# Table of Contents

1. Introduction ................................................................. 2
   1.1. What is OpenXR? ...................................................... 2
   1.2. The Programmer's View of OpenXR ............................. 2
   1.3. The Implementor's View of OpenXR ........................... 2
   1.4. Our View of OpenXR ............................................. 3
   1.5. Filing Bug Reports .............................................. 3
   1.6. Document Conventions .......................................... 3

2. Fundamentals .................................................................... 5
   2.1. API Version Numbers and Semantics ......................... 5
   2.2. String Encoding ................................................... 7
   2.3. Threading Behavior ............................................... 7
   2.4. Multiprocessing Behavior ...................................... 8
   2.5. Runtime .............................................................. 8
   2.6. Extensions .......................................................... 8
   2.7. API Layers .......................................................... 9
   2.8. Return Codes ....................................................... 16
   2.9. Handles .............................................................. 22
   2.10. Object Handle Types ............................................. 22
   2.11. Buffer Size Parameters ......................................... 23
   2.12. Time ................................................................. 25
   2.13. Duration ............................................................ 26
   2.14. Prediction Time Limits .......................................... 27
   2.15. Colors .............................................................. 27
   2.16. Coordinate System ............................................... 28
   2.17. Common Object Types ........................................... 31
   2.18. Angles ............................................................... 34
   2.19. Boolean Values .................................................... 35
   2.20. Events .............................................................. 35
   2.21. System resource lifetime ....................................... 40

3. API Initialization ........................................................... 41
   3.1. Exported Functions ............................................... 41
   3.2. Function Pointers .................................................. 41

4. Instance ........................................................................... 45
   4.1. API Layers and Extensions .................................... 45
   4.2. Instance Lifecycle .................................................. 51
   4.3. Instance Information .............................................. 56
4.4. Platform-Specific Instance Creation ............................................................. 58
4.5. Instance Enumerated Type String Functions ................................................. 59
5. System ............................................................................................................. 62
  5.1. Form Factors ............................................................................................... 62
  5.2. Getting the XrSystemId .............................................................................. 63
  5.3. System Properties ....................................................................................... 66
6. Path Tree and Semantic Paths ........................................................................ 70
  6.1. Path Atom Type ......................................................................................... 70
  6.2. Well-Formed Path Strings .......................................................................... 72
  6.3. Reserved Paths ............................................................................................ 75
  6.4. Interaction Profile Paths ............................................................................. 80
7. Spaces ............................................................................................................ 88
  7.1. Reference Spaces ....................................................................................... 89
  7.2. Action Spaces ............................................................................................. 94
  7.3. Space Lifecycle ........................................................................................... 95
  7.4. Locating Spaces ......................................................................................... 102
8. View Configurations ....................................................................................... 110
  8.1. Primary View Configurations .................................................................... 110
  8.2. View Configuration API .......................................................................... 111
  8.3. Example View Configuration Code ............................................................ 118
9. Session ........................................................................................................... 121
  9.1. Session Lifecycle ....................................................................................... 121
  9.2. Session Creation ....................................................................................... 123
  9.3. Session Control ......................................................................................... 127
  9.4. Session States ........................................................................................... 131
10. Rendering ..................................................................................................... 136
  10.1. Swapchain Image Management ............................................................... 136
  10.2. View and Projection State ....................................................................... 153
  10.3. Frame Synchronization .......................................................................... 158
  10.4. Frame Submission .................................................................................. 162
11. Input and Haptics ......................................................................................... 180
  11.1. Action Overview ...................................................................................... 180
  11.2. Action Sets ............................................................................................... 182
  11.3. Creating Actions ..................................................................................... 186
  11.4. Suggested Bindings .................................................................................. 191
  11.5. Reading Input Action State ..................................................................... 201
  11.6. Output Actions and Haptics .................................................................... 212
  11.7. Input Action State Synchronization ......................................................... 217
Copyright (c) 2017-2022, The Khronos Group Inc. All rights reserved.

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. 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.

Vulkan and Khronos are registered trademarks of The Khronos Group Inc. 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 1. Introduction

This chapter is informative except for the section on Normative Terminology.

This document, referred to as the "OpenXR Specification" or just the "Specification" hereafter, describes OpenXR: what it is, how it acts, and what is required to implement it. We assume that the reader has a basic understanding of computer graphics and the technologies involved in virtual and augmented reality. This means familiarity with the essentials of computer graphics algorithms and terminology, modern GPUs (Graphic Processing Units), tracking technologies, head mounted devices, and input modalities.

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

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

1.1. What is OpenXR?

OpenXR is an API (Application Programming Interface) for XR applications. XR refers to a continuum of real-and-virtual combined environments generated by computers through human-machine interaction and is inclusive of the technologies associated with virtual reality (VR), augmented reality (AR) and mixed reality (MR). OpenXR is the interface between an application and an in-process or out-of-process "XR runtime system", or just "runtime" hereafter. The runtime may handle such functionality as frame composition, peripheral management, and raw tracking information.

Optionally, a runtime may support device layer plugins which allow access to a variety of hardware across a commonly defined interface.

1.2. The Programmer’s View of OpenXR

To the application programmer, OpenXR is a set of functions that interface with a runtime to perform commonly required operations such as accessing controller/peripheral state, getting current and/or predicted tracking positions, and submitting rendered frames.

A typical OpenXR program begins with a call to create an instance which establishes a connection to a runtime. Then a call is made to create a system which selects for use a physical display and a subset of input, tracking, and graphics devices. Subsequently a call is made to create buffers into which the application will render one or more views using the appropriate graphics APIs for the platform. Finally calls are made to create a session and begin the application's XR rendering loop.

1.3. The Implementor’s View of OpenXR

To the runtime implementor, OpenXR is a set of functions that control the operation of the XR system and establishes the lifecycle of a XR application.
The implementor's task is to provide a software library on the host which implements the OpenXR API, while mapping the work for each OpenXR function to the graphics hardware as appropriate for the capabilities of the device.

### 1.4. Our View of OpenXR

We view OpenXR as a mechanism for interacting with VR/AR/MR systems in a platform-agnostic way.

We expect this model to result in a specification that satisfies the needs of both programmers and runtime implementors. It does not, however, necessarily provide a model for implementation. A runtime implementation **must** produce results conforming to those produced by the specified methods, but **may** carry out particular procedures in ways that are more efficient than the one specified.

### 1.5. Filing Bug Reports

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

https://github.com/KhronosGroup/OpenXR-Docs

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

### 1.6. Document Conventions

The OpenXR 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.

#### 1.6.1. Normative Terminology

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


**must**

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

**should**

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

**may**

This word, or the adjective **optional**, means that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item.

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

**can**

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

**cannot**

This word means that the particular behavior described is not achievable by an application, for example, an entry point does not exist.

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 runtime.
Chapter 2. Fundamentals

2.1. API Version Numbers and Semantics

Multi-part version numbers are used in several places in the OpenXR API.

```c
typedef uint64_t XrVersion;
```

In each such use, the API major version number, minor version number, and patch version number are packed into a 64-bit integer, referred to as `XrVersion`, as follows:

**Version Numbers**

- The major version number is a 16-bit integer packed into bits 63-48.
- The minor version number is a 16-bit integer packed into bits 47-32.
- The patch version number is a 32-bit integer packed into bits 31-0.

Differences in any of the version numbers indicate a change to the API, with each part of the version number indicating a different scope of change, as follows.

Note

The rules below apply to OpenXR versions 1.0 or later. Prerelease versions of OpenXR may use different rules for versioning.

A difference in patch version numbers indicates that some usually small part of the specification or header has been modified, typically to fix a bug, and may have an impact on the behavior of existing functionality. Differences in the patch version number must affect neither full compatibility nor backwards compatibility between two versions, nor may it add additional interfaces to the API.

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

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

The following table attempts to detail the changes that may occur versus when they must not be updated (indicating the next version number must be updated instead) during an update to any of the major, minor, or patch version numbers:

**Table 1. Scenarios Which May Cause a Version Change**

<table>
<thead>
<tr>
<th>Reason</th>
<th>Major Version</th>
<th>Minor Version</th>
<th>Patch Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extensions Added/Removed*</td>
<td>may</td>
<td>may</td>
<td>may</td>
</tr>
<tr>
<td>Spec-Optional Behavior Changed*</td>
<td>may</td>
<td>may</td>
<td>must not</td>
</tr>
<tr>
<td>Spec Required Behavior Changed*</td>
<td>may</td>
<td>may</td>
<td>must not</td>
</tr>
<tr>
<td>Core Interfaces Added*</td>
<td>may</td>
<td>may</td>
<td>must not</td>
</tr>
<tr>
<td>Weak Deprecation*</td>
<td>may</td>
<td>may</td>
<td>must not</td>
</tr>
<tr>
<td>Strong Deprecation*</td>
<td>may</td>
<td>must not</td>
<td>must not</td>
</tr>
<tr>
<td>Core Interfaces Changed/Removed*</td>
<td>may</td>
<td>must not</td>
<td>must not</td>
</tr>
</tbody>
</table>

In the above table, the following identify the various cases in detail:

**Extensions Added/Removed**
An extension may be added or removed with a change at this patch level.

**Specification-Optional Behavior Changed**
Some optional behavior laid out in this specification has changed. Usually this will involve a change in behavior that is marked with the normatives *should* or *may*. For example, a runtime that previously did not validate a particular use case may now begin validating that use case.

**Specification-Required Behavior Changed**
A behavior of runtimes that is required by this specification *may* have changed. For example, a previously optional validation *may* now have become mandatory for runtimes.

**Core Interfaces Added**
New interfaces *may* have been added to this specification (and to the OpenXR header file) in revisions at this level.
**Weak Deprecation**  
An interface may have been weakly deprecated at this level. This may happen if there is now a better way to accomplish the same thing. Applications making this call should behave the same as before the deprecation, but following the new path may be more performant, less latent, or otherwise yield better results. It is possible that some runtimes may choose to give run-time warnings that the feature has been weakly deprecated and will likely be strongly deprecated or removed in the future.

**Strong Deprecation**  
An interface may have been strongly deprecated at this level. This means that the interface must still exist (so applications that are compiled against it will still run) but it may now be a no-op, or it may be that its behavior has been significantly changed. It may be that this functionality is no longer necessary, or that its functionality has been subsumed by another call. This should not break an application, but some behavior may be different or unanticipated.

**Interfaces Changed/Removed**  
An interface may have been changed — with different parameters or return types — at this level. An interface or feature may also have been removed entirely. It is almost certain that rebuilding applications will be required.

### 2.2. String Encoding

This API uses strings as input and output for some functions. Unless otherwise specified, all such strings are **NULL** terminated UTF-8 encoded case-sensitive character arrays.

### 2.3. Threading Behavior

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

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

For all functions which destroy an object handle, the application must externally synchronize the object handle parameter and any child handles.
2.4. Multiprocessing Behavior

The OpenXR API does not explicitly recognize nor require support for multiple processes using the runtime simultaneously, nor does it prevent a runtime from providing such support.

2.5. Runtime

An OpenXR runtime is software which implements the OpenXR API. There may be more than one OpenXR runtime installed on a system, but only one runtime can be active at any given time.

2.6. Extensions

OpenXR is an extensible API that can grow through the addition of new features. Similar to other Khronos APIs, extensions can be used to expose new OpenXR functions or modify the behavior of existing OpenXR functions. Extensions are optional and therefore must be enabled by the application before the extended functionality is made available. Because extensions are optional, they may be implemented only on a subset of runtimes, graphics platforms, or operating systems. Therefore, an application should first query which extensions are available before enabling.

