism to modify or replace an existing swapchain with one targeting the same surface?

\textbf{RESOLVED}: Yes. This is described above in the text.

21) Should there be a way to set prerotation and mirroring using native APIs when presenting using a Vulkan swapchain?

\textbf{RESOLVED}: Yes. The transforms that can be expressed in this extension are a subset of those possible on native platforms. If a platform exposes a method to specify the transform of presented images for a given surface using native methods and exposes more transforms or other properties for surfaces than Vulkan supports, it might be impossible, difficult, or inconvenient to set some of those properties using Vulkan KHR extensions and some using the native interfaces. To avoid overwriting properties set using native commands when presenting using a Vulkan swapchain, the application can set the pretransform to “inherit”, in which case the current native properties will be used, or if none are available, a platform-specific default will be used. Platforms that do not specify a reasonable default or do not provide native mechanisms to specify such transforms should not include the inherit bits in the \texttt{supportedTransforms} bitmask they return in \texttt{VkSurfaceCapabilitiesKHR}.

22) Should the content of presentable images be clipped by objects obscuring their target surface?

\textbf{RESOLVED}: Applications can choose which behavior they prefer. Allowing the content to be clipped could enable more efficient presentation methods on some platforms, but some applications might rely on the content of presentable images to perform techniques such as partial updates or motion blurs.

23) What is the purpose of specifying a \texttt{VkColorSpaceKHR} along with \texttt{VkFormat} when creating a swapchain?

\textbf{RESOLVED}: While Vulkan itself is color space agnostic (e.g. even the meaning of R, G, B and A can
be freely defined by the rendering application), the swapchain eventually will have to present the images on a display device with specific color reproduction characteristics. If any color space transformations are necessary before an image can be displayed, the color space of the presented image must be known to the swapchain. A swapchain will only support a restricted set of color format and -space pairs. This set can be discovered via `vkGetPhysicalDeviceSurfaceFormatsKHR`. As it can be expected that most display devices support the sRGB color space, at least one format/color-space pair has to be exposed, where the color space is `VK_COLOR_SPACE_SRGB_NONLINEAR_KHR`.

24) How are sRGB formats and the sRGB color space related?

**RESOLVED**: While Vulkan exposes a number of SRGB texture formats, using such formats does not guarantee working in a specific color space. It merely means that the hardware can directly support applying the non-linear transfer functions defined by the sRGB standard color space when reading from or writing to images of those formats. Still, it is unlikely that a swapchain will expose a `_SRGB` format along with any color space other than `VK_COLOR_SPACE_SRGB_NONLINEAR_KHR`.

On the other hand, non-`*_SRGB` formats will be very likely exposed in pair with a SRGB color space. This means, the hardware will not apply any transfer function when reading from or writing to such images, yet they will still be presented on a device with sRGB display characteristics. In this case the application is responsible for applying the transfer function, for instance by using shader math.

25) How are the lifetimes of surfaces and swapchains targeting them related?

**RESOLVED**: A surface must outlive any swapchains targeting it. A `VkSurfaceKHR` owns the binding of the native window to the Vulkan driver.

26) How can the client control the way the alpha component of swapchain images is treated by the presentation engine during compositing?

**RESOLVED**: We should add new enum values to allow the client to negotiate with the presentation engine on how to treat image alpha values during the compositing process. Since not all platforms can practically control this through the Vulkan driver, a value of `VK_COMPOSITE_ALPHA_INHERIT_BIT_KHR` is provided like for surface transforms.

27) Is `vkCreateSwapchainKHR` the right function to return `VK_ERROR_NATIVE_WINDOW_IN_USE_KHR`, or should the various platform-specific `VkSurfaceKHR` factory functions catch this error earlier?

**RESOLVED**: For most platforms, the `VkSurfaceKHR` structure is a simple container holding the data that identifies a native window or other object representing a surface on a particular platform. For the surface factory functions to return this error, they would likely need to register a reference on the native objects with the native display server somehow, and ensure no other such references exist. Surfaces were not intended to be that heavyweight.

Swapchains are intended to be the objects that directly manipulate native windows and communicate with the native presentation mechanisms. Swapchains will already need to communicate with the native display server to negotiate allocation and/or presentation of presentable images for a native surface. Therefore, it makes more sense for swapchain creation to be the point at which native object exclusivity is enforced. Platforms may choose to enforce further restrictions on the number of `VkSurfaceKHR` objects that may be created for the same native
window if such a requirement makes sense on a particular platform, but a global requirement is only sensible at the swapchain level.

**Version History**

- **Revision 1, 2015-05-20 (James Jones)**
  - Initial draft, based on LunarG KHR spec, other KHR specs, patches attached to bugs.

- **Revision 2, 2015-05-22 (Ian Elliott)**
  - Made many agreed-upon changes from 2015-05-21 KHR TSG meeting. This includes using only a queue for presentation, and having an explicit function to acquire the next image.
  - Fixed typos and other minor mistakes.

- **Revision 3, 2015-05-26 (Ian Elliott)**
  - Improved the Description section.
  - Added or resolved issues that were found in improving the Description. For example, pSurfaceDescription is used consistently, instead of sometimes using pSurface.

- **Revision 4, 2015-05-27 (James Jones)**
  - Fixed some grammatical errors and typos
  - Filled in the description of imageUseFlags when creating a swapchain.
  - Added a description of swapInterval.
  - Replaced the paragraph describing the order of operations on a queue for image ownership and presentation.

- **Revision 5, 2015-05-27 (James Jones)**
  - Imported relevant issues from the (abandoned) vk_wsi_persistent_swapchain_images extension.
  - Added issues 6 and 7, regarding behavior of the acquire next image and present commands with respect to queues.
  - Updated spec language and examples to align with proposed resolutions to issues 6 and 7.

- **Revision 6, 2015-05-27 (James Jones)**
  - Added issue 8, regarding atomic presentation of multiple swapchains
  - Updated spec language and examples to align with proposed resolution to issue 8.

- **Revision 7, 2015-05-27 (James Jones)**
  - Fixed compilation errors in example code, and made related spec fixes.

- **Revision 8, 2015-05-27 (James Jones)**
  - Added issue 9, and the related VK_SUBOPTIMAL_KHR result code.
  - Renamed VK_OUT_OF_DATE_KHR to VK_ERROR_OUT_OF_DATE_KHR.

- **Revision 9, 2015-05-27 (James Jones)**
  - Added inline proposed resolutions (marked with [JRJ]) to some XXX questions/issues. These should be moved to the issues section in a subsequent update if the proposals are adopted.
• Revision 10, 2015-05-28 (James Jones)
  ◦ Converted vkAcquireNextImageKHR back to a non-queue operation that uses a
    VkSemaphore object for explicit synchronization.
  ◦ Added issue 10 to determine whether vkAcquireNextImageKHR generates or returns
    semaphores, or whether it operates on a semaphore provided by the application.

• Revision 11, 2015-05-28 (James Jones)
  ◦ Marked issues 6, 7, and 8 resolved.
  ◦ Renamed VkSurfaceCapabilityPropertiesKHR to VkSurfacePropertiesKHR to better convey
    the mutable nature of the information it contains.

• Revision 12, 2015-05-28 (James Jones)
  ◦ Added issue 11 with a proposed resolution, and the related issue 12.
  ◦ Updated various sections of the spec to match the proposed resolution to issue 11.

• Revision 13, 2015-06-01 (James Jones)
  ◦ Moved some structures to VK_EXT_KHR_swap_chain to resolve the specification’s issues 1
    and 2.

• Revision 14, 2015-06-01 (James Jones)
  ◦ Added code for example 4 demonstrating how an application might make use of the two
    different present and acquire next image KHR result codes.
  ◦ Added issue 13.

• Revision 15, 2015-06-01 (James Jones)
  ◦ Added issues 14 - 16 and related spec language.
  ◦ Fixed some spelling errors.
  ◦ Added language describing the meaningful return values for vkAcquireNextImageKHR and
    vkQueuePresentKHR.

• Revision 16, 2015-06-02 (James Jones)
  ◦ Added issues 17 and 18, as well as related spec language.
  ◦ Removed some erroneous text added by mistake in the last update.

• Revision 17, 2015-06-15 (Ian Elliott)
  ◦ Changed special value from "-1" to "0" so that the data types can be unsigned.

• Revision 18, 2015-06-15 (Ian Elliott)
  ◦ Clarified the values of VkSurfacePropertiesKHR::minImageCount and the timeout parameter
    of the vkAcquireNextImageKHR function.

• Revision 19, 2015-06-17 (James Jones)
  ◦ Misc. cleanup. Removed resolved inline issues and fixed typos.
  ◦ Fixed clarification of VkSurfacePropertiesKHR::minImageCount made in version 18.
  ◦ Added a brief "Image Ownership" definition to the list of terms used in the spec.

• Revision 20, 2015-06-17 (James Jones)
• Updated enum-extending values using new convention.

• Revision 21, 2015-06-17 (James Jones)
  ◦ Added language describing how to use VK_IMAGE_LAYOUT_PRESENT_SOURCE_KHR.
  ◦ Cleaned up an XXX comment regarding the description of which queues vkQueuePresentKHR can be used on.

• Revision 22, 2015-06-17 (James Jones)
  ◦ Rebased on Vulkan API version 126.

• Revision 23, 2015-06-18 (James Jones)
  ◦ Updated language for issue 12 to read as a proposed resolution.
  ◦ Marked issues 11, 12, 13, 16, and 17 resolved.
  ◦ Temporarily added links to the relevant bugs under the remaining unresolved issues.
  ◦ Added issues 19 and 20 as well as proposed resolutions.

• Revision 24, 2015-06-19 (Ian Elliott)
  ◦ Changed special value for VkSurfacePropertiesKHR::currentExtent back to “-1” from “0”. This value will never need to be unsigned, and “0” is actually a legal value.

• Revision 25, 2015-06-23 (Ian Elliott)
  ◦ Examples now show use of function pointers for extension functions.
  ◦ Eliminated extraneous whitespace.

• Revision 26, 2015-06-25 (Ian Elliott)
  ◦ Resolved Issues 9 & 10 per KHR TSG meeting.

• Revision 27, 2015-06-25 (James Jones)
  ◦ Added oldSwapchain member to VkSwapchainCreateInfoKHR.

• Revision 28, 2015-06-25 (James Jones)
  ◦ Added the “inherit” bits to the rotation and mirroring flags and the associated issue 21.

• Revision 29, 2015-06-25 (James Jones)
  ◦ Added the “clipped” flag to VkSwapchainCreateInfoKHR, and the associated issue 22.
  ◦ Specified that presenting an image does not modify it.

• Revision 30, 2015-06-25 (James Jones)
  ◦ Added language to the spec that clarifies the behavior of vkCreateSwapchainKHR() when the oldSwapchain field of VkSwapchainCreateInfoKHR is not NULL.

• Revision 31, 2015-06-26 (Ian Elliott)
  ◦ Example of new VkSwapchainCreateInfoKHR members, “oldSwapchain” and “clipped”.
  ◦ Example of using VkSurfacePropertiesKHR::{min|max}ImageCount to set VkSwapchainCreateInfoKHR::minImageCount.
  ◦ Rename vkGetSurfaceInfoKHR()'s 4th parameter to “pDataSize”, for consistency with other functions.
* Add macro with C-string name of extension (just to header file).

- **Revision 32, 2015-06-26 (James Jones)**
  - Minor adjustments to the language describing the behavior of “oldSwapchain”
  - Fixed the version date on my previous two updates.

- **Revision 33, 2015-06-26 (Jesse Hall)**
  - Add usage flags to VkSwapchainCreateInfoKHR

- **Revision 34, 2015-06-26 (Ian Elliott)**
  - Rename vkQueuePresentKHR()'s 2nd parameter to “pPresentInfo”, for consistency with other functions.

- **Revision 35, 2015-06-26 (Ian Elliott)**
  - Merged the VkRotationFlagBitsKHR and VkMirrorFlagBitsKHR enums into a single VkSurfaceTransformFlagBitsKHR enum.

- **Revision 36, 2015-06-26 (Ian Elliott)**
  - Added a VkSurfaceTransformKHR enum that is not a bitmask. Each value in VkSurfaceTransformKHR corresponds directly to one of the bits in VkSurfaceTransformFlagBitsKHR so transforming from one to the other is easy. Having a separate enum means that currentTransform and preTransform are now unambiguous by definition.

- **Revision 37, 2015-06-29 (Ian Elliott)**
  - Corrected one of the signatures of vkAcquireNextImageKHR, which had the last two parameters switched from what it is elsewhere in the specification and header files.

- **Revision 38, 2015-06-30 (Ian Elliott)**
  - Corrected a typo in description of the vkGetSwapchainInfoKHR() function.
  - Corrected a typo in header file comment for VkPresentInfoKHR::sType.

- **Revision 39, 2015-07-07 (Daniel Rakos)**
  - Added error section describing when each error is expected to be reported.
  - Replaced bool32_t with VkBool32.

- **Revision 40, 2015-07-10 (Ian Elliott)**
  - Updated to work with version 138 of the `vulkan.h` header. This includes declaring the VkSwapchainKHR type using the new VK_DEFINE_NONDISP_HANDLE macro, and no longer extending VkObjectType (which was eliminated).

- **Revision 41, 2015-07-09 (Mathias Heyer)**
  - Added color space language.

- **Revision 42, 2015-07-10 (Daniel Rakos)**
  - Updated query mechanism to reflect the convention changes done in the core spec.
  - Removed “queue” from the name of VK_STRUCTURE_TYPE_QUEUE_PRESENT_INFO_KHR to be consistent with the established naming convention.
• Removed reference to the no longer existing VkObjectType enum.

• Revision 43, 2015-07-17 (Daniel Rakos)
  • Added support for concurrent sharing of swapchain images across queue families.
  • Updated sample code based on recent changes

• Revision 44, 2015-07-27 (Ian Elliott)
  • Noted that support for VK_PRESENT_MODE_FIFO_KHR is required. That is ICDs may optionally support IMMEDIATE and MAILBOX, but must support FIFO.

• Revision 45, 2015-08-07 (Ian Elliott)
  • Corrected a typo in spec file (type and variable name had wrong case for the imageColorSpace member of the VkSwapchainCreateInfoKHR struct).
  • Corrected a typo in header file (last parameter in PFN_vkGetSurfacePropertiesKHR was missing “KHR” at the end of type: VkSurfacePropertiesKHR).

• Revision 46, 2015-08-20 (Ian Elliott)
  • Renamed this extension and all of its enumerations, types, functions, etc. This makes it compliant with the proposed standard for Vulkan extensions.
  • Switched from “revision” to “version”, including use of the VK_MAKE_VERSION macro in the header file.
  • Made improvements to several descriptions.
  • Changed the status of several issues from PROPOSED to RESOLVED, leaving no unresolved issues.
  • Resolved several TODOs, did miscellaneous cleanup, etc.

• Revision 47, 2015-08-20 (Ian Elliott—porting a 2015-07-29 change from James Jones)
  • Moved the surface transform enums to VK_WSI_swapchain so they could be reused by VK_WSI_display.

• Revision 48, 2015-09-01 (James Jones)
  • Various minor cleanups.

• Revision 49, 2015-09-01 (James Jones)
  • Restore single-field revision number.

• Revision 50, 2015-09-01 (James Jones)
  • Update Example #4 to include code that illustrates how to use the oldSwapchain field.

• Revision 51, 2015-09-01 (James Jones)
  • Fix example code compilation errors.

• Revision 52, 2015-09-08 (Matthaeus G. Chajdas)
  • Corrected a typo.

• Revision 53, 2015-09-10 (Alon Or-bach)
  • Removed underscore from SWAP_CHAIN left in VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR.
• Revision 54, 2015-09-11 (Jesse Hall)
  ◦ Described the execution and memory coherence requirements for image transitions to and from VK_IMAGE_LAYOUT_PRESENT_SOURCE_KHR.

• Revision 55, 2015-09-11 (Ray Smith)
  ◦ Added errors for destroying and binding memory to presentable images

• Revision 56, 2015-09-18 (James Jones)
  ◦ Added fence argument to vkAcquireNextImageKHR
  ◦ Added example of how to meter a host thread based on presentation rate.

• Revision 57, 2015-09-26 (Jesse Hall)
  ◦ Replace VkSurfaceDescriptionKHR with VkSurfaceKHR.
  ◦ Added issue 25 with agreed resolution.

• Revision 58, 2015-09-28 (Jesse Hall)
  ◦ Renamed from VK_EXT_KHR_device_swapchain to VK_EXT_KHR_swapchain.

• Revision 59, 2015-09-29 (Ian Elliott)
  ◦ Changed vkDestroySwapchainKHR() to return void.

• Revision 60, 2015-10-01 (Jeff Vigil)
  ◦ Added error result VK_ERROR_SURFACE_LOST_KHR.

• Revision 61, 2015-10-05 (Jason Ekstrand)
  ◦ Added the VkCompositeAlpha enum and corresponding structure fields.

• Revision 62, 2015-10-12 (Daniel Rakos)
  ◦ Added VK_PRESENT_MODE_FIFO_RELAXED_KHR.

• Revision 63, 2015-10-15 (Daniel Rakos)
  ◦ Moved surface capability queries to VK_EXT_KHR_surface.

• Revision 64, 2015-10-26 (Ian Elliott)
  ◦ Renamed from VK_EXT_KHR_swapchain to VK_KHR_swapchain.

• Revision 65, 2015-10-28 (Ian Elliott)
  ◦ Added optional pResult member to VkPresentInfoKHR, so that per-swapchain results can be obtained from vkQueuePresentKHR().

• Revision 66, 2015-11-03 (Daniel Rakos)
  ◦ Added allocation callbacks to create and destroy functions.
  ◦ Updated resource transition language.
  ◦ Updated sample code.

• Revision 67, 2015-11-10 (Jesse Hall)
  ◦ Add reserved flags bitmask to VkSwapchainCreateInfoKHR.
  ◦ Modify naming and member ordering to match API style conventions, and so the
VkSwapchainCreateInfoKHR image property members mirror corresponding VkImageCreateInfo members but with an 'image' prefix.

- Make VkPresentInfoKHR::pResults non-const; it is an output array parameter.
- Make pPresentInfo parameter to vkQueuePresentKHR const.

Revision 68, 2016-04-05 (Ian Elliott)

- Moved the “validity” include for vkAcquireNextImage to be in its proper place, after the prototype and list of parameters.
- Clarified language about presentable images, including how they are acquired, when applications can and cannot use them, etc. As part of this, removed language about “ownership” of presentable images, and replaced it with more-consistent language about presentable images being “acquired” by the application.

2016-08-23 (Ian Elliott)

- Update the example code, to use the final API command names, to not have so many characters per line, and to split out a new example to show how to obtain function pointers. This code is more similar to the LunarG “cube” demo program.

2016-08-25 (Ian Elliott)

- A note was added at the beginning of the example code, stating that it will be removed from future versions of the appendix.

Revision 69, 2017-09-07 (Tobias Hector)

- Added interactions with Vulkan 1.1

Revision 70, 2017-10-06 (Ian Elliott)

- Corrected interactions with Vulkan 1.1

VK_KHR_swapchain mutable_format

Name String

VK_KHR_swapchain mutable_format

Extension Type

Device extension

Registered Extension Number

201

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_swapchain
- Requires VK_KHR_maintenance2
- Requires VK_KHR_image_format_list
Contact
  • Daniel Rakos

Other Extension Metadata

Last Modified Date
  2018-03-28

IP Status
  No known IP claims.

Contributors
  • Jason Ekstrand, Intel
  • Jan-Harald Fredriksen, ARM
  • Jesse Hall, Google
  • Daniel Rakos, AMD
  • Ray Smith, ARM

Description

This extension allows processing of swapchain images as different formats to that used by the window system, which is particularly useful for switching between sRGB and linear RGB formats.

It adds a new swapchain creation flag that enables creating image views from presentable images with a different format than the one used to create the swapchain.

New Enum Constants

- VK_KHR_SWAPCHAIN_MUTABLE_FORMAT_EXTENSION_NAME
- VK_KHR_SWAPCHAIN_MUTABLE_FORMAT_SPEC_VERSION
- Extending VkSwapchainCreateFlagBitsKHR:
  - VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR

Issues

1) Are there any new capabilities needed?

RESOLVED: No. It is expected that all implementations exposing this extension support swapchain image format mutability.

2) Do we need a separate VK_SWAPCHAIN_CREATE_EXTENDED_USAGE_BIT_KHR?

RESOLVED: No. This extension requires VK_KHR_maintenance2 and presentable images of swapchains created with VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR are created internally in a way equivalent to specifying both VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT and VK_IMAGE_CREATE_EXTENDED_USAGE_BIT_KHR.
3) Do we need a separate structure to allow specifying an image format list for swapchains?

**RESOLVED:** No. We simply use the same `VkImageFormatListCreateInfoKHR` structure introduced by `VK_KHR_image_format_list`. The structure is required to be included in the `pNext` chain of `VkSwapchainCreateInfoKHR` for swapchains created with `VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR`.

**Version History**

- Revision 1, 2018-03-28 (Daniel Rakos)
  - Internal revisions.

**VK_KHR_synchronization2**

**Name String**

`VK_KHR_synchronization2`

**Extension Type**

Device extension

**Registered Extension Number**

315

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

**Contact**

- Tobias Hector (@tobski)

**Other Extension Metadata**

**Last Modified Date**

2020-12-03

**Interactions and External Dependencies**

- Interacts with `VK_KHR_create_renderpass2`

**Contributors**

- Tobias Hector

**Description**

This extension modifies the original core synchronization APIs to simplify the interface and improve usability of these APIs. It also adds new pipeline stage and access flag types that extend...
into the 64-bit range, as we have run out within the 32-bit range. The new flags are identical to the old values within the 32-bit range, with new stages and bits beyond that.

Pipeline stages and access flags are now specified together in memory barrier structures, making the connection between the two more obvious. Additionally, scoping the pipeline stages into the barrier structure allows the use of the MEMORY_READ and MEMORY_WRITE flags without sacrificing precision. The per-stage access flags should be used to disambiguate specific accesses in a given stage or set of stages - for instance, between uniform reads and sampling operations.

Layout transitions have been simplified as well; rather than requiring a different set of layouts for depth/stencil/color attachments, there are generic VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL_KHR and VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR layouts which are contextually applied based on the image format. For example, for a depth format image, VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR is equivalent to VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR. VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR also functionally replaces VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL.

Events are now more efficient, because they include memory dependency information when you set them on the device. Previously, this information was only known when waiting on an event, so the dependencies could not be satisfied until the wait occurred. That sometimes meant stalling the pipeline when the wait occurred. The new API provides enough information for implementations to satisfy these dependencies in parallel with other tasks.

Queue submission has been changed to wrap command buffers and semaphores in extensible structures, which incorporate changes from Vulkan 1.1, VK_KHR_device_group, and VK_KHR_timeline_semaphore. This also adds a pipeline stage to the semaphore signal operation, mirroring the existing pipeline stage specification for wait operations.

Other miscellaneous changes include:

- Events can now be specified as interacting only with the device, allowing more efficient access to the underlying object.
- Image memory barriers that do not perform an image layout transition can be specified by setting oldLayout equal to newLayout.
  - E.g. the old and new layout can both be set to VK_IMAGE_LAYOUT_UNDEFINED, without discarding data in the image.
- Queue family ownership transfer parameters are simplified in some cases.
- Where two synchronization commands need to be matched up (queue transfer operations, events), the dependency information specified in each place must now match completely for consistency.
- Extensions with commands or functions with a VkPipelineStageFlags or VkPipelineStageFlagBits parameter have had those APIs replaced with equivalents using VkPipelineStageFlags2KHR.
- The new event and barrier interfaces are now more extensible for future changes.
- Relevant pipeline stage masks can now be specified as empty with the new VK_PIPELINE_STAGE_NONE_KHR and VK_PIPELINE_STAGE_2_NONE_KHR values.
- VkMemoryBarrier2KHR can be chained to VkSubpassDependency2, overriding the original 32-bit stage and access masks.
New Base Types

- VkFlags64

New Commands

- vkCmdPipelineBarrier2KHR
- vkCmdResetEvent2KHR
- vkCmdSetEvent2KHR
- vkCmdWaitEvents2KHR
- vkCmdWriteTimestamp2KHR
- vkQueueSubmit2KHR

New Structures

- VkBufferMemoryBarrier2KHR
- VkCommandBufferSubmitInfoKHR
- VkDependencyInfoKHR
- VkImageMemoryBarrier2KHR
- VkSemaphoreSubmitInfoKHR
- VkSubmitInfo2KHR
- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceSynchronization2FeaturesKHR
- Extending VkSubpassDependency2:
  - VkMemoryBarrier2KHR

New Enums

- VkAccessFlagBits2KHR
- VkPipelineStageFlagBits2KHR
- VkSubmitFlagBitsKHR

New Bitmasks

- VkAccessFlags2KHR
- VkPipelineStageFlags2KHR
- VkSubmitFlagsKHR

New Enum Constants

- VK_KHR_SYNCHRONIZATION_2_EXTENSION_NAME
• **VK_KHR_SYNCHRONIZATION_2_SPEC_VERSION**

Extending **VkAccessFlagBits**:
  ◦ **VK_ACCESS_NONE_KHR**

Extending **VkEventCreateFlagBits**:
  ◦ **VK_EVENT_CREATE_DEVICE_ONLY_BIT_KHR**

Extending **VkImageLayout**:
  ◦ **VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL_KHR**
  ◦ **VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR**

Extending **VkPipelineStageFlagBits**:
  ◦ **VK_PIPELINE_STAGE_NONE_KHR**

Extending **VkStructureType**:
  ◦ **VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER_2_KHR**
  ◦ **VK_STRUCTURE_TYPE_COMMAND_BUFFER_SUBMIT_INFO_KHR**
  ◦ **VK_STRUCTURE_TYPE_DEPENDENCY_INFO_KHR**
  ◦ **VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER_2_KHR**
  ◦ **VK_STRUCTURE_TYPE_MEMORY_BARRIER_2_KHR**
  ◦ **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SYNCHRONIZATION_2_FEATURES_KHR**
  ◦ **VK_STRUCTURE_TYPE_SEMAPHORE_SUBMIT_INFO_KHR**
  ◦ **VK_STRUCTURE_TYPE_SUBMIT_INFO_2_KHR**

If **VK_EXT_blend_operation_advanced** is supported:

• Extending **VkAccessFlagBits2KHR**:
  ◦ **VK_ACCESS_2_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT**

If **VK_KHR_fragment_shading_rate** is supported:

• Extending **VkAccessFlagBits2KHR**:
  ◦ **VK_ACCESS_2_FRAGMENT_SHADING_RATE_ATTACHMENT_READ_BIT_KHR**

• Extending **VkPipelineStageFlagBits2KHR**:
  ◦ **VK_PIPELINE_STAGE_2_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR**

**Examples**

See [https://github.com/KhronosGroup/Vulkan-Docs/wiki/Synchronization-Examples](https://github.com/KhronosGroup/Vulkan-Docs/wiki/Synchronization-Examples)

**Version History**

• Revision 1, 2020-12-03 (Tobias Hector)
  ◦ Internal revisions
VK_EXT_4444_formats

Name String
VK_EXT_4444_formats

Extension Type
Device extension

Registered Extension Number
341

Revision
1

Extension and Version Dependencies
- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

Contact
- Joshua Ashton 🌐Joshua-Ashton

Other Extension Metadata

Last Modified Date
2020-07-28

IP Status
No known IP claims.

Contributors
- Joshua Ashton, Valve
- Jason Ekstrand, Intel

Description
This extension defines the VK_FORMAT_A4R4G4B4_UNORM_PACK16_EXT and VK_FORMAT_A4B4G4R4_UNORM_PACK16_EXT formats which are defined in other current graphics APIs.

This extension may be useful for building translation layers for those APIs or for porting applications that use these formats without having to resort to swizzles.

When VK_EXT_custom_border_color is used, these formats are not subject to the same restrictions for border color without format as with VK_FORMAT_B4G4R4A4_UNORM_PACK16.