The application queries the available list of extensions using the xrEnumerateInstanceExtensionProperties function. Once an application determines which target extensions are supported, it can enable some subset of them during the call to xrCreateInstance.

OpenXR extensions have unique names that convey information about what functionality is provided.
The names have the following format:

**Extension Name Formatting**

- The prefix "XR_" to identify this as an OpenXR extension
- A string identifier for the vendor tag, which corresponds to the company or group exposing the extension. The vendor tag **must** use only uppercase letters and decimal digits. Some examples include:
  - "KHR" for Khronos extensions, supported by multiple vendors.
  - "EXT" for non-Khronos extensions supported by multiple vendors.
- An underscore ".".
- A string uniquely identifying the extension. The string is a compound of substrings which **must** use only lower case letters and decimal digits. The substrings are delimited with single underscores.

For example: **XR_KHR_composition_layer_cube** is an OpenXR extension created by the Khronos (KHR) OpenXR Working Group to support cube composition layers.

The public list of available extensions known at the time of this specification being generated appears in the [List of Extensions](#) appendix at the end of this document.

### 2.7. API Layers

OpenXR is designed to be a layered API, which means that a user or application **may** insert API layers between the application and the runtime implementation. These API layers provide additional functionality by intercepting OpenXR functions from the layer above and then performing different operations than would otherwise be performed without the layer. In the simplest cases, the layer simply calls the next layer down with the same arguments, but a more complex layer may implement API functionality that is not present in the layers or runtime below it. This mechanism is essentially an architected "function shimming" or "intercept" feature that is designed into OpenXR and meant to replace more informal methods of "hooking" API calls.

#### 2.7.1. Examples of API Layers

**Validation Layer**

The layered API approach employed by OpenXR allows for the expensive validation of correct API usage to be implemented in a "validation" layer. This layer allows the application developer to develop their application with the validation layer active to ensure that the application is using the API correctly. The validation layer confirms that the application has set up object state correctly, has provided the required data for each function, ensures that required resources are available, etc. If the validation layer detects a problem, it issues an error message that can be logged or captured by the
application via a callback. After the developer has determined that the application is correct, they turn off the validation layer to allow the application to run in a production environment without repeatedly incurring the validation expense.

**API Logging Layer**

Another example of an API layer is an API logging layer that simply serializes all the API calls to an output sink in a text format, including printing out argument values and structure contents.

**API Trace Layer**

A related API trace layer produces a trace file that contains all the information provided to the API so that the trace file can be played back by a replay program.

### 2.7.2. Naming API Layers

To organize API layer names and prevent collisions in the API layer name namespace, API layers **must** be named using the following convention:

```
XR_APILAYER_<VENDOR-TAG>_short_name
```

Vendors are responsible for registering a vendor tag with the OpenXR working group and just like for implementors, they must maintain their vendor namespace.

Example of an API layer name produced by the Acme company for the "check best practices" API layer:

```
XR_APILAYER_ACME_check_best_practices
```

### 2.7.3. Activating API Layers

**Application Activation**

Applications **can** determine the API layers that are available to them by calling the `xrEnumerateApiLayerProperties` function to obtain a list of available API layers. Applications then **can** select the desired API layers from this list and provide them to the `xrCreateInstance` function when creating an instance.

**System Activation**

Application users or users performing roles such as system integrator or system administrator **may** configure a system to activate API layers without involvement from the applications. These platform-dependent steps **may** include the installation of API layer-related files, setting environment variables, or other platform-specific operations. The options that are available for configuring the API layers in this manner are also dependent on the platform and/or runtime.
2.7.4. API Layer Extensions

API layers **may** implement OpenXR functions that may or may not be supported by the underlying runtime. In order to expose these new features, the API layer must expose this functionality in the form of an OpenXR **extension**. It **must** not expose new OpenXR functions without an associated extension.

For example, an OpenXR API-logging API layer might expose an API function to allow the application to turn logging on for only a portion of its execution. Since new functions **must** be exposed through an extension, the vendor has created an extension called XR_ACME_logging_on_off to contain these new functions. The application **should** query if the API layer supports the extension and then, only if it exists, enable both the extension and the API layer by name during `xrCreateInstance`.

To find out what extensions an API layer supports, an application **must** first verify that the API layer exists on the current system by calling `xrEnumerateApiLayerProperties`. After verifying an API layer of interest exists, the application then **should** call `xrEnumerateInstanceExtensionProperties` and provide the API layer name as the first parameter. This will return the list of extensions implemented internally in that API layer.

2.7.5. Type Aliasing

Type aliasing refers to the situation in which the actual type of a element does not match the declared type. Some C and C++ compilers can be configured to assume that the actual type matches the declared type, and may be so configured by default at common optimization levels. Without this, otherwise undefined behavior may occur. This compiler feature is typically referred to as "strict aliasing," and it can usually be enabled or disabled via compiler options. The OpenXR specification does not support strict aliasing, as there are some cases in which an application intentionally provides a struct with a type that differs from the declared type. For example, `XrFrameEndInfo::layers` is an array of type `const XrCompositionLayerBaseHeader * const`. However, the array **must** be of one of the specific layer types, such as `XrCompositionLayerQuad`. Similarly, `xrEnumerateSwapchainImages` accepts an array of `XrSwapchainImageBaseHeader`, whereas the actual type passed **must** be an array of a type such as `XrSwapchainImageVulkanKHR`. For OpenXR to work correctly, the compiler **must** support the type aliasing described here.

```c
#if !defined(XR_MAY_ALIAS)
#if defined(__clang__) || (defined(__GNUC__) && (__GNUC__ > 4))
#define XR_MAY_ALIAS __attribute__((__may_alias__))
#else
#define XR_MAY_ALIAS
#endif
#endif
```

As a convenience, some types and pointers that are known at specification time to alias values of
different types have been annotated with the `XR_MAY_ALIAS` definition. If this macro is not defined before including OpenXR headers, and a new enough Clang or GCC compiler is used, it will be defined to the compiler-specific attribute annotation to inform these compilers that those pointers may alias. However, there is no guarantee that all aliasing types or pointers have been correctly marked with this macro, so thorough testing is still recommended if you choose (at your own risk) to permit your compiler to perform type-based aliasing analysis.

2.7.6. Valid Usage

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

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

Valid usage conditions should apply to a function or structure where complete information about the condition would be known during execution of an application. This is such that a validation API layer or linter can be written directly against these statements at the point they are specified.

2.7.7. Implicit Valid Usage

Some valid usage conditions apply to all functions and structures in the API, unless explicitly denoted otherwise for a specific function or structure. These conditions are considered implicit. Implicit valid usage conditions are described in detail below.

Valid Usage for Object Handles

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

**Object Handle Validity Conditions**

- it has been created or allocated by a previous, successful call to the API,
- it has not been destroyed by a previous call to the API, and
- its parent handle is also valid.

There are contexts in which an object handle is optional or otherwise unspecified. In those cases, the API uses `XR_NULL_HANDLE`, which has the integer value 0.

Valid Usage for Pointers

Any parameter that is a pointer must be a valid pointer when the specification indicates that the
runtime uses the pointer. A pointer is valid if and only if it points at memory containing values of the number and type(s) expected by the function, and all fundamental types accessed through the pointer (e.g. as elements of an array or as members of a structure) satisfy the alignment requirements of the host processor.

**Valid Usage for Enumerated Types**

Any parameter of an enumerated type must be a valid enumerant for that type. An enumerant is valid if and only if the enumerant is defined as part of the enumerated type in question.

**Valid Usage for Flags**

A collection of flags is represented by a bitmask using the type XrFlags64:

```c
typedef uint64_t XrFlags64;
```

Bitmasks are passed to many functions and structures to compactly represent options and are stored in memory defined by the XrFlags64 type. But the API does not use the XrFlags64 type directly. Instead, a Xr*Flags type is used which is an alias of the XrFlags64 type. The API also defines a set of constant bit definitions used to set the bitmasks.

Any Xr*Flags member or parameter used in the API 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 and only if:

**Bit Flag Validity**

- The bit flag is one of the constant bit definitions defined by the same Xr*Flags type as the Xr*Flags member or parameter. Valid flag values may also be defined by extensions.
- 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.

**Valid Usage for Structure Types**

Any parameter that is a structure containing a type member must have a value of type which is a valid XrStructureType value matching the type of the structure. As a general rule, the name of this value is obtained by taking the structure name, stripping the leading Xr, prefixing each capital letter with an underscore, converting the entire resulting string to upper case, and prefixing it with XR_TYPE_.

The only exceptions to this rule are API and Operating System names which are converted in a way that produces a more readable value:
Structure Type Format Exceptions

- OpenGL → _OPENGL
- OpenGL ES → _OPENGL_ES
- EGL → _EGL
- D3D → _D3D
- VULKAN → _VULKAN

Valid Usage for Structure Pointer Chains

Any structure containing a void* next member must have a value of next that is either NULL, or points to a valid structure that also contains type and next member values. The set of structures connected by next pointers is referred to as a next chain.

In order to use a structure type defined by an extension in a next chain, the proper extension must have been previously enabled during xrCreateInstance. A runtime must ignore all unrecognized structures in a next chain, including those associated with an extension that has not been enabled.

Some structures for use in a chain are described in the core OpenXR specification and are mentioned in the Member Descriptions. Any structure described in this document intended for use in a chain is mentioned in a "See also" list in the implicit valid usage of the structure they chain to. Most chained structures are associated with extensions, and are described in the base OpenXR Specification under the List of Extensions. Vendor-specific extensions may be found there as well, or may only be available from the vendor’s website or internal document repositories.

Unless otherwise specified: Chained structs which are output structs may be modified by the runtime with the exception of the type and next fields. Upon return from any function, all type and next fields in the chain must be unmodified.

Useful Base Structures

As a convenience to runtimes and layers needing to iterate through a structure pointer chain, the OpenXR API provides the following base structures:

The XrBaseInStructure structure is defined as:

typedef struct XrBaseInStructure {
    XrStructureType type;
    const struct XrBaseInStructure* next;
} XrBaseInStructure;
Member Descriptions

- `type` is the `XrStructureType` of this structure. This base structure itself has no associated `XrStructureType` value.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

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

The `XrBaseOutStructure` structure is defined as:

```c
typedef struct XrBaseOutStructure {
  XrStructureType              type;
  struct XrBaseOutStructure*   next;
} XrBaseOutStructure;
```

Member Descriptions

- `type` is the `XrStructureType` of this structure. This base structure itself has no associated `XrStructureType` value.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

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

These structures allow for some type safety and can be used by OpenXR API functions that operate on generic inputs and outputs.

Next Chain Structure Uniqueness

Applications **should** ensure that they create and insert no more than one occurrence of each type of extension structure in a given `next` chain. Other components of OpenXR (such as the OpenXR loader or an API Layer) **may** insert duplicate structures into this chain. This provides those components the ability to update a structure that appears in the `next` chain by making a modified copy of that same structure and placing the new version at the beginning of the chain. The benefit of allowing this duplication is each component is no longer required to create a copy of the entire `next` chain just to update one structure. When duplication is present, all other OpenXR components **must** process only the first instance of a structure of a given type, and then ignore all instances of a structure of that same type.
If a component makes such a structure copy, and the original structure is also used to return content, then that component must copy the necessary content from the copied structure and into the original version of the structure upon completion of the function prior to proceeding back up the call stack. This is to ensure that OpenXR behavior is consistent whether or not that particular OpenXR component is present and/or enabled on the system.