New Structures
- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDevice4444FormatsFeaturesEXT
New Enum Constants

- VK_EXT_4444_FORMATS_EXTENSION_NAME
- VK_EXT_4444_FORMATS_SPEC_VERSION

Extending VkFormat:
  - VK_FORMAT_A4B4G4R4_UNORM_PACK16_EXT
  - VK_FORMAT_A4R4G4B4_UNORM_PACK16_EXT

Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_4444_FORMATS_FEATURES_EXT

Version History

- Revision 1, 2020-07-04 (Joshua Ashton)
  - Initial draft

VK_EXT_application_parameters

Name String
  VK_EXT_application_parameters

Extension Type
  Instance extension

Registered Extension Number
  436

Revision
  1

Extension and Version Dependencies
  - Requires Vulkan 1.0

Contact
  - Daniel Koch @dgkoch

Other Extension Metadata

Last Modified Date
  2021-12-14

Contributors
  - Daniel Koch, NVIDIA
  - Jonathan Mccaffrey, NVIDIA
  - Aidan Fabius, CoreAVI
Description

This instance extension enables an application to pass application parameters to the implementation at instance or device creation time.

The application parameters consist of a set of vendor-specific keys and values. Each key is a 32-bit enum, and each value is a 64-bit integer. The valid keys, range of values, and default values are documented external to this specification in implementation-specific documentation.

This extension is an instance extension rather than a device extension so that the implementation can modify reported \texttt{VkPhysicalDevice} properties or features as needed.

New Structures

- Extending \texttt{VkApplicationInfo}, \texttt{VkDeviceCreateInfo}:
  - \texttt{VkApplicationParametersEXT}

New Enum Constants

- \texttt{VK_EXT_APPLICATION_PARAMETERS_EXTENSION_NAME}
- \texttt{VK_EXT_APPLICATION_PARAMETERS_SPEC_VERSION}
- Extending \texttt{VkStructureType}:
  - \texttt{VK_STRUCTURE_TYPE_APPLICATION_PARAMETERS_EXT}

Issues

1. How should the \texttt{key} enumerants be assigned?

  \textbf{RESOLVED}: The \texttt{key} enumerants are completely implementation-specific and do not need to be centrally reserved. They should be documented in the implementation-specific documentation. The vendor ID and optionally the device ID are provided to disambiguate between multiple ICDs or devices.

2. How does an application know what application parameters are valid on a particular implementation?

  \textbf{DISCUSSION}: There is no ability to enumerate device or system properties before an instance is created, however \texttt{key} and \texttt{values} \textbf{must} be recognized by an implementation in order for instance or device creation to succeed. The vendor and optionally the device ID are provided to identify which ICD or device the application parameters are targeted at.

3. Is it OK if the "valid value" for specified keys is not from static documented values, but must be consistent-with/interdependent-on other \texttt{VkApplicationParametersEXT}?

  \textbf{DISCUSSION}: Yes this is fine. Examples for how this could be used include:

  - a checksum \texttt{key} where the \texttt{value} is computed based on other \texttt{VkApplicationParametersEXT} structures in the \texttt{pNext} chain.
- an "application key" which either implies or explicitly lists a set of prevalidated key/value pairs.

Version History

- Revision 1, 2021-12-14 (Daniel Koch)
  - Initial revision

**VK_EXT_astc_decode_mode**

**Name String**
- VK_EXT_astc_decode_mode

**Extension Type**
- Device extension

**Registered Extension Number**
- 68

**Revision**
- 1

**Extension and Version Dependencies**
- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

**Contact**
- Jan-Harald Fredriksen janharaldfredriksen-arm

**Other Extension Metadata**

**Last Modified Date**
- 2018-08-07

**Contributors**
- Jan-Harald Fredriksen, Arm

**Description**

The existing specification requires that low dynamic range (LDR) ASTC textures are decompressed to FP16 values per component. In many cases, decompressing LDR textures to a lower precision intermediate result gives acceptable image quality. Source material for LDR textures is typically authored as 8-bit UNORM values, so decoding to FP16 values adds little value. On the other hand, reducing precision of the decoded result reduces the size of the decompressed data, potentially improving texture cache performance and saving power.

The goal of this extension is to enable this efficiency gain on existing ASTC texture data. This is achieved by giving the application the ability to select the intermediate decoding precision.
Three decoding options are provided:

- Decode to **VK_FORMAT_R16G16B16A16_SFLOAT** precision: This is the default, and matches the required behavior in the core API.
- Decode to **VK_FORMAT_R8G8B8A8_UNORM** precision: This is provided as an option in LDR mode.
- Decode to **VK_FORMAT_E5B9G9R9_UFLOAT_PACK32** precision: This is provided as an option in both LDR and HDR mode. In this mode, negative values cannot be represented and are clamped to zero. The alpha component is ignored, and the results are as if alpha was 1.0. This decode mode is optional and support can be queried via the physical device properties.

**New Structures**

- Extending **VkImageViewCreateInfo**:
  - **VkImageViewASTCDecodeModeEXT**
- Extending **VkPhysicalDeviceFeatures2, VkDeviceCreateInfo**:
  - **VkPhysicalDeviceASTCDecodeFeaturesEXT**

**New Enum Constants**

- **VK_EXT_ASTC_DECODE_MODE_EXTENSION_NAME**
- **VK_EXT_ASTC_DECODE_MODE_SPEC_VERSION**
- Extending **VkStructureType**:
  - **VK_STRUCTURE_TYPE_IMAGE_VIEW_ASTC_DECODE_MODE_EXT**
  - **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ASTC_DECODE_FEATURES_EXT**

**Issues**

1) Are implementations allowed to decode at a higher precision than what is requested?

RESOLUTION: No.

If we allow this, then this extension could be exposed on all implementations that support ASTC. But developers would have no way of knowing what precision was actually used, and thus whether the image quality is sufficient at reduced precision.

2) Should the decode mode be image view state and/or sampler state?

RESOLUTION: Image view state only.

Some implementations treat the different decode modes as different texture formats.
Example

Create an image view that decodes to `VK_FORMAT_R8G8B8A8_UNORM` precision:

```c
VkImageViewASTCDecodeModeEXT decodeMode =
{
    VK_STRUCTURE_TYPE_IMAGE_VIEW_ASTC DECODE_MODE_EXT, // sType
    NULL, // pNext
    VK_FORMAT_R8G8B8A8_UNORM // decode mode
};

VkImageViewCreateInfo createInfo =
{
    VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO, // sType
    &decodeMode, // pNext
    // flags, image, viewType set to application-desired values
    VK_FORMAT_ASTC_8x8_UNORM_BLOCK, // format
    // components, subresourceRange set to application-desired values
};

VkImageView imageView;
VkResult result = vkCreateImageView(
    device,
    &createInfo,
    NULL,
    &imageView);
```

Version History

- Revision 1, 2018-08-07 (Jan-Harald Fredriksen)
  - Initial revision

VK_EXT_blend_operation_advanced

Name String

`VK_EXT_blend_operation_advanced`

Extension Type

Device extension

Registered Extension Number

149

Revision

2

Extension and Version Dependencies

- Requires Vulkan 1.0
Description

This extension adds a number of “advanced” blending operations that can be used to perform new color blending operations, many of which are more complex than the standard blend modes provided by unextended Vulkan. This extension requires different styles of usage, depending on the level of hardware support and the enabled features:

- If `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT::advancedBlendCoherentOperations` is `VK_FALSE`, the new blending operations are supported, but a memory dependency must separate each advanced blend operation on a given sample. `VK_ACCESS_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT` is used to synchronize reads using advanced blend operations.

- If `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT::advancedBlendCoherentOperations` is `VK_TRUE`, advanced blend operations obey primitive order just like basic blend operations.

In unextended Vulkan, the set of blending operations is limited, and can be expressed very simply. The `VK_BLEND_OP_MIN` and `VK_BLEND_OP_MAX` blend operations simply compute component-wise minimums or maximums of source and destination color components. The `VK_BLEND_OP_ADD`, `VK_BLEND_OP_SUBTRACT`, and `VK_BLEND_OP_REVERSE_SUBTRACT` modes multiply the source and destination colors by source and destination factors and either add the two products together or subtract one from the other. This limited set of operations supports many common blending operations but precludes the use of more sophisticated transparency and blending operations commonly available in many dedicated imaging APIs.

This extension provides a number of new “advanced” blending operations. Unlike traditional blending operations using `VK_BLEND_OP_ADD`, these blending equations do not use source and destination factors specified by `VkBlendFactor`. Instead, each blend operation specifies a complete equation based on the source and destination colors. These new blend operations are used for both RGB and alpha components; they must not be used to perform separate RGB and alpha blending (via different values of color and alpha `VkBlendOp`).

These blending operations are performed using premultiplied colors, where RGB colors can be considered premultiplied or non-premultiplied by alpha, according to the `srcPremultiplied` and `dstPremultiplied` members of `VkPipelineColorBlendAdvancedStateCreateInfoEXT`. If a color is considered non-premultiplied, the (R,G,B) color components are multiplied by the alpha component prior to blending. For non-premultiplied color components in the range [0,1], the corresponding premultiplied color component would have values in the range [0 × A, 1 × A].
Many of these advanced blending equations are formulated where the result of blending source and destination colors with partial coverage have three separate contributions: from the portions covered by both the source and the destination, from the portion covered only by the source, and from the portion covered only by the destination. The blend parameter `VkPipelineColorBlendAdvancedStateCreateInfoEXT:blendOverlap` can be used to specify a correlation between source and destination pixel coverage. If set to `VK_BLEND_OVERLAP_CONJOINT_EXT`, the source and destination are considered to have maximal overlap, as would be the case if drawing two objects on top of each other. If set to `VK_BLEND_OVERLAP_DISJOINT_EXT`, the source and destination are considered to have minimal overlap, as would be the case when rendering a complex polygon tessellated into individual non-intersecting triangles. If set to `VK_BLEND_OVERLAP_UNCORRELATED_EXT`, the source and destination coverage are assumed to have no spatial correlation within the pixel.

In addition to the coherency issues on implementations not supporting `advancedBlendCoherentOperations`, this extension has several limitations worth noting. First, the new blend operations have a limit on the number of color attachments they can be used with, as indicated by `VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT::advancedBlendMaxColorAttachments`. Additionally, blending precision may be limited to 16-bit floating-point, which may result in a loss of precision and dynamic range for framebuffer formats with 32-bit floating-point components, and in a loss of precision for formats with 12- and 16-bit signed or unsigned normalized integer components.

**New Structures**

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT`
- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT`
- Extending `VkPipelineColorBlendStateCreateInfo`:
  - `VkPipelineColorBlendAdvancedStateCreateInfoEXT`

**New Enums**

- `VkBlendOverlapEXT`

**New Enum Constants**

- `VK_EXT_BLEND_OPERATION_ADVANCED_EXTENSION_NAME`
- `VK_EXT_BLEND_OPERATION_ADVANCED_SPEC_VERSION`
- Extending `VkAccessFlagBits`:
  - `VK_ACCESS_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT`
- Extending `VkBlendOp`:
  - `VK_BLEND_OP_BLUE_EXT`
  - `VK_BLEND_OP_COLORBURN_EXT`
- VK_BLEND_OP_COLOR_DODGE_EXT
- VK_BLEND_OP_CONTRAST_EXT
- VK_BLEND_OP_DARKEN_EXT
- VK_BLEND_OP_DIFFERENCE_EXT
- VK_BLEND_OP_DST_ATOP_EXT
- VK_BLEND_OP_DST_EXT
- VK_BLEND_OP_DST_IN_EXT
- VK_BLEND_OP_DST_OUT_EXT
- VK_BLEND_OP_DST_OVER_EXT
- VK_BLEND_OP_EXCLUSION_EXT
- VK_BLEND_OP_GREEN_EXT
- VK_BLEND_OP_HARDLIGHT_EXT
- VK_BLEND_OP_HARDCOPY_EXT
- VK_BLEND_OP_HSL_COLOR_EXT
- VK_BLEND_OP_HSL_HUE_EXT
- VK_BLEND_OP_HSL_LUMINOSITY_EXT
- VK_BLEND_OP_HSL_SATURATION_EXT
- VK_BLEND_OP_INVERSE_EXT
- VK_BLEND_OP_INVersed_RGB_EXT
- VK_BLEND_OP_INVersed_RGBA_EXT
- VK_BLEND_OP_LIGHTEN_EXT
- VK_BLEND_OP_LINEARBURN_EXT
- VK_BLEND_OP_LINEARDODGE_EXT
- VK_BLEND_OP_LINEARLIGHT_EXT
- VK_BLEND_OP_MINUS_CLAMPED_EXT
- VK_BLEND_OP_MINUS_EXT
- VK_BLEND_OP_MULTIPLY_EXT
- VK_BLEND_OP_OVERLAY_EXT
- VK_BLEND_OP_PINLIGHT_EXT
- VK_BLEND_OP_PLUS_CLAMPED_ALPHA_EXT
- VK_BLEND_OP_PLUS_CLAMPED_EXT
- VK_BLEND_OP_PLUS_DARKER_EXT
- VK_BLEND_OP_PLUS_EXT
- VK_BLEND_OP_RED_EXT
- VK_BLEND_OP_SCREEN_EXT
• VK_BLEND_OP_SOFTLIGHT_EXT
• VK_BLEND_OP_SRC_ATOP_EXT
• VK_BLEND_OP_SRC_EXT
• VK_BLEND_OP_SRC_IN_EXT
• VK_BLEND_OP_SRC_OUT_EXT
• VK_BLEND_OP_SRC_OVER_EXT
• VK_BLEND_OP_VIVIDLIGHT_EXT
• VK_BLEND_OP_XOR_EXT
• VK_BLEND_OP_ZERO_EXT

• Extending VkStructureType:
  • VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BLEND_OPERATION_ADVANCED_FEATURES_EXT
  • VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BLEND_OPERATION_ADVANCED_PROPERTIES_EXT
  • VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_ADVANCED_STATE_CREATE_INFO_EXT

Issues
None.

Version History
• Revision 1, 2017-06-12 (Jeff Bolz)
  ◦ Internal revisions
• Revision 2, 2017-06-12 (Jeff Bolz)
  ◦ Internal revisions

VK_EXT_calibrated_timestamps

Name String
VK_EXT_calibrated_timestamps

Extension Type
Device extension

Registered Extension Number
185

Revision
2

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2
Contact

• Daniel Rakos drakos-amd

Other Extension Metadata

Last Modified Date
2018-10-04

IP Status
No known IP claims.

Contributors

• Matthaeus G. Chajdas, AMD
• Alan Harrison, AMD
• Derrick Owens, AMD
• Daniel Rakos, AMD
• Jason Ekstrand, Intel
• Keith Packard, Valve

Description

This extension provides an interface to query calibrated timestamps obtained quasi simultaneously from two time domains.

New Commands

• vkGetCalibratedTimestampsEXT
• vkGetPhysicalDeviceCalibrateableTimeDomainsEXT

New Structures

• VkCalibratedTimestampInfoEXT

New Enums

• VkTimeDomainEXT

New Enum Constants

• VK_EXT_CALIBRATED_TIMESTAMPS_EXTENSION_NAME
• VK_EXT_CALIBRATED_TIMESTAMPS_SPEC_VERSION

Extending VkStructureType:

• VK_STRUCTURE_TYPE_CALIBRATED_TIMESTAMP_INFO_EXT
## Issues

1) Is the device timestamp value returned in the same time domain as the timestamp values written by `vkCmdWriteTimestamp`?

**RESOLVED:** Yes.

2) What time domain is the host timestamp returned in?

**RESOLVED:** A query is provided to determine the calibrateable time domains. The expected host time domain used on Windows is that of QueryPerformanceCounter, and on Linux that of CLOCK_MONOTONIC.

3) Should we support other time domain combinations than just one host and the device time domain?

**RESOLVED:** Supporting that would need the application to query the set of supported time domains, while supporting only one host and the device time domain would only need a query for the host time domain type. The proposed API chooses the general approach for the sake of extensibility.

4) Should we use CLOCK_MONOTONIC_RAW instead of CLOCK_MONOTONIC?

**RESOLVED:** CLOCK_MONOTONIC is usable in a wider set of situations, however, it is subject to NTP adjustments so some use cases may prefer CLOCK_MONOTONIC_RAW. Thus this extension allows both to be exposed.

5) How can the application extrapolate future device timestamp values from the calibrated timestamp value?

**RESOLVED:** `VkPhysicalDeviceLimits::timestampPeriod` makes it possible to calculate future device timestamps as follows:

\[
\text{futureTimestamp} = \text{calibratedTimestamp} + \frac{\text{deltaNanoseconds}}{\text{timestampPeriod}}
\]

6) In what queue are timestamp values in time domain `VK_TIME_DOMAIN_DEVICE_EXT` captured by `vkGetCalibratedTimestampsEXT`?

**RESOLVED:** An implementation supporting this extension will have all its VkQueue share the same time domain.

6) Can the host and device timestamp values drift apart over longer periods of time?

**RESOLVED:** Yes, especially as some time domains by definition allow for that to happen (e.g. CLOCK_MONOTONIC is subject to NTP adjustments). Thus it is recommended that applications re-calibrate from time to time.

7) Should we add a query for reporting the maximum deviation of the timestamp values returned by calibrated timestamp queries?
RESOLVED: A global query seems inappropriate and difficult to enforce. However, it is possible to return the maximum deviation any single calibrated timestamp query can have by sampling one of the time domains twice as follows:

```plaintext
timestampX = timestampX_before = SampleTimeDomain(X)
for each time domain Y != X
    timestampY = SampleTimeDomain(Y)
    timestampX_after = SampleTimeDomain(X)
    maxDeviation = timestampX_after - timestampX_before
```

8) Can the maximum deviation reported ever be zero?

RESOLVED: Unless the tick of each clock corresponding to the set of time domains coincides and all clocks can literally be sampled simultaneously, there is not really a possibility for the maximum deviation to be zero, so by convention the maximum deviation is always at least the maximum of the length of the ticks of the set of time domains calibrated and thus can never be zero.

**Version History**

- Revision 2, 2021-03-16 (Lionel Landwerlin)
  - Specify requirement on device timestamps
- Revision 1, 2018-10-04 (Daniel Rakos)
  - Internal revisions.

**VK_EXT_color_write_enable**

**Name String**

VK_EXT_color_write_enable

**Extension Type**

Device extension

**Registered Extension Number**

382

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

**Contact**

- Sharif Elcott selcott
Other Extension Metadata

Last Modified Date
2020-02-25

IP Status
No known IP claims.

Contributors
- Sharif Elcott, Google
- Tobias Hector, AMD
- Piers Daniell, NVIDIA

Description
This extension allows for selectively enabling and disabling writes to output color attachments via a pipeline dynamic state.

The intended use cases for this new state are mostly identical to those of colorWriteMask, such as selectively disabling writes to avoid feedback loops between subpasses or bandwidth savings for unused outputs. By making the state dynamic, one additional benefit is the ability to reduce pipeline counts and pipeline switching via shaders that write a superset of the desired data of which subsets are selected dynamically. The reason for a new state, colorWriteEnable, rather than making colorWriteMask dynamic is that, on many implementations, the more flexible per-component semantics of the colorWriteMask state cannot be made dynamic in a performant manner.

New Commands
- `vkCmdSetColorWriteEnableEXT`

New Structures
- Extending `VkPhysicalDeviceFeatures2, VkDeviceCreateInfo`:
  - `VkPhysicalDeviceColorWriteEnableFeaturesEXT`
- Extending `VkPipelineColorBlendStateCreateInfo`:
  - `VkPipelineColorWriteCreateInfoEXT`

New Enum Constants
- `VK_EXT_COLOR_WRITE_ENABLE_EXTENSION_NAME`
- `VK_EXT_COLOR_WRITE_ENABLE_SPEC_VERSION`
- Extending `VkDynamicState`:
  - `VK_DYNAMIC_STATE_COLOR_WRITE_ENABLE_EXT`
- Extending `VkStructureType`:
• VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COLOR_WRITE_ENABLE_FEATURES_EXT
• VK_STRUCTURE_TYPE_PIPELINE_COLOR_WRITE_CREATE_INFO_EXT

Version History

• Revision 1, 2020-01-25 (Sharif Elcott)
  ◦ Internal revisions

VK_EXT_conservative_rasterization

Name String

VK_EXT_conservative_rasterization

Extension Type

Device extension

Registered Extension Number

102

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Contact

• Piers Daniell

Other Extension Metadata

Last Modified Date

2020-06-09

Interactions and External Dependencies

• This extension requires SPV_EXT_fragment_fully_covered if the VkPhysicalDeviceConservativeRasterizationPropertiesEXT::fullyCoveredFragmentShaderInputVariable feature is used.
• This extension requires SPV_KHR_post_depth_coverage if the VkPhysicalDeviceConservativeRasterizationPropertiesEXT::conservativeRasterizationPostDepthCoverage feature is used.
• This extension provides API support for GL_NV_conservative_raster_underestimation if the VkPhysicalDeviceConservativeRasterizationPropertiesEXT::fullyCoveredFragmentShaderInputVariable feature is used.

Contributors

• Daniel Koch, NVIDIA
Description

This extension adds a new rasterization mode called conservative rasterization. There are two modes of conservative rasterization; overestimation and underestimation.

When overestimation is enabled, if any part of the primitive, including its edges, covers any part of the rectangular pixel area, including its sides, then a fragment is generated with all coverage samples turned on. This extension allows for some variation in implementations by accounting for differences in overestimation, where the generating primitive size is increased at each of its edges by some sub-pixel amount to further increase conservative pixel coverage. Implementations can allow the application to specify an extra overestimation beyond the base overestimation the implementation already does. It also allows implementations to either cull degenerate primitives or rasterize them.

When underestimation is enabled, fragments are only generated if the rectangular pixel area is fully covered by the generating primitive. If supported by the implementation, when a pixel rectangle is fully covered the fragment shader input variable builtin called FullyCoveredEXT is set to true. The shader variable works in either overestimation or underestimation mode.

Implementations can process degenerate triangles and lines by either discarding them or generating conservative fragments for them. Degenerate triangles are those that end up with zero area after the rasterizer quantizes them to the fixed-point pixel grid. Degenerate lines are those with zero length after quantization.

New Structures

- Extending VkPhysicalDeviceProperties2:
  - VkPhysicalDeviceConservativeRasterizationPropertiesEXT

- Extending VkPipelineRasterizationStateCreateInfo:
  - VkPipelineRasterizationConservativeStateCreateInfoEXT

New Enums

- VkConservativeRasterizationModeEXT

New Bitmasks

- VkPipelineRasterizationConservativeStateCreateFlagsEXT
New Enum Constants

- `VK_EXT_CONSERVATIVE_RASTERIZATION_EXTENSION_NAME`
- `VK_EXT_CONSERVATIVE_RASTERIZATION_SPEC_VERSION`

Extending `VkStructureType`:
- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CONSERVATIVE_RASTERIZATION_PROPERTIES_EXT`
- `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_CONSERVATIVE_STATE_CREATE_INFO_EXT`

Version History

- Revision 1.1, 2020-09-06 (Piers Daniell)
  - Add missing SPIR-V and GLSL dependencies.
- Revision 1, 2017-08-28 (Piers Daniell)
  - Internal revisions

**VK_EXT_custom_border_color**

Name String

`VK_EXT_custom_border_color`

Extension Type

Device extension

Registered Extension Number

288

Revision

12

Extension and Version Dependencies

- Requires Vulkan 1.0

Special Uses

- OpenGL / ES support
- D3D support

Contact

- Liam Middlebrook [liam-middlebrook](mailto:liam-middlebrook)

Other Extension Metadata

Last Modified Date

2020-04-16

IP Status

No known IP claims.
Contributors

- Joshua Ashton, Valve
- Hans-Kristian Arntzen, Valve
- Philip Rebohle, Valve
- Liam Middlebrook, NVIDIA
- Jeff Bolz, NVIDIA
- Tobias Hector, AMD
- Jason Ekstrand, Intel
- Spencer Fricke, Samsung Electronics
- Graeme Leese, Broadcom
- Jesse Hall, Google
- Jan-Harald Fredriksen, ARM
- Tom Olson, ARM
- Stuart Smith, Imagination Technologies
- Donald Scorgie, Imagination Technologies
- Alex Walters, Imagination Technologies
- Peter Quayle, Imagination Technologies

Description

This extension provides cross-vendor functionality to specify a custom border color for use when the sampler address mode `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO BORDER` is used.

To create a sampler which uses a custom border color set `VkSamplerCreateInfo::borderColor` to one of:

- `VK_BORDER_COLOR_FLOAT_CUSTOM_EXT`
- `VK_BORDER_COLOR_INT_CUSTOM_EXT`

When `VK_BORDER_COLOR_FLOAT_CUSTOM_EXT` or `VK_BORDER_COLOR_INT_CUSTOM_EXT` is used, applications must provide a `VkSamplerCustomBorderColorCreateInfoEXT` in the `pNext` chain for `VkSamplerCreateInfo`.

New Structures

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceCustomBorderColorFeaturesEXT`
- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDeviceCustomBorderColorPropertiesEXT`
- Extending `VkSamplerCreateInfo`:
New Enum Constants

- `VK_EXT_CUSTOM_BORDER_COLOR_EXTENSION_NAME`
- `VK_EXT_CUSTOM_BORDER_COLOR_SPEC_VERSION`

Extending `VkBorderColor`:

- `VK_BORDER_COLOR_FLOAT_CUSTOM_EXT`
- `VK_BORDER_COLOR_INT_CUSTOM_EXT`

Extending `VkStructureType`:

- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CUSTOM_BORDER_COLOR_FEATURES_EXT`
- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CUSTOM_BORDER_COLOR_PROPERTIES_EXT`
- `VK_STRUCTURE_TYPE_SAMPLER_CUSTOM_BORDER_COLOR_CREATE_INFO_EXT`

Issues

1) Should `VkClearColorValue` be used for the border color value, or should we have our own struct/union? Do we need to specify the type of the input values for the components? This is more of a concern if `VkClearColorValue` is used here because it provides a union of float,int,uint types.

**RESOLVED:** Will reuse existing `VkClearColorValue` structure in order to easily take advantage of float,int,uint borderColor types.

2) For hardware which supports a limited number of border colors what happens if that number is exceeded? Should this be handled by the driver unbeknownst to the application? In Revision 1 we had solved this issue using a new Object type, however that may have lead to additional system resource consumption which would otherwise not be required.

**RESOLVED:** Added `VkPhysicalDeviceCustomBorderColorPropertiesEXT::maxCustomBorderColorSamplers` for tracking implementation-specific limit, and Valid Usage statement handling overflow.

3) Should this be supported for immutable samplers at all, or by a feature bit? Some implementations may not be able to support custom border colors on immutable samplers — is it worthwhile enabling this to work on them for implementations that can support it, or forbidding it entirely.

**RESOLVED:** Samplers created with a custom border color are forbidden from being immutable. This resolves concerns for implementations where the custom border color is an index to a LUT instead of being directly embedded into sampler state.

4) Should UINT and SINT (unsigned integer and signed integer) border color types be separated or should they be combined into one generic INT (integer) type?

**RESOLVED:** Separating these does not make much sense as the existing fixed border color types do not have this distinction, and there is no reason in hardware to do so. This separation would also create unnecessary work and considerations for the application.
Version History

- Revision 1, 2019-10-10 (Joshua Ashton)
  - Internal revisions.
- Revision 2, 2019-10-11 (Liam Middlebrook)
  - Remove VkCustomBorderColor object and associated functions
  - Add issues concerning HW limitations for custom border color count
- Revision 3, 2019-10-12 (Joshua Ashton)
  - Re-expose the limits for the maximum number of unique border colors
  - Add extra details about border color tracking
  - Fix typos
- Revision 4, 2019-10-12 (Joshua Ashton)
  - Changed maxUniqueCustomBorderColors to a uint32_t from a VkDeviceSize
- Revision 5, 2019-10-14 (Liam Middlebrook)
  - Added features bit
- Revision 6, 2019-10-15 (Joshua Ashton)
  - Type-ize VK_BORDER_COLOR_CUSTOM
  - Fix const-ness on pNext of VkSamplerCustomBorderColorCreateInfoEXT
- Revision 7, 2019-11-26 (Liam Middlebrook)
  - Renamed maxUniqueCustomBorderColors to maxCustomBorderColors
- Revision 8, 2019-11-29 (Joshua Ashton)
  - Renamed borderColor member of VkSamplerCustomBorderColorCreateInfoEXT to customBorderColor
- Revision 9, 2020-02-19 (Joshua Ashton)
  - Renamed maxCustomBorderColors to maxCustomBorderColorSamplers
- Revision 10, 2020-02-21 (Joshua Ashton)
  - Added format to VkSamplerCustomBorderColorCreateInfoEXT and feature bit
- Revision 11, 2020-04-07 (Joshua Ashton)
  - Dropped UINT/SINT border color differences, consolidated types
- Revision 12, 2020-04-16 (Joshua Ashton)
  - Renamed VK_BORDER_COLOR_CUSTOM_FLOAT_EXT to VK_BORDER_COLOR_FLOAT_CUSTOM_EXT for consistency

VK_EXT_debug_utils

Name String

VK_EXT_debug_utils
Extension Type
  Instance extension

Registered Extension Number
  129

Revision
  2

Extension and Version Dependencies
  • Requires Vulkan 1.0

Special Use
  • Debugging tools

Contact
  • Mark Young marky-lunarg

Other Extension Metadata

Last Modified Date
  2020-04-03

Revision
  2

IP Status
  No known IP claims.

Dependencies
  • This extension is written against version 1.0 of the Vulkan API.
  • Requires VkObjectType

Contributors
  • Mark Young, LunarG
  • Baldur Karlsson
  • Ian Elliott, Google
  • Courtney Goeltzenleuchter, Google
  • Karl Schultz, LunarG
  • Mark Lobodzinski, LunarG
  • Mike Schuchardt, LunarG
  • Jaakko Konttinen, AMD
  • Dan Ginsburg, Valve Software
  • Rolando Olivares, Epic Games
  • Dan Baker, Oxide Games
Description

Due to the nature of the Vulkan interface, there is very little error information available to the developer and application. By using the VK_EXT_debug_utils extension, developers can obtain more information. When combined with validation layers, even more detailed feedback on the application’s use of Vulkan will be provided.

This extension provides the following capabilities:

- The ability to create a debug messenger which will pass along debug messages to an application supplied callback.
- The ability to identify specific Vulkan objects using a name or tag to improve tracking.
- The ability to identify specific sections within a VkQueue or VkCommandBuffer using labels to aid organization and offline analysis in external tools.

The main difference between this extension and VK_EXT_debug_report and VK_EXT_debug_marker is that those extensions use VkDebugReportObjectTypeEXT to identify objects. This extension uses the core VkObjectType in place of VkDebugReportObjectTypeEXT. The primary reason for this move is that no future object type handle enumeration values will be added to VkDebugReportObjectTypeEXT since the creation of VkObjectType.

In addition, this extension combines the functionality of both VK_EXT_debug_report and VK_EXT_debug_marker by allowing object name and debug markers (now called labels) to be returned to the application’s callback function. This should assist in clarifying the details of a debug message including: what objects are involved and potentially which location within a VkQueue or VkCommandBuffer the message occurred.

New Object Types

- VkDebugUtilsMessengerEXT

New Commands

- vkCmdBeginDebugUtilsLabelEXT
- vkCmdEndDebugUtilsLabelEXT
- vkCmdInsertDebugUtilsLabelEXT
- vkCreateDebugUtilsMessengerEXT
- vkDestroyDebugUtilsMessengerEXT
- vkQueueBeginDebugUtilsLabelEXT
- vkQueueEndDebugUtilsLabelEXT
- vkQueueInsertDebugUtilsLabelEXT
• vkSetDebugUtilsObjectNameEXT
• vkSetDebugUtilsObjectTagEXT
• vkSubmitDebugUtilsMessageEXT

New Structures

• VkDebugUtilsLabelEXT
• VkDebugUtilsMessengerCallbackDataEXT
• VkDebugUtilsObjectNameInfoEXT
• VkDebugUtilsObjectTagInfoEXT
• Extending VkInstanceCreateInfo:
  ◦ VkDebugUtilsMessengerCreateInfoEXT

New Function Pointers

•PFN_vkDebugUtilsMessengerCallbackEXT

New Enums

• VkDebugUtilsMessageSeverityFlagBitsEXT
• VkDebugUtilsMessageTypeFlagBitsEXT

New Bitmasks

• VkDebugUtilsMessageSeverityFlagsEXT
• VkDebugUtilsMessageTypeFlagsEXT
• VkDebugUtilsMessengerCallbackDataFlagsEXT
• VkDebugUtilsMessengerCreateInfoFlagsEXT

New Enum Constants

• VK_EXT_DEBUG_UTILS_EXTENSION_NAME
• VK_EXT_DEBUG_UTILS_SPEC_VERSION
• Extending VkObjectType:
  ◦ VK_OBJECT_TYPE_DEBUG_UTILS_MESSENGER_EXT
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT
  ◦ VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CALLBACK_DATA_EXT
  ◦ VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT
  ◦ VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_NAME_INFO_EXT
  ◦ VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_TAG_INFO_EXT
Examples

Example 1

VK_EXT_debug_utils allows an application to register multiple callbacks with any Vulkan component wishing to report debug information. Some callbacks may log the information to a file, others may cause a debug break point or other application defined behavior. An application can register callbacks even when no validation layers are enabled, but they will only be called for loader and, if implemented, driver events.

To capture events that occur while creating or destroying an instance an application can link a VkDebugUtilsMessengerCreateInfoEXT structure to the pNext element of the VkInstanceCreateInfo structure given to vkCreateInstance.

Example uses: Create three callback objects. One will log errors and warnings to the debug console using Windows OutputDebugString. The second will cause the debugger to break at that callback when an error happens and the third will log warnings to stdout.

```c
extern VkInstance instance;
VkResult res;
VkDebugUtilsMessengerEXT cb1, cb2, cb3;

// Must call extension functions through a function pointer:
PFN_vkCreateDebugUtilsMessengerEXT pfnCreateDebugUtilsMessengerEXT =
(PFN_vkCreateDebugUtilsMessengerEXT)vkGetInstanceProcAddr(instance, "vkCreateDebugUtilsMessengerEXT");
PFN_vkDestroyDebugUtilsMessengerEXT pfnDestroyDebugUtilsMessengerEXT =
(PFN_vkDestroyDebugUtilsMessengerEXT)vkGetInstanceProcAddr(instance, "vkDestroyDebugUtilsMessengerEXT");

VkDebugUtilsMessengerCreateInfoEXT callback1 = {
    VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT, // sType
    NULL, // pNext
    0, // flags
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT | messageSeverity
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT, // flags
    VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT | messageSeverity
    VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT,
    myOutputDebugString,
    NULL // pUserCallback
};
res = pfnCreateDebugUtilsMessengerEXT(instance, &callback1, NULL, &cb1);
if (res != VK_SUCCESS) {
    // Do error handling for VK_ERROR_OUT_OF_MEMORY
}

callback1.messageSeverity = VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
callback1.pfnUserCallback = myDebugBreak;
```
callback1.pUserData = NULL;
res = pfnCreateDebugUtilsMessengerEXT(instance, &callback1, NULL, &cb2);
if (res != VK_SUCCESS) {
    // Do error handling for VK_ERROR_OUT_OF_MEMORY
}

VkDebugUtilsMessengerCreateInfoEXT callback3 = {
    VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT, // sType
    NULL, // pNext
    0, // flags
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT, // messageSeverity
    VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT | VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT, // messageType
    mystdOutLogger, // pfnUserCallback
    NULL // pUserData
};
res = pfnCreateDebugUtilsMessengerEXT(instance, &callback3, NULL, &cb3);
if (res != VK_SUCCESS) {
    // Do error handling for VK_ERROR_OUT_OF_MEMORY
}
...

// Remove callbacks when cleaning up
pfnDestroyDebugUtilsMessengerEXT(instance, cb1, NULL);
pfnDestroyDebugUtilsMessengerEXT(instance, cb2, NULL);
pfnDestroyDebugUtilsMessengerEXT(instance, cb3, NULL);

Example 2

Associate a name with an image, for easier debugging in external tools or with validation layers that can print a friendly name when referring to objects in error messages.

extern VkInstance instance;
extern VkDevice device;
extern VkImage image;

// Must call extension functions through a function pointer:
PFN_vkSetDebugUtilsObjectNameEXT pfnSetDebugUtilsObjectNameEXT =
    (PFN_vkSetDebugUtilsObjectNameEXT)vkGetInstanceProcAddr(instance, "vkSetDebugUtilsObjectNameEXT");

// Set a name on the image
const VkDebugUtilsObjectNameInfoEXT imageNameInfo =
    {VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_NAME_INFO_EXT, // sType
    NULL, // pNext
    VK_OBJECT_TYPE_IMAGE, // objectType
    &imageName}
Example 3

Annotating regions of a workload with naming information so that offline analysis tools can display a more usable visualization of the commands submitted.

```c
extern VkInstance instance;
extern VkCommandBuffer commandBuffer;

// Must call extension functions through a function pointer:
PFN_vkQueueBeginDebugUtilsLabelEXT pfnQueueBeginDebugUtilsLabelEXT =
    (PFN_vkQueueBeginDebugUtilsLabelEXT)vkGetInstanceProcAddr(instance,
    "vkQueueBeginDebugUtilsLabelEXT");
PFN_vkQueueEndDebugUtilsLabelEXT pfnQueueEndDebugUtilsLabelEXT =
    (PFN_vkQueueEndDebugUtilsLabelEXT)vkGetInstanceProcAddr(instance,
    "vkQueueEndDebugUtilsLabelEXT");
PFN_vkCmdBeginDebugUtilsLabelEXT pfnCmdBeginDebugUtilsLabelEXT =
    (PFN_vkCmdBeginDebugUtilsLabelEXT)vkGetInstanceProcAddr(instance,
    "vkCmdBeginDebugUtilsLabelEXT");
PFN_vkCmdEndDebugUtilsLabelEXT pfnCmdEndDebugUtilsLabelEXT =
    (PFN_vkCmdEndDebugUtilsLabelEXT)vkGetInstanceProcAddr(instance,
    "vkCmdEndDebugUtilsLabelEXT");
PFN_vkCmdInsertDebugUtilsLabelEXT pfnCmdInsertDebugUtilsLabelEXT =
    (PFN_vkCmdInsertDebugUtilsLabelEXT)vkGetInstanceProcAddr(instance,
    "vkCmdInsertDebugUtilsLabelEXT");

// Describe the area being rendered
correct
const VkDebugUtilsLabelEXT houseLabel =
    {
        VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT, // sType
        NULL, // pNext
        "Brick House", // pLabelName
        { 1.0f, 0.0f, 0.0f, 1.0f }, // color
    }

// Start an annotated group of calls under the 'Brick House' name
pfnCmdBeginDebugUtilsLabelEXT(commandBuffer, &houseLabel);
{
    // A mutable structure for each part being rendered
    VkDebugUtilsLabelEXT housePartLabel =
    {
        VK_STRUCTURE_TYPE_DEBUG_UTILS
```
VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT, // sType
NULL,       // pNext
NULL,       // pLabelName
{ 0.0f, 0.0f, 0.0f, 0.0f },          // color
};

// Set the name and insert the marker
housePartLabel.pLabelName = "Walls";
PFNCmdInsertDebugUtilsLabelEXT(commandBuffer, &housePartLabel);

// Insert the drawcall for the walls
vkCmdDrawIndexed(commandBuffer, 1000, 1, 0, 0);

// Insert a recursive region for two sets of windows
housePartLabel.pLabelName = "Windows";
PFNCmdBeginDebugUtilsLabelEXT(commandBuffer, &housePartLabel);
{
    vkCmdDrawIndexed(commandBuffer, 75, 6, 1000, 0, 0);
    vkCmdDrawIndexed(commandBuffer, 100, 2, 1450, 0, 0);
}
PFNCmdEndDebugUtilsLabelEXT(commandBuffer);

housePartLabel.pLabelName = "Front Door";
PFNCmdInsertDebugUtilsLabelEXT(commandBuffer, &housePartLabel);

vkCmdDrawIndexed(commandBuffer, 350, 1, 1650, 0, 0);

housePartLabel.pLabelName = "Roof";
PFNCmdInsertDebugUtilsLabelEXT(commandBuffer, &housePartLabel);

vkCmdDrawIndexed(commandBuffer, 500, 1, 2000, 0, 0);
}
// End the house annotation started above
PFNCmdEndDebugUtilsLabelEXT(commandBuffer);

// Do other work
vkEndCommandBuffer(commandBuffer);

// Describe the queue being used
const VkDebugUtilsLabelEXT queueLabel =
{
    VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT, // sType
    NULL,       // pNext
    "Main Render Work",    // pLabelName
    { 0.0f, 1.0f, 0.0f, 1.0f },          // color
};

// Identify the queue label region
PFNQueueBeginDebugUtilsLabelEXT(queue, &queueLabel);
Issues

1) Should we just name this extension VK_EXT_debug_report2

RESOLVED: No. There is enough additional changes to the structures to break backwards compatibility. So, a new name was decided that would not indicate any interaction with the previous extension.

2) Will validation layers immediately support all the new features.

RESOLVED: Not immediately. As one can imagine, there is a lot of work involved with converting the validation layer logging over to the new functionality. Basic logging, as seen in the origin VK_EXT_debug_report extension will be made available immediately. However, adding the labels and object names will take time. Since the priority for Khronos at this time is to continue focusing on Valid Usage statements, it may take a while before the new functionality is fully exposed.

3) If the validation layers will not expose the new functionality immediately, then what is the point of this extension?

RESOLVED: We needed a replacement for VK_EXT_debug_report because the VkDebugReportObjectTypeEXT enumeration will no longer be updated and any new objects will need to be debugged using the new functionality provided by this extension.

4) Should this extension be split into two separate parts (1 extension that is an instance extension providing the callback functionality, and another device extension providing the general debug marker and annotation functionality)?

RESOLVED: No, the functionality for this extension is too closely related. If we did split up the extension, where would the structures and enums live, and how would you define that the device behavior in the instance extension is really only valid if the device extension is enabled, and the functionality is passed in. It is cleaner to just define this all as an instance extension, plus it allows the application to enable all debug functionality provided with one enable string during vkCreateInstance.
Version History

- Revision 1, 2017-09-14 (Mark Young and all listed Contributors)
  - Initial draft, based on VK_EXT_debug_report and VK_EXT_debug_marker in addition to previous feedback supplied from various companies including Valve, Epic, and Oxide games.
- Revision 2, 2020-04-03 (Mark Young and Piers Daniell)
  - Updated to allow either NULL or an empty string to be passed in for pObjectName in VkDebugUtilsObjectNameInfoEXT, because the loader and various drivers support NULL already.

VK_EXT_depth_clip_enable

Name String
  VK_EXT_depth_clip_enable

Extension Type
  Device extension

Registered Extension Number
  103

Revision
  1

Extension and Version Dependencies
  - Requires Vulkan 1.0

Special Use
  - D3D support

Contact
  - Piers Daniell pdaniell-nv

Other Extension Metadata

Last Modified Date
  2018-12-20

Contributors
  - Daniel Rakos, AMD
  - Henri Verbeet, CodeWeavers
  - Jeff Bolz, NVIDIA
  - Philip Rebohle, DXVK
  - Tobias Hector, AMD
Description

This extension allows the depth clipping operation, that is normally implicitly controlled by `VkPipelineRasterizationStateCreateInfo::depthClampEnable`, to instead be controlled explicitly by `VkPipelineRasterizationDepthClipStateCreateInfoEXT::depthClipEnable`.

This is useful for translating DX content which assumes depth clamping is always enabled, but depth clip can be controlled by the DepthClipEnable rasterization state (D3D12_RASTERIZER_DESC).

New Structures

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceDepthClipEnableFeaturesEXT`
- Extending `VkPipelineRasterizationStateCreateInfo`:
  - `VkPipelineRasterizationDepthClipStateCreateInfoEXT`

New Bitmasks

- `VkPipelineRasterizationDepthClipStateCreateFlagsEXT`

New Enum Constants

- `VK_EXT_DEPTH_CLIP_ENABLE_EXTENSION_NAME`
- `VK_EXT_DEPTH_CLIP_ENABLE_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEPTH_CLIP_ENABLE_FEATURES_EXT`
  - `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_DEPTH_CLIP_STATE_CREATE_INFO_EXT`

Version History

- Revision 1, 2018-12-20 (Piers Daniell)
  - Internal revisions

**VK_EXT_depth_range_unrestricted**

Name String

- `VK_EXT_depth_range_unrestricted`

Extension Type

- Device extension

Registered Extension Number

- 14

Revision

- 1
Extension and Version Dependencies

• Requires Vulkan 1.0

Contact

• Piers Daniell

Other Extension Metadata

Last Modified Date

2017-06-22

Contributors

• Daniel Koch, NVIDIA
• Jeff Bolz, NVIDIA

Description

This extension removes the VkViewport minDepth and maxDepth restrictions that the values must be between 0.0 and 1.0, inclusive. It also removes the same restriction on VkPipelineDepthStencilStateCreateInfo minDepthBounds and maxDepthBounds. Finally it removes the restriction on the depth value in VkClearDepthStencilValue.

New Enum Constants

• VK_EXT_DEPTH_RANGE_UNRESTRICTED_EXTENSION_NAME
• VK_EXT_DEPTH_RANGE_UNRESTRICTED_SPEC_VERSION

Issues

1) How do VkViewport minDepth and maxDepth values outside of the 0.0 to 1.0 range interact with Primitive Clipping?

RESOLVED: The behavior described in Primitive Clipping still applies. If depth clamping is disabled the depth values are still clipped to 0 ≤ z ≤ w before the viewport transform. If depth clamping is enabled the above equation is ignored and the depth values are instead clamped to the VkViewport minDepth and maxDepth values, which in the case of this extension can be outside of the 0.0 to 1.0 range.

2) What happens if a resulting depth fragment is outside of the 0.0 to 1.0 range and the depth buffer is fixed-point rather than floating-point?

RESOLVED: The supported range of a fixed-point depth buffer is 0.0 to 1.0 and depth fragments are clamped to this range.

Version History

• Revision 1, 2017-06-22 (Piers Daniell)
  ◦ Internal revisions
VK_EXT_direct_mode_display

Name String
VK_EXT_direct_mode_display

Extension Type
Instance extension

Registered Extension Number
89

Revision
1

Extension and Version Dependencies
- Requires Vulkan 1.0
- Requires VK_KHR_display

Contact
- James Jones cubanismo

Other Extension Metadata

Last Modified Date
2016-12-13

IP Status
No known IP claims.

Contributors
- Pierre Boudier, NVIDIA
- James Jones, NVIDIA
- Damien Leone, NVIDIA
- Pierre-Loup Griffais, Valve
- Liam Middlebrook, NVIDIA

Description
This is extension, along with related platform extensions, allows applications to take exclusive control of displays associated with a native windowing system. This is especially useful for virtual reality applications that wish to hide HMDs (head mounted displays) from the native platform’s display management system, desktop, and/or other applications.

New Commands
- vkReleaseDisplayEXT
New Enum Constants

- VK_EXT_DIRECT_MODE_DISPLAY_EXTENSION_NAME
- VK_EXT_DIRECT_MODE_DISPLAY_SPEC_VERSION

Issues

1) Should this extension and its related platform-specific extensions leverage VK_KHR_display, or provide separate equivalent interfaces.

**RESOLVED:** Use VK_KHR_display concepts and objects. VK_KHR_display can be used to enumerate all displays on the system, including those attached to/in use by a window system or native platform, but VK_KHR_display_swapchain will fail to create a swapchain on in-use displays. This extension and its platform-specific children will allow applications to grab in-use displays away from window systems and/or native platforms, allowing them to be used with VK_KHR_display_swapchain.

2) Are separate calls needed to acquire displays and enable direct mode?

**RESOLVED:** No, these operations happen in one combined command. Acquiring a display puts it into direct mode.

Version History

- Revision 1, 2016-12-13 (James Jones)
  - Initial draft

**VK_EXT_discard_rectangles**

Name String

VK_EXT_discard_rectangles

Extension Type

Device extension

Registered Extension Number

100

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

Contact

- Piers Daniell pdaniell-nv
This extension provides additional orthogonally aligned “discard rectangles” specified in framebuffer-space coordinates that restrict rasterization of all points, lines and triangles.

From zero to an implementation-dependent limit (specified by `maxDiscardRectangles`) number of discard rectangles can be operational at once. When one or more discard rectangles are active, rasterized fragments can either survive if the fragment is within any of the operational discard rectangles (`VK_DISCARD_RECTANGLE_MODE_INCLUSIVE_EXT` mode) or be rejected if the fragment is within any of the operational discard rectangles (`VK_DISCARD_RECTANGLE_MODE_EXCLUSIVE_EXT` mode).

These discard rectangles operate orthogonally to the existing scissor test functionality. The discard rectangles can be different for each physical device in a device group by specifying the device mask and setting discard rectangle dynamic state.

**New Commands**

- `vkCmdSetDiscardRectangleEXT`

**New Structures**

- Extending `VkGraphicsPipelineCreateInfo`:
  - `VkPipelineDiscardRectangleStateCreateInfoEXT`

- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDeviceDiscardRectanglePropertiesEXT`

**New Enums**

- `VkDiscardRectangleModeEXT`

**New Bitmasks**

- `VkPipelineDiscardRectangleStateCreateFlagsEXT`
New Enum Constants

- VK_EXT_DISCARD_RECTANGLES_EXTENSION_NAME
- VK_EXT_DISCARD_RECTANGLES_SPEC_VERSION

Extending VkDynamicState:
- VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT

Extending VkStructureType:
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DISCARD_RECTANGLE_PROPERTIES_EXT
- VK_STRUCTURE_TYPE_PIPELINE_DISCARD_RECTANGLE_STATE_CREATE_INFO_EXT

Version History

- Revision 1, 2016-12-22 (Piers Daniell)
  - Internal revisions

VK_EXT_display_control

Name String
VK_EXT_display_control

Extension Type
Device extension

Registered Extension Number
92

Revision
1

Extension and Version Dependencies
- Requires Vulkan 1.0
- Requires VK_EXT_display_surface_counter
- Requires VK_KHR_swapchain

Contact
- James Jones @cubanismo

Other Extension Metadata

Last Modified Date
2016-12-13

IP Status
No known IP claims.
Contributors

- Pierre Boudier, NVIDIA
- James Jones, NVIDIA
- Damien Leone, NVIDIA
- Pierre-Loup Griffais, Valve
- Daniel Vetter, Intel

Description

This extension defines a set of utility functions for use with the VK_KHR_display and VK_KHR_display_swapchain extensions.

New Commands

- vkDisplayPowerControlEXT
- vkGetSwapchainCounterEXT
- vkRegisterDeviceEventEXT
- vkRegisterDisplayEventEXT

New Structures

- VkDeviceEventInfoEXT
- VkDisplayEventInfoEXT
- VkDisplayPowerInfoEXT

Extending VkSwapchainCreateInfoKHR:

- VkSwapchainCounterCreateInfoEXT

New Enums

- VkDeviceEventTypeEXT
- VkDisplayEventTypeEXT
- VkDisplayPowerStateEXT

New Enum Constants

- VK_EXT_DISPLAY_CONTROL_EXTENSION_NAME
- VK_EXT_DISPLAY_CONTROL_SPEC_VERSION

Extending VkStructureType:

- VK_STRUCTURE_TYPE_DEVICE_EVENT_INFO_EXT
- VK_STRUCTURE_TYPE_DISPLAY_EVENT_INFO_EXT
- VK_STRUCTURE_TYPE_DISPLAY_POWER_INFO_EXT
Issues

1) Should this extension add an explicit “WaitForVsync” API or a fence signaled at vsync that the application can wait on?

**RESOLVED:** A fence. A separate API could later be provided that allows exporting the fence to a native object that could be inserted into standard run loops on POSIX and Windows systems.

2) Should callbacks be added for a vsync event, or in general to monitor events in Vulkan?

**RESOLVED:** No, fences should be used. Some events are generated by interrupts which are managed in the kernel. In order to use a callback provided by the application, drivers would need to have the userspace driver spawn threads that would wait on the kernel event, and hence the callbacks could be difficult for the application to synchronize with its other work given they would arrive on a foreign thread.

3) Should vblank or scanline events be exposed?

**RESOLVED:** Vblank events. Scanline events could be added by a separate extension, but the latency of processing an interrupt and waking up a userspace event is high enough that the accuracy of a scanline event would be rather low. Further, per-scanline interrupts are not supported by all hardware.

Version History

- Revision 1, 2016-12-13 (James Jones)
  - Initial draft

**VK_EXT_display_surface_counter**

Name String  
VK_EXT_display_surface_counter

Extension Type  
Instance extension

Registered Extension Number  
91

Revision  
1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires **VK_KHR_display**
Description

This extension defines a vertical blanking period counter associated with display surfaces. It provides a mechanism to query support for such a counter from a VkSurfaceKHR object.

New Commands

- vkGetPhysicalDeviceSurfaceCapabilities2EXT

New Structures

- VkSurfaceCapabilities2EXT

New Enums

- VkSurfaceCounterFlagBitsEXT

New Bitmasks

- VkSurfaceCounterFlagsEXT

New Enum Constants

- VK_EXT_DISPLAY_SURFACE_COUNTER_EXTENSION_NAME
- VK_EXT_DISPLAY_SURFACE_COUNTER_SPEC_VERSION

Extending VkStructureType:

- VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_EXT
Version History

• Revision 1, 2016-12-13 (James Jones)
  ◦ Initial draft

VK_EXT_extended_dynamic_state

Name String

VK_EXT_extended_dynamic_state

Extension Type

Device extension

Registered Extension Number

268

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Contact

• Piers Daniell (@pdaniell-nv)

Other Extension Metadata

Last Modified Date

2019-12-09

IP Status

No known IP claims.

Contributors

• Dan Ginsburg, Valve Corporation
• Graeme Leese, Broadcom
• Hans-Kristian Arntzen, Valve Corporation
• Jan-Harald Fredriksen, Arm Limited
• Jason Ekstrand, Intel
• Jeff Bolz, NVIDIA
• Jesse Hall, Google
• Philip Rebohle, Valve Corporation
• Stuart Smith, Imagination Technologies
Tobias Hector, AMD

Description

This extension adds some more dynamic state to support applications that need to reduce the number of pipeline state objects they compile and bind.

New Commands

- vkCmdBindVertexBuffers2EXT
- vkCmdSetCullModeEXT
- vkCmdSetDepthBoundsTestEnableEXT
- vkCmdSetDepthCompareOpEXT
- vkCmdSetDepthTestEnableEXT
- vkCmdSetDepthWriteEnableEXT
- vkCmdSetFrontFaceEXT
- vkCmdSetPrimitiveTopologyEXT
- vkCmdSetScissorWithCountEXT
- vkCmdSetStencilOpEXT
- vkCmdSetStencilTestEnableEXT
- vkCmdSetViewportWithCountEXT

New Structures

- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceExtendedDynamicStateFeaturesEXT

New Enum Constants

- VK_EXT_EXTENDED_DYNAMIC_STATE_EXTENSION_NAME
- VK_EXT_EXTENDED_DYNAMIC_STATE_SPEC_VERSION

- Extending VkDynamicState:
  - VK_DYNAMIC_STATE_CULL_MODE_EXT
  - VK_DYNAMIC_STATE_DEPTH_BOUNDS_TEST_ENABLE_EXT
  - VK_DYNAMIC_STATE_DEPTH_COMPARE_OP_EXT
  - VK_DYNAMIC_STATE_DEPTH_TEST_ENABLE_EXT
  - VK_DYNAMIC_STATE_DEPTH_WRITE_ENABLE_EXT
  - VK_DYNAMIC_STATE_FRONT_FACE_EXT
  - VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT
  - VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT
• VK_DYNAMIC_STATE_STENCIL_OP_EXT
• VK_DYNAMIC_STATE_STENCIL_TEST_ENABLE_EXT
• VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT
• VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT

• Extending VkStructureType:
  • VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTENDED_DYNAMIC_STATE_FEATURES_EXT

Version History

• Revision 1, 2019-12-09 (Piers Daniell)
  • Internal revisions

VK_EXT_extended_dynamic_state2

Name String
  VK_EXT_extended_dynamic_state2

Extension Type
  Device extension

Registered Extension Number
  378

Revision
  1

Extension and Version Dependencies
  • Requires Vulkan 1.0
  • Requires VK_KHR_get_physical_device_properties2

Contact
  • Vikram Kushwaha vkushwaha-nv

Other Extension Metadata

Last Modified Date
  2021-04-12

IP Status
  No known IP claims.

Contributors
  • Vikram Kushwaha, NVIDIA
  • Piers Daniell, NVIDIA
  • Jeff Bolz, NVIDIA
Description
This extension adds some more dynamic state to support applications that need to reduce the number of pipeline state objects they compile and bind.