Valid Usage for Nested Structures

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

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

2.8. Return Codes

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

Return Code Categories

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

```c
typedef enum XrResult {
    XR_SUCCESS = 0,
    XR_TIMEOUT_EXPIRED = 1,
    XR_SESSION_LOSS_PENDING = 3,
    XR_EVENT_UNAVAILABLE = 4,
    XR_SPACE_BOUNDS_UNAVAILABLE = 7,
    XR_SESSION_NOT_FOCUSED = 8,
    XR_FRAME_DISCARDED = 9,
    XR_ERROR_VALIDATION_FAILURE = -1,
    XR_ERROR_RUNTIME_FAILURE = -2,
    XR_ERROR_OUT_OF_MEMORY = -3,
    XR_ERROR_API_VERSION_UNSUPPORTED = -4,
    XR_ERROR_INITIALIZATION_FAILED = -6,
    XR_ERROR_FUNCTION_UNSUPPORTED = -7,
    XR_ERROR_FEATURE_UNSUPPORTED = -8,
    XR_ERROR_EXTENSION_NOT_PRESENT = -9,

```
All return codes in the API are reported via XrResult return values.
Some common suffixes shared across many of the return codes are defined below:

- **_INVALID**: The specified handle, atom or value is formatted incorrectly, or the specified handle was never created or has been destroyed.
- **_UNSUPPORTED**: The specified handle, atom, enumerant or value is formatted correctly but cannot be used for the lifetime of this function’s parent handle.
- **_UNAVAILABLE**: The specified handle, atom, enumerant or value is supported by this function’s parent handle but not at this moment.

### Success Codes

<table>
<thead>
<tr>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XR_SUCCESS</td>
<td>Function successfully completed.</td>
</tr>
<tr>
<td>XR_TIMEOUT_EXPIRED</td>
<td>The specified timeout time occurred before the operation could complete.</td>
</tr>
<tr>
<td>XR_SESSION_LOSS_PENDING</td>
<td>The session will be lost soon.</td>
</tr>
<tr>
<td>XR_EVENT_UNAVAILABLE</td>
<td>No event was available.</td>
</tr>
<tr>
<td>XR_SPACE_BOUNDS_UNAVAILABLE</td>
<td>The space's bounds are not known at the moment.</td>
</tr>
<tr>
<td>XR_SESSION_NOT_FOCUSED</td>
<td>The session is not in the focused state.</td>
</tr>
<tr>
<td>XR_FRAME_DISCARDED</td>
<td>A frame has been discarded from composition.</td>
</tr>
</tbody>
</table>

### Error Codes

<table>
<thead>
<tr>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XR_ERROR_VALIDATION_FAILURE</td>
<td>The function usage was invalid in some way.</td>
</tr>
<tr>
<td>XR_ERROR_RUNTIME_FAILURE</td>
<td>The runtime failed to handle the function in an unexpected way that is not covered by another error result.</td>
</tr>
<tr>
<td>XR_ERROR_OUT_OF_MEMORY</td>
<td>A memory allocation has failed.</td>
</tr>
<tr>
<td>XR_ERROR_API_VERSION_UNSUPPORTED</td>
<td>The runtime does not support the requested API version.</td>
</tr>
<tr>
<td>XR_ERROR_INITIALIZATION_FAILED</td>
<td>Initialization of object could not be completed.</td>
</tr>
<tr>
<td>XR_ERROR_FUNCTION_UNSUPPORTED</td>
<td>The requested function was not found or is otherwise unsupported.</td>
</tr>
<tr>
<td>XR_ERROR_FEATURE_UNSUPPORTED</td>
<td>The requested feature is not supported.</td>
</tr>
<tr>
<td>XR_ERROR_EXTENSION_NOT_PRESENT</td>
<td>A requested extension is not supported.</td>
</tr>
<tr>
<td>XR_ERROR_LIMIT_REACHED</td>
<td>The runtime supports no more of the requested resource.</td>
</tr>
<tr>
<td>Enum</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>XR_ERROR_SIZE_INSUFFICIENT</td>
<td>The supplied size was smaller than required.</td>
</tr>
<tr>
<td>XR_ERROR_HANDLE_INVALID</td>
<td>A supplied object handle was invalid.</td>
</tr>
<tr>
<td>XR_ERROR_INSTANCE_LOST</td>
<td>The XrInstance was lost or could not be found. It will need to be destroyed</td>
</tr>
<tr>
<td></td>
<td>and optionally recreated.</td>
</tr>
<tr>
<td>XR_ERROR_SESSION_RUNNING</td>
<td>The session is already running.</td>
</tr>
<tr>
<td>XR_ERROR_SESSION_NOT_RUNNING</td>
<td>The session is not yet running.</td>
</tr>
<tr>
<td>XR_ERROR_SESSION_LOST</td>
<td>The XrSession was lost. It will need to be destroyed and optionally</td>
</tr>
<tr>
<td></td>
<td>recreated.</td>
</tr>
<tr>
<td>XR_ERROR_SYSTEM_INVALID</td>
<td>The provided XrSystemId was invalid.</td>
</tr>
<tr>
<td>XR_ERROR_PATH_INVALID</td>
<td>The provided XrPath was not valid.</td>
</tr>
<tr>
<td>XR_ERROR_PATH_COUNT_EXCEEDED</td>
<td>The maximum number of supported semantic paths has been reached.</td>
</tr>
<tr>
<td>XR_ERROR_PATH_FORMAT_INVALID</td>
<td>The semantic path character format is invalid.</td>
</tr>
<tr>
<td>XR_ERROR_PATH_UNSUPPORTED</td>
<td>The semantic path is unsupported.</td>
</tr>
<tr>
<td>XR_ERROR_LAYER_INVALID</td>
<td>The layer was NULL or otherwise invalid.</td>
</tr>
<tr>
<td>XR_ERROR_LAYER_LIMIT_EXCEEDED</td>
<td>The number of specified layers is greater than the supported number.</td>
</tr>
<tr>
<td>XR_ERROR_SWAPCHAIN_RECT_INVALID</td>
<td>The image rect was negatively sized or otherwise invalid.</td>
</tr>
<tr>
<td>XR_ERROR_SWAPCHAIN_FORMAT_UNSUPPORTED</td>
<td>The image format is not supported by the runtime or platform.</td>
</tr>
<tr>
<td>XR_ERROR_ACTION_TYPE_MISMATCH</td>
<td>The API used to retrieve an action's state does not match the action's</td>
</tr>
<tr>
<td></td>
<td>type.</td>
</tr>
<tr>
<td>XR_ERROR_SESSION_NOT_READY</td>
<td>The session is not in the ready state.</td>
</tr>
<tr>
<td>XR_ERROR_SESSION_NOT_STOPPING</td>
<td>The session is not in the stopping state.</td>
</tr>
<tr>
<td>XR_ERROR_TIME_INVALID</td>
<td>The provided XrTime was zero, negative, or out of range.</td>
</tr>
<tr>
<td>XR_ERROR_REFERENCE_SPACE_UNSUPPORTED</td>
<td>The specified reference space is not supported by the runtime or system.</td>
</tr>
<tr>
<td>XR_ERROR_FILE_ACCESS_ERROR</td>
<td>The file could not be accessed.</td>
</tr>
<tr>
<td>XR_ERROR_FILE_CONTENTS_INVALID</td>
<td>The file's contents were invalid.</td>
</tr>
<tr>
<td>XR_ERROR_FORM_FACTOR_UNSUPPORTED</td>
<td>The specified form factor is not supported by the current runtime or</td>
</tr>
<tr>
<td></td>
<td>platform.</td>
</tr>
<tr>
<td>Enum</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>XR_ERROR_FORM_FACTOR_UNAVAILABLE</td>
<td>The specified form factor is supported, but the device is currently not available, e.g. not plugged in or powered off.</td>
</tr>
<tr>
<td>XR_ERROR_API_LAYER_NOT_PRESENT</td>
<td>A requested API layer is not present or could not be loaded.</td>
</tr>
<tr>
<td>XR_ERROR_CALL_ORDER_INVALID</td>
<td>The call was made without having made a previously required call.</td>
</tr>
<tr>
<td>XR_ERROR_GRAPHICS_DEVICE_INVALID</td>
<td>The given graphics device is not in a valid state. The graphics device could be lost or initialized without meeting graphics requirements.</td>
</tr>
<tr>
<td>XR_ERROR_POSE_INVALID</td>
<td>The supplied pose was invalid with respect to the requirements.</td>
</tr>
<tr>
<td>XR_ERROR_INDEX_OUT_OF_RANGE</td>
<td>The supplied index was outside the range of valid indices.</td>
</tr>
<tr>
<td>XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED</td>
<td>The specified view configuration type is not supported by the runtime or platform.</td>
</tr>
<tr>
<td>XR_ERROR_ENVIRONMENT_BLEND_MODE_UNSUPPORTED</td>
<td>The specified environment blend mode is not supported by the runtime or platform.</td>
</tr>
<tr>
<td>XR_ERROR_NAME_DUPLICATED</td>
<td>The name provided was a duplicate of an already-existing resource.</td>
</tr>
<tr>
<td>XR_ERROR_NAME_INVALID</td>
<td>The name provided was invalid.</td>
</tr>
<tr>
<td>XR_ERROR_ACTIONSET_NOT_ATTACHED</td>
<td>A referenced action set is not attached to the session.</td>
</tr>
<tr>
<td>XR_ERROR_ACTIONSETS_ALREADY_ATTACHED</td>
<td>The session already has attached action sets.</td>
</tr>
<tr>
<td>XR_ERROR_LOCALIZED_NAME_DUPLICATED</td>
<td>The localized name provided was a duplicate of an already-existing resource.</td>
</tr>
<tr>
<td>XR_ERROR_LOCALIZED_NAME_INVALID</td>
<td>The localized name provided was invalid.</td>
</tr>
<tr>
<td>XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING</td>
<td>The <code>xrGetGraphicsRequirements</code> call was not made before calling <code>xrCreateSession</code>.</td>
</tr>
<tr>
<td>XR_ERROR_RUNTIME_UNAVAILABLE</td>
<td>The loader was unable to find or load a runtime.</td>
</tr>
<tr>
<td>XR_ERROR_ANDROIDTHREADSETTINGSIDINVALID_KHR</td>
<td><code>xrSetAndroidApplicationThreadKHR</code> failed as thread id is invalid. (Added by the <code>XR_KHR_android_thread_settings</code> extension)</td>
</tr>
<tr>
<td>XR_ERROR_ANDROIDTHREADSETTINGSFAILURE_KHR</td>
<td><code>xrSetAndroidApplicationThreadKHR</code> failed setting the thread attributes/priority. (Added by the <code>XR_KHR_android_thread_settings</code> extension)</td>
</tr>
</tbody>
</table>
2.8.1. Convenience Macros

```c
#define XR_SUCCEEDED(result) ((result) >= 0)
```

A convenience macro that can be used to test if a function succeeded. This may be a qualified success such as `XR_FRAME_DISCARDED`.

```c
#define XR_FAILED(result) ((result) < 0)
```

A convenience macro that can be used to test if a function has failed in some way.

```c
#define XR_UNQUALIFIED_SUCCESS(result) ((result) == 0)
```

A convenience macro that can be used to test a function's failure. The `XR_UNQUALIFIED_SUCCESS` macro is a convenience macro which may be used to compare an `XrResult` to 0 (`XR_SUCCESS`) exclusively.