New Commands
- vkCmdSetDepthBiasEnableEXT
- vkCmdSetLogicOpEXT
- vkCmdSetPatchControlPointsEXT
- vkCmdSetPrimitiveRestartEnableEXT
- vkCmdSetRasterizerDiscardEnableEXT

New Structures
- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceExtendedDynamicState2FeaturesEXT

New Enum Constants
- VK_EXT_EXTENDED_DYNAMIC_STATE_2_EXTENSION_NAME
- VK_EXT_EXTENDED_DYNAMIC_STATE_2_SPEC_VERSION
- Extending VkDynamicState:
  - VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT
  - VK_DYNAMIC_STATE_LOGIC_OP_EXT
  - VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT
  - VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT
  - VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTENDED_DYNAMIC_STATE_2_FEATURES_EXT

Version History
- Revision 1, 2021-04-12 (Vikram Kushwaha)
  - Internal revisions

VK_EXT_external_memory_dma_buf
Name String
  VK_EXT_external_memory_dma_buf

Extension Type
  Device extension
Description

A `dma_buf` is a type of file descriptor, defined by the Linux kernel, that allows sharing memory across kernel device drivers and across processes. This extension enables applications to import a `dma_buf` as `VkDeviceMemory`, to export `VkDeviceMemory` as a `dma_buf`, and to create `VkBuffer` objects that can be bound to that memory.

New Enum Constants

- `VK_EXT_EXTERNAL_MEMORY_DMA_BUF_EXTENSION_NAME`
- `VK_EXT_EXTERNAL_MEMORY_DMA_BUF_SPEC_VERSION`
- Extending `VkExternalMemoryHandleTypeFlagBits`:
  - `VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT`

Issues

1) How does the application, when creating a `VkImage` that it intends to bind to `dma_buf` `VkDeviceMemory` containing an externally produced image, specify the memory layout (such as row pitch and DRM format modifier) of the `VkImage`? In other words, how does the application achieve behavior comparable to that provided by `EGL_EXT_image_dma_buf_import` and
RESOLVED: Features comparable to those in `EGL_EXT_image_dma_buf_import` and `EGL_EXT_image_dma_buf_import_modifiers` will be provided by an extension layered atop this one.

2) Without the ability to specify the memory layout of external `dma_buf` images, how is this extension useful?

RESOLVED: This extension provides exactly one new feature: the ability to import/export between `dma_buf` and `VkDeviceMemory`. This feature, together with features provided by `VK_KHR_external_memory_fd`, is sufficient to bind a `VkBuffer` to `dma_buf`.

**Version History**

- Revision 1, 2017-10-10 (Chad Versace)
  - Squashed internal revisions

**VK_EXT_external_memory_host**

**Name String**

- `VK_EXT_external_memory_host`

**Extension Type**

- Device extension

**Registered Extension Number**

- 179

**Revision**

- 1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_external_memory`

**Contact**

- Daniel Rakos @drakos-amd

**Other Extension Metadata**

**Last Modified Date**

- 2017-11-10

**IP Status**

- No known IP claims.

**Contributors**

- Jaakko Konttinen, AMD
Description

This extension enables an application to import host allocations and host mapped foreign device memory to Vulkan memory objects.

New Commands

- vkGetMemoryHostPointerPropertiesEXT

New Structures

- VkMemoryHostPointerPropertiesEXT
- Extending VkMemoryAllocateInfo:
  - VkImportMemoryHostPointerInfoEXT
- Extending VkPhysicalDeviceProperties2:
  - VkPhysicalDeviceExternalMemoryHostPropertiesEXT

New Enum Constants

- VK_EXT_EXTERNAL_MEMORY_HOST_EXTENSION_NAME
- VK_EXT_EXTERNAL_MEMORY_HOST_SPEC_VERSION
- Extending VkExternalMemoryHandleTypeFlagBits:
  - VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT
  - VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_IMPORT_MEMORY_HOST_POINTER_INFO_EXT
  - VK_STRUCTURE_TYPE_MEMORY_HOST_POINTER_PROPERTIES_EXT
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_MEMORY_HOST_PROPERTIES_EXT

Issues

1) What memory type has to be used to import host pointers?

RESOLVED: Depends on the implementation. Applications have to use the new vkGetMemoryHostPointerPropertiesEXT command to query the supported memory types for a particular host pointer. The reported memory types may include memory types that come from a
memory heap that is otherwise not usable for regular memory object allocation and thus such a heap’s size may be zero.

2) Can the application still access the contents of the host allocation after importing?

**RESOLVED:** Yes. However, usual synchronization requirements apply.

3) Can the application free the host allocation?

**RESOLVED:** No, it violates valid usage conditions. Using the memory object imported from a host allocation that is already freed thus results in undefined behavior.

4) Is `vkMapMemory` expected to return the same host address which was specified when importing it to the memory object?

**RESOLVED:** No. Implementations are allowed to return the same address but it is not required. Some implementations might return a different virtual mapping of the allocation, although the same physical pages will be used.

5) Is there any limitation on the alignment of the host pointer and/or size?

**RESOLVED:** Yes. Both the address and the size have to be an integer multiple of `minImportedHostPointerAlignment`. In addition, some platforms and foreign devices may have additional restrictions.

6) Can the same host allocation be imported multiple times into a given physical device?

**RESOLVED:** No, at least not guaranteed by this extension. Some platforms do not allow locking the same physical pages for device access multiple times, so attempting to do it may result in undefined behavior.

7) Does this extension support exporting the new handle type?

**RESOLVED:** No.

8) Should we include the possibility to import host mapped foreign device memory using this API?

**RESOLVED:** Yes, through a separate handle type. Implementations are still allowed to support only one of the handle types introduced by this extension by not returning import support for a particular handle type as returned in `VkExternalMemoryPropertiesKHR`.

**Version History**

- Revision 1, 2017-11-10 (Daniel Rakos)
  - Internal revisions

**VK_EXT_filter_cubic**

**Name String**

`VK_EXT_filter_cubic`
Extension Type

Device extension

Registered Extension Number

171

Revision

3

Extension and Version Dependencies

• Requires Vulkan 1.0

Contact

• Bill Licea-Kane

Other Extension Metadata

Last Modified Date

2019-12-13

Contributors

• Bill Licea-Kane, Qualcomm Technologies, Inc.
• Andrew Garrard, Samsung
• Daniel Koch, NVIDIA
• Donald Scorgie, Imagination Technologies
• Graeme Leese, Broadcom
• Jan-Herald Fredericksen, ARM
• Jeff Leger, Qualcomm Technologies, Inc.
• Tobias Hector, AMD
• Tom Olson, ARM
• Stuart Smith, Imagination Technologies

Description

VK_EXT_filter_cubic extends VK_IMG_filter_cubic.

It documents cubic filtering of other image view types. It adds new structures that can be added to the pNext chain of VkPhysicalDeviceImageFormatInfo2 and VkImageFormatProperties2 that can be used to determine which image types and which image view types support cubic filtering.

New Structures

• Extending VkImageFormatProperties2:
  ○ VkFilterCubicImageViewImageFormatPropertiesEXT
• Extending `VkPhysicalDeviceImageFormatInfo2`:
  ◦ `VkPhysicalDeviceImageViewImageFormatInfoEXT`

**New Enum Constants**

• `VK_EXT_FILTER_CUBIC_EXTENSION_NAME`
• `VK_EXT_FILTER_CUBIC_SPEC_VERSION`

Extending `VkFilter`:

  ◦ `VK_FILTER_CUBIC_EXT`

Extending `VkFormatFeatureFlagBits`:

  ◦ `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`

Extending `VkStructureType`:

  ◦ `VK_STRUCTURE_TYPE_FILTER_CUBIC_IMAGE_VIEW_IMAGE_FORMAT_PROPERTIES_EXT`
  ◦ `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_VIEW_IMAGE_FORMAT_INFO_EXT`

**Version History**

• Revision 3, 2019-12-13 (wwlk)
  ◦ Delete requirement to cubic filter the formats USCALED_PACKED32, SSCALED_PACKED32, UINT_PACK32, and SINT_PACK32 (cut/paste error)

• Revision 2, 2019-06-05 (wwlk)
  ◦ Clarify 1D optional

• Revision 1, 2019-01-24 (wwlk)
  ◦ Initial version

**VK_EXT_fragment_shader_interlock**

**Name String**

  `VK_EXT_fragment_shader_interlock`

**Extension Type**

  Device extension

**Registered Extension Number**

  252

**Revision**

  1

**Extension and Version Dependencies**

• Requires Vulkan 1.0

• Requires `VK_KHR_get_physical_device_properties2`
Other Extension Metadata

Last Modified Date
2019-05-02

Interactions and External Dependencies
• This extension requires SPV_EXT_fragment_shader_interlock
• This extension provides API support for GL_ARB_fragment_shader_interlock

Contributors
• Daniel Koch, NVIDIA
• Graeme Leese, Broadcom
• Jan-Harald Fredriksen, Arm
• Jason Ekstrand, Intel
• Jeff Bolz, NVIDIA
• Ruihao Zhang, Qualcomm
• Slawomir Grajewski, Intel
• Spencer Fricke, Samsung

Description
This extension adds support for the FragmentShaderPixelInterlockEXT, FragmentShaderSampleInterlockEXT, and FragmentShaderShadingRateInterlockEXT capabilities from the SPV_EXT_fragment_shader_interlock extension to Vulkan.

Enabling these capabilities provides a critical section for fragment shaders to avoid overlapping pixels being processed at the same time, and certain guarantees about the ordering of fragment shader invocations of fragments of overlapping pixels.

This extension can be useful for algorithms that need to access per-pixel data structures via shader loads and stores. Algorithms using this extension can access per-pixel data structures in critical sections without other invocations accessing the same per-pixel data. Additionally, the ordering guarantees are useful for cases where the API ordering of fragments is meaningful. For example, applications may be able to execute programmable blending operations in the fragment shader, where the destination buffer is read via image loads and the final value is written via image stores.

New Structures
• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT
New Enum Constants

- VK_EXT_FRAGMENT_SHADER_INTERLOCK_EXTENSION_NAME
- VK_EXT_FRAGMENT_SHADER_INTERLOCK_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADER_INTERLOCK_FEATURES_EXT

New SPIR-V Capabilities

- FragmentShaderInterlockEXT
- FragmentShaderPixelInterlockEXT
- FragmentShaderShadingRateInterlockEXT

Version History

- Revision 1, 2019-05-24 (Piers Daniell)
  - Internal revisions

VK_EXT_global_priority

Name String

VK_EXT_global_priority

Extension Type

Device extension

Registered Extension Number

175

Revision

2

Extension and Version Dependencies

- Requires Vulkan 1.0

Contact

- Andres Rodriguez

Other Extension Metadata

Last Modified Date

2017-10-06

IP Status

No known IP claims.
Contributors

- Andres Rodriguez, Valve
- Pierre-Loup Griffais, Valve
- Dan Ginsburg, Valve
- Mitch Singer, AMD

Description

In Vulkan, users can specify device-scope queue priorities. In some cases it may be useful to extend this concept to a system-wide scope. This extension provides a mechanism for callers to set their system-wide priority. The default queue priority is `VK_QUEUE_GLOBAL_PRIORITY_MEDIUM_EXT`.

The driver implementation will attempt to skew hardware resource allocation in favour of the higher-priority task. Therefore, higher-priority work may retain similar latency and throughput characteristics even if the system is congested with lower priority work.

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

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

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

New Structures

- Extending `VkDeviceQueueCreateInfo`:
  - `VkDeviceQueueGlobalPriorityCreateInfoEXT`

New Enums

- `VkQueueGlobalPriorityEXT`

New Enum Constants

- `VK_EXT_GLOBAL_PRIORITY_EXTENSION_NAME`
- `VK_EXT_GLOBAL_PRIORITY_SPEC_VERSION`

Extending `VkResult`:

- `VK_ERROR_NOT_PERMITTED_EXT`

Extending `VkStructureType`:

- `VK_STRUCTURE_TYPE_DEVICE_QUEUE_GLOBAL_PRIORITY_CREATE_INFO_EXT`
**Version History**

- Revision 2, 2017-11-03 (Andres Rodriguez)
  - Fixed VkQueueGlobalPriorityEXT missing _EXT suffix
- Revision 1, 2017-10-06 (Andres Rodriguez)
  - First version.

**VK_EXT_hdr_metadata**

**Name String**

VK_EXT_hdr_metadata

**Extension Type**

Device extension

**Registered Extension Number**

106

**Revision**

2

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires VK_KHR_swapchain

**Contact**

- Courtney Goeltzenleuchter @courtney-g

**Other Extension Metadata**

**Last Modified Date**

2018-12-19

**IP Status**

No known IP claims.

**Contributors**

- Courtney Goeltzenleuchter, Google

**Description**

This extension defines two new structures and a function to assign SMPTE (the Society of Motion Picture and Television Engineers) 2086 metadata and CTA (Consumer Technology Association) 861.3 metadata to a swapchain. The metadata includes the color primaries, white point, and luminance range of the reference monitor, which all together define the color volume containing all the possible colors the reference monitor can produce. The reference monitor is the display where creative work is done and creative intent is established. To preserve such creative intent as much
as possible and achieve consistent color reproduction on different viewing displays, it is useful for the display pipeline to know the color volume of the original reference monitor where content was created or tuned. This avoids performing unnecessary mapping of colors that are not displayable on the original reference monitor. The metadata also includes the \texttt{maxContentLightLevel} and \texttt{maxFrameAverageLightLevel} as defined by CTA 861.3.

While the general purpose of the metadata is to assist in the transformation between different color volumes of different displays and help achieve better color reproduction, it is not in the scope of this extension to define how exactly the metadata should be used in such a process. It is up to the implementation to determine how to make use of the metadata.

**New Commands**

- \texttt{vkSetHdrMetadataEXT}

**New Structures**

- \texttt{VkHdrMetadataEXT}
- \texttt{VkXYColorEXT}

**New Enum Constants**

- \texttt{VK_EXT_HDR_METADATA_EXTENSION_NAME}
- \texttt{VK_EXT_HDR_METADATA_SPEC_VERSION}
- Extending \texttt{VkStructureType}:
  - \texttt{VK_STRUCTURE_TYPE_HDR_METADATA_EXT}

**Issues**

1) Do we need a query function?

**PROPOSED:** No, Vulkan does not provide queries for state that the application can track on its own.

2) Should we specify default if not specified by the application?

**PROPOSED:** No, that leaves the default up to the display.

**Version History**

- Revision 1, 2016-12-27 (Courtney Goeltzenleuchter)
  - Initial version
- Revision 2, 2018-12-19 (Courtney Goeltzenleuchter)
  - Correct implicit validity for \texttt{VkHdrMetadataEXT} structure
VK_EXT_headless_surface

Name String
VK_EXT_headless_surface

Extension Type
Instance extension

Registered Extension Number
257

Revision
1

Extension and Version Dependencies
- Requires Vulkan 1.0
- Requires VK_KHR_surface

Contact
- Lisa Wu 📱 chengtianww

Other Extension Metadata

Last Modified Date
2019-03-21

IP Status
No known IP claims.

Contributors
- Ray Smith, Arm

Description

The VK_EXT_headless_surface extension is an instance extension. It provides a mechanism to create VkSurfaceKHR objects independently of any window system or display device. The presentation operation for a swapchain created from a headless surface is by default a no-op, resulting in no externally-visible result.

Because there is no real presentation target, future extensions can layer on top of the headless surface to introduce arbitrary or customisable sets of restrictions or features. These could include features like saving to a file or restrictions to emulate a particular presentation target.

This functionality is expected to be useful for application and driver development because it allows any platform to expose an arbitrary or customisable set of restrictions and features of a presentation engine. This makes it a useful portable test target for applications targeting a wide range of presentation engines where the actual target presentation engines might be scarce, unavailable or otherwise undesirable or inconvenient to use for general Vulkan application...
development.

**New Commands**
- `vkCreateHeadlessSurfaceEXT`

**New Structures**
- `VkHeadlessSurfaceCreateInfoEXT`

**New Bitmasks**
- `VkHeadlessSurfaceCreateFlagsEXT`

**New Enum Constants**
- `VK_EXT_HEADLESS_SURFACE_EXTENSION_NAME`
- `VK_EXT_HEADLESS_SURFACE_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_HEADLESS_SURFACE_CREATE_INFO_EXT`

**Version History**
- Revision 1, 2019-03-21 (Ray Smith)
  - Initial draft

**VK_EXT_image_drm_format_modifier**

**Name String**
- `VK_EXT_image_drm_format_modifier`

**Extension Type**
- Device extension

**Registered Extension Number**
- 159

**Revision**
- 2

**Extension and Version Dependencies**
- Requires Vulkan 1.0
- Requires `.VK_KHR_bind_memory2`
- Requires `VK_KHR_get_physical_device_properties2`
- Requires `VK_KHR_image_format_list`
This extension provides the ability to use DRM format modifiers with images, enabling Vulkan to better integrate with the Linux ecosystem of graphics, video, and display APIs.

Its functionality closely overlaps with EGL_EXT_image_dma_buf_import_modifiers\(^2\) and EGL_MESA_image_dma_buf_export\(^3\). Unlike the EGL extensions, this extension does not require the use of a specific handle type (such as a dma_buf) for external memory and provides more explicit control of image creation.

**Introduction to DRM Format Modifiers**

A DRM format modifier is a 64-bit, vendor-prefixed, semi-opaque unsigned integer. Most modifiers represent a concrete, vendor-specific tiling format for images. Some exceptions are DRM_FORMAT_MOD_LINEAR (which is not vendor-specific); DRM_FORMAT_MOD_NONE (which is an alias of DRM_FORMAT_MOD_LINEAR due to historical accident); and DRM_FORMAT_MOD_INVALID (which does not represent a tiling format). The modifier’s vendor prefix consists of the 8 most significant bits. The canonical list of modifiers and vendor prefixes is found in drm_fourcc.h in the Linux kernel source. The other dominant source of modifiers are vendor kernel trees.

One goal of modifiers in the Linux ecosystem is to enumerate for each vendor a reasonably sized set of tiling formats that are appropriate for images shared across processes, APIs, and/or devices, where each participating component may possibly be from different vendors. A non-goal is to
enumerate all tiling formats supported by all vendors. Some tiling formats used internally by vendors are inappropriate for sharing; no modifiers should be assigned to such tiling formats.

Modifier values typically do not describe memory layouts. More precisely, a modifier's lower 56 bits usually have no structure. Instead, modifiers name memory layouts; they name a small set of vendor-preferred layouts for image sharing. As a consequence, in each vendor namespace the modifier values are often sequentially allocated starting at 1.

Each modifier is usually supported by a single vendor and its name matches the pattern {VENDOR}_FORMAT_MOD_* or DRM_FORMAT_MOD_{VENDOR}_*. Examples are I915_FORMAT_MOD_X_TILED and DRM_FORMAT_MOD_BROADCOM_VC4_T_TILED. An exception is DRM_FORMAT_MOD_LINEAR, which is supported by most vendors.

Many APIs in Linux use modifiers to negotiate and specify the memory layout of shared images. For example, a Wayland compositor and Wayland client may, by relaying modifiers over the Wayland protocol zwp_linux_dmabuf_v1, negotiate a vendor-specific tiling format for a shared wl_buffer. The client may allocate the underlying memory for the wl_buffer with GBM, providing the chosen modifier to gbm_bo_create_with_modifiers. The client may then import the wl_buffer into Vulkan for producing image content, providing the resource’s dma_buf to VkImportMemoryFdInfoKHR and its modifier to VkImageDrmFormatModifierExplicitCreateInfoEXT. The compositor may then import the wl_buffer into OpenGL for sampling, providing the resource’s dma_buf and modifier to eglCreateImage. The compositor may also bypass OpenGL and submit the wl_buffer directly to the kernel's display API, providing the dma_buf and modifier through drm_mode_fb_cmd2.

Format Translation

Modifier-capable APIs often pair modifiers with DRM formats, which are defined in drm_fourcc.h. However, VK_EXT_image_drm_format_modifier uses VkFormat instead of DRM formats. The application must convert between VkFormat and DRM format when it sends or receives a DRM format to or from an external API.

The mapping from VkFormat to DRM format is lossy. Therefore, when receiving a DRM format from an external API, often the application must use information from the external API to accurately map the DRM format to a VkFormat. For example, DRM formats do not distinguish between RGB and sRGB (as of 2018-03-28); external information is required to identify the image’s colorspace.

The mapping between VkFormat and DRM format is also incomplete. For some DRM formats there exist no corresponding Vulkan format, and for some Vulkan formats there exist no corresponding DRM format.

Usage Patterns

Three primary usage patterns are intended for this extension:

- **Negotiation.** The application negotiates with modifier-aware, external components to determine sets of image creation parameters supported among all components.

  In the Linux ecosystem, the negotiation usually assumes the image is a 2D, single-sampled, non-mipmapped, non-array image; this extension permits that assumption but does not require it.
The result of the negotiation usually resembles a set of tuples such as \( (\text{drmFormat, drmFormatModifier}) \), where each participating component supports all tuples in the set.

Many details of this negotiation—such as the protocol used during negotiation, the set of image creation parameters expressable in the protocol, and how the protocol chooses which process and which API will create the image—are outside the scope of this specification.

In this extension, `vkGetPhysicalDeviceFormatProperties2` with `VkDrmFormatModifierPropertiesListEXT` serves a primary role during the negotiation, and `vkGetPhysicalDeviceImageFormatProperties2` with `VkPhysicalDeviceImageDrmFormatModifierInfoEXT` serves a secondary role.

- **Import.** The application imports an image with a modifier.

  In this pattern, the application receives from an external source the image's memory and its creation parameters, which are often the result of the negotiation described above. Some image creation parameters are implicitly defined by the external source; for example, `VK_IMAGE_TYPE_2D` is often assumed. Some image creation parameters are usually explicit, such as the image’s `format, drmFormatModifier, and extent`; and each plane’s `offset` and `rowPitch`.

  Before creating the image, the application first verifies that the physical device supports the received creation parameters by querying `vkGetPhysicalDeviceFormatProperties2` with `VkDrmFormatModifierPropertiesListEXT` and `vkGetPhysicalDeviceImageFormatProperties2` with `VkPhysicalDeviceImageDrmFormatModifierInfoEXT`. Then the application creates the image by chaining `VkImageDrmFormatModifierExplicitCreateInfoEXT` and `VkExternalMemoryImageCreateInfo` onto `VkImageCreateInfo`.

- **Export.** The application creates an image and allocates its memory. Then the application exports to modifier-aware consumers the image's memory handles; its creation parameters; its `modifier`; and the `offset, size, and rowPitch` of each memory plane.

  In this pattern, the Vulkan device is the authority for the image; it is the allocator of the image's memory and the decider of the image's creation parameters. When choosing the image's creation parameters, the application usually chooses a tuple \( (\text{format, drmFormatModifier}) \) from the result of the negotiation described above. The negotiation's result often contains multiple tuples that share the same format but differ in their `modifier`. In this case, the application should defer the choice of the image's `modifier` to the Vulkan implementation by providing all such `modifiers` to `VkImageDrmFormatModifierListCreateInfoEXT::pDrmFormatModifiers`; and the implementation should choose from `pDrmFormatModifiers` the optimal `modifier` in consideration with the other image parameters.

  The application creates the image by chaining `VkImageDrmFormatModifierListCreateInfoEXT` and `VkExternalMemoryImageCreateInfo` onto `VkImageCreateInfo`. The protocol and APIs by which the application will share the image with external consumers will likely determine the value of `VkExternalMemoryImageCreateInfo::handleTypes`. The implementation chooses for the image an optimal `modifier` from `VkImageDrmFormatModifierListCreateInfoEXT::pDrmFormatModifiers`. The application then queries the implementation-chosen `modifier` with `vkGetImageDrmFormatModifierPropertiesEXT`, and queries the memory layout of each plane with `vkGetImageSubresourceLayout`.
The application then allocates the image's memory with `VkMemoryAllocateInfo`, adding chained extending structures for external memory; binds it to the image; and exports the memory, for example, with `vkGetMemoryFdKHR`.

Finally, the application sends the image's creation parameters, its modifier, its per-plane memory layout, and the exported memory handle to the external consumers. The details of how the application transmits this information to external consumers is outside the scope of this specification.

**Prior Art**

Extension `EGL_EXT_image_dma_buf_import` introduced the ability to create an `EGLImage` by importing for each plane a dma_buf, offset, and row pitch.

Later, extension `EGL_EXT_image_dma_buf_import_modifiers` introduced the ability to query which combination of formats and modifiers the implementation supports and to specify modifiers during creation of the `EGLImage`.

Extension `EGL_MESA_image_dma_buf_export` is the inverse of `EGL_EXT_image_dma_buf_import_modifiers`.

The Linux kernel modesetting API (KMS), when configuring the display's framebuffer with `struct drm_mode_fb_cmd2`, allows one to specify the framebuffer's modifier as well as a per-plane memory handle, offset, and row pitch.

GBM, a graphics buffer manager for Linux, allows creation of a `gbm_bo` (that is, a graphics buffer object) by importing data similar to that in `EGL_EXT_image_dma_buf_import_modifiers`; and symmetrically allows exporting the same data from the `gbm_bo`. See the references to modifier and plane in `gbm.h`.

**New Commands**

- `vkGetImageDrmFormatModifierPropertiesEXT`

**New Structures**

- `VkDrmFormatModifierPropertiesEXT`
- `VkImageDrmFormatModifierPropertiesEXT`
- Extending `VkFormatProperties2`:
  - `VkDrmFormatModifierPropertiesListEXT`
- Extending `VkImageCreateInfo`:
  - `VkImageDrmFormatModifierExplicitCreateInfoEXT`
  - `VkImageDrmFormatModifierListCreateInfoEXT`
- Extending `VkPhysicalDeviceImageFormatInfo2`:
  - `VkPhysicalDeviceImageDrmFormatModifierInfoEXT`
New Enum Constants

- VK_EXT_IMAGE_DRM_FORMAT_MODIFIER_EXTENSION_NAME
- VK_EXT_IMAGE_DRM_FORMAT_MODIFIER_SPEC_VERSION

Extending VkImageAspectFlagBits:

- VK_IMAGE_ASPECT_MEMORY_PLANE_0_BIT_EXT
- VK_IMAGE_ASPECT_MEMORY_PLANE_1_BIT_EXT
- VK_IMAGE_ASPECT_MEMORY_PLANE_2_BIT_EXT
- VK_IMAGE_ASPECT_MEMORY_PLANE_3_BIT_EXT

Extending VkImageTiling:

- VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT

Extending VkResult:

- VK_ERROR_INVALID_DRM_FORMAT_MODIFIER_PLANE_LAYOUT_EXT

Extending VkStructureType:

- VK_STRUCTURE_TYPE_DRM_FORMAT_MODIFIER_PROPERTIES_LIST_EXT
- VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_EXPLICIT_CREATE_INFO_EXT
- VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_LIST_CREATE_INFO_EXT
- VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_PROPERTIES_EXT
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_DRM_FORMAT_MODIFIER_INFO_EXT

Issues

1) Should this extension define a single DRM format modifier per VkImage? Or define one per plane?

RESOLVED: There exists a single DRM format modifier per VkImage.

DISCUSSION: Prior art, such as EGL_EXT_image_dma_buf_import_modifiers, struct drm_mode_fb_cmd, and struct gbm_import_fd_modifier_data, allows defining one modifier per plane. However, developers of the GBM and kernel APIs concede it was a mistake. Beginning in Linux 4.10, the kernel requires that the application provide the same DRM format modifier for each plane. (See Linux commit bae781b259269590109e8a4a8227331362b88212). And GBM provides an entry point, gbm_bo_get_modifier, for querying the modifier of the image but does not provide one to query the modifier of individual planes.

2) When creating an image with VkImageDrmFormatModifierExplicitCreateInfoEXT, which is typically used when importing an image, should the application explicitly provide the size of each plane?

RESOLVED: No. The application must not provide the size. To enforce this, the API requires that
**DISCUSSION:** Prior art, such as `EGL_EXT_image_dma_buf_importModifiers`, `struct drm_mode_fb_cmd2`, and `struct gbm_import_fd_modifier_data`, omits from the API the size of each plane. Instead, the APIs infer each plane's size from the import parameters, which include the image's pixel format and a `dma_buf`, offset, and row pitch for each plane.