2.8.2. Validation

Except as noted below or in individual API specifications, valid API usage may be required by the runtime. Runtimes may choose to validate some API usage and return an appropriate error code.

Application developers should use validation layers to catch and eliminate errors during development. Once validated, applications should not enable validation layers by default.

If a function returns a run time error, unless otherwise specified any output parameters will have undefined contents, except that if the output parameter is a structure with type and next fields, those fields will be unmodified. Any output structures chained from next will also have undefined contents, except that the type and next will be unmodified.

Unless otherwise specified, errors do not affect existing OpenXR objects. Objects that have already been successfully created may still be used by the application.

`XrResult` code returns may be added to a given function in future versions of the specification. Runtimes must return only `XrResult` codes from the set documented for the given application API version.

Runtimes must ensure that incorrect usage by an application does not affect the integrity of the operating system, the API implementation, or other API client applications in the system, and does not
allow one application to access data belonging to another application.

2.9. Handles

Objects which are allocated by the runtime on behalf of applications are represented by handles. Handles are opaque identifiers for objects whose lifetime is controlled by applications via the create and destroy functions. Example handle types include XrInstance, XrSession, and XrSwapchain. Handles which have not been destroyed are unique for a given application process, but may be reused after being destroyed. Unless otherwise specified, a successful handle creation function call returns a new unique handle. Unless otherwise specified, handles are implicitly destroyed when their parent handle is destroyed. Applications may destroy handles explicitly before the parent handle is destroyed, and should do so if no longer needed, in order to conserve resources. Runtimes may detect XR_NULL_HANDLE and other invalid handles passed where a valid handle is required and return XR_ERROR_HANDLE_INVALID. However, runtimes are not required to do so unless otherwise specified, and so use of any invalid handle may result in undefined behavior. When a function has an optional handle parameter, XR_NULL_HANDLE must be used unless passing a valid handle.

All functions that take a handle parameter may return XR_ERROR_HANDLE_INVALID.

Handles form a hierarchy in which child handles fall under the validity and lifetime of parent handles. For example, to create an XrSwapchain handle, applications must call xrCreateSwapchain and pass an XrSession handle. Thus XrSwapchain is a child handle to XrSession.

2.10. Object Handle Types

The type of an object handle used in a function is usually determined by the specification of that function, as discussed in Valid Usage for Object Handles. However, some functions accept or return object handle parameters where the type of the object handle is unknown at execution time and is not specified in the description of the function itself. For these functions, the XrObjectType may be used to explicitly specify the type of a handle.

For example, an information-gathering or debugging mechanism implemented in a runtime extension or API layer extension may return a list of object handles that are generated by the mechanism’s operation. The same mechanism may also return a parallel list of object handle types that allow the recipient of this information to easily determine the types of the handles.

In general, anywhere an object handle of more than one type can occur, the object handle type may be provided to indicate its type.
The **XrObjectType** enumeration defines values, each of which corresponds to a specific OpenXR handle type. These values can be used to associate debug information with a particular type of object through one or more extensions.

The following table defines **XrObjectType** and OpenXR Handle relationships:

<table>
<thead>
<tr>
<th>XrObjectType</th>
<th>OpenXR Handle Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>XR_OBJECT_TYPE_UNKNOWN</td>
<td>Unknown/Undefined Handle</td>
</tr>
<tr>
<td>XR_OBJECT_TYPE_INSTANCE</td>
<td>XrInstance</td>
</tr>
<tr>
<td>XR_OBJECT_TYPE_SESSION</td>
<td>XrSession</td>
</tr>
<tr>
<td>XR_OBJECT_TYPE_SWAPCHAIN</td>
<td>XrSwapchain</td>
</tr>
<tr>
<td>XR_OBJECT_TYPE_SPACE</td>
<td>XrSpace</td>
</tr>
<tr>
<td>XR_OBJECT_TYPE_ACTION_SET</td>
<td>XrActionSet</td>
</tr>
<tr>
<td>XR_OBJECT_TYPE_ACTION</td>
<td>XrAction</td>
</tr>
</tbody>
</table>

### 2.11. Buffer Size Parameters

Functions with input/output buffer parameters take on either parameter form or struct form, looking like one of the following examples, with the element type being **float** in this case:

**Parameter form:**

```c
XrResult xrFunction(uint32_t elementCapacityInput, uint32_t* elementCountOutput, float* elements);
```

**Struct form:**
A two-call idiom may be employed, first calling `xrFunction` (with a valid `elementCountOutput` pointer if in parameter form), but passing `NULL` as `elements` and `0` as `elementCapacityInput`, to retrieve the required buffer size as number of elements (number of floats in this example). After allocating a buffer at least as large as `elementCountOutput` (in a struct) or the value pointed to by `elementCountOutput` (as parameters), a pointer to the allocated buffer should be passed as `elements`, along with the buffer's length in `elementCapacityInput`, to a second call to `xrFunction` to perform the retrieval of the data. In case that `elements` is a struct with `type` and `next` fields, the application must set the `type` to the correct value as well as `next` either to `NULL` or a struct with extension related data in which `type` and `next` also need to be well defined.

In the following discussion, "set `elementCountOutput" should be interpreted as "set the value pointed to by `elementCountOutput" in parameter form and "set the value of `elementCountOutput" in struct form. These functions have the below-listed behavior with respect to the buffer size parameters:
Buffer Size Parameter Behavior

• The element capacity and count arguments precede the array to which they refer, in argument order.

• `elementCapacityInput` specifies the capacity in number of elements of the buffer to be written, or 0 to indicate a request for the required buffer size.

• Independent of `elementCapacityInput` or `elements` parameters, the function sets `elementCountOutput`. `elementCountOutput` must be a valid pointer if the function uses parameter form.

• Where the `elementCapacityInput` is 0, the function sets `elementCountOutput` to the required size in number of elements and must return XR_SUCCESS. `elements` is ignored.

• Where the `elementCapacityInput` is non-zero but less than required, the function sets `elementCountOutput` to the required capacity, and must return XR_ERROR_SIZE_INSUFFICIENT. The data in `elements` is undefined.

• Where the `elementCapacityInput` is non-zero and the function returned successfully, the function sets `elementCountOutput` to the count of the elements that have been written to `elements`.

• Upon a failure for reasons unrelated to the element array capacity, the contents of `elementCountOutput` and `elements` are undefined.

• In the case that the element array refers to a string (is of type `char*`), `elementCapacityInput` and `elementCountOutput` refer to the string `strlen` plus 1 for a NULL terminator.

Some functions fill multiple buffers in one call. For these functions, the `elementCapacityInput`, `elementCountOutput` and `elements` parameters or fields are repeated, once per buffer, with different prefixes. In that case, the semantics above still apply, with the additional behavior that if any `elementCapacityInput` parameter or field is set to 0 by the application, the runtime must treat all `elementCapacityInput` values as if they were set to 0. If any `elementCapacityInput` value is too small to fit all elements of the buffer, XR_ERROR_SIZE_INSUFFICIENT must be returned, and the data in all buffers is undefined.

2.12. Time

Time is represented by a 64-bit signed integer representing nanoseconds (XrTime). The passage of time must be monotonic and not real-time (i.e. wall clock time). Thus the time is always increasing at a constant rate and is unaffected by clock changes, time zones, daylight savings, etc.

2.12.1. XrTime
typedef int64_t XrTime;

XrTime is a base value type that represents time as a signed 64-bit integer, representing the monotonically-increasing count of nanoseconds that have elapsed since a runtime-chosen epoch. XrTime always represents the time elapsed since that constant epoch, rather than a duration or a time point relative to some moving epoch such as vsync time, etc. Durations are instead represented by XrDuration.

A single runtime must use the same epoch for all simultaneous applications. Time must be represented the same regardless of multiple processors or threads present in the system.

The period precision of time reported by the runtime is runtime-dependent, and may change. One nanosecond is the finest possible period precision. A runtime may, for example, report time progression with only microsecond-level granularity.

Time must not be assumed to correspond to a system clock time.

Unless specified otherwise, zero or a negative value is not a valid XrTime, and related functions must return error XR_ERROR_TIME_INVALID. Applications must not initialize such XrTime fields to a zero value. Instead, applications should always assign XrTime fields to the meaningful point in time they are choosing to reason about, such as a frame's predicted display time, or an action's last change time.

The behavior of a runtime is undefined when time overflows beyond the maximum positive value that can be represented by an XrTime. Runtimes should choose an epoch that minimizes the chance of overflow. Runtimes should also choose an epoch that minimizes the chance of underflow below 0 for applications performing a reasonable amount of historical pose lookback. For example, if the runtime chooses an epoch relative to its startup time, it should push the epoch into the past by enough time to avoid applications performing reasonable pose lookback from reaching a negative XrTime value.

An application cannot assume that the system's clock and the runtime's clock will maintain a constant relationship across frames and should avoid storing such an offset, as this may cause time drift. Applications should instead always use time interop functions to convert a relevant time point across the system's clock and the runtime's clock using extensions, for example, XR_KHR_win32_convert_performance_counter_time or XR_KHR_convert_timespec_time.

2.13. Duration

Duration refers to an elapsed period of time, as opposed to an absolute timepoint.

2.13.1. XrDuration
typedef int64_t XrDuration;

The difference between two timepoints is a duration, and thus the difference between two XrTime values is an XrDuration value.

Functions that refer to durations use XrDuration as opposed to XrTime.

#define XR_NO_DURATION 0

For the case of timeout durations, XR_NO_DURATION may be used to indicate that the timeout is immediate.

#define XR_INFINITE_DURATION 0x7fffffffffffffffLL

XR_INFINITE_DURATION is a special value that may be used to indicate that the timeout never occurs. A timeout with a duration that refers to the past has the same effect as a timeout of XR_NO_DURATION.

### 2.14. Prediction Time Limits

Some functions involve prediction. For example, xrLocateViews accepts a display time for which to return the resulting data. Prediction times provided by applications may refer to time in the past or the future. Times in the past may be interpolated historical data. Runtimes have different practical limits with respect to how far forward or backward prediction times can be accurate. There is no prescribed forward limit the application can successfully request predictions for, though predictions may become less accurate as they get farther into the future. With respect to backward prediction, the application can pass a prediction time equivalent to the timestamp of the most recently received pose plus as much as 50 milliseconds in the past to retrieve accurate historical data. Requested times predating this time window, or requested times predating the earliest received pose, may result in a best effort data whose accuracy reduced or unspecified.

### 2.15. Colors

The XrColor4f structure is defined as:
typedef struct XrColor4f {
    float    r;
    float    g;
    float    b;
    float    a;
} XrColor4f;

Member Descriptions

- r is the red component of the color.
- g is the green component of the color.
- b is the blue component of the color.
- a is the alpha component of the color.

Unless otherwise specified, colors are encoded as linear (not with sRGB nor other gamma compression) values with individual components being in the range of 0.0 through 1.0, and without the RGB components being premultiplied by the alpha component.

If color encoding is specified as being premultiplied by the alpha component, the RGB components are set to zero if the alpha component is zero.

2.16. Coordinate System

This API uses a Cartesian right-handed coordinate system.

Figure 1. Right Handed Coordinate System

The conventions for mapping coordinate axes of any particular space to meaningful directions depend on and are documented with the description of the space.

The API uses 2D, 3D, and 4D floating-point vectors to describe points and directions in a space.

A two-dimensional vector is defined by the XrVector2f structure:
typedef struct XrVector2f {
    float    x;
    float    y;
} XrVector2f;

**Member Descriptions**

- `x` is the x coordinate of the vector.
- `y` is the y coordinate of the vector.