However, Vulkan differs from EGL and GBM with regards to image creation in the following ways:

**Differences in Image Creation**

- **Undedicated allocation by default.** When importing or exporting a set of `dma_buvs` as an `EGLImage` or `gbm_bo`, common practice mandates that each `dma_buf`'s memory be dedicated (in the sense of `VK_KHR_dedicated_allocation`) to the image (though not necessarily dedicated to a single plane). In particular, neither the GBM documentation nor the EGL extension specifications explicitly state this requirement, but in light of common practice this is likely due to under-specification rather than intentional omission. In contrast, `VK_EXT_image_drm_format_modifier` permits, but does not require, the implementation to require dedicated allocations for images created with `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`.

- **Separation of image creation and memory allocation.** When importing a set of `dma_buvs` as an `EGLImage` or `gbm_bo`, EGL and GBM create the image resource and bind it to memory (the `dma_buvs`) simultaneously. This allows EGL and GBM to query each `dma_buf`'s size during image creation. In Vulkan, image creation and memory allocation are independent unless a dedicated allocation is used (as in `VK_KHR_dedicated_allocation`). Therefore, without requiring dedicated allocation, Vulkan cannot query the size of each `dma_buf` (or other external handle) when calculating the image's memory layout. Even if dedication allocation were required, Vulkan cannot calculate the image's memory layout until after the image is bound to its `dma_buvs`.

The above differences complicate the potential inference of plane size in Vulkan. Consider the following problematic cases:

**Problematic Plane Size Calculations**

- **Padding.** Some plane of the image may require implementation-dependent padding.

- **Metadata.** For some modifiers, the image may have a metadata plane which requires a non-trivial calculation to determine its size.

- **Mipmapped, array, and 3D images.** The implementation may support `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT` for images whose `mplevels`, `arrayLayers`, or `depth` is greater than 1. For such images with certain modifiers, the calculation of each plane's size may be non-trivial.

However, an application-provided plane size solves none of the above problems.

For simplicity, consider an external image with a single memory plane. The implementation is obviously capable calculating the image's size when its tiling is `VK_IMAGE_TILING_OPTIMAL`. Likewise, any reasonable implementation is capable of calculating the image's size when its tiling uses a supported modifier.

Suppose that the external image's size is smaller than the implementation-calculated size. If the
application provided the external image’s size to \texttt{vkCreateImage}, the implementation would observe the mismatched size and recognize its inability to comprehend the external image’s layout (unless the implementation used the application-provided size to select a refinement of the tiling layout indicated by the \textit{modifier}, which is strongly discouraged). The implementation would observe the conflict, and reject image creation with \texttt{VK\_ERROR\_INVALID\_DRM\_FORMAT\_MODIFIER\_PLANE\_LAYOUT\_EXT}. On the other hand, if the application did not provide the external image’s size to \texttt{vkCreateImage}, then the application would observe after calling \texttt{vkGetImageMemoryRequirements} that the external image’s size is less than the size required by the implementation. The application would observe the conflict and refuse to bind the \texttt{VkImage} to the external memory. In both cases, the result is explicit failure.

Suppose that the external image’s size is larger than the implementation-calculated size. If the application provided the external image’s size to \texttt{vkCreateImage}, for reasons similar to above the implementation would observe the mismatched size and recognize its inability to comprehend the image data residing in the extra size. The implementation, however, must assume that image data resides in the entire size provided by the application. The implementation would observe the conflict and reject image creation with \texttt{VK\_ERROR\_INVALID\_DRM\_FORMAT\_MODIFIER\_PLANE\_LAYOUT\_EXT}. On the other hand, if the application did not provide the external image’s size to \texttt{vkCreateImage}, then the application would observe after calling \texttt{vkGetImageMemoryRequirements} that the external image’s size is larger than the implementation-usable size. The application would observe the conflict and refuse to bind the \texttt{VkImage} to the external memory. In both cases, the result is explicit failure.

Therefore, an application-provided size provides no benefit, and this extension should not require it. This decision renders \texttt{VkSubresourceLayout::size} an unused field during image creation, and thus introduces a risk that implementations may require applications to submit sideband creation parameters in the unused field. To prevent implementations from relying on sideband data, this extension requires the application to set \texttt{size} to 0.

**References**

1. \texttt{EGL\_EXT\_image\_dma\_buf\_import}
2. \texttt{EGL\_EXT\_image\_dma\_buf\_import\_modifiers}
3. \texttt{EGL\_MESA\_image\_dma\_buf\_export}
4. \texttt{struct\_drm\_mode\_fb\_cmd2}
5. \texttt{gbm.h}

**Version History**

- Revision 1, 2018-08-29 (Chad Versace)
  - First stable revision
- Revision 2, 2021-09-30 (Jon Leech)
  - Add interaction with \texttt{VK\_KHR\_format\_feature\_flags2} to \texttt{vk.xml}
VK_EXT_image_robustness

Name String
   VK_EXT_image_robustness

Extension Type
   Device extension

Registered Extension Number
   336

Revision
   1

Extension and Version Dependencies
   • Requires Vulkan 1.0
   • Requires VK_KHR_get_physical_device_properties2

Contact
   • Graeme Leese ᴅgnl21

Other Extension Metadata

Last Modified Date
   2020-04-27

IP Status
   No known IP claims.

Contributors
   • Graeme Leese, Broadcom
   • Jan-Harald Fredriksen, ARM
   • Jeff Bolz, NVIDIA
   • Spencer Fricke, Samsung
   • Courtney Goeltzenleuchter, Google
   • Slawomir Cygan, Intel

Description

This extension adds stricter requirements for how out of bounds reads from images are handled. Rather than returning undefined values, most out of bounds reads return R, G, and B values of zero and alpha values of either zero or one. Components not present in the image format may be set to zero or to values based on the format as described in Conversion to RGBA.
New Structures

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceImageRobustnessFeaturesEXT`

New Enum Constants

- `VK_EXT_IMAGE_ROBUSTNESS_EXTENSION_NAME`
- `VK_EXT_IMAGE_ROBUSTNESS_SPEC_VERSION`

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICALDEVICEIMAGEROBUSTNESSFEATURES_EXT`

Issues

1. How does this extension differ from VK_EXT_robustness2?

The guarantees provided by this extension are a subset of those provided by the robustImageAccess2 feature of VK_EXT_robustness2. Where this extension allows return values of (0, 0, 0, 0) or (0, 0, 0, 1), robustImageAccess2 requires that a particular value dependent on the image format be returned. This extension provides no guarantees about the values returned for an access to an invalid Lod.

Examples

None.

Version History

- Revision 1, 2020-04-27 (Graeme Leese)
  - Initial draft

**VK_EXT_index_type_uint8**

Name String

- `VK_EXT_index_type_uint8`

Extension Type

- Device extension

Registered Extension Number

- 266

Revision

- 1

Extension and Version Dependencies

- Requires Vulkan 1.0
This extension allows `uint8_t` indices to be used with `vkCmdBindIndexBuffer`.

### New Structures

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceIndexTypeUint8FeaturesEXT`

### New Enum Constants

- `VK_EXT_INDEX_TYPE_UINT8_EXTENSION_NAME`
- `VK_EXT_INDEX_TYPE_UINT8_SPEC_VERSION`

- Extending `VkIndexType`:
  - `VK_INDEX_TYPE_UINT8_EXT`

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INDEX_TYPE_UINT8_FEATURES_EXT`

### Version History

- Revision 1, 2019-05-02 (Piers Daniell)
  - Internal revisions
Description
This extension adds some line rasterization features that are commonly used in CAD applications and supported in other APIs like OpenGL. Bresenham-style line rasterization is supported, smooth rectangular lines (coverage to alpha) are supported, and stippled lines are supported for all three line rasterization modes.

New Commands
• vkCmdSetLineStippleEXT

New Structures
• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceLineRasterizationFeaturesEXT
• ExtendingVkPhysicalDeviceProperties2:
  ◦ VkPhysicalDeviceLineRasterizationPropertiesEXT
• Extending VkPipelineRasterizationStateCreateInfo:
  ◦ VkPipelineRasterizationLineStateCreateInfoEXT

New Enums

• VkLineRasterizationModeEXT

New Enum Constants

• VK_EXT_LINE_RASTERIZATION_EXTENSION_NAME
• VK_EXT_LINE_RASTERIZATION_SPEC_VERSION
• Extending VkDynamicState:
  ◦ VK_DYNAMIC_STATE_LINE_STIPPLE_EXT
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_LINE_RASTERIZATION_FEATURES_EXT
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_LINE_RASTERIZATION_PROPERTIES_EXT
  ◦ VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_LINE_STATE_CREATE_INFO_EXT

Issues

(1) Do we need to support Bresenham-style and smooth lines with more than one rasterization sample? i.e. the equivalent of glEnable(GL_MULTISAMPLE) in OpenGL when the framebuffer has more than one sample?

RESOLVED: Yes.
For simplicity, Bresenham line rasterization carries forward a few restrictions from OpenGL, such as not supporting per-sample shading, alpha to coverage, or alpha to one.

Version History

• Revision 1, 2019-05-09 (Jeff Bolz)
  ◦ Initial draft

VK_EXT_memory_budget

Name String

VK_EXT_memory_budget

Extension Type

Device extension
Description

While running a Vulkan application, other processes on the machine might also be attempting to use the same device memory, which can pose problems. This extension adds support for querying the amount of memory used and the total memory budget for a memory heap. The values returned by this query are implementation-dependent and can depend on a variety of factors including operating system and system load.

The `VkPhysicalDeviceMemoryBudgetPropertiesEXT::heapBudget` values can be used as a guideline for how much total memory from each heap the current process can use at any given time, before allocations may start failing or causing performance degradation. The values may change based on other activity in the system that is outside the scope and control of the Vulkan implementation.

The `VkPhysicalDeviceMemoryBudgetPropertiesEXT::heapUsage` will display the current process estimated heap usage.

With this information, the idea is for an application at some interval (once per frame, per few seconds, etc) to query `heapBudget` and `heapUsage`. From here the application can notice if it is over budget and decide how it wants to handle the memory situation (free it, move to host memory, changing mipmap levels, etc). This extension is designed to be used in concert with `VK_EXT_memory_priority` to help with this part of memory management.

New Structures

- Extending `VkPhysicalDeviceMemoryProperties2`:
  - `VkPhysicalDeviceMemoryBudgetPropertiesEXT`
New Enum Constants

• VK_EXT_MEMORY_BUDGET_EXTENSION_NAME
• VK_EXT_MEMORY_BUDGET_SPEC_VERSION
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_BUDGET_PROPERTIES_EXT

Version History

• Revision 1, 2018-10-08 (Jeff Bolz)
  ◦ Initial revision

VK_EXT_pci_bus_info

Name String

VK_EXT_pci_bus_info

Extension Type

Device extension

Registered Extension Number

213

Revision

2

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Contact

• Matthaeus G. Chajdas anteru

Other Extension Metadata

Last Modified Date

2018-12-10

IP Status

No known IP claims.

Contributors

• Matthaeus G. Chajdas, AMD
• Daniel Rakos, AMD
Description

This extension adds a new query to obtain PCI bus information about a physical device.

Not all physical devices have PCI bus information, either due to the device not being connected to the system through a PCI interface or due to platform specific restrictions and policies. Thus this extension is only expected to be supported by physical devices which can provide the information.

As a consequence, applications should always check for the presence of the extension string for each individual physical device for which they intend to issue the new query for and should not have any assumptions about the availability of the extension on any given platform.

New Structures

- Extending VkPhysicalDeviceProperties2:
  - VkPhysicalDevicePCIBusInfoPropertiesEXT

New Enum Constants

- VK_EXT_PCI_BUS_INFO_EXTENSION_NAME
- VK_EXT_PCI_BUS_INFO_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PCI_BUS_INFO_PROPERTIES_EXT

Version History

- Revision 2, 2018-12-10 (Daniel Rakos)
  - Changed all members of the new structure to have the uint32_t type
- Revision 1, 2018-10-11 (Daniel Rakos)
  - Initial revision

VK_EXT_post_depth_coverage

Name String

VK_EXT_post_depth_coverage

Extension Type

Device extension

Registered Extension Number

156

Revision

1
Extension and Version Dependencies

- Requires Vulkan 1.0

Contact

- Daniel Koch @dgkoch

Other Extension Metadata

Last Modified Date

2017-07-17

Interactions and External Dependencies

- This extension requires SPV_KHR_post_depth_coverage
- This extension provides API support for GL_ARB_post_depth_coverage and GL_EXT_post_depth_coverage

Contributors

- Jeff Bolz, NVIDIA

Description

This extension adds support for the following SPIR-V extension in Vulkan:

- SPV_KHR_post_depth_coverage

which allows the fragment shader to control whether values in the SampleMask built-in input variable reflect the coverage after early depth and stencil tests are applied.

This extension adds a new PostDepthCoverage execution mode under the SampleMaskPostDepthCoverage capability. When this mode is specified along with EarlyFragmentTests, the value of an input variable decorated with the SampleMask built-in reflects the coverage after the early fragment tests are applied. Otherwise, it reflects the coverage before the depth and stencil tests.

When using GLSL source-based shading languages, the post_depth_coverage layout qualifier from GL_ARB_post_depth_coverage or GL_EXT_post_depth_coverage maps to the PostDepthCoverage execution mode.

New Enum Constants

- VK_EXT_POST_DEPTH_COVERAGE_EXTENSION_NAME
- VK_EXT_POST_DEPTH_COVERAGE_SPEC_VERSION

New SPIR-V Capabilities

- SampleMaskPostDepthCoverage
Version History

- Revision 1, 2017-07-17 (Daniel Koch)
  - Internal revisions

VK_EXT_queue_family_foreign

Name String
VK_EXT_queue_family_foreign

Extension Type
Device extension

Registered Extension Number
127

Revision
1

Extension and Version Dependencies
- Requires Vulkan 1.0
- Requires VK_KHR_external_memory

Contact
- Chad Versace [chadversary](mailto:chadversary)

Other Extension Metadata

Last Modified Date
2017-11-01

IP Status
No known IP claims.

Contributors
- Chad Versace, Google
- James Jones, NVIDIA
- Jason Ekstrand, Intel
- Jesse Hall, Google
- Daniel Rakos, AMD
- Ray Smith, ARM

Description
This extension defines a special queue family, VK_QUEUE_FAMILY_FOREIGN_EXT, which can be used to transfer ownership of resources backed by external memory to foreign, external queues. This is
similar to VK_QUEUE_FAMILY_EXTERNAL_KHR, defined in VK_KHR_external_memory. The key differences between the two are:

- The queues represented by VK_QUEUE_FAMILY_EXTERNAL_KHR must share the same physical device and the same driver version as the current VkInstance. VK_QUEUE_FAMILY_FOREIGN_EXT has no such restrictions. It can represent devices and drivers from other vendors, and can even represent non-Vulkan-capable devices.
- All resources backed by external memory support VK_QUEUE_FAMILY_EXTERNAL_KHR. Support for VK_QUEUE_FAMILY_FOREIGN_EXT is more restrictive.
- Applications should expect transitions to/from VK_QUEUE_FAMILY_FOREIGN_EXT to be more expensive than transitions to/from VK_QUEUE_FAMILY_EXTERNAL_KHR.

New Enum Constants

- VK_EXT_QUEUE_FAMILY_FOREIGN_EXTENSION_NAME
- VK_EXT_QUEUE_FAMILY_FOREIGN_SPEC_VERSION
- VK_QUEUE_FAMILY_FOREIGN_EXT

Version History

- Revision 1, 2017-11-01 (Chad Versace)
  - Squashed internal revisions

VK_EXT_robustness2

Name String

VK_EXT_robustness2

Extension Type

Device extension

Registered Extension Number

287

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0

Contact

- Liam Middlebrook liam-middlebrook

Other Extension Metadata

Last Modified Date

2020-01-29
IP Status

No known IP claims.

Contributors

- Liam Middlebrook, NVIDIA
- Jeff Bolz, NVIDIA

Description

This extension adds stricter requirements for how out of bounds reads and writes are handled. Most accesses must be tightly bounds-checked, out of bounds writes must be discarded, out of bound reads must return zero. Rather than allowing multiple possible \((0,0,0,x)\) vectors, the out of bounds values are treated as zero, and then missing components are inserted based on the format as described in Conversion to RGBA and vertex input attribute extraction.

These additional requirements may be expensive on some implementations, and should only be enabled when truly necessary.

This extension also adds support for “null descriptors”, where \(VK_NULL_HANDLE\) can be used instead of a valid handle. Accesses to null descriptors have well-defined behavior, and do not rely on robustness.

New Structures

- Extending \(VkPhysicalDeviceFeatures2, VkDeviceCreateInfo\):
  - \(VkPhysicalDeviceRobustness2FeaturesEXT\)
- Extending \(VkPhysicalDeviceProperties2\):
  - \(VkPhysicalDeviceRobustness2PropertiesEXT\)

New Enum Constants

- \(VK_EXT_ROBUSTNESS_2_EXTENSION_NAME\)
- \(VK_EXT_ROBUSTNESS_2_SPEC_VERSION\)
- Extending \(VkStructureType\):
  - \(VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ROBUSTNESS_2_FEATURES_EXT\)
  - \(VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ROBUSTNESS_2_PROPERTIES_EXT\)

Issues

1. Why do \(VkPhysicalDeviceRobustness2PropertiesEXT::robustUniformBufferAccessSizeAlignment\) and \(VkPhysicalDeviceRobustness2PropertiesEXT::robustStorageBufferAccessSizeAlignment\) exist?

RESOLVED: Some implementations cannot efficiently tightly bounds-check all buffer accesses. Rather, the size of the bound range is padded to some power of two multiple, up to 256 bytes for uniform buffers and up to 4 bytes for storage buffers, and that padded size is bounds-checked. This is sufficient to implement D3D-like behavior, because D3D only allows binding whole uniform
buffers or ranges that are a multiple of 256 bytes, and D3D raw and structured buffers only support 32-bit accesses.

**Examples**

None.

**Version History**

- Revision 1, 2019-11-01 (Jeff Bolz, Liam Middlebrook)
  - Initial draft

**VK_EXT_sample_locations**

**Name String**

VK_EXT_sample_locations

**Extension Type**

Device extension

**Registered Extension Number**

144

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

**Contact**

- Daniel Rakos drakos-amd

**Other Extension Metadata**

**Last Modified Date**

2017-08-02

**Contributors**

- Mais Alnasser, AMD
- Matthaeus G. Chajdas, AMD
- Maciej Jesionowski, AMD
- Daniel Rakos, AMD
- Slawomir Grajewski, Intel
- Jeff Bolz, NVIDIA
Description

This extension allows an application to modify the locations of samples within a pixel used in rasterization. Additionally, it allows applications to specify different sample locations for each pixel in a group of adjacent pixels, which can increase antialiasing quality (particularly if a custom resolve shader is used that takes advantage of these different locations).

It is common for implementations to optimize the storage of depth values by storing values that can be used to reconstruct depth at each sample location, rather than storing separate depth values for each sample. For example, the depth values from a single triangle may be represented using plane equations. When the depth value for a sample is needed, it is automatically evaluated at the sample location. Modifying the sample locations causes the reconstruction to no longer evaluate the same depth values as when the samples were originally generated, thus the depth aspect of a depth/stencil attachment must be cleared before rendering to it using different sample locations.

Some implementations may need to evaluate depth image values while performing image layout transitions. To accommodate this, instances of the VkSampleLocationsInfoEXT structure can be specified for each situation where an explicit or automatic layout transition has to take place. VkSampleLocationsInfoEXT can be chained from VkImageMemoryBarrier structures to provide sample locations for layout transitions performed by vkCmdWaitEvents and vkCmdPipelineBarrier calls, and VkRenderPassSampleLocationsBeginInfoEXT can be chained from VkRenderPassBeginInfo to provide sample locations for layout transitions performed implicitly by a render pass instance.

New Commands

- vkCmdSetSampleLocationsEXT
- vkGetPhysicalDeviceMultisamplePropertiesEXT

New Structures

- VkAttachmentSampleLocationsEXT
- VkMultisamplePropertiesEXT
- VkSampleLocationEXT
- VkSubpassSampleLocationsEXT
- Extending VkImageMemoryBarrier, VkImageMemoryBarrier2KHR:
  - VkSampleLocationsInfoEXT
- Extending VkPhysicalDeviceProperties2:
  - VkPhysicalDeviceSampleLocationsPropertiesEXT
- Extending VkPipelineMultisampleStateCreateInfo:
  - VkPipelineSampleLocationsStateCreateInfoEXT
- Extending VkRenderPassBeginInfo:
New Enum Constants

- VK_EXT_SAMPLE_LOCATIONS_EXTENSION_NAME
- VK_EXT_SAMPLE_LOCATIONS_SPEC_VERSION

Extending VkDynamicState:

- VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT

Extending VkImageCreateFlagBits:

- VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT

Extending VkStructureType:

- VK_STRUCTURE_TYPE_MULTISAMPLE_PROPERTIES_EXT
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLE_LOCATIONS_PROPERTIES_EXT
- VK_STRUCTURE_TYPE_PIPELINE_SAMPLE_LOCATIONS_STATE_CREATE_INFO_EXT
- VK_STRUCTURE_TYPE_RENDER_PASS_SAMPLE_LOCATIONS_BEGIN_INFO_EXT
- VK_STRUCTURE_TYPE_SAMPLE_LOCATIONS_INFO_EXT

Version History

- Revision 1, 2017-08-02 (Daniel Rakos)
  - Internal revisions

VK_EXT_shader_atomic_float

Name String

VK_EXT_shader_atomic_float

Extension Type

Device extension

Registered Extension Number

261

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

Contact

- Vikram Kushwaha @vkushwaha-nv
Other Extension Metadata

Last Modified Date
2020-07-15

IP Status
No known IP claims.

Interactions and External Dependencies
- This extension requires `SPV_EXT_shader_atomic_float_add`
- This extension provides API support for `GL_EXT_shader_atomic_float`

Contributors
- Vikram Kushwaha, NVIDIA
- Jeff Bolz, NVIDIA

Description
This extension allows a shader to contain floating-point atomic operations on buffer, workgroup, and image memory. It also advertises the SPIR-V `AtomicFloat32AddEXT` and `AtomicFloat64AddEXT` capabilities that allows atomic addition on floating-points numbers. The supported operations include `OpAtomicFAddEXT`, `OpAtomicExchange`, `OpAtomicLoad` and `OpAtomicStore`.

New Structures
- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceShaderAtomicFloatFeaturesEXT`

New Enum Constants
- `VK_EXT_SHADER_ATOMIC_FLOAT_EXTENSION_NAME`
- `VK_EXT_SHADER_ATOMIC_FLOAT_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_FLOAT_FEATURES_EXT`

New SPIR-V Capabilities
- `AtomicFloat32AddEXT`
- `AtomicFloat64AddEXT`

Version History
- Revision 1, 2020-07-15 (Vikram Kushwaha)
  - Internal revisions
VK_EXT_shader_demote_to_helper_invocation

Name String

VK_EXT_shader_demote_to_helper_invocation

Extension Type

Device extension

Registered Extension Number

277

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Contact

• Jeff Bolz (Jeffbolznv)

Other Extension Metadata

Last Modified Date

2019-06-01

IP Status

No known IP claims.

Interactions and External Dependencies

• This extension requires SPV_EXT_demote_to_helper_invocation
• This extension provides API support for GL_EXT_demote_to_helper_invocation

Contributors

• Jeff Bolz, NVIDIA

Description

This extension adds Vulkan support for the SPV_EXT_demote_to_helper_invocation SPIR-V extension. That SPIR-V extension provides a new instruction OpDemoteToHelperInvocationEXT allowing shaders to “demote” a fragment shader invocation to behave like a helper invocation for its duration. The demoted invocation will have no further side effects and will not output to the framebuffer, but remains active and can participate in computing derivatives and in group operations. This is a better match for the “discard” instruction in HLSL.

New Structures

• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
New Enum Constants

- **VK_EXT_SHADER_DEMOTE_TO_HELPER_INVOCATION_EXTENSION_NAME**
- **VK_EXT_SHADER_DEMOTE_TO_HELPER_INVOCATION_SPEC_VERSION**

Extending `VkStructureType`:

- **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DEMOTE_TO_HELPER_INVOCATION_FEATURES_EXT**

New SPIR-V Capability

- **DemoteToHelperInvocationEXT**

Version History

- Revision 1, 2019-06-01 (Jeff Bolz)
  - Initial draft

**VK_EXT_shader_image_atomic_int64**

Name String

- `VK_EXT_shader_image_atomic_int64`

Extension Type

- Device extension

Registered Extension Number

- 235

Revision

- 1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

Contact

- Tobias Hector @tobski

Other Extension Metadata

Last Modified Date

- 2020-07-14

IP Status

- No known IP claims.
Interactions and External Dependencies

• This extension requires `SPV_EXT_shader_image_int64`
• This extension provides API support for `GLSL_EXT_shader_image_int64`

Contributors

• Matthaeus Chajdas, AMD
• Graham Wihldal, Epic Games
• Tobias Hector, AMD
• Jeff Bolz, Nvidia
• Jason Ekstrand, Intel

Description

This extension extends existing 64-bit integer atomic support to enable these operations on images as well.

When working with large 2- or 3-dimensional data sets (e.g. rasterization or screen-space effects), image accesses are generally more efficient than equivalent buffer accesses. This extension allows applications relying on 64-bit integer atomics in this manner to quickly improve performance with only relatively minor code changes.

64-bit integer atomic support is guaranteed for optimally tiled images with the `VK_FORMAT_R64_UINT` and `VK_FORMAT_R64_SINT` formats.

New Structures

• Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  ◦ `VkPhysicalDeviceShaderImageAtomicInt64FeaturesEXT`

New Enum Constants

• `VK_EXT_SHADER_IMAGE_ATOMIC_INT64_EXTENSION_NAME`
• `VK_EXT_SHADER_IMAGE_ATOMIC_INT64_SPEC_VERSION`
• Extending `VkStructureType`:
  ◦ `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_IMAGE_ATOMIC_INT64_FEATURES_EXT`

Version History

• Revision 1, 2020-07-14 (Tobias Hector)
  ◦ Initial draft

VK_EXT_shader_stencil_export

Name String

`VK_EXT_shader_stencil_export`
Extension Type
   Device extension

Registered Extension Number
   141

Revision
   1

Extension and Version Dependencies
   • Requires Vulkan 1.0

Contact
   • Dominik Witczak ♦ominatorwitzakamd

Other Extension Metadata

Last Modified Date
   2017-07-19

IP Status
   No known IP claims.

Interactions and External Dependencies
   • This extension requires SPV_EXT_shader_stencil_export
   • This extension provides API support for GL_ARB_shader_stencil_export

Contributors
   • Dominik Witczak, AMD
   • Daniel Rakos, AMD
   • Rex Xu, AMD

Description
This extension adds support for the SPIR-V extension SPV_EXT_shader_stencil_export, providing a mechanism whereby a shader may generate the stencil reference value per invocation. When stencil testing is enabled, this allows the test to be performed against the value generated in the shader.

New Enum Constants

• VK_EXT_SHADER_STENCIL_EXPORT_EXTENSION_NAME
• VK_EXT_SHADER_STENCIL_EXPORT_SPEC_VERSION

Version History
   • Revision 1, 2017-07-19 (Dominik Witczak)
VK_EXT_subgroup_size_control

Name String
VK_EXT_subgroup_size_control

Extension Type
Device extension

Registered Extension Number
226

Revision
2

Extension and Version Dependencies
• Requires Vulkan 1.1

Contact
• Neil Henning 🌐sheredom

Other Extension Metadata

Last Modified Date
2019-03-05

Contributors
• Jeff Bolz, NVIDIA
• Jason Ekstrand, Intel
• Sławek Grajewski, Intel
• Jesse Hall, Google
• Neil Henning, AMD
• Daniel Koch, NVIDIA
• Jeff Leger, Qualcomm
• Graeme Leese, Broadcom
• Allan MacKinnon, Google
• Mariusz Merecki, Intel
• Graham Wihlidal, Electronic Arts

Description
This extension enables an implementation to control the subgroup size by allowing a varying subgroup size and also specifying a required subgroup size.
It extends the subgroup support in Vulkan 1.1 to allow an implementation to expose a varying subgroup size. Previously Vulkan exposed a single subgroup size per physical device, with the expectation that implementations will behave as if all subgroups have the same size. Some implementations may dispatch shaders with a varying subgroup size for different subgroups. As a result they could implicitly split a large subgroup into smaller subgroups or represent a small subgroup as a larger subgroup, some of whose invocations were inactive on launch.

To aid developers in understanding the performance characteristics of their programs, this extension exposes a minimum and maximum subgroup size that a physical device supports and a pipeline create flag to enable that pipeline to vary its subgroup size. If enabled, any SubgroupSize decorated variables in the SPIR-V shader modules provided to pipeline creation may vary between the minimum and maximum subgroup sizes.

An implementation is also optionally allowed to support specifying a required subgroup size for a given pipeline stage. Implementations advertise which stages support a required subgroup size, and any pipeline of a supported stage can be passed a VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT structure to set the subgroup size for that shader stage of the pipeline. For compute shaders, this requires the developer to query the maxComputeWorkgroupSubgroups and ensure that:

$$s = \text{WorkGroupSize}.x \times \text{WorkGroupSize}.y \times \text{WorkGroupSize}.z \leq \text{SubgroupSize} \times \text{maxComputeWorkgroupSubgroups}$$

Developers can also specify a new pipeline shader stage create flag that requires the implementation to have fully populated subgroups within local workgroups. This requires the workgroup size in the X dimension to be a multiple of the subgroup size.

**New Structures**

- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceSubgroupSizeControlFeaturesEXT
- Extending VkPhysicalDeviceProperties2:
  - VkPhysicalDeviceSubgroupSizeControlPropertiesEXT
- Extending VkPipelineShaderStageCreateInfo:
  - VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT

**New Enum Constants**

- VK_EXT_SUBGROUP_SIZE_CONTROL_EXTENSION_NAME
- VK_EXT_SUBGROUP_SIZE_CONTROL_SPEC_VERSION
- Extending VkPipelineShaderStageCreateFlagBits:
  - VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT
  - VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_FEATURES_EXT
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_PROPERTIES_EXT
Version History

- Revision 1, 2019-03-05 (Neil Henning)
  - Initial draft
- Revision 2, 2019-07-26 (Jason Ekstrand)
  - Add the missing `VkPhysicalDeviceSubgroupSizeControlFeaturesEXT` for querying subgroup size control features.

**VK_EXT_swapchain_colorspace**

**Name String**

`VK_EXT_swapchain_colorspace`

**Extension Type**

Instance extension

**Registered Extension Number**

105

**Revision**

4

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_surface`

**Contact**

- Courtney Goeltzenleuchter

**Other Extension Metadata**

**Last Modified Date**

2019-04-26

**IP Status**

No known IP claims.

**Contributors**

- Courtney Goeltzenleuchter, Google

**Description**

To be done.
New Enum Constants

- `VK_EXT_SWAPCHAIN_COLOR_SPACE_EXTENSION_NAME`
- `VK_EXT_SWAPCHAIN_COLOR_SPACE_SPEC_VERSION`

Extending `VkColorSpaceKHR`:

- `VK_COLOR_SPACE_ADOBERGB_LINEAR_EXT`
- `VK_COLOR_SPACE_ADOBERGB_NONLINEAR_EXT`
- `VK_COLOR_SPACE_BT2020_LINEAR_EXT`
- `VK_COLOR_SPACE_BT709_LINEAR_EXT`
- `VK_COLOR_SPACE_BT709_NONLINEAR_EXT`
- `VK_COLOR_SPACE_DCI_P3_NONLINEAR_EXT`
- `VK_COLOR_SPACE_DISPLAY_P3_LINEAR_EXT`
- `VK_COLOR_SPACE_DISPLAY_P3_NONLINEAR_EXT`
- `VK_COLOR_SPACE_DOLBYVISION_EXT`
- `VK_COLOR_SPACE_EXTENDED_SRGB_LINEAR_EXT`
- `VK_COLOR_SPACE_EXTENDED_SRGB_NONLINEAR_EXT`
- `VK_COLOR_SPACE_HDR10_HLG_EXT`
- `VK_COLOR_SPACE_HDR10_ST2084_EXT`
- `VK_COLOR_SPACE_PASS_THROUGH_EXT`

Issues

1) Does the spec need to specify which kinds of image formats support the color spaces?

**RESOLVED**: Pixel format is independent of color space (though some color spaces really want / need floating point color components to be useful). Therefore, do not plan on documenting what formats support which colorspace. An application can call `vkGetPhysicalDeviceSurfaceFormatsKHR` to query what a particular implementation supports.

2) How does application determine if HW supports appropriate transfer function for a colorspace?

**RESOLVED**: Extension indicates that implementation must not do the OETF encoding if it is not sRGB. That responsibility falls to the application shaders. Any other native OETF / EOTF functions supported by an implementation can be described by separate extension.

Version History

- Revision 1, 2016-12-27 (Courtney Goeltzenleuchter)
  - Initial version
- Revision 2, 2017-01-19 (Courtney Goeltzenleuchter)
  - Add pass through and multiple options for BT2020.
Clean up some issues with equations not displaying properly.

• Revision 3, 2017-06-23 (Courtney Goeltzenleuchter)
  • Add extended sRGB non-linear enum.

• Revision 4, 2019-04-26 (Graeme Leese)
  • Clarify colorspace transfer function usage.
  • Refer to normative definitions in the Data Format Specification.
  • Clarify DCI-P3 and Display P3 usage.

**VK_EXT_texel_buffer_alignment**

Name String

- **VK_EXT_texel_buffer_alignment**

Extension Type

- Device extension

Registered Extension Number

- 282

Revision

- 1

Extension and Version Dependencies

- Requires Vulkan 1.0
  - Requires **VK_KHR_get_physical_device_properties2**

Contact

- Jeff Bolz [jeffbolznv](mailto:jeffbolznv)

Other Extension Metadata

Last Modified Date

- 2019-06-06

IP Status

- No known IP claims.

Interactions and External Dependencies

Contributors

- Jeff Bolz, NVIDIA

Description

This extension adds more expressive alignment requirements for uniform and storage texel buffers. Some implementations have single texel alignment requirements that cannot be expressed
via `VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment`.

New Structures

- Extending `VkPhysicalDeviceFeatures2, VkDeviceCreateInfo`:
  - `VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT`
- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT`

New Enum Constants

- `VK_EXT_TEXEL_BUFFER_ALIGNMENT_EXTENSION_NAME`
- `VK_EXT_TEXEL_BUFFER_ALIGNMENT_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_FEATURES_EXT`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_PROPERTIES_EXT`

Version History

- Revision 1, 2019-06-06 (Jeff Bolz)
  - Initial draft

**VK_EXT_texture_compression_astc_hdr**

Name String

`VK_EXT_texture_compression_astc_hdr`

Extension Type

Device extension

Registered Extension Number

67

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

Contact

- Jan-Harald Fredriksen 👥janharaldfredriksen-arm
Other Extension Metadata

Last Modified Date
2019-05-28

IP Status
No known issues.

Contributors
• Jan-Harald Fredriksen, Arm

Description
This extension adds support for textures compressed using the Adaptive Scalable Texture Compression (ASTC) High Dynamic Range (HDR) profile.

When this extension is enabled, the HDR profile is supported for all ASTC formats listed in ASTC Compressed Image Formats.

New Structures
• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT

New Enum Constants
• VK_EXT_TEXTURE_COMPRESSION_ASTC_HDR_EXTENSION_NAME
• VK_EXT_TEXTURE_COMPRESSION_ASTC_HDR_SPEC_VERSION
• Extending VkFormat:
  ◦ VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK_EXT
  ◦ VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK_EXT
  ◦ VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK_EXT
  ◦ VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK_EXT
  ◦ VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK_EXT
  ◦ VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK_EXT
  ◦ VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK_EXT
  ◦ VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK_EXT
  ◦ VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK_EXT
  ◦ VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK_EXT
  ◦ VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK_EXT
  ◦ VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK_EXT
  ◦ VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK_EXT
Issues

1) Should we add a feature or limit for this functionality?

Yes. It is consistent with the ASTC LDR support to add a feature like textureCompressionASTC_HDR. The feature is strictly speaking redundant as long as this is just an extension; it would be sufficient to just enable the extension. But adding the feature is more forward-looking if wanted to make this an optional core feature in the future.

2) Should we introduce new format enums for HDR?

Yes. Vulkan 1.0 describes the ASTC format enums as UNORM, e.g. VK_FORMAT_ASTC_4x4_UNORM_BLOCK, so it is confusing to make these contain HDR data. Note that the OpenGL (ES) extensions did not make this distinction because a single ASTC HDR texture may contain both unorm and float blocks. Implementations may not be able to distinguish between LDR and HDR ASTC textures internally and just treat them as the same format, i.e. if this extension is supported then sampling from a VK_FORMAT_ASTC_4x4_UNORM_BLOCK image format may return HDR results. Applications can get predictable results by using the appropriate image format.

Version History

- Revision 1, 2019-05-28 (Jan-Harald Fredriksen)
  - Initial version

VK_EXT_validation_features

Name String

VK_EXT_validation_features

Extension Type

Instance extension

Registered Extension Number

248

Revision

5

Extension and Version Dependencies

- Requires Vulkan 1.0

Special Use

- Debugging tools
Description

This extension provides the `VkValidationFeaturesEXT` struct that can be included in the `pNext` chain of the `VkInstanceCreateInfo` structure passed as the `pCreateInfo` parameter of `vkCreateInstance`. The structure contains an array of `VkValidationFeatureEnableEXT` enum values that enable specific validation features that are disabled by default. The structure also contains an array of `VkValidationFeatureDisableEXT` enum values that disable specific validation layer features that are enabled by default.

Note

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

New Structures

- Extending `VkInstanceCreateInfo`:
  - `VkValidationFeaturesEXT`

New Enums

- `VkValidationFeatureDisableEXT`
- `VkValidationFeatureEnableEXT`

New Enum Constants

- `VK_EXT_VALIDATION_FEATURES_EXTENSION_NAME`
• **VK_EXT_VALIDATION_FEATURES_SPEC_VERSION**

  Extending **VkStructureType**:
  - **VK_STRUCTURE_TYPE_VALIDATION_FEATURES_EXT**

**Version History**

- Revision 1, 2018-11-14 (Karl Schultz)
  - Initial revision
- Revision 2, 2019-08-06 (Mark Lobodzinski)
  - Add Best Practices enable
- Revision 3, 2020-03-04 (Tony Barbour)
  - Add Debug Printf enable
- Revision 4, 2020-07-29 (John Zulauf)
  - Add Synchronization Validation enable
- Revision 5, 2021-05-18 (Tony Barbour)
  - Add Shader Validation Cache disable

**VK_EXT_vertex_attribute_divisor**

**Name String**

  VK_EXT_vertex_attribute_divisor

**Extension Type**

  Device extension

**Registered Extension Number**

  191

**Revision**

  3

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires **VK_KHR_get_physical_device_properties2**

**Contact**

  - Vikram Kushwaha 🌐vkushwaha

**Other Extension Metadata**

**Last Modified Date**

  2018-08-03
IP Status
No known IP claims.

Contributors
• Vikram Kushwaha, NVIDIA
• Jason Ekstrand, Intel

Description
This extension allows instance-rate vertex attributes to be repeated for certain number of instances instead of advancing for every instance when instanced rendering is enabled.

New Structures
• VkVertexInputBindingDivisorDescriptionEXT
• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT
• Extending VkPhysicalDeviceProperties2:
  ◦ VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT
• Extending VkPipelineVertexInputStateCreateInfo:
  ◦ VkPipelineVertexInputDivisorStateCreateInfoEXT

New Enum Constants
• VK_EXT_VERTEX_ATTRIBUTE_DIVISOR_EXTENSION_NAME
• VK_EXT_VERTEX_ATTRIBUTE_DIVISOR_SPEC_VERSION
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VERTEX_ATTRIBUTE_DIVISOR_FEATURES_EXT
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VERTEX_ATTRIBUTE_DIVISOR_PROPERTIES_EXT
  ◦ VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_DIVISOR_STATE_CREATE_INFO_EXT

Issues
1) What is the effect of a non-zero value for firstInstance?

RESOLVED: The Vulkan API should follow the OpenGL convention and offset attribute fetching by firstInstance while computing vertex attribute offsets.

2) Should zero be an allowed divisor?

RESOLVED: Yes. A zero divisor means the vertex attribute is repeated for all instances.
Examples

To create a vertex binding such that the first binding uses instanced rendering and the same attribute is used for every 4 draw instances, an application could use the following set of structures:

```c
const VkVertexInputBindingDivisorDescriptionEXT divisorDesc = {
    0,
    4
};

const VkPipelineVertexInputDivisorStateCreateInfoEXT divisorInfo = {
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_DIVISOR_STATE_CREATE_INFO_EXT, // sType
    NULL, // pNext
    1, // vertexBindingDivisorCount
    &divisorDesc // pVertexBindingDivisors
};

const VkVertexInputBindingDescription binding = {
    0, // binding
    sizeof(Vertex), // stride
    VK_VERTEX_INPUT_RATE_INSTANCE // inputRate
};

const VkPipelineVertexInputStateCreateInfo viInfo = {
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_CREATE_INFO, // sType
    &divisorInfo, // pNext
    ...
};
```

Version History

- Revision 1, 2017-12-04 (Vikram Kushwaha)
  - First Version
- Revision 2, 2018-07-16 (Jason Ekstrand)
  - Adjust the interaction between divisor and firstInstance to match the OpenGL convention.
  - Disallow divisors of zero.
- Revision 3, 2018-08-03 (Vikram Kushwaha)
  - Allow a zero divisor.
Add a physical device features structure to query/enable this feature.

**VK_EXT_vertex_input_dynamic_state**

**Name String**

VK_EXT_vertex_input_dynamic_state

**Extension Type**

Device extension

**Registered Extension Number**

353

**Revision**

2

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

**Contact**

- Piers Daniell pdaniell-nv

**Other Extension Metadata**

**Last Modified Date**

2020-08-21

**IP Status**

No known IP claims.

**Contributors**

- Jeff Bolz, NVIDIA
- Spencer Fricke, Samsung
- Stu Smith, AMD

**Description**

One of the states that contributes to the combinatorial explosion of pipeline state objects that need to be created, is the vertex input binding and attribute descriptions. By allowing them to be dynamic applications may reduce the number of pipeline objects they need to create.

This extension adds dynamic state support for what is normally static state in VkPipelineVertexInputStateCreateInfo.
New Commands

- `vkCmdSetVertexInputEXT`

New Structures

- `VkVertexInputAttributeDescription2EXT`
- `VkVertexInputBindingDescription2EXT`
- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceVertexInputDynamicStateFeaturesEXT`

New Enum Constants

- `VK_EXT_VERTEX_INPUT_DYNAMIC_STATE_EXTENSION_NAME`
- `VK_EXT_VERTEX_INPUT_DYNAMIC_STATE_SPEC_VERSION`
- Extending `VkDynamicState`:
  - `VK_DYNAMIC_STATE_VERTEX_INPUT_EXT`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VERTEX_INPUT_DYNAMIC_STATE_FEATURES_EXT`
  - `VK_STRUCTURE_TYPE_VERTEX_INPUT_ATTRIBUTE_DESCRIPTION_2_EXT`
  - `VK_STRUCTURE_TYPE_VERTEX_INPUT_BINDING_DESCRIPTION_2_EXT`

Version History

- Revision 2, 2020-11-05 (Piers Daniell)
  - Make `VkVertexInputBindingDescription2EXT` extensible
  - Add new `VkVertexInputAttributeDescription2EXT` struct for the `pVertexAttributeDescriptions` parameter to `vkCmdSetVertexInputEXT` so it is also extensible
- Revision 1, 2020-08-21 (Piers Daniell)
  - Internal revisions

**VK_EXT_ycbcr_2plane_444_formats**

Name String

- `VK_EXT_ycbcr_2plane_444_formats`

Extension Type

- Device extension

Registered Extension Number

- 331
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_sampler_ycbcr_conversion

Contact

- Tony Zlatinski tzlatinski

Other Extension Metadata

Last Modified Date

2020-07-28

IP Status

No known IP claims.

Contributors

- Piers Daniell, NVIDIA
- Ping Liu, Intel

Description

This extension adds some Y'CbCr formats that are in common use for video encode and decode, but were not part of the VK_KHR_sampler_ycbcr_conversion extension.

New Structures

- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceYcbcr2Plane444FormatsFeaturesEXT

New Enum Constants

- VK_EXT_YCBCR_2PLANE_444_FORMATS_EXTENSION_NAME
- VK_EXT_YCBCR_2PLANE_444_FORMATS_SPEC_VERSION
- Extending VkFormat:
  ◦ VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16_EXT
  ◦ VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_EXT
  ◦ VK_FORMAT_G16_B16R16_2PLANE_444_UNORM_EXT
  ◦ VK_FORMAT_G8_B8R8_2PLANE_444_UNORM_EXT
- Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_YCBCR_2_PLANE_444_FORMATS_FEATURES_EXT
**Version History**

- Revision 1, 2020-03-08 (Piers Daniell)
  - Initial draft

**VK_EXT_ycbcr_image_arrays**

**Name String**

VK_EXT_ycbcr_image_arrays

**Extension Type**

Device extension

**Registered Extension Number**

253

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires VK_KHR_sampler_ycbcr_conversion

**Contact**

- Piers Daniell  (pdaniell-nv)

**Other Extension Metadata**

**Last Modified Date**

2019-01-15

**Contributors**

- Piers Daniell, NVIDIA

**Description**

This extension allows images of a format that requires $Y'CbCr$ conversion to be created with multiple array layers, which is otherwise restricted.

**New Structures**

- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceYcbcrImageArraysFeaturesEXT

**New Enum Constants**

- VK_EXT_YCBCR_IMAGE ARRAYS_EXTENSION_NAME
• **VK_EXT_YCBCR_IMAGE ARRAYS_SPEC_VERSION**

• Extending **VkStructureType**:
  ◦ **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_YCBCR_IMAGE ARRAYS FEATURES_EXT**

**Version History**

• Revision 1, 2019-01-15 (Piers Daniell)
  ◦ Initial revision

**VK_NV_external_memory_sci_buf**

**Name String**

    VK_NV_external_memory_sci_buf

**Extension Type**

    Device extension

**Registered Extension Number**

    375

**Revision**

    2

**Extension and Version Dependencies**

• Requires Vulkan 1.1

**Contact**

• Kai Zhang kazhang

**Other Extension Metadata**

**Last Modified Date**

    2022-04-12

**Contributors**

• Kai Zhang, NVIDIA

• Jeff Bolz, NVIDIA

• Jonathan McCaffrey, NVIDIA

• Daniel Koch, NVIDIA

**Description**

This extension enables an application to access external memory via **NvSciBufObj**. To import a **NvSciBufObj** to **VkDeviceMemory**, applications need to:

• Create an unreconciled **NvSciBufAttrList** via **NvSciBufAttrListCreate()**
Fill in the private attribute list via `vkGetPhysicalDeviceSciBufAttributesNV()`

Fill in the public attribute list via `NvSciBufAttrListSetAttrs`

Reconcile the `NvSciBufAttrList` via `NvSciBufAttrListReconcile()``

Create a `NvSciBufObj` via `NvSciBufObjAlloc()`

Import the `NvSciBufObj` to a `VkDeviceMemory` by chaining `VkImportMemorySciBufInfoNV` structure to the command `vkAllocateMemory`

For details of the `NvSciBuf` APIs and data structures, see the `NvStreams Documentation`.

**New Commands**

- `vkGetMemorySciBufNV`
- `vkGetPhysicalDeviceExternalMemorySciBufPropertiesNV`
- `vkGetPhysicalDeviceSciBufAttributesNV`

**New Structures**

- `VkMemoryGetSciBufInfoNV`
- `VkMemorySciBufPropertiesNV`
- Extending `VkMemoryAllocateInfo`:
  - `VkExportMemorySciBufInfoNV`
  - `VkImportMemorySciBufInfoNV`
- Extending `VkPhysicalDeviceFeatures2, VkDeviceCreateInfo`:
  - `VkPhysicalDeviceExternalMemorySciBufFeaturesNV`
  - `VkPhysicalDeviceExternalSciBufFeaturesNV`

**New Enum Constants**

- `VK_NV_EXTERNAL_MEMORY_SCI_BUF_EXTENSION_NAME`
- `VK_NV_EXTERNAL_MEMORY_SCI_BUF_SPEC_VERSION`
- Extending `VkExternalMemoryHandleTypeFlagBits`:
  - `VK_EXTERNAL_MEMORY_HANDLE_TYPE_SCI_BUF_BIT_NV`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_EXPORT_MEMORY_SCI_BUF_INFO_NV`
  - `VK_STRUCTURE_TYPE_IMPORT_MEMORY_SCI_BUF_INFO_NV`
  - `VK_STRUCTURE_TYPE_MEMORY_GET_SCI_BUF_INFO_NV`
  - `VK_STRUCTURE_TYPE_MEMORY_SCI_BUF_PROPERTIES_NV`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_MEMORY_SCI_BUF_FEATURES_NV`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SCI_BUF_FEATURES_NV`
Issues

1) What should we call this extension?

RESOLVED. The external API is NvSciBuf, but the Vulkan convention is to append the vendor suffix at the end of an identifier. Using NvSciBufNV seems awkward, so we have chosen to use just the SciBuf portion of the name in Vulkan commands and tokens. Since this is for interacting with memory objects allocated from outside Vulkan, we use “external_memory” in the name, similar to VK_KHR_external_memory_fd. To avoid an explosion of extensions, we include the capability to import and export memory in one extension but include separate features in case implementations only implement (or safety certify) a subset.

2) What changed in revision 2?

RESOLVED. The VkPhysicalDeviceExternalSciBufFeaturesNV struct was renamed to VkPhysicalDeviceExternalMemorySciBufFeaturesNV to follow naming conventions (previous names retained as aliases), and drop const on pNext pointer.

Version History

- Revision 1, 2022-04-12 (Kai Zhang, Daniel Koch)
  - Internal revisions
- Revision 2, 2023-01-03 (Daniel Koch)
  - fix the feature structure to address naming convention and cts autogeneration issues

VK_NV_external_sci_sync2

Name String

VK_NV_external_sci_sync2

Extension Type

Device extension

Registered Extension Number

490

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.1

Contact

- Kai Zhang kazhang

Other Extension Metadata
Description

An application using external memory may wish to synchronize access to that memory using semaphores and fences. This extension enables an application to import semaphore and import/export fence payloads to and from `NvSciSync` objects. To import a `NvSciSyncObj` to a `VkSemaphore` or `VkFence`, applications need to:

- Create an unreconciled `NvSciSyncAttrList` via `NvSciSyncAttrListCreate()`
- Fill the private attribute list via `vkGetPhysicalDeviceSciSyncAttributesNV()`
- Fill the public attribute list via `NvSciSyncAttrListSetAttrs()`
- Reconcile the `NvSciSyncAttrList` via `NvSciSyncAttrListReconcile()`
- Create a `NvSciSyncObj` via `NvSciSyncObjAlloc()`

To import a `NvSciSyncObj` to a `VkSemaphore`, create a `VkSemaphoreSciSyncPoolNV` for the `NvSciSyncObj` and then select the semaphore from `VkSemaphoreSciSyncPoolNV` by passing the `VkSemaphoreSciSyncCreateInfoNV` structure to `vkCreateSemaphore`

To import a `NvSciSyncObj` to a `VkFence`, pass the `VkImportFenceSciSyncInfoNV` structure to the `vkImportFenceSciSyncObjNV` command.

To import/export a `NvSciSyncFence` to a `VkFence` object, that `VkFence` object must already have a `NvSciSyncObj` previously imported.

This extension does not support exporting semaphores from `NvSciSync` objects.

For details of the `NvSciSync` APIs and data structures, see the `NvStreams Documentation`.

New Object Types

- `VkSemaphoreSciSyncPoolNV`

New Commands

- `vkCreateSemaphoreSciSyncPoolNV`
- `vkDestroySemaphoreSciSyncPoolNV`
- `vkGetFenceSciSyncFenceNV`
- `vkGetFenceSciSyncObjNV`
New Structures

- VkFenceGetSciSyncInfoNV
- VkImportFenceSciSyncInfoNV
- VkSciSyncAttributesInfoNV
- VkSemaphoreSciSyncPoolCreateInfoNV

Extending VkFenceCreateInfo:
  - VkExportFenceSciSyncInfoNV

Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceExternalSciSync2FeaturesNV

Extending VkSemaphoreCreateInfo:
  - VkSemaphoreSciSyncCreateInfoNV

If Vulkan SC 1.0 is supported:

Extending VkDeviceObjectReservationCreateInfo:
  - VkDeviceSemaphoreSciSyncPoolReservationCreateInfoNV

New Enums

- VkSciSyncClientTypeNV
- VkSciSyncPrimitiveTypeNV

New Enum Constants

- VK_NV_EXTERNAL_SCI_SYNC_2_EXTENSION_NAME
- VK_NV_EXTERNAL_SCI_SYNC_2_SPEC_VERSION

Extending VkExternalFenceHandleTypeFlagBits:
  - VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_FENCE_BIT_NV
  - VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_OBJ_BIT_NV

Extending VkObjectType:
  - VK_OBJECT_TYPE_SEMAPHORE_SCI_SYNC_POOL_NV

Extending VkStructureType:
  - VK_STRUCTURE_TYPE_EXPORT_FENCE_SCI_SYNC_INFO_NV
  - VK_STRUCTURE_TYPE_FENCE_GET_SCI_SYNC_INFO_NV
  - VK_STRUCTURE_TYPE_IMPORT_FENCE_SCI_SYNC_INFO_NV
If Vulkan SC 1.0 is supported:

- Extending *VkStructureType*:
  - **VK_STRUCTURE_TYPE_DEVICE_SEMAPHORE_SCI_SYNC_POOL_RESERVATION_CREATE_INFO_NV**

**Issues**

1) **Does this extension extend or replace** *VK_NV_external_sci_sync*?

RESOLVED. Replaces - expect to deprecate it and eventually remove it.

2) **What part of** *VK_NV_external_sci_sync* **is deprecated/removed in this extension?**

RESOLVED. The commands to import and export semaphores from *VK_NV_external_sci_sync* are removed and have been replaced with an alternate mechanism to import semaphores. Fence import and export functionality is unchanged.

In particular:

- **Removed Commands:**
  - *vkImportSemaphoreSciSyncObjNV*
  - *vkGetSemaphoreSciSyncObjNV*

- **Removed Structures:**
  - *VkImportSemaphoreSciSyncInfoNV*
  - *VkExportSemaphoreSciSyncInfoNV*
  - *VkSemaphoreGetSciSyncInfoNV*

3) **Application migration guide from** *VK_NV_external_sci_sync to VK_NV_external_sci_sync2***

- In *VK_NV_external_sci_sync*, to import a *NvSciSyncObj* to *VkSemaphore*, applications need to:
  - Create a *VkSemaphore* by command *vkCreateSemaphore*.
  - Call *vkImportSemaphoreSciSyncObjNV* command to import the *NvSciSyncObj* to *VkSemaphore* created.
  - Call *vkDestroySemaphore* to destroy the *VkSemaphore* after all submitted batches that refer to it have completed execution.

- In order to migrate to *VK_NV_external_sci_sync2*, applications need to:
  - Chain *VkDeviceSemaphoreSciSyncPoolReservationCreateInfoNV* to *VkDeviceObjectReservationCreateInfo* and specify the *semaphoreSciSyncPoolRequestCount* maximum number of semaphore SciSync pools that will be used simultaneously.
◦ Import the a \texttt{NvSciSyncObj} to a \texttt{VkSemaphoreSciSyncPoolNV} by command \texttt{vkCreateSemaphoreSciSyncPoolNV}.

◦ Select the \texttt{VkSemaphore} from \texttt{VkSemaphoreSciSyncPoolNV} by passing the \texttt{VkSemaphoreSciSyncCreateInfoNV} structure to \texttt{vkCreateSemaphore}.

◦ Can call \texttt{vkDestroySemaphore} to destroy the \texttt{VkSemaphore} immediately after all the batches that refer to it are submitted.

**Version History**

- Revision 1, 2022-09-07 (Kai Zhang, Daniel Koch)
  - Initial revision

**VK_NV_private_vendor_info**

**Name String**

\texttt{VK_NV_private_vendor_info}

**Extension Type**

Device extension

**Registered Extension Number**

52

**Revision**

2

**Extension and Version Dependencies**

- Requires Vulkan 1.0

**Contact**

- Daniel Koch \(\text{dgkoch}\)

**Other Extension Metadata**

**Last Modified Date**

2022-08-10

**Contributors**

- Daniel Koch, NVIDIA
- Jonathan McCaffrey, NVIDIA
- Jeff Bolz, NVIDIA

**Description**

This extension provides the application with access to vendor-specific enums and structures that are not expected to be publicly documented.
New Enum Constants

- VK_NV_PRIVATE_VENDOR_INFO_EXTENSION_NAME
- VK_NV_PRIVATE_VENDOR_INFO_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PRIVATE_VENDOR_INFO_RESERVED_OFFSET_0_NV

Issues

1) What should we call this extension?