If used to represent physical distances (rather than e.g. normalized direction) and not otherwise specified, values **must** be in meters.

A three-dimensional vector is defined by the `XrVector3f` structure:

typedef struct XrVector3f {
    float    x;
    float    y;
    float    z;
} XrVector3f;

**Member Descriptions**

- `x` is the x coordinate of the vector.
- `y` is the y coordinate of the vector.
- `z` is the z coordinate of the vector.

If used to represent physical distances (rather than e.g. velocity or angular velocity) and not otherwise specified, values **must** be in meters.

A four-dimensional or homogeneous vector is defined by the `XrVector4f` structure:
```c
typedef struct XrVector4f {
    float x;
    float y;
    float z;
    float w;
} XrVector4f;
```

### Member Descriptions

- **x** is the x coordinate of the vector.
- **y** is the y coordinate of the vector.
- **z** is the z coordinate of the vector.
- **w** is the w coordinate of the vector.

If used to represent physical distances, x, y, and z values must be in meters.

Rotation is represented by a unit quaternion defined by the `XrQuaternionf` structure:

```c
typedef struct XrQuaternionf {
    float x;
    float y;
    float z;
    float w;
} XrQuaternionf;
```

### Member Descriptions

- **x** is the x coordinate of the quaternion.
- **y** is the y coordinate of the quaternion.
- **z** is the z coordinate of the quaternion.
- **w** is the w coordinate of the quaternion.

A pose is defined by the `XrPosef` structure:
typedef struct XrPosef {
    XrQuaternionf orientation;
    XrVector3f    position;
} XrPosef;

Member Descriptions

- **orientation** is an `XrQuaternionf` representing the orientation within a space.
- **position** is an `XrVector3f` representing position within a space.

A construct representing a position and orientation within a space, with position expressed in meters, and orientation represented as a unit quaternion. When using `XrPosef` the rotation described by `orientation` is always applied before the translation described by `position`.

A runtime **must** return `XR_ERROR_POSE_INVALID` if the `orientation` norm deviates by more than 1% from unit length.

### 2.17. Common Object Types

Some types of OpenXR objects are used in multiple structures. Those include the `XrVector*f` and types specified above but also the following structures: offset, extents and rectangle.

Offsets are used to describe the magnitude of an offset in two dimensions.

A floating-point offset is defined by the structure:

typedef struct XrOffset2Df {
    float    x;
    float    y;
} XrOffset2Df;

Member Descriptions

- **x** the floating-point offset in the x direction.
- **y** the floating-point offset in the y direction.

This structure is used for component values that may be fractional (floating-point). If used to represent physical distances, values **must** be in meters.
An integer offset is defined by the structure:

typedef struct XrOffset2Di {
    int32_t    x;
    int32_t    y;
} XrOffset2Di;

**Member Descriptions**

- **x** the integer offset in the x direction.
- **y** the integer offset in the y direction.

This variant is for representing discrete values such as texels. For representing physical distances, the floating-point variant **must** be used instead.

Extents are used to describe the size of a rectangular region in two dimensions.

A two-dimensional floating-point extent is defined by the structure:

typedef struct XrExtent2Df {
    float    width;
    float    height;
} XrExtent2Df;

**Member Descriptions**

- **width** the floating-point width of the extent.
- **height** the floating-point height of the extent.

This structure is used for component values that may be fractional (floating-point). If used to represent physical distances, values **must** be in meters.

The **width** and **height** value **must** be non-negative.

A two-dimensional integer extent is defined by the structure:
typedef struct XrExtent2Di {
    int32_t width;
    int32_t height;
} XrExtent2Di;

### Member Descriptions

- **width** the integer width of the extent.
- **height** the integer height of the extent.

This variant is for representing discrete values such as texels. For representing physical distances, the floating-point variant **must** be used instead.

The **width** and **height** value **must** be non-negative.

Rectangles are used to describe a specific rectangular region in two dimensions. Rectangles **must** include both an offset and an extent defined in the same units. For instance, if a rectangle is in meters, both offset and extent **must** be in meters.

A rectangle with floating-point values is defined by the structure:

typedef struct XrRect2Df {
    XrOffset2Df offset;
    XrExtent2Df extent;
} XrRect2Df;

### Member Descriptions

- **offset** is the XrOffset2Df specifying the rectangle offset.
- **extent** is the XrExtent2Df specifying the rectangle extent.

This structure is used for component values that may be fractional (floating-point).

A rectangle with integer values is defined by the structure:
typedef struct XrRect2Di {
    XrOffset2Di    offset;
    XrExtent2Di    extent;
} XrRect2Di;

**Member Descriptions**

- **offset** is the XrOffset2Di specifying the integer rectangle offset.
- **extent** is the XrExtent2Di specifying the integer rectangle extent.

This variant is for representing discrete values such as texels. For representing physical distances, the floating-point variant **must** be used instead.

### 2.18. Angles

Where a value is provided as a function parameter or as a structure member and will be interpreted as an angle, the value is defined to be in radians.

Field of view (FoV) is defined by the structure:

typedef struct XrFovf {
    float    angleLeft;
    float    angleRight;
    float    angleUp;
    float    angleDown;
} XrFovf;

**Member Descriptions**

- **angleLeft** is the angle of the left side of the field of view. For a symmetric field of view this value is negative.
- **angleRight** is the angle of the right side of the field of view.
- **angleUp** is the angle of the top part of the field of view.
- **angleDown** is the angle of the bottom part of the field of view. For a symmetric field of view this value is negative.

Angles to the right of the center and upwards from the center are positive, and angles to the left of the
center and down from the center are negative. The total horizontal field of view is \texttt{angleRight} minus \texttt{angleLeft}, and the total vertical field of view is \texttt{angleUp} minus \texttt{angleDown}. For a symmetric FoV, \texttt{angleRight} and \texttt{angleUp} will have positive values, \texttt{angleLeft} will be \(-\texttt{angleRight}\), and \texttt{angleDown} will be \(-\texttt{angleUp}\).

The angles \textbf{must} be specified in radians, and \textbf{must} be between \(-\pi/2\) and \(\pi/2\) exclusively.

When \texttt{angleLeft} > \texttt{angleRight}, the content of the view \textbf{must} be flipped horizontally. When \texttt{angleDown} > \texttt{angleUp}, the content of the view \textbf{must} be flipped vertically.

### 2.19. Boolean Values

```c
typedef uint32_t XrBool32;
```

Boolean values used by OpenXR are of type \texttt{XrBool32} and are 32-bits wide as suggested by the name. The only valid values are the following:

<table>
<thead>
<tr>
<th>Enumerant Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{XR_TRUE} represents a true value.</td>
</tr>
<tr>
<td>\texttt{XR_FALSE} represents a false value.</td>
</tr>
</tbody>
</table>

### 2.20. Events

Events are messages sent from the runtime to the application.

#### 2.20.1. Event Polling

These events are placed in a queue and the application \textbf{must} read from the queue with regularity. Events are read from the queue one at a time via \texttt{xrPollEvent}. Every event is identified by an individual struct, with each struct beginning with an \texttt{XrEventDataBaseHeader}.
Example 1. Proper Method for Receiving OpenXR Event Data

```cpp
XrInstance instance; // previously initialized

// Initialize an event buffer to hold the output.
XrEventDataBuffer event;
// Only the header needs to be initialized.
event.type = XR_TYPE_EVENT_DATA_BUFFER;
event.next = nullptr;
XrResult result = xrPollEvent(instance, &event);
if (result == XR_SUCCESS) {
    switch (event.type) {
        case XR_TYPE_EVENT_DATA_SESSION_STATE_CHANGED: {
            const XrEventDataSessionStateChanged& session_state_changed_event = *
                reinterpret_cast<XrEventDataSessionStateChanged*>(&event);
            // ...
            break;
        }
        case XR_TYPE_EVENT_DATA_INSTANCE_LOSS_PENDING: {
            const XrEventDataInstanceLossPending& instance_loss_pending_event = *
                reinterpret_cast<XrEventDataInstanceLossPending*>(&event);
            // ...
            break;
        }
    }
}
```

**xrPollEvent**

```cpp
XrResult xrPollEvent(
    XrInstance instance,
    XrEventDataBuffer* eventData);
```

**xrPollEvent** polls for the next event and returns an event if one is available. **xrPollEvent** returns immediately regardless of whether an event was available. The event (if present) is unilaterally removed from the queue if a valid **XrInstance** is provided. On return the **eventData** parameter is filled with the event's data and the type field is changed to the event's type. Runtimes **may** create valid next chains depending on enabled extensions, but they **must** guarantee that any such chains point only to objects which fit completely within the original **XrEventDataBuffer** pointed to by **eventData**.
Parameter Descriptions

- `instance` is a valid `XrInstance`.
- `eventData` is a pointer to a valid `XrEventDataBuffer`.

Valid Usage (Implicit)

- `instance` must be a valid `XrInstance` handle
- `eventData` must be a pointer to an `XrEventDataBuffer` structure

Return Codes

**Success**

- `XR_SUCCESS`
- `XR_EVENT_UNAVAILABLE`

**Failure**

- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`

The runtime must discard queued events which contain destroyed or otherwise invalid handles.

Table 2. Event Descriptions

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>XrEventDataEventsLost</code></td>
<td>event queue has overflowed and some events were lost</td>
</tr>
<tr>
<td><code>XrEventDataInstanceLossPending</code></td>
<td>application is about to lose the instance</td>
</tr>
<tr>
<td><code>XrEventDataInteractionProfileChanged</code></td>
<td>active input form factor for one or more top level user paths has changed</td>
</tr>
<tr>
<td><code>XrEventDataReferenceSpaceChangePending</code></td>
<td>runtime will begin operating with updated space bounds</td>
</tr>
<tr>
<td><code>XrEventDataSessionStateChanged</code></td>
<td>application has changed lifecycle state</td>
</tr>
</tbody>
</table>

The `XrEventDataBaseHeader` structure is defined as:
typedef struct XrEventDataBaseHeader {
    XrStructureType type;
    const void* next;
} XrEventDataBaseHeader;

Parameter Descriptions

- **type** is the `XrStructureType` of this structure. This base structure itself has no associated `XrStructureType` value.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

The `XrEventDataBaseHeader` is a generic structure used to identify the common event data elements.

Upon receipt, the `XrEventDataBaseHeader` pointer should be type-cast to a pointer of the appropriate event data based on the `type` parameter.

Valid Usage (Implicit)

- **type** must be one of the following `XrStructureType` values: `XR_TYPE_EVENT_DATA_EVENTS_LOST`, `XR_TYPE_EVENT_DATA_INSTANCE_LOSS_PENDING`, `XR_TYPE_EVENT_DATA_INTERACTION_PROFILE_CHANGED`, `XR_TYPE_EVENT_DATA_REFERENCE_SPACE_CHANGE_PENDING`, `XR_TYPE_EVENT_DATA_SESSION_STATE_CHANGED`, `XR_TYPE_EVENT_DATA_VISIBILITY_MASK_CHANGED_KHR`.
- **next** must be `NULL` or a valid pointer to the next structure in a structure chain.

The `XrEventDataBuffer` is a structure passed to `xrPollEvent` large enough to contain any returned event data element. The maximum size is specified by `XR_MAX_EVENT_DATA_SIZE`.

It is sufficient to clear the `type` and `next` parameters of an `XrEventDataBuffer` when passing it as an input to `xrPollEvent`.