RESOLVED. VK_NV_private_vendor_info as this contains details of NVIDIA's implementation that we do not expect to publicly document.

Version History

- Revision 1, 2022-05-03 (Daniel Koch)
  - Internal revisions
- Revision 2, 2022-08-10 (Daniel Koch)
  - change number for extension (373 to 52) to avoid conflict

List of Deprecated Extensions

- VK_NV_external_sci_sync
**VK_NV_external_sci_sync**

**Name String**

VK_NV_external_sci_sync

**Extension Type**

Device extension

**Registered Extension Number**

374

**Revision**

2

**Extension and Version Dependencies**

- Requires Vulkan 1.1

**Deprecation state**

- Deprecated by VK_NV_external_sci_sync2 extension

**Contact**

- Kai Zhang @kazhang

**Other Extension Metadata**

**Last Modified Date**

2022-04-12

**Contributors**

- Kai Zhang, NVIDIA
- Jeff Bolz, NVIDIA
- Jonathan McCaffrey, NVIDIA
- Daniel Koch, NVIDIA

**Description**

An application using external memory may wish to synchronize access to that memory using semaphores and fences. This extension enables an application to import and export semaphore and fence payloads to and from NvSciSync objects. To import a NvSciSyncObj to a VkSemaphore or VkFence, applications need to:

- Create an unreconciled NvSciSyncAttrList via NvSciSyncAttrListCreate()
- Fill the private attribute list via vkGetPhysicalDeviceSciSyncAttributesNV()
- Fill the public attribute list via NvSciSyncAttrListSetAttrs()
- Reconcile the NvSciSyncAttrList via NvSciSyncAttrListReconcile()
- Create a NvSciSyncObj via NvSciSyncObjAlloc()
• Import the `NvSciSyncObj` to a `VkSemaphore` by passing the `VkImportSemaphoreSciSyncInfoNV` structure to the `vkImportSemaphoreSciSyncObjNV` command, or to a `VkFence` by passing the `VkImportFenceSciSyncInfoNV` structure to the `vkImportFenceSciSyncObjNV` command.

To import/export a `NvSciSyncFence` to a `VkFence` object, that `VkFence` object **must** already have a `NvSciSyncObj` previously imported.

For details of the `NvSciSync` APIs and data structures, see the `NvStreams Documentation`.

### New Commands

- `vkGetFenceSciSyncFenceNV`
- `vkGetFenceSciSyncObjNV`
- `vkGetPhysicalDeviceSciSyncAttributesNV`
- `vkGetSemaphoreSciSyncObjNV`
- `vkImportFenceSciSyncFenceNV`
- `vkImportFenceSciSyncObjNV`
- `vkImportSemaphoreSciSyncObjNV`

### New Structures

- `VkFenceGetSciSyncInfoNV`
- `VkImportFenceSciSyncInfoNV`
- `VkImportSemaphoreSciSyncInfoNV`
- `VkSciSyncAttributesInfoNV`
- `VkSemaphoreGetSciSyncInfoNV`

  Extending `VkFenceCreateInfo`:
  - `VkExportFenceSciSyncInfoNV`

  Extending `VkPhysicalDeviceFeatures2, VkDeviceCreateInfo`:
  - `VkPhysicalDeviceExternalSciSyncFeaturesNV`

  Extending `VkSemaphoreCreateInfo`:
  - `VkExportSemaphoreSciSyncInfoNV`

### New Enums

- `VkSciSyncClientTypeNV`
- `VkSciSyncPrimitiveTypeNV`

### New Enum Constants

- `VK_NV_EXTERNAL_SCI_SYNC_EXTENSION_NAME`
- `VK_NV_EXTERNAL_SCI_SYNC_SPEC_VERSION`
• Extending VkExternalFenceHandleTypeFlagBits:
  ◦ VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_FENCE_BIT_NV
  ◦ VK_EXTERNAL_FENCE_HANDLE_TYPE_SCI_SYNC_OBJ_BIT_NV

• Extending VkExternalSemaphoreHandleTypeFlagBits:
  ◦ VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SCI_SYNC_OBJ_BIT_NV

• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_EXPORT_FENCE_SCI_SYNC_INFO_NV
  ◦ VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_SCI_SYNC_INFO_NV
  ◦ VK_STRUCTURE_TYPE_FENCE_GET_SCI_SYNC_INFO_NV
  ◦ VK_STRUCTURE_TYPE_IMPORT_FENCE_SCI_SYNC_INFO_NV
  ◦ VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_SCI_SYNC_INFO_NV
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SCI_SYNC_FEATURES_NV
  ◦ VK_STRUCTURE_TYPE_SCI_SYNC_ATTRIBUTES_INFO_NV
  ◦ VK_STRUCTURE_TYPE_SEMAPHORE_GET_SCI_SYNC_INFO_NV

Issues

1) What should we call this extension?

RESOLVED. The external API is NvSciSync, but the Vulkan convention is to append the vendor suffix at the end of an identifier. Using NvSciSyncNV seems awkward, so we have chosen to use just the SciSync portion of the name in Vulkan commands and tokens. Since this is for interacting with objects from outside Vulkan, we use "external" in the name, similar to VK_KHR_external_fence_fd. To avoid an explosion of extensions, we include the capability to import and export both semaphores and fences in one extension but include separate features in case implementations only implement (or safety certify) a subset.

2) How do we resolve the NvStreams terminology of NvSciSyncFence which conflicts with the Vulkan SC terminology of VkFence.

RESOLVED: "fence" refers to VkFence. "NvSciSyncFence" refers to the NvStreams type and "VkFence" refers to the Vulkan SC type.

Version History

• Revision 2, 2022-03-29 (Daniel Koch)
  ◦ use separate entry points for NvSciSyncFence and NvSciSyncObj handles

• Revision 1, 2020-11-25 (Kai Zhang, Daniel Koch)
  ◦ Initial revision
Appendix F: API Boilerplate

This appendix defines Vulkan API features that are infrastructure required for a complete functional description of Vulkan, but do not logically belong elsewhere in the Specification.

Vulkan Header Files

Vulkan is defined as an API in the C99 language. Khronos provides a corresponding set of header files for applications using the API, which may be used in either C or C++ code. The interface descriptions in the specification are the same as the interfaces defined in these header files, and both are derived from the *vk.xml* XML API Registry, which is the canonical machine-readable description of the Vulkan API. The Registry, scripts used for processing it into various forms, and documentation of the registry schema are available as described at https://www.khronos.org/registry/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:

```
VKAPI_ATTR <return_type> VKAPI_CALL <command_name>(<command_parameters>);
```

Additionally, a Vulkan function pointer type declaration takes the form of:

```
typedef <return_type> (VKAPI_PTR *PFN_<command_name>)(<command_parameters>);
```

Platform-Specific Header Control

If the VK_NO_STDINT_H macro is defined by the application at compile time, extended integer types used by the Vulkan API, such as uint8_t, must also be defined by the application. Otherwise, the Vulkan headers will not compile. If VK_NO_STDINT_H is not defined, the system <stdint.h> is used to define these types. There is a fallback path when Microsoft Visual Studio version 2008 and earlier versions are detected at compile time.

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.

```
// Provided by VK_VERSION_1_0
// Version of this file
#define VK_HEADER_VERSION 12
```

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 \texttt{VkApplicationInfo::apiVersion}. Instead applications 
should explicitly select a specific fixed major/minor API version using, for example, one of the \texttt{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

\textbf{VK_DEFINE_HANDLE} defines a \textit{dispatchable handle} type.

```c
// Provided by VK_VERSION_1_0

#define VK_DEFINE_HANDLE(object) typedef struct object##_T* (object);
```

- \textbf{object} is the name of the resulting C type.

The only dispatchable handle types are those related to device and instance management, such as \texttt{VkDevice}.

\textbf{VK_DEFINE_NON_DISPATCHABLE_HANDLE} defines a \textit{non-dispatchable handle} type.

```c
// Provided by VK_VERSION_1_0

#ifndef VK_DEFINE_NON_DISPATCHABLE_HANDLE
#if (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
```

- \textbf{object} is the name of the resulting C type.

Most Vulkan handle types, such as \texttt{VkBuffer}, are non-dispatchable.

\textbf{Note}

The \texttt{vulkan_sc_core.h} header allows the \texttt{VK_DEFINE_NON_DISPATCHABLE_HANDLE} and 
\texttt{VK_NULL_HANDLE} definitions to be overridden by the application. If \texttt{VK_DEFINE_NON_DISPATCHABLE_HANDLE} is already defined when \texttt{vulkan_sc_core.h} is 
compiled, the default definitions for \texttt{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
#ifndef VK_NULL_HANDLE
    #define VK_NULL_HANDLE 0
#endif
```

VK_USE_64_BIT_PTR_DEFINES defines whether the default non-dispatchable handles are declared using either a 64-bit pointer type or a 64-bit unsigned integer type.

VK_USE_64_BIT_PTR_DEFINES is set to '1' to use a 64-bit pointer type or any other value to use a 64-bit unsigned integer type.

```c
// Provided by VK_VERSION_1_0

#ifndef VK_USE_64_BIT_PTR_DEFINES
    #if defined(__LP64__) || defined(_WIN64) || (defined(__x86_64__) && !defined(__ILP32__)) || defined(_M_X64) || defined(__ia64) || defined (_M_IA64) || defined(__aarch64__) || defined(__powerpc64__)
        #define VK_USE_64_BIT_PTR_DEFINES 1
    #else
        #define VK_USE_64_BIT_PTR_DEFINES 0
    #endif
#endif
```

**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 systems **must** be included at compile time, or platform-specific types **must** be forward-declared. The Vulkan header files cannot determine whether or not an external header is available at compile time, so platform-specific extensions are provided in separate headers from the core API and platform-independent extensions, allowing applications to decide which ones should be defined and how the external headers are included.

Extensions dependent on particular sets of platform headers, or that forward-declare platform-specific types, are declared in a header named for that platform. Before including these platform-specific Vulkan headers, applications **must** include both `vulkan_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>VK_USE_PLATFORM_ANDROID_KHR</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>VK_USE_PLATFORM_WAYLAND_KHR</td>
</tr>
<tr>
<td>VK_KHR_xcb_surface</td>
<td>X11 Xcb</td>
<td><code>vulkan_xcb.h</code></td>
<td><code>&lt;xcb/xcb.h&gt;</code></td>
<td>VK_USE_PLATFORM_XCB_KHR</td>
</tr>
<tr>
<td>VK_KHR_xlib_surface</td>
<td>X11 Xlib</td>
<td><code>vulkan_xlib.h</code></td>
<td><code>&lt;X11/Xlib.h&gt;</code></td>
<td>VK_USE_PLATFORM_XLIB_KHR</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_NV_external_sci_sync</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>
<tr>
<td>VK_NV_external_memory_sci_buf</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note**

This section describes the purpose of the headers independently of the specific underlying functionality of the window system extensions themselves. Each extension name will only link to a description of that extension when viewing a specification built with that extension included.

**Provisional Extension Header Control (Informative)**

*Provisional* extensions should not be used in production applications. The functionality defined by such extensions may change in ways that break backwards compatibility between revisions, and before final release of a non-provisional version of that extension.

Provisional extensions are defined in a separate *provisional header*, `vulkan_beta.h`, allowing applications to decide whether or not to include them. The mechanism is similar to *window system-specific headers*: before including `vulkan_beta.h`, applications must include `vulkan_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 TessCoord will be in the range [0, 1]. Additionally, for any defined component x of TessCoord, the results of computing 1.0-x in a tessellation evaluation shader will be exact. If any floating-point values in the range [0, 1] fail to satisfy this property, such values must not be used as tessellation coordinate components.
Appendix 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>
<tbody>
<tr>
<td>VK_KHR_object_refresh</td>
<td>Optional</td>
</tr>
</tbody>
</table>

The following items have been added to Vulkan SC:

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Additions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fundamentals</td>
<td>• extending VkResult</td>
</tr>
<tr>
<td></td>
<td>◦ VK_ERROR_VALIDATION_FAILED [SCID-1]</td>
</tr>
<tr>
<td></td>
<td>◦ VK_ERROR_INVALID_PIPELINE_CACHE_DATA [SCID-1]</td>
</tr>
<tr>
<td></td>
<td>◦ VK_ERROR_NO_PIPELINE_MATCH [SCID-1]</td>
</tr>
<tr>
<td>Devices and Queues</td>
<td>• VkPhysicalDeviceVulkanSC10Properties [SCID-1]</td>
</tr>
<tr>
<td></td>
<td>• VkDeviceObjectReservationCreateInfo [SCID-4]</td>
</tr>
<tr>
<td></td>
<td>• VkPerformanceQueryReservationInfoKHR [SCID-4]</td>
</tr>
<tr>
<td></td>
<td>• VkPipelinePoolSize [SCID-4]</td>
</tr>
<tr>
<td>Command Buffers</td>
<td>• VkCommandPoolMemoryReservationCreateInfo [SCID-4]</td>
</tr>
<tr>
<td></td>
<td>• vkGetCommandPoolMemoryConsumption [SCID-1]</td>
</tr>
<tr>
<td></td>
<td>• VkCommandPoolMemoryConsumption [SCID-1]</td>
</tr>
<tr>
<td>Pipelines</td>
<td>• extending VkPipelineCacheCreateFlagBits</td>
</tr>
<tr>
<td></td>
<td>◦ VK_PIPELINE_CACHE_CREATE_READ_ONLY_BIT [SCID-1], [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>◦ VK_PIPELINE_CACHE_CREATE_USE_APPLICATION_STORAGE_BIT [SCID-2]</td>
</tr>
<tr>
<td></td>
<td>• extending VkPipelineCacheHeaderVersion</td>
</tr>
<tr>
<td></td>
<td>◦ VK_PIPELINE_CACHE_HEADER_VERSION_SAFETY_CRITICAL_ONE [SCID-1], [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkPipelineCacheHeaderVersionSafetyCriticalOne [SCID-1], [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkPipelineCacheValidationVersion [SCID-1], [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkPipelineCacheSafetyCriticalIndexEntry [SCID-1], [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkPipelineCacheStageValidationIndexEntry[SCID-1], [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkPipelineOfflineCreateInfo [SCID-1], [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkPipelineMatchControl [SCID-1]</td>
</tr>
</tbody>
</table>
### Additions

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Additions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Allocation</td>
<td>- extending VkMemoryHeapFlagBits</td>
</tr>
<tr>
<td></td>
<td>- VK_MEMORY_HEAP_SEU_SAFE_BIT [SCID-1]</td>
</tr>
<tr>
<td>Features</td>
<td>- VkPhysicalDeviceVulkanSC10Features [SCID-1]</td>
</tr>
<tr>
<td>Debugging</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>Fundamentals</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>Devices and Queues</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::pPipelineCacheCreateInfo points 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>Command Buffers</td>
<td>• The <code>VkCommandPoolCreateInfo::pNext</code> chain must include a valid <code>VkCommandPoolMemoryReservationCreateInfo</code> structure [SCID-4].</td>
</tr>
<tr>
<td></td>
<td>• If <code>commandPoolResetCommandBuffer</code> is not supported [SCID-8], <code>vkResetCommandBuffer</code> must not be called.</td>
</tr>
<tr>
<td></td>
<td>• <code>vkFreeCommandBuffers</code> does not return the memory used by command recording back to its parent command pool [SCID-4]. This memory is reclaimed when <code>vkResetCommandPool</code> is next called.</td>
</tr>
<tr>
<td></td>
<td>• If <code>VkPhysicalDeviceVulkanSC10Properties::commandPoolMultipleCommandBuffersRecording</code> is <code>VK_FALSE</code>, then only one command buffer from a command pool can be in the recording state at a time [SCID-8].</td>
</tr>
<tr>
<td></td>
<td>• If <code>VkPhysicalDeviceVulkanSC10Properties::commandBufferSimultaneousUse</code> is <code>VK_FALSE</code>, then <code>VkCommandBufferBeginInfo::flags</code> must not include <code>VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT</code> [SCID-8].</td>
</tr>
<tr>
<td></td>
<td>• If <code>commandPoolResetCommandBuffer</code> is not supported, <code>commandBuffer</code> must be in the initial state when <code>vkBeginCommandBuffer</code> is called [SCID-8].</td>
</tr>
<tr>
<td></td>
<td>• If <code>VkPhysicalDeviceVulkanSC10Properties::secondaryCommandBufferNullOrImagelessFramebuffer</code> is <code>VK_FALSE</code>, then <code>VkCommandBufferInheritanceInfo::framebuffer</code> must not be <code>VK_NULL_HANDLE</code> and <code>flags</code> must not have been created with a <code>VkFramebufferCreateInfo::flags</code> value that includes <code>VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT</code> if the command buffer will be executed within a render pass instance [SCID-8].</td>
</tr>
<tr>
<td>Chapter</td>
<td>Modifications</td>
</tr>
<tr>
<td>---------</td>
<td>---------------</td>
</tr>
<tr>
<td>Pipelines</td>
<td>• <code>vkCreateComputePipelines</code> returns <code>VK_ERROR_NO_PIPELINE_MATCH</code> if the <code>VkComputePipelineCreateInfo::pNext</code> chain does not include a valid <code>VkPipelineOfflineCreateInfo</code> structure [SCID-1].</td>
</tr>
<tr>
<td></td>
<td>• <code>vkCreateComputePipelines::pipelineCache</code> must not be <code>VK_NULL_HANDLE</code> [SCID-1], [SCID-8].</td>
</tr>
<tr>
<td></td>
<td>• <code>VkComputePipelineCreateInfo::basePipelineHandle</code> must be <code>VK_NULL_HANDLE</code> [SCID-8].</td>
</tr>
<tr>
<td></td>
<td>• <code>VkComputePipelineCreateInfo::basePipelineIndex</code> must be zero [SCID-8].</td>
</tr>
<tr>
<td></td>
<td>• <code>vkCreateGraphicsPipelines</code> returns <code>VK_ERROR_NO_PIPELINE_MATCH</code> if the <code>VkGraphicsPipelineCreateInfo::pNext</code> chain does not include a valid <code>VkPipelineOfflineCreateInfo</code> structure [SCID-1].</td>
</tr>
<tr>
<td></td>
<td>• <code>vkCreateGraphicsPipelines::pipelineCache</code> must not be <code>VK_NULL_HANDLE</code> [SCID-1], [SCID-8].</td>
</tr>
<tr>
<td></td>
<td>• <code>VkGraphicsPipelineCreateInfo::basePipelineHandle</code> must be <code>VK_NULL_HANDLE</code> [SCID-8].</td>
</tr>
<tr>
<td></td>
<td>• <code>VkGraphicsPipelineCreateInfo::basePipelineIndex</code> must be zero [SCID-8].</td>
</tr>
<tr>
<td></td>
<td>• <code>VkPipelineCacheCreateInfo::pInitialData</code> must point to a valid pipeline cache that has been generated offline [SCID-1], [SCID-8].</td>
</tr>
<tr>
<td></td>
<td>• <code>VkPipelineCacheCreateInfo::initialDataSize</code> must not be 0 [SCID-1], [SCID-8].</td>
</tr>
<tr>
<td></td>
<td>• <code>VkPipelineCacheCreateInfo::pInitialData</code> must not be <code>NULL</code> [SCID-1], [SCID-8].</td>
</tr>
<tr>
<td></td>
<td>• <code>VkPipelineCacheCreateInfo::flags</code> must include <code>VK_PIPELINE_CACHE_CREATE_READ_ONLY_BIT</code> [SCID-1], [SCID-8].</td>
</tr>
<tr>
<td></td>
<td>• <code>VkPipelineCacheCreateInfo::flags</code> must include <code>VK_PIPELINE_CACHE_CREATE_USE_APPLICATION_STORAGE_BIT</code> [SCID-2].</td>
</tr>
<tr>
<td></td>
<td>• The contents of <code>VkPipelineCacheCreateInfo</code>, including the data pointed to by <code>VkPipelineCacheCreateInfo::pInitialData</code>, passed to <code>vkCreatePipelineCache</code> must be the same as specified in one of the <code>VkDeviceObjectReservationCreateInfo::pPipelineCacheCreateInfo</code> structures when the device was created [SCID-1].</td>
</tr>
<tr>
<td></td>
<td>• <code>VkPipelineCacheHeaderVersionOne::headerSize</code> must be 56 [SCID-1].</td>
</tr>
<tr>
<td></td>
<td>• <code>VkPipelineCacheHeaderVersionOne::headerVersion</code> must be <code>VK_PIPELINE_CACHE_HEADER_VERSION_SAFETY_CRITICAL_ONE</code> [SCID-1].</td>
</tr>
<tr>
<td>Chapter</td>
<td>Modifications</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Memory Allocation</td>
<td>• <code>vkCreate*::pAllocator</code> must be <code>NULL</code> [SCID-2], [SCID-8].&lt;br&gt;• <code>vkDestroy*::pAllocator</code> must be <code>NULL</code> [SCID-2], [SCID-8].&lt;br&gt;• <code>vk*Memory::pAllocator</code> must be <code>NULL</code> [SCID-2], [SCID-8].&lt;br&gt;• <code>vkRegisterDeviceEventEXT::pAllocator</code> must be <code>NULL</code> [SCID-8].</td>
</tr>
<tr>
<td>Resource Creation</td>
<td>• <code>VkBufferCreateInfo::flags</code> must not contain any of the <code>VK_BUFFER_CREATE_SPARSE_BINDING_BIT</code>, <code>VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT</code>, or <code>VK_BUFFER_CREATE_SPARSE_ALIASED_BIT</code> flags [SCID-8].&lt;br&gt;• <code>VkImageCreateInfo::flags</code> must not contain any of the <code>VK_IMAGE_CREATE_SPARSE_BINDING_BIT</code>, <code>VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT</code>, or <code>VK_IMAGE_CREATE_SPARSE_ALIASED_BIT</code> flags [SCID-8].&lt;br&gt;• <code>VkBindImageMemoryDeviceGroupInfo::splitInstanceBindRegionCount</code> must be zero [SCID-8].</td>
</tr>
<tr>
<td>Resource Descriptors</td>
<td>• If <code>recycleDescriptorSetMemory</code> is <code>VK_FALSE</code>, 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].</td>
</tr>
<tr>
<td>Sparse Resources</td>
<td>• <code>VkPhysicalDeviceSparseProperties::residencyStandard2DBlockShape</code> must be reported as <code>VK_FALSE</code> [SCID-8].&lt;br&gt;• <code>VkPhysicalDeviceSparseProperties::residencyStandard2DMultisampleBlockShape</code> must be reported as <code>VK_FALSE</code> [SCID-8].&lt;br&gt;• <code>VkPhysicalDeviceSparseProperties::residencyStandard3DBlockShape</code> must be reported as <code>VK_FALSE</code> [SCID-8].&lt;br&gt;• <code>VkPhysicalDeviceSparseProperties::residencyAlignedMipSize</code> must be reported as <code>VK_FALSE</code> [SCID-8].&lt;br&gt;• <code>VkPhysicalDeviceSparseProperties::residencyNonResidentStrict</code> must be reported as <code>VK_FALSE</code> [SCID-8].</td>
</tr>
<tr>
<td>WSI Swapchain</td>
<td>• <code>VkSwapchainCreateInfoKHR::flags</code> must not contain <code>VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR</code> [SCID-8].&lt;br&gt;• <code>VkSwapchainCreateInfoKHR::oldSwapchain</code> must be <code>VK_NULL_HANDLE</code> [SCID-4].</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Chapter</th>
<th>Modifications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Features</strong></td>
<td></td>
</tr>
<tr>
<td>• <em>VkPhysicalDeviceFeatures::shaderResourceResidency</em> must be reported as <code>VK_FALSE</code> [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• <em>VkPhysicalDeviceFeatures::sparseBinding</em> must be reported as <code>VK_FALSE</code> [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• <em>VkPhysicalDeviceFeatures::sparseResidencyBuffer</em> must be reported as <code>VK_FALSE</code> [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• <em>VkPhysicalDeviceFeatures::sparseResidencyImage2D</em> must be reported as <code>VK_FALSE</code> [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• <em>VkPhysicalDeviceFeatures::sparseResidencyImage3D</em> must be reported as <code>VK_FALSE</code> [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• <em>VkPhysicalDeviceFeatures::sparseResidency2Samples</em> must be reported as <code>VK_FALSE</code> [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• <em>VkPhysicalDeviceFeatures::sparseResidency4Samples</em> must be reported as <code>VK_FALSE</code> [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• <em>VkPhysicalDeviceFeatures::sparseResidency8Samples</em> must be reported as <code>VK_FALSE</code> [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• <em>VkPhysicalDeviceFeatures::sparseResidency16Samples</em> must be reported as <code>VK_FALSE</code> [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• <em>VkPhysicalDeviceFeatures::sparseResidencyAliased</em> must be reported as <code>VK_FALSE</code> [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• <em>VkPhysicalDeviceVulkanSC10Features::shaderAtomicInstructions</em> are made optional [SCID-1].</td>
<td></td>
</tr>
<tr>
<td>• <em>VkPhysicalDeviceVulkan11Features::multiview</em> is made optional [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• <em>VkPhysicalDeviceVulkan12Features::timelineSemaphore</em> is made optional [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• <em>VkPhysicalDeviceVulkan12Features::vulkanMemoryModel</em> must be reported as <code>VK_TRUE</code> [SCID-1].</td>
<td></td>
</tr>
<tr>
<td><strong>Limits</strong></td>
<td></td>
</tr>
<tr>
<td>• <em>VkPhysicalDeviceLimits::maxFramebufferLayers</em> may be 1 if neither <code>geometryShader</code> or <code>shaderOutputLayer</code> are supported [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• <em>VkPhysicalDeviceVulkan12Properties::supportedDepthResolveModes</em> may be only <code>VK_RESOLVE_MODE_NONE</code> [SCID-8].</td>
<td></td>
</tr>
<tr>
<td>• <em>VkPhysicalDeviceVulkan12Properties::supportedStencilResolveModes</em> may be only <code>VK_RESOLVE_MODE_NONE</code> [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>
</table>
| Fundamentals                  | • **VkStructureType** (deprecated aliases)**
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTER_FEATURES [SCID-8]
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DRAW_PARAMETER_FEATURES [SCID-8]
  ◦ VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES2_EXT [SCID-8]                                                                                                                                 |
| Devices and Queues            | • **VkQueueFlagBits****
  ◦ VK_QUEUE_SPARSE_BINDING_BIT [SCID-8]                                                                                                                                 |
| Command Buffers               | • **vkTrimCommandPool, vkTrimCommandPoolKHR [SCID-8]**
  • **VkCommandPoolTrimFlags, VkCommandPoolTrimFlagsKHR [SCID-8]**
  • **VkCommandPoolResetFlagBits**
  ◦ VK_COMMAND_POOL_RESET_RELEASE_RESOURCES_BIT [SCID-4]                                                                                                                                 |
| Synchronization and Cache Control | • **vkDestroySemaphoreSciSyncPoolNV [SCID-4]**                                                                                                                                 |
| Shaders                       | • **VkStructureType**
  ◦ VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO [SCID-8]**
  • **VkObjectType**
  ◦ VK_OBJECT_TYPE_SHADER_MODULE [SCID-8]**
  • **vkCreateShaderModule, vkDestroyShaderModule [SCID-8]**
  • **VkShaderModule, VkShaderModuleCreateInfo [SCID-8]**
  • **VkShaderModuleCreateFlags [SCID-8]**
  • **VkShaderModuleCreateFlagBits [SCID-8]**                                                                                                                                 |
| Pipelines                     | • **VkPipelineCreateFlagBits**
  ◦ VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT [SCID-8]
  ◦ VK_PIPELINE_CREATE_DERIVATIVE_BIT [SCID-8]                                                                                                                                 |
| Memory Allocation             | • **VkSystemAllocationScope, VkInternalAllocationType [SCID-8]**
  • **vkFreeMemory [SCID-4]**                                                                                                                                 |
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Removals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Descriptors</td>
<td>• vkDestroyDescriptorPool [SCID-4]</td>
</tr>
<tr>
<td></td>
<td>• VkStructureType</td>
</tr>
<tr>
<td></td>
<td>◦ VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO_KHR [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkObjectType</td>
</tr>
<tr>
<td></td>
<td>◦ VK_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_KHR [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• vkCreateDescriptorUpdateTemplateKHR,</td>
</tr>
<tr>
<td></td>
<td>vkDestroyDescriptorUpdateTemplateKHR,</td>
</tr>
<tr>
<td></td>
<td>vkUpdateDescriptorSetWithTemplateKHR,</td>
</tr>
<tr>
<td></td>
<td>vkCmdPushDescriptorSetWithTemplateKHR [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkDescriptorUpdateTemplateKHR, VkDescriptorUpdateTemplateEntryKHR,</td>
</tr>
<tr>
<td></td>
<td>VkDescriptorUpdateTemplateCreateInfoKHR [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>• VkDescriptorUpdateTemplateType</td>
</tr>
<tr>
<td></td>
<td>◦ VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET_KHR [SCID-8]</td>
</tr>
<tr>
<td></td>
<td>◦ VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR [SCID-8]</td>
</tr>
<tr>
<td>Queries</td>
<td>• vkDestroyQueryPool [SCID-4]</td>
</tr>
<tr>
<td>Fragment Operations</td>
<td>• VkStencilFaceFlagBits (deprecated alias)</td>
</tr>
<tr>
<td></td>
<td>◦ VK_STENCIL_FRONT_AND_BACK [SCID-8]</td>
</tr>
<tr>
<td>Chapter</td>
<td>Removals</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Sparse Resources</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>Window System Integration</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 [fundamentals-errors].

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 <em>char</em> and <em>float</em> 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 typedef struct VkStruct { … } VkStruct; and typedef enum VkEnum { … } VkEnum; where the tag declaration is unused.</td>
</tr>
<tr>
<td>Rule</td>
<td>2.4: &quot;A project should not contain unused tag declarations&quot;</td>
</tr>
<tr>
<td>------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>Rationale</td>
<td>The <code>vulkan_sc_core.h</code> provides a complete API definition and it is expected that an application may 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>

<table>
<thead>
<tr>
<th>Rule</th>
<th>2.5: &quot;A project should not contain unused macro declarations&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>Advisory</td>
</tr>
<tr>
<td>Note</td>
<td>This is reported for every unused macro defined in the header.</td>
</tr>
<tr>
<td>Rationale</td>
<td>The <code>vulkan_sc_core.h</code> provides a complete API definition and it is expected that an application may not use all the provided macro declarations.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>5.1: &quot;External identifiers shall be distinct&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>Required</td>
</tr>
<tr>
<td>Note</td>
<td>This is reported for identifiers with names that do not differ in the first 31 characters, such as <code>vkGetPhysicalDeviceFormatProperties</code> and <code>vkGetPhysicalDeviceFormatProperties2</code>.</td>
</tr>
<tr>
<td>Rationale</td>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>5.2: &quot;Identifiers declared in the same scope and name space shall be distinct&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>Required</td>
</tr>
<tr>
<td>Note</td>
<td>This is reported for many <code>typedef</code> statements with long identifiers.</td>
</tr>
<tr>
<td>Rationale</td>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>5.4: &quot;Macro identifiers shall be distinct&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>Required</td>
</tr>
<tr>
<td>Note</td>
<td>This is reported for macros with names that do not differ in the first 31 characters, such as <code>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT</code> and <code>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT</code>.</td>
</tr>
<tr>
<td>Rationale</td>
<td>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.</td>
</tr>
</tbody>
</table>

<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>Category</td>
<td>Required</td>
</tr>
</tbody>
</table>

1784
<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 <code>union</code> 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 <code>VkClearColorValue</code>, <code>VkClearValue</code>, and <code>VkPerformanceCounterResultKHR</code> 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 <code>extern &quot;C&quot;</code> 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 <code>#</code> and <code>##</code> 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>
</tbody>
</table>
| | ```
#define VK_DEFINE_HANDLE(object) typedef struct object##_T* (object);
#define VK_DEFINE_NON_DISPATCHABLE_HANDLE(object) typedef struct object##_T *(object);
``` |
| Rationale | This is expected usage of the macro expansion operation and there are not multiple operators used in the statement. |
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.

Advanced Blend Operation

Blending performed using one of the blend operation enums introduced by the `VK_EXT_blend_operation_advanced` extension. See Advanced Blending Operations.

Alias (API type/command)

An identical definition of another API type/command with the same behavior but a different name.

Aliased Range (Memory)

A range of a device memory allocation that is bound to multiple resources simultaneously.

Allocation Scope

An association of a host memory allocation to a parent object or command, where the allocation's lifetime ends before or at the same time as the parent object is freed or destroyed, or during the parent command.

Aspect (Image)

An image may contain multiple kinds, or aspects, of data for each pixel, where each aspect is used in a particular way by the pipeline and may be stored differently or separately from other aspects. For example, the color components of an image format make up the color aspect of the image, and may be used as a framebuffer color attachment. Some operations, like depth testing, operate only on specific aspects of an image.

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
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 unsignaled. 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 that vertex positions (Position decoration) are written in 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`
- `VkDrmFormatModifierPropertiesEXT::drmFormatModifierTilingFeatures`

**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 mipmap levels that can be provided for an image, from the largest application specified mipmap level size down to the *minimum mipmap size*. See *Image Mipmap Sizing*.

**Component (Format)**
A distinct part of a format. Color components are represented with R, G, B, and A. Depth and stencil components are represented with D and S. Formats can have multiple instances of the same component. Some formats have other notations such as E or X which are not considered a component of the format.

**Compressed Texel Block**
An element of an image having a block-compressed format, comprising a rectangular block of texel values that are encoded as a single value in memory. Compressed texel blocks of a particular block-compressed format have a corresponding width, height, and depth defining the dimensions of these elements in units of texels, and a size in bytes of the encoding in memory.

**Constant Integral Expressions**
A SPIR-V constant instruction whose type is OpTypeInt. See *Constant Instruction* in section 2.2.1 “Instructions” of the *Khronos SPIR-V Specification*.

**Coverage Index**
The index of a sample in the coverage mask.

**Coverage Mask**
A bitfield associated with a fragment representing the samples that were determined to be covered based on the result of rasterization, and then subsequently modified by fragment operations or the fragment shader.

**Cull Distance**
A built-in output from pre-rasterization shader stages defining a cull half-space where the primitive is rejected if all vertices have a negative value for the same cull distance.

**Cull Volume**
The intersection of the view volume with all cull half-spaces.
Decoration (SPIR-V)
Auxiliary information such as built-in variables, stream numbers, invariance, interpolation type, relaxed precision, etc., added to variables or structure-type members through decorations.

Deprecated (feature)
A feature is deprecated if it is no longer recommended as the correct or best way to achieve its intended purpose.

Depth/Stencil Attachment
A subpass attachment point, or image view, that is the target of depth and/or stencil test operations and writes.

Depth/Stencil Format
A VkFormat that includes depth and/or stencil components.

Depth/Stencil Image (or ImageView)
A VkImage (or VkImageView) with a depth/stencil format.

Depth/Stencil Resolve Attachment
A subpass attachment point, or image view, that is the target of a multisample resolve operation from the corresponding depth/stencil attachment at the end of the subpass.

Derivative Group
A set of fragment shader invocations that cooperate to compute derivatives, including implicit derivatives for sampled image operations.

Descriptor
Information about a resource or resource view written into a descriptor set that is used to access the resource or view from a shader.

Descriptor Binding
An entry in a descriptor set layout corresponding to zero or more descriptors of a single descriptor type in a set. Defined by a VkDescriptorSetLayoutBinding structure.

Descriptor Pool
An object that descriptor sets are allocated from, and that owns the storage of those descriptor sets. Descriptor pools aid multithreaded performance by enabling different threads to use different allocators, without internal synchronization on each use. Represented by a VkDescriptorPool object.

Descriptor Set
An object that resource descriptors are written into via the API, and that can be bound to a command buffer such that the descriptors contained within it can be accessed from shaders. Represented by a VkDescriptorSet object.

Descriptor Set Layout
An object defining the set of resources (types and counts) and their relative arrangement (in the binding namespace) within a descriptor set. Used when allocating descriptor sets and when
creating pipeline layouts. Represented by a `VkDescriptorSetLayout` object.

**Device**

The processor(s) and execution environment that perform tasks requested by the application via the Vulkan API.

**Device Group**

A set of physical devices that support accessing each other’s memory and recording a single command buffer that can be executed on all the physical devices.

**Device Index**

A zero-based integer that identifies one physical device from a logical device. A device index is valid if it is less than the number of physical devices in the logical device.

**Device Mask**

A bitmask where each bit represents one device index. A device mask value is valid if every bit that is set in the mask is at a bit position that is less than the number of physical devices in the logical device.

**Device Memory**

Memory accessible to the device. Represented by a `VkDeviceMemory` object.

**Device-Level Command**

Any command that is dispatched from a logical device, or from a child object of a logical device.

**Device-Level Functionality**

All device-level commands and objects, and their structures, enumerated types, and enumerants.

**Device-Level Object**

Logical device objects and their child objects. For example, `VkDevice`, `VkQueue`, and `VkCommandBuffer` objects are device-level objects.

**Device-Local Memory**

Memory that is connected to the device, and may be more performant for device access than host-local memory.

**Direct Drawing Commands**

*Drawing commands* that take all their parameters as direct arguments to the command (and not sourced via structures in buffer memory as the *indirect drawing commands*). Includes `vkCmdDraw` and `vkCmdDrawIndexed`.

**Disjoint**

*Disjoint planes* are *image planes* to which memory is bound independently. A *disjoint image* consists of multiple *disjoint planes*, and is created with the `VK_IMAGE_CREATE_DISJOINT_BIT` bit set.

**Dispatchable Command**

A non-global command. The first argument to each dispatchable command is a dispatchable
handle type.

**Dispatchable Handle**
A handle of a pointer handle type which may be used by layers as part of intercepting API commands.

**Dispatching Commands**
Commands that provoke work using a compute pipeline. Includes `vkCmdDispatch` and `vkCmdDispatchIndirect`.

**Drawing Commands**
Commands that provoke work using a graphics pipeline. Includes `vkCmdDraw`, `vkCmdDrawIndexed`, `vkCmdDrawIndirectCount`, `vkCmdDrawIndexedIndirectCount`, `vkCmdDrawIndirect`, and `vkCmdDrawIndexedIndirect`.

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

**Dynamic Storage Buffer**
A storage buffer whose offset is specified each time the storage buffer is bound to a command buffer via a descriptor set.

**Dynamic Uniform Buffer**
A uniform buffer whose offset is specified each time the uniform buffer is bound to a command buffer via a descriptor set.

**Dynamically Uniform**
See *Dynamically Uniform* in section 2.2 “Terms” of the Khronos SPIR-V Specification.

**Element**
Arrays are composed of multiple elements, where each element exists at a unique index within that array. Used primarily to describe data passed to or returned from the Vulkan API.

**Explicitly-Enabled Layer**
A layer enabled by the application by adding it to the enabled layer list in `vkCreateInstance` or `vkCreateDevice`.

**Event**
A synchronization primitive that is signaled when execution of previous commands completes through a specified set of pipeline stages. Events can be waited on by the device and polled by the host. Represented by a `VkEvent` object.

**Executable State (Command Buffer)**
A command buffer that has ended recording commands and can be executed. See also Initial State and Recording State.
Execution Dependency
A dependency that guarantees that certain pipeline stages’ work for a first set of commands has completed execution before certain pipeline stages’ work for a second set of commands begins execution. This is accomplished via pipeline barriers, subpass dependencies, events, or implicit ordering operations.

Execution Dependency Chain
A sequence of execution dependencies that transitively act as a single execution dependency.

Explicit chroma reconstruction
An implementation of sampler Y’C_aC_bC_r conversion which reconstructs reduced-resolution chroma samples to luma resolution and then separately performs texture sample interpolation. This is distinct from an implicit implementation, which incorporates chroma sample reconstruction into texture sample interpolation.

Extension Scope
The set of objects and commands that can be affected by an extension. Extensions are either device scope or instance scope.

Extending Structure
A structure type which may appear in the pNext chain of another structure, extending the functionality of the other structure. Extending structures may be defined by either core API versions or extensions.

External Handle
A resource handle which has meaning outside of a specific Vulkan device or its parent instance. External handles may be used to share resources between multiple Vulkan devices in different instances, or between Vulkan and other APIs. Some external handle types correspond to platform-defined handles, in which case the resource may outlive any particular Vulkan device or instance and may be transferred between processes, or otherwise manipulated via functionality defined by the platform for that handle type.

External synchronization
A type of synchronization required of the application, where parameters defined to be externally synchronized must not be used simultaneously in multiple threads.

Facingness (Polygon)
A classification of a polygon as either front-facing or back-facing, depending on the orientation (winding order) of its vertices.

Facingness (Fragment)
A fragment is either front-facing or back-facing, depending on the primitive it was generated from. If the primitive was a polygon (regardless of polygon mode), the fragment inherits the facingness of the polygon. All other fragments are front-facing.

Fence
A synchronization primitive that is signaled when a set of batches or sparse binding operations complete execution on a queue. Fences can be waited on by the host. Represented by a VkFence
object.

Flat Shading
A property of a vertex attribute that causes the value from a single vertex (the provoking vertex) to be used for all vertices in a primitive, and for interpolation of that attribute to return that single value unaltered.

Format Features
A set of features from VkFormatFeatureFlagBits that a VkFormat is capable of using for various commands. The list is determined by factors such as VkImageTiling.

Fragment
A rectangular framebuffer region with associated data produced by rasterization and processed by fragment operations including the fragment shader.

Fragment Area
The width and height, in pixels, of a fragment.

Fragment Input Attachment Interface
Variables with UniformConstant storage class and a decoration of InputAttachmentIndex that are statically used by a fragment shader’s entry point, which receive values from input attachments.

Fragment Output Interface
A fragment shader entry point’s variables with Output storage class, which output to color and/or depth/stencil attachments.

Framebuffer
A collection of image views and a set of dimensions that, in conjunction with a render pass, define the inputs and outputs used by drawing commands. Represented by a VkFramebuffer object.

Framebuffer Attachment
One of the image views used in a framebuffer.

Framebuffer Coordinates
A coordinate system in which adjacent pixels' coordinates differ by 1 in x and/or y, with (0,0) in the upper left corner and pixel centers at half-integers.

Framebuffer-Space
Operating with respect to framebuffer coordinates.

Framebuffer-Local
A framebuffer-local dependency guarantees that only for a single framebuffer region, the first set of operations happens-before the second set of operations.

Framebuffer-Global
A framebuffer-global dependency guarantees that for all framebuffer regions, the first set of operations happens-before the second set of operations.
Framebuffer Region
A framebuffer region is a set of sample (x, y, layer, sample) coordinates that is a subset of the entire framebuffer.

Front-Facing
See Facingness.

Full Compatibility
A given version of the API is fully compatible with another version if an application, relying only on valid behavior and functionality defined by either of those specifications, is able to correctly run against each version without any modification. This assumes no active attempt by that application to not run when it detects a different version.

Global Command
A Vulkan command for which the first argument is not a dispatchable handle type.

Global Workgroup
A collection of local workgroups dispatched by a single dispatching command.

Handle
An opaque integer or pointer value used to refer to a Vulkan object. Each object type has a unique handle type.

Happen-after, happens-after
A transitive, irreflexive and antisymmetric ordering relation between operations. An execution dependency with a source of A and a destination of B enforces that B happens-after A. The inverse relation of happens-before.

Happen-before, happens-before
A transitive, irreflexive and antisymmetric ordering relation between operations. An execution dependency with a source of A and a destination of B enforces that A happens-before B. The inverse relation of happens-after.

Helper Invocation
A fragment shader invocation that is created solely for the purposes of evaluating derivatives for use in non-helper fragment shader invocations, and which does not have side effects.

Host
The processor(s) and execution environment that the application runs on, and that the Vulkan API is exposed on.

Host Mapped Device Memory
Device memory that is mapped for host access using vkMapMemory.

Host Mapped Foreign Memory
Memory owned by a foreign device that is mapped for host access.
Host Memory
Memory not accessible to the device, used to store implementation data structures.

Host-Accessible Subresource
A buffer, or a linear image subresource in either the `VK_IMAGE_LAYOUT_PREINITIALIZED` or `VK_IMAGE_LAYOUT_GENERAL` layout. Host-accessible subresources have a well-defined addressing scheme which can be used by the host.

Host-Local Memory
Memory that is not local to the device, and may be less performant for device access than device-local memory.

Host-Visible Memory
Device memory that can be mapped on the host and can be read and written by the host.

Identically Defined Objects
Objects of the same type where all arguments to their creation or allocation functions, with the exception of `pAllocator`, are

1. Vulkan handles which refer to the same object or
2. identical scalar or enumeration values or
3. Host pointers which point to an array of values or structures which also satisfy these three constraints.

Image
A resource that represents a multi-dimensional formatted interpretation of device memory. Represented by a `VkImage` object.

Image Subresource
A specific mipmap level and layer of an image.

Image Subresource Range
A set of image subresources that are contiguous mipmap levels and layers.

Image View
An object that represents an image subresource range of a specific image, and state controlling how the contents are interpreted. Represented by a `VkImageView` object.

Immutable Sampler
A sampler descriptor provided at descriptor set layout creation time, and that is used for that binding in all descriptor sets allocated from the layout, and cannot be changed.

Implicit chroma reconstruction
An implementation of sampler YC_bC_r 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 VkImage created with VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT and whose Linux DRM format modifier is DRM_FORMAT_MOD_LINEAR

A resource is non-linear if it is one of the following:

- a VkImage created with VK_IMAGE_TILING_OPTIMAL
- a VkImage created with VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT and whose Linux DRM format modifier is not DRM_FORMAT_MOD_LINEAR

Linux DRM Format Modifier

A 64-bit, vendor-prefixed, semi-opaque unsigned integer describing vendor-specific details of an image's memory layout. In Linux graphics APIs, modifiers are commonly used to specify the memory layout of externally shared images. An image has a modifier if and only if it is created with tiling equal to VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT. For more details, refer to the appendix for extension VK_EXT_image_drm_format_modifier.

Local Workgroup

A collection of compute shader invocations invoked by a single 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 \texttt{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 \textbf{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 \textbf{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 \textbf{can be} made.

Memory Type

An index used to select a set of memory properties (e.g. mappable, cached) for a device memory allocation.

Minimum Miplevel Size

The smallest size that is permitted for a mipmap level. For conventional images this is 1x1x1. See Image Miplevel Sizing.

Mip Tail Region

The set of mipmap levels of a sparse residency texture that are too small to fill a sparse block,
and that **must** all be bound to memory collectively and opaquely.

**Multi-planar**

A *multi-planar format* (or “planar format”) is an image format consisting of more than one *plane*, identifiable with a `_2PLANE` or `_3PLANE` component to the format name and listed in Formats requiring sampler Y’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 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 required to represent the texels in the image and with a contiguous mapping of coordinates to bound memory. Most images consist only of a single plane, but some formats spread the components across multiple image planes. The host-accessible properties of each image plane are accessed in a linear layout using vkGetImageSubresourceLayout. If a multi-planar image is created with the VK_IMAGE_CREATE_DISJOINT_BIT bit set, the image is described as disjoint, and its planes are therefore bound to memory independently.

**Point Sampling (Rasterization)**

A rule that determines whether a fragment sample location is covered by a polygon primitive by testing whether the sample location is in the interior of the polygon in framebuffer-space, or on the boundary of the polygon according to the tie-breaking rules.

**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.

**Presentable image**

A VkImage object obtained from a VkSwapchainKHR used to present to a VkSurfaceKHR object.

**Preserve Attachment**

One of a list of attachments in a subpass description that is not read or written by the subpass, but that is read or written on earlier and later subpasses and whose contents must be preserved through this subpass.

**Primary Command Buffer**

A command buffer that can execute secondary command buffers, and can be submitted directly to a queue.

**Primitive 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.

Retired Swapchain
A swapchain that has been used as the oldSwapchain parameter to vkCreateSwapchainKHR. Images cannot be acquired from a retired swapchain, however images that were acquired (but not presented) before the swapchain was retired can be presented.
**Sample 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.

**Shading Rate**

The ratio of the number of fragment shader invocations generated in a fully covered framebuffer region to the size (in pixels) of that region.
**Shared presentable image**

A presentable image created from a swapchain with `VkPresentModeKHR` set to either `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR` or `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`.

**Side Effect**

A store to memory or atomic operation on memory from a shader invocation.

**Single 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. 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.

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

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 Ksiezykowski, 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)
• Aidan Fabius, Core Avionics & Industrial Inc. (version 1.2)
• Alan Baker, Google (versions 1.1, 1.2)
• 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 (version 1.2)
• Alex Walters, Imagination Technologies (version 1.2)
• Alexander Galazin, Arm (versions 1.0, 1.1, 1.2)
• Allen Hux, Intel (version 1.0)
• Alon Or-bach, Samsung Electronics (versions 1.0, 1.1, 1.2) (WSI technical sub-group chair)
• Anastasia Stulova, Arm (version 1.2)
• Andreas Vasilakis, Think Silicon (version 1.2)
• Andres Gomez, Igalia (version 1.1)
• Andrew Cox, Samsung Electronics (version 1.0)
• Andrew Garrard, Samsung Electronics (versions 1.0, 1.1, 1.2) (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 (version 1.2)
• Antoine Labour, Google (versions 1.0, 1.1)
• Aras Pranckevičius, Unity Technologies (version 1.0)
• Ashwin Kolhe, NVIDIA (version 1.0)
• Baldur Karlsson, Independent (versions 1.1, 1.2)
• David Miller, Miller & Mattson (versions 1.0, 1.1) (Vulkan reference card)
• David Neto, Google (versions 1.0, 1.1, 1.2)
• David Wilkinson, AMD (version 1.2)
• David Yu, Pixar (version 1.0)
• Dejan Mircevski, Google (version 1.1)
• Dominik Witczak, AMD (versions 1.0, 1.1)
• Donald Scorgie, Imagination Technologies (version 1.2)
• Dzmitry Malyshau, Mozilla (versions 1.1, 1.2)
• Ed Hutchins, Oculus (version 1.2)
• Emily Stearns, Khronos (version 1.2)
• 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)
• Graham Connor, Imagination Technologies (version 1.0)
• Graham Sellers, AMD (versions 1.0, 1.1)
• Greg Fischer, LunarG (version 1.1)
• Hai Nguyen, Google (version 1.2)
• Hans-Kristian Arntzen, Arm (versions 1.1, 1.2)
• Henri Verbeet, Codeweavers (version 1.2)
• 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)
• James Hughes, Oculus VR (version 1.0)
• James Jones, NVIDIA (versions 1.0, 1.1, 1.2)
• James Riordon, Khronos (version 1.2)
• Jan Hermes, Continental Corporation (versions 1.0, 1.1)
• Jan-Harald Fredriksen, Arm (versions 1.0, 1.1, 1.2)
• Jason Ekstrand, Intel (versions 1.0, 1.1, 1.2)
• Jean-François Roy, Google (versions 1.1, 1.2)
• Jeff Bolz, NVIDIA (versions 1.0, 1.1, 1.2)
• Jeff Juliano, NVIDIA (versions 1.0, 1.1, 1.2)
• Jeff Leger, Qualcomm Technologies, Inc. (version 1.1)
• Jeff Vigil, Samsung Electronics (versions 1.0, 1.1, 1.2)
• 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)
• Jesse Hall, Google (versions 1.0, 1.1, 1.2)
• Joe Davis, Samsung Electronics (version 1.1)
• Johannes van Waveren, Oculus VR (versions 1.0, 1.1)
• John Kessenich, Google (versions 1.0, 1.1, 1.2) (SPIR-V and GLSL for Vulkan specification author)
• John McDonald, Valve Software (versions 1.0, 1.1, 1.2)
• John Zulauf, LunarG (versions 1.1, 1.2)
• Jon Ashburn, LunarG (version 1.0)
• Jon Leech, Independent (versions 1.0, 1.1, 1.2) (XML toolchain, normative language, release wrangler)
• Jonas Gustavsson, Samsung Electronics (versions 1.0, 1.1)
• Jonas Meyer, Epic Games (version 1.2)
• Jonathan Hamilton, Imagination Technologies (version 1.0)
• Jordan Justen, Intel (version 1.1)
• 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)
• 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)
• Kris Rose, Khronos (version 1.2)
• Kristian Kristensen, Intel (versions 1.0, 1.1)
• Krzysztof Iwanicki, Samsung Electronics (version 1.0)
• Larry Seiler, Intel (version 1.0)
• Lauri Ilola, Nokia (version 1.1)
• Lei Zhang, Google (version 1.2)
• Lenny Komow, LunarG (versions 1.1, 1.2)
• Lionel Landwerlin, Intel (versions 1.1, 1.2)
• Liz Maitral, Khronos (version 1.2)
• Lutz Latta, Lucasfilm (version 1.0)
• Maciej Jesionowski, AMD (version 1.1)
• Mais Alnasser, AMD (version 1.1)
• Marcin Rogucki, Mobica (version 1.1)
• Maria Rovatsou, Codeplay Software Ltd. (version 1.0)
• Mark Callow, Independent (versions 1.0, 1.1, 1.2)
• Mark Kilgard, NVIDIA (versions 1.1, 1.2)
• Mark Lobodzinski, LunarG (versions 1.0, 1.1, 1.2)
• Mark Young, LunarG (version 1.1)
• Markus Tavenrath, NVIDIA (version 1.1)
• Mateusz Przybylski, Intel (version 1.0)
• Mathias Heyer, NVIDIA (versions 1.0, 1.1)
• Mathias Schott, NVIDIA (versions 1.0, 1.1)
• Matt Netsch, Qualcomm Technologies, Inc. (version 1.1)
• Matthäus Chajdas, AMD (versions 1.1, 1.2)
• Maurice Ribble, Qualcomm Technologies, Inc. (versions 1.0, 1.1)
• Maxim Lukyanov, Samsung Electronics (version 1.0)
• 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 (version 1.0)
• 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)
• Minyoung Son, Samsung Electronics (version 1.0)
• Mitch Singer, AMD (versions 1.0, 1.1, 1.2)
• Mythri Venugopal, Samsung Electronics (version 1.0)
• Naveen Leekha, Google (version 1.0)
• Neil Henning, AMD (versions 1.0, 1.1, 1.2)
• Neil Hickey, Arm (version 1.2)
• Neil Trevett, NVIDIA (versions 1.0, 1.1, 1.2)
• 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)
• 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)
• Pierre Boudier, NVIDIA (versions 1.0, 1.1, 1.2)
• Pierre-Loup Griffais, Valve Software (versions 1.0, 1.1, 1.2)
• Piers Daniell, NVIDIA (versions 1.0, 1.1, 1.2)
• Piotr Bialecki, Intel (version 1.0)
• Prabindh Sundareson, Samsung Electronics (version 1.0)
• Pyry Haulos, Google (versions 1.0, 1.1) (Vulkan conformance test subcommittee chair)
• Rajeev Rao, Qualcomm (version 1.2)
• Ralph Potter, Codeplay Software Ltd. (versions 1.1, 1.2)
• Ray Smith, Arm (versions 1.0, 1.1, 1.2)
• Richard Huddy, Samsung Electronics (version 1.2)
• Rob Barris, NVIDIA (version 1.1)
• Rob Stepinski, Transgaming (version 1.0)
• Robert Simpson, Qualcomm Technologies, Inc. (versions 1.0, 1.1)
• Rolando Caloca Olivares, Epic Games (versions 1.0, 1.1, 1.2)
• Roy Ju, Mediatek (version 1.0)
• Rufus Hamade, Imagination Technologies (version 1.0)
• Ruihao Zhang, Qualcomm Technologies, Inc. (versions 1.1, 1.2)
• 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)
• Slawomir Cygan, Intel (versions 1.0, 1.1)
• Slawomir Grajewski, Intel (versions 1.0, 1.1)
• Sorel Bosan, AMD (version 1.1)
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 and 1.2 was provided by Khronos staff including Angela Cheng, Ann Thorsnes, Blaine Kohl, Dominic Agoro-Ombaka, Emily Stearns, Jeff Phillips, Lisie Aartsen, and Liz Maitral; and by Alex Crabb of Caster Communications.

Administrative support for Vulkan 1.0 was provided by Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, Kathleen Mattson and Michelle Clark of Gold Standard Group.

Technical support was provided by James Riordon, site administration of Khronos.org and OpenGL.org.