An `XrEventDataBuffer` may be type-cast to a `XrEventDataBaseHeader` pointer or a pointer to any other appropriate event data based on the `type` parameter.
typedef struct XrEventDataBuffer {
    XrStructureType    type;
    const void*        next;
    uint8_t            varying[4000];
} XrEventDataBuffer;

Parameter Descriptions

- **type** is the `XrStructureType` of this structure.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **varying** is a fixed sized output buffer big enough to hold returned data elements for all specified event data types.

Valid Usage (Implicit)

- **type must** be `XR_TYPE_EVENT_DATA_BUFFER`
- **next must** be `NULL` or a valid pointer to the next structure in a structure chain

`XR_MAX_EVENT_DATA_SIZE` is the maximum size of an `XrEventDataBuffer`.

```c
#define XR_MAX_EVENT_DATA_SIZE sizeof(XrEventDataBuffer)
```

**XrEventDataEventsLost**

The `XrEventDataEventsLost` structure is defined as:

```c
typedef struct XrEventDataEventsLost {
    XrStructureType    type;
    const void*        next;
    uint32_t           lostEventCount;
} XrEventDataEventsLost;
```
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **lostEventCount** is the number of events which have overflowed since the last call to xrPollEvent.

Receiving the XrEventDataEventsLost event structure indicates that the event queue overflowed and some events were removed at the position within the queue at which this event was found.

Valid Usage (Implicit)

- **type** must be XR_TYPE_EVENT_DATA_EVENTS_LOST
- **next** must be NULL or a valid pointer to the next structure in a structure chain

Other event structures are defined in later chapters in the context where their definition is most relevant.

### 2.21. System resource lifetime

The creator of an underlying system resource is responsible for ensuring the resource’s lifetime matches the lifetime of the associated OpenXR handle.

Resources passed as inputs from the application to the runtime when creating an OpenXR handle **should** not be freed while that handle is valid. A runtime **must** not free resources passed as inputs or decrease their reference counts (if applicable) from the initial value. For example, the graphics device handle (or pointer) passed in to xrCreateSession in XrGraphicsBinding* structure **should** be kept alive when the corresponding XrSession handle is valid, and **should** be freed by the application after the XrSession handle is destroyed.

Resources created by the runtime should not be freed by the application, and the application **should** maintain the same reference count (if applicable) at the destruction of the OpenXR handle as it had at its creation. For example, the ID3D*Texture2D objects in the XrSwapchainImageD3D* are created by the runtime and associated with the lifetime of the XrSwapchain handle. The application **should** not keep additional reference counts on any ID3D*Texture2D objects past the lifetime of the XrSwapchain handle, or make extra reference count decrease after destroying the XrSwapchain handle.
Chapter 3. API Initialization

Before using an OpenXR runtime, an application **must** initialize it by creating an XrInstance object. The following functions are useful for gathering information about the API layers and extensions installed on the system and creating the instance.

### Instance Creation Functions

- xrEnumerateApiLayerProperties
- xrEnumerateInstanceExtensionProperties
- xrCreateInstance

xrEnumerateApiLayerProperties and xrEnumerateInstanceExtensionProperties can be called before calling xrCreateInstance.

### 3.1. Exported Functions

A dynamically linked library (.dll or .so) that implements the API loader **must** export all core OpenXR API functions. However, the application **can** gain access to extension functions by obtaining pointers to these functions through the use of xrGetInstanceProcAddr.

### 3.2. Function Pointers

Function pointers for all OpenXR functions **can** be obtained with the function xrGetInstanceProcAddr.

```c
XrResult xrGetInstanceProcAddr(
    XrInstance instance,
    const char* name,
    PFN_xrVoidFunction* function);
```

**Parameter Descriptions**

- **instance** is the instance that the function pointer will be compatible with, or NULL for functions not dependent on any instance.
- **name** is the name of the function to obtain.
- **function** is the address of the function pointer to get.
xrGetInstanceProcAddr itself is obtained in a platform- and loader- specific manner. Typically, the loader library will export this function as a function symbol, so applications can link against the loader library, or load it dynamically and look up the symbol using platform-specific APIs. Loaders must export function symbols for all core OpenXR functions. Because of this, applications that use only the core OpenXR functions have no need to use xrGetInstanceProcAddr.

Because an application can call xrGetInstanceProcAddr before creating an instance, xrGetInstanceProcAddr returns a valid function pointer when the instance parameter is XR_NULL_HANDLE and the name parameter is one of the following strings:

No Instance Required

- xrEnumerateInstanceExtensionProperties
- xrEnumerateApiLayerProperties
- xrCreateInstance

xrGetInstanceProcAddr must return XR_ERROR_HANDLE_INVALID if name is not one of the above strings and instance is XR_NULL_HANDLE. xrGetInstanceProcAddr may return XR_ERROR_HANDLE_INVALID if name is not one of the above strings and instance is invalid but not XR_NULL_HANDLE.

xrGetInstanceProcAddr must return XR_ERROR_FUNCTION_UNSUPPORTED if instance is a valid instance and the string specified in name is not the name of an OpenXR core or enabled extension function.

If name is the name of an extension function, then the result returned by xrGetInstanceProcAddr will depend upon how the instance was created. If instance was created with the related extension's name appearing in the XrInstanceCreateInfo::enabledExtensionNames array, then xrGetInstanceProcAddr returns a valid function pointer. If the related extension's name did not appear in the XrInstanceCreateInfo::enabledExtensionNames array during the creation of instance, then xrGetInstanceProcAddr returns XR_ERROR_FUNCTION_UNSUPPORTED. Because of this, function pointers returned by xrGetInstanceProcAddr using one XrInstance may not be valid when used with objects related to a different XrInstance.

The returned function pointer is of type PFN_xrVoidFunction, and must be cast to the type of the function being queried.

The table below defines the various use cases for xrGetInstanceProcAddr and return value (“fp” is “function pointer”) for each case.

<table>
<thead>
<tr>
<th>instance parameter</th>
<th>name parameter</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>invalid instance</td>
<td>*</td>
<td>undefined</td>
</tr>
<tr>
<td>*</td>
<td>NULL</td>
<td>undefined</td>
</tr>
</tbody>
</table>

Table 3. xrGetInstanceProcAddr behavior
<table>
<thead>
<tr>
<th>instance parameter</th>
<th>name parameter</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>xrEnumerateInstanceExtensionProperties</td>
<td>fp</td>
</tr>
<tr>
<td>NULL</td>
<td>xrEnumerateApiLayerProperties</td>
<td>fp</td>
</tr>
<tr>
<td>NULL</td>
<td>xrCreateInstance</td>
<td>fp</td>
</tr>
<tr>
<td>NULL</td>
<td>* (any name not covered above)</td>
<td>NULL</td>
</tr>
<tr>
<td>instance</td>
<td>core OpenXR function</td>
<td>fp</td>
</tr>
<tr>
<td>instance</td>
<td>enabled extension function for instance</td>
<td>fp</td>
</tr>
<tr>
<td>instance</td>
<td>* (any name not covered above)</td>
<td>NULL</td>
</tr>
</tbody>
</table>

The returned function pointer **must** only be called with a handle (the first parameter) that is `instance` or a child of `instance`.

## Valid Usage (Implicit)
- If `instance` is not `XR_NULL_HANDLE`, `instance` **must** be a valid `XrInstance` handle
- `name` **must** be a null-terminated UTF-8 string
- `function` **must** be a pointer to a `PFN_xrVoidFunction` value

## Return Codes

### Success
- `XR_SUCCESS`

### Failure
- `XR_ERROR_FUNCTION_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_OUT_OF_MEMORY`
typedef void (XRAPI_PTR *PFN_xrVoidFunction)(void);

**Parameter Descriptions**

- no parameters.

PFN_xrVoidFunction is a generic function pointer type returned by queries, specifically those to xrGetInstanceProcAddr.
Chapter 4. Instance

XR_DEFINE_HANDLE(XrInstance)

An OpenXR instance is an object that allows an OpenXR application to communicate with an OpenXR runtime. The application accomplishes this communication by calling `xrCreateInstance` and receiving a handle to the resulting `XrInstance` object.

The `XrInstance` object stores and tracks OpenXR-related application state, without storing any such state in the application’s global address space. This allows the application to create multiple instances as well as safely encapsulate the application’s OpenXR state since this object is opaque to the application. OpenXR runtimes may limit the number of simultaneous `XrInstance` objects that may be created and used, but they must support the creation and usage of at least one `XrInstance` object per process.

Physically, this state may be stored in any of the OpenXR loader, OpenXR API layers or the OpenXR runtime components. The exact storage and distribution of this saved state is implementation-dependent, except where indicated by this specification.

The tracking of OpenXR state in the instance allows the streamlining of the API, where the intended instance is inferred from the highest ascendant of an OpenXR function’s target object. For example, in:

```
myResult = xrEndFrame(mySession, &myEndFrameDescription);
```

the `XrSession` object was created from an `XrInstance` object. The OpenXR loader typically keeps track of the `XrInstance` that is the parent of the `XrSession` object in this example and directs the function to the runtime associated with that instance. This tracking of OpenXR objects eliminates the need to specify an `XrInstance` in every OpenXR function.

4.1. API Layers and Extensions

Additional functionality may be provided by API layers or extensions. An API layer must not add or modify the definition of OpenXR functions, while an extension may do so.

The set of API layers to enable is specified when creating an instance, and those API layers are able to intercept any functions dispatched to that instance or any of its child objects.

Example API layers may include (but are not limited to):

- an API layer to dump out OpenXR API calls
- an API layer to perform OpenXR validation
To determine what set of API layers are available, OpenXR provides the `xrEnumerateApiLayerProperties` function:

```c
XrResult xrEnumerateApiLayerProperties(
    uint32_t propertyCapacityInput,
    uint32_t* propertyCountOutput,
    XrApiLayerProperties* properties);
```

**Parameter Descriptions**

- `propertyCapacityInput` is the capacity of the properties array, or 0 to indicate a request to retrieve the required capacity.
- `propertyCountOutput` is a pointer to the count of properties written, or a pointer to the required capacity in the case that `propertyCapacityInput` is insufficient.
- `properties` is a pointer to an array of `XrApiLayerProperties` structures, but can be NULL if `propertyCapacityInput` is 0.
- See the Buffer Size Parameters section for a detailed description of retrieving the required properties size.

The list of available layers may change at any time due to actions outside of the OpenXR runtime, so two calls to `xrEnumerateApiLayerProperties` with the same parameters may return different results, or retrieve different `propertyCountOutput` values or `properties` 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)**

- `propertyCountOutput` must be a pointer to a `uint32_t` value
- If `propertyCapacityInput` is not 0, `properties` must be a pointer to an array of `propertyCapacityInput` `XrApiLayerProperties` structures
Return Codes

Success
• XR_SUCCESS

Failure
• XR_ERROR_VALIDATION_FAILURE
• XR_ERROR_RUNTIME_FAILURE
• XR_ERROR_OUT_OF_MEMORY
• XR_ERROR_SIZE_INSUFFICIENT

The XrApiLayerProperties structure is defined as:

typedef struct XrApiLayerProperties {
    XrStructureType    type;
    void*              next;
    char               layerName[XR_MAX_API_LAYER_NAME_SIZE];
    XrVersion          specVersion;
    uint32_t           layerVersion;
    char               description[XR_MAX_API_LAYER_DESCRIPTION_SIZE];
} XrApiLayerProperties;

Member Descriptions

• type is the XrStructureType of this structure.

• next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

• layerName is a string specifying the name of the API layer. Use this name in the XrInstanceCreateInfo::enabledApiLayerNames array to enable this API layer for an instance.

• specVersion is the API version the API layer was written to, encoded as described in the API Version Numbers and Semantics section.

• layerVersion is the version of this API layer. It is an integer, increasing with backward compatible changes.

• description is a string providing additional details that can be used by the application to identify the API layer.
To enable a layer, the name of the layer **should** be added to the `enabledApiLayerNames` member of `XrInstanceCreateInfo` when creating an `XrInstance`.

Loader implementations **may** provide mechanisms outside this API for enabling specific API layers. API layers enabled through such a mechanism are implicitly enabled, while API layers enabled by including the API layer name in `XrInstanceCreateInfo::enabledApiLayerNames` are explicitly enabled. Except where otherwise specified, implicitly enabled and explicitly enabled API layers differ only in the way they are enabled. Explicitly enabling an API layer that is implicitly enabled has no additional effect.

Instance extensions are able to affect the operation of the instance and any of its child objects. As stated earlier, extensions can expand the OpenXR API and provide new functions or augment behavior.

**Examples of extensions** **may** be (but are not limited to):

**Extension Examples**

- an extension to include OpenXR functions to work with a new graphics API
- an extension to expose debug information via a callback

The application can determine the available instance extensions by calling `xrEnumerateInstanceExtensionProperties`:

```c
XrResult xrEnumerateInstanceExtensionProperties(
    const char*                              layerName,
    uint32_t                                  propertyCapacityInput,
    uint32_t*                                 propertyCountOutput,
    XrExtensionProperties*                   properties);
```
Parameter Descriptions

- **layerName** is either **NULL** or a pointer to a string naming the API layer to retrieve extensions from, as returned by `xrEnumerateApiLayerProperties`.

- **propertyCapacityInput** is the capacity of the properties array, or 0 to indicate a request to retrieve the required capacity.

- **propertyCountOutput** is a pointer to the count of properties written, or a pointer to the required capacity in the case that **propertyCapacityInput** is insufficient.

- **properties** is a pointer to an array of `XrExtensionProperties` structures, but can be **NULL** if **propertyCapacityInput** is 0.

- See the **Buffer Size Parameters** section for a detailed description of retrieving the required properties size.

If **properties** is **NULL**, then the number of extensions properties available is returned in **propertyCountOutput**. Otherwise, **propertyCountInput** must point to a variable set by the user to the number of elements in the properties array. If **propertyCountInput** is less than the number of extension properties available, the contents of properties will be undefined. If **propertyCountInput** is smaller than the number of extensions available, the runtime must return the failure code `XR_ERROR_SIZE_INSUFFICIENT` and the contents of properties are undefined.

Because the list of available layers may change externally between calls to `xrEnumerateInstanceExtensionProperties`, two calls may retrieve different results if a **layerName** is available in one call but not in another. The extensions supported by a layer may also change between two calls, e.g. if the layer implementation is replaced by a different version between those calls.

Valid Usage (Implicit)

- If **layerName** is not **NULL**, **layerName** must be a null-terminated UTF-8 string

- **propertyCountOutput** must be a pointer to a `uint32_t` value

- If **propertyCapacityInput** is not 0, **properties** must be a pointer to an array of `propertyCapacityInput` `XrExtensionProperties` structures
Return Codes

Success
- XR_SUCCESS

Failure
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_SIZE_INSUFFICIENT
- XR_ERROR_RUNTIME_UNAVAILABLE
- XR_ERROR_API_LAYER_NOT_PRESENT

The XrExtensionProperties structure is defined as:

typedef struct XrExtensionProperties {
    XrStructureType    type;
    void*              next;
    char               extensionName[XR_MAX_EXTENSION_NAME_SIZE];
    uint32_t           extensionVersion;
} XrExtensionProperties;

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- extensionName is a NULL terminated string specifying the name of the extension.
- extensionVersion is the version of this extension. It is an integer, incremented with backward compatible changes.

Valid Usage (Implicit)

- type must be XR_TYPE_EXTENSION_PROPERTIES
- next must be NULL or a valid pointer to the next structure in a structure chain
4.2. Instance Lifecycle

The `xrCreateInstance` function is defined as:

```c
XrResult xrCreateInstance(
    const XrInstanceCreateInfo* createInfo,  
    XrInstance* instance);
```

**Parameter Descriptions**

- `createInfo` points to an instance of `XrInstanceCreateInfo` controlling creation of the instance.
- `instance` points to an `XrInstance` handle in which the resulting instance is returned.

`xrCreateInstance` creates the `XrInstance`, then enables and initializes global API layers and extensions requested by the application. If an extension is provided by an API layer, both the API layer and extension **must** be specified at `xrCreateInstance` time. If a specified API layer cannot be found, no `XrInstance` will be created and the function will return `XR_ERROR_API_LAYER_NOT_PRESENT`. Likewise, if a specified extension cannot be found, the call **must** return `XR_ERROR_EXTENSION_NOT_PRESENT` and no `XrInstance` will be created. Additionally, some runtimes **may** limit the number of concurrent instances that may be in use. If the application attempts to create more instances than a runtime can simultaneously support, `xrCreateInstance` **may** return `XR_ERROR_LIMIT_REACHED`.

If the `XrApplicationInfo::applicationName` is the empty string the runtime **must** return `XR_ERROR_NAME_INVALID`.

If the `XrInstanceCreateInfo` structure contains a platform-specific extension for a platform other than the target platform, `XR_ERROR_INITIALIZATION_FAILED` **may** be returned. If a mandatory platform-specific extension is defined for the target platform but no matching extension struct is provided in `XrInstanceCreateInfo` the runtime **must** return `XR_ERROR_INITIALIZATION_FAILED`.

**Valid Usage (Implicit)**

- `createInfo` **must** be a pointer to a valid `XrInstanceCreateInfo` structure
- `instance` **must** be a pointer to an `XrInstance` handle
Return Codes

Success
- XR_SUCCESS

Failure
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_LIMIT_REACHED
- XR_ERROR_RUNTIME_UNAVAILABLE
- XR_ERROR_NAME_INVALID
- XR_ERROR_INITIALIZATION_FAILED
- XR_ERROR_EXTENSION_NOT_PRESENT
- XR_ERROR_API_VERSION_UNSUPPORTED
- XR_ERROR_API_LAYER_NOT_PRESENT

The `XrInstanceCreateInfo` structure is defined as:

```c
typedef struct XrInstanceCreateInfo {
    XrStructureType          type;
    const void*              next;
    XrInstanceCreateFlags    createFlags;
    XrApplicationInfo        applicationInfo;
    uint32_t                 enabledApiLayerCount;
    const char* const*       enabledApiLayerNames;
    uint32_t                 enabledExtensionCount;
    const char* const*       enabledExtensionNames;
} XrInstanceCreateInfo;
```
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **createFlags** is a bitmask of XrInstanceCreateFlags that identifies options that apply to the creation.
- **applicationInfo** is an instance of XrApplicationInfo. This information helps runtimes recognize behavior inherent to classes of applications. XrApplicationInfo is defined in detail below.
- **enabledApiLayerCount** is the number of global API layers to enable.
- **enabledApiLayerNames** is a pointer to an array of enabledApiLayerCount strings containing the names of API layers to enable for the created instance. See the API Layers And Extensions section for further details.
- **enabledExtensionCount** is the number of global extensions to enable.
- **enabledExtensionNames** is a pointer to an array of enabledExtensionCount strings containing the names of extensions to enable.

Valid Usage (Implicit)

- **type** must be XR_TYPE_INSTANCE_CREATE_INFO
- **next** must be NULL or a valid pointer to the next structure in a structure chain. See also: XrInstanceCreateInfoAndroidKHR
- **createFlags** must be 0
- **applicationInfo** must be a valid XrApplicationInfo structure
- If **enabledApiLayerCount** is not 0, **enabledApiLayerNames** must be a pointer to an array of enabledApiLayerCount null-terminated UTF-8 strings
- If **enabledExtensionCount** is not 0, **enabledExtensionNames** must be a pointer to an array of enabledExtensionCount null-terminated UTF-8 strings

The XrInstanceCreateFlags include:

```c
// Flag bits for XrInstanceCreateFlags
```

There are currently no instance creation flags. This is reserved for future use.
The `XrApplicationInfo` structure is defined as:

```c
typedef struct XrApplicationInfo {
    char         applicationName[XR_MAX_APPLICATION_NAME_SIZE];
    uint32_t     applicationVersion;
    char         engineName[XR_MAX_ENGINE_NAME_SIZE];
    uint32_t     engineVersion;
    XrVersion    apiVersion;
} XrApplicationInfo;
```

**Member Descriptions**

- `applicationName` is a non-empty string containing the name of the application.
- `applicationVersion` is an unsigned integer variable containing the developer-supplied version number of the application.
- `engineName` is a string containing the name of the engine (if any) used to create the application. It may be empty to indicate no specified engine.
- `engineVersion` is an unsigned integer variable containing the developer-supplied version number of the engine used to create the application. May be zero to indicate no specified engine.
- `apiVersion` is the version of this API against which the application will run, encoded as described in the API Version Numbers and Semantics section. If the runtime does not support the requested `apiVersion` it must return `XR_ERROR_API_VERSION_UNSUPPORTED`.

**Valid Usage (Implicit)**

- `applicationName` **must** be a null-terminated UTF-8 string whose length is less than or equal to `XR_MAX_APPLICATION_NAME_SIZE`
- `engineName` **must** be a null-terminated UTF-8 string whose length is less than or equal to `XR_MAX_ENGINE_NAME_SIZE`
**Note**

When using the OpenXR API to implement a reusable engine that will be used by many applications, `engineName` **should** be set to a unique string that identifies the engine, and `engineVersion` **should** encode a representation of the engine’s version. This way, all applications that share this engine version will provide the same `engineName` and `engineVersion` to the runtime. The engine **should** then enable individual applications to choose their specific `applicationName` and `applicationVersion`, enabling one application to be distinguished from another application.

When using the OpenXR API to implement an individual application without a shared engine, the input `engineName` **should** be left empty and `engineVersion` **should** be set to 0. The `applicationName` **should** then be filled in with a unique string that identifies the app and the `applicationVersion` **should** encode a representation of the application’s version.

The `xrDestroyInstance` function is defined as:

```c
XrResult xrDestroyInstance(
    XrInstance instance);
```

The `xrDestroyInstance` function is used to destroy an `XrInstance`.

### Parameter Descriptions

- `instance` is the handle to the instance to destroy.

`XrInstance` handles are destroyed using `xrDestroyInstance`. When an `XrInstance` is destroyed, all handles that are children of that `XrInstance` are also destroyed.

### Valid Usage (Implicit)

- `instance` **must** be a valid `XrInstance` handle

### Thread Safety

- Access to `instance`, and any child handles, **must** be externally synchronized
Return Codes

Success
  • XR_SUCCESS

Failure
  • XR_ERROR_HANDLE_INVALID

4.3. Instance Information

The `xrGetInstanceProperties` function provides information about the instance and the associated runtime.

```c
XrResult xrGetInstanceProperties(
    XrInstance                                  instance,
    XrInstanceProperties*                       instanceProperties);
```

Parameter Descriptions

• `instance` is a handle to an `XrInstance` previously created with `xrCreateInstance`.
• `instanceProperties` points to an `XrInstanceProperties` which describes the `instance`.

The `instanceProperties` parameter must be filled out by the runtime in response to this call, with information as defined in `XrInstanceProperties`.

Valid Usage (Implicit)

• `instance` must be a valid `XrInstance` handle
• `instanceProperties` must be a pointer to an `XrInstanceProperties` structure
Return Codes

Success
• XR_SUCCESS

Failure
• XR_ERROR_VALIDATION_FAILURE
• XR_ERROR_RUNTIME_FAILURE
• XR_ERROR_HANDLE_INVALID
• XR_ERROR_INSTANCE_LOST

The XrInstanceProperties structure is defined as:

typedef struct XrInstanceProperties {
    XrStructureType    type;
    void*              next;
    XrVersion          runtimeVersion;
    char               runtimeName[XR_MAX_RUNTIME_NAME_SIZE];
} XrInstanceProperties;

Member Descriptions

• type is the XrStructureType of this structure.

• next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

• runtimeVersion is the runtime's version (not necessarily related to an OpenXR API version), expressed in the format of XR_MAKE_VERSION.

• runtimeName is the name of the runtime.

Valid Usage (Implicit)

• type must be XR_TYPE_INSTANCE_PROPERTIES
• next must be NULL or a valid pointer to the next structure in a structure chain
4.4. Platform-Specific Instance Creation

Some amount of data required for instance creation is exposed through chained structures defined in extensions. These structures may be optional or even required for instance creation on specific platforms, but not on other platforms. Separating off platform-specific functionality into extension structures prevents the primary XrInstanceCreateInfo structure from becoming too bloated with unnecessary information.

See the List of Extensions appendix for the list of available extensions and their related structures. These structures expand the XrInstanceCreateInfo parent struct using the XrInstanceCreateInfo::next member. The specific list of structures that may be used for extending XrInstanceCreateInfo::next can be found in the "Valid Usage (Implicit)" block immediately following the definition of the structure.

4.4.1. The Instance Lost Error

The XR_ERROR_INSTANCE_LOST error indicates that the XrInstance has become unusable. This can happen if a critical runtime process aborts, if the connection to the runtime is otherwise no longer available, or if the runtime encounters an error during any function execution which prevents it from being able to support further function execution. Once XR_ERROR_INSTANCE_LOST is first returned, it must henceforth be returned by all non-destroy functions that involve an XrInstance or child handle type until the instance is destroyed. Applications must destroy the XrInstance. Applications may then attempt to continue by recreating all relevant OpenXR objects, starting with a new XrInstance. A runtime may generate an XrEventDataInstanceLossPending event when instance loss is detected.

4.4.2. XrEventDataInstanceLossPending

```c
typedef struct XrEventDataInstanceLossPending {
    XrStructureType    type;
    const void*        next;
    XrTime             lossTime;
} XrEventDataInstanceLossPending;
```

Receiving the XrEventDataInstanceLossPending event structure indicates that the application is about to lose the indicated XrInstance at the indicated lossTime in the future. The application should call xrDestroyInstance and relinquish any instance-specific resources. This typically occurs to make way for a replacement of the underlying runtime, such as via a software update.

After the application has destroyed all of its instances and their children and waited past the specified time, it may then re-try xrCreateInstance in a loop waiting for whatever maintenance the runtime is performing to complete. The runtime will return XR_ERROR_RUNTIME_UNAVAILABLE from xrCreateInstance as long as it is unable to create the instance. Once the runtime has returned and is able to continue, it must resume returning XR_SUCCESS from xrCreateInstance if valid data is passed in.
Member Descriptions

• **type** is the XrStructureType of this structure.

• **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

• **lossTime** is the absolute time at which the indicated instance will be considered lost and become unusable.

Valid Usage (Implicit)

• **type** must be XR_TYPE_EVENT_DATA_INSTANCE_LOSS_PENDING

• **next** must be NULL or a valid pointer to the next structure in a structure chain

4.5. Instance Enumerated Type String Functions

Applications often want to turn certain enum values from the runtime into strings for use in log messages, to be localized in UI, or for various other reasons. OpenXR provides functions that turn common enum types into UTF-8 strings for use in applications.

```c
XrResult xrResultToString(  
    XrInstance instance,  
    XrResult value,  
    char buffer[XR_MAX_RESULT_STRING_SIZE]);
```

Parameter Descriptions

• **instance** is the handle of the instance to ask for the string.

• **value** is the XrResult value to turn into a string.

• **buffer** is the buffer that will be used to return the string in.

Returns the text version of the provided XrResult value as a UTF-8 string.

In all cases the returned string must be one of:
Result String Return Values

- The literal string defined for the provide numeric value in the core spec or extension. (e.g. the value 0 results in the string `XR_SUCCESS`)
- `XR_UNKNOWN_SUCCESS_` concatenated with the positive result number expressed as a decimal number.
- `XR_UNKNOWN_FAILURE_` concatenated with the negative result number expressed as a decimal number.

Valid Usage (Implicit)

- `instance` must be a valid `XrInstance` handle
- `value` must be a valid `XrResult` value
- `buffer` must be a character array of length `XR_MAX_RESULT_STRING_SIZE`

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE�OST`

The `xrStructureTypeToString` function is defined as:

```c
XrResult xrStructureTypeToString(
    XrInstance instance,
    XrStructureType value,
    char buffer[XR_MAX_STRUCTURE_NAME_SIZE]);
```
**Parameter Descriptions**

- **instance** is the handle of the instance to ask for the string.
- **value** is the XrStructureType value to turn into a string.
- **buffer** is the buffer that will be used to return the string in.

Returns the text version of the provided XrStructureType value as a UTF-8 string.

In all cases the returned string **must** be one of:

**Structure Type String Return Values**

- The literal string defined for the provide numeric value in the core spec or extension. (e.g. the value of XR_TYPE_INSTANCE_CREATE_INFO results in the string XR_TYPE_INSTANCE_CREATE_INFO)
- XR_UNKNOWN_STRUCTURE_TYPE_ concatenated with the structure type number expressed as a decimal number.

**Valid Usage (Implicit)**

- **instance** **must** be a valid XrInstance handle
- **value** **must** be a valid XrStructureType value
- **buffer** **must** be a character array of length XR_MAX_STRUCTURE_NAME_SIZE

**Return Codes**

**Success**

- XR_SUCCESS

**Failure**

- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
Chapter 5. System

This API separates the concept of physical systems of XR devices from the logical objects that applications interact with directly. A system represents a collection of related devices in the runtime, often made up of several individual hardware components working together to enable XR experiences. An \texttt{XrSystemId} is returned by \texttt{xrGetSystem} representing the system of devices the runtime will use to support a given \textbf{form factor}. Each system may include: a VR/AR display, various forms of input (gamepad, touchpad, motion controller), and other trackable objects.

The application uses the system to create a \textbf{session}, which can then be used to accept input from the user and output rendered frames. The application also provides a default set of bindings from its actions to any number of input sources. The runtime \textbf{may} use this action information to activate only a subset of devices and avoid wasting resources on devices that are not in use. Exactly which devices are active once an XR system is selected will depend on the features provided by the runtime, and \textbf{may} vary from runtime to runtime. For example, a runtime that is capable of mapping from one tracking system's space to another’s \textbf{may} support devices from multiple tracking systems simultaneously.

5.1. Form Factors

The first step in selecting a system is for the application to request its desired \textbf{form factor}. The form factor defines how the display(s) moves in the environment relative to the user's head and how the user will interact with the XR experience. A runtime \textbf{may} support multiple form factors, such as on a mobile phone that supports both slide-in VR headset experiences and handheld AR experiences.

While an application’s core XR rendering may span across form factors, its user interface will often be written to target a particular form factor, requiring explicit tailoring to function well on other form factors. For example, screen-space UI designed for a handheld phone will produce an uncomfortable experience for users if presented in screen-space on an AR headset.

```c
typedef enum XrFormFactor {
    XR_FORM_FACTOR_HEAD_MOUNTED_DISPLAY = 1,
    XR_FORM_FACTOR_HANDHELD_DISPLAY = 2,
    XR_FORM_FACTOR_MAX_ENUM = 0x7FFFFFFF
} XrFormFactor;
```

The predefined form factors which \textbf{may} be supported by OpenXR runtimes are:
Enumerant Descriptions

- **XR_FORM_FACTOR_HEAD_MOUNTED_DISPLAY.** The tracked display is attached to the user's head. The user cannot touch the display itself. A VR headset would be an example of this form factor.

- **XR_FORM_FACTOR_HANDHELD_DISPLAY.** The tracked display is held in the user's hand, independent from the user's head. The user may be able to touch the display, allowing for screen-space UI. A mobile phone running an AR experience using pass-through video would be an example of this form factor.

## 5.2. Getting the XrSystemId

**XR_DEFINE_ATOM(XrSystemId)**

An **XrSystemId** is an opaque atom used by the runtime to identify a system. The value **XR_NULL_SYSTEM_ID** is considered an invalid system.

```c
#define XR_NULL_SYSTEM_ID 0
```

The only **XrSystemId** value defined to be constant across all instances is the invalid system **XR_NULL_SYSTEM_ID**. No supported system is associated with **XR_NULL_SYSTEM_ID**. Unless explicitly permitted, it should not be passed to API calls or used as a structure attribute when a valid **XrSystemId** is required.

The **xrGetSystem** function is defined as:

```c
XrResult xrGetSystem(
    XrInstance instance,
    const XrSystemGetInfo* getInfo,
    XrSystemId* systemId);
```
Parameter Descriptions

- **instance** is the handle of the instance from which to get the information.
- **getInfo** is a pointer to an `XrSystemGetInfo` structure containing the application’s requests for a system.
- **systemId** is the returned `XrSystemId`.

To get an `XrSystemId`, an application specifies its desired form factor to `xrGetSystem` and gets the runtime’s `XrSystemId` associated with that configuration.

If the form factor is supported but temporarily unavailable, `xrGetSystem` must return `XR_ERROR_FORM_FACTOR_UNAVAILABLE`. A runtime may return `XR_SUCCESS` on a subsequent call for a form factor it previously returned `XR_ERROR_FORM_FACTOR_UNAVAILABLE`. For example, connecting or warming up hardware might cause an unavailable form factor to become available.

Valid Usage (Implicit)

- **instance** must be a valid `XrInstance` handle
- **getInfo** must be a pointer to a valid `XrSystemGetInfo` structure
- **systemId** must be a pointer to an `XrSystemId` value

Return Codes

**Success**

- `XR_SUCCESS`

**Failure**

- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_FORM_FACTOR_UNSUPPORTED`
- `XR_ERROR_FORM_FACTOR_UNAVAILABLE`

The `XrSystemGetInfo` structure is defined as:
typedef struct XrSystemGetInfo {
    XrStructureType    type;
    const void*        next;
    XrFormFactor       formFactor;
} XrSystemGetInfo;

Member Descriptions

- **type** is the **XrStructureType** of this structure.
- **next** is **NULL** or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **formFactor** is the **XrFormFactor** requested by the application.

The **XrSystemGetInfo** structure specifies attributes about a system as desired by an application.

Valid Usage (Implicit)

- **type** must be **XR_TYPE_SYSTEM_GET_INFO**
- **next** must be **NULL** or a valid pointer to the next structure in a structure chain
- **formFactor** must be a valid **XrFormFactor** value
XrInstance instance; // previously initialized

XrSystemGetInfo system_get_info;
memset(&system_get_info, 0, sizeof(system_get_info));
system_get_info.type = XR_TYPE_SYSTEM_GET_INFO;
system_get_info.formFactor = XR_FORM_FACTOR_HEAD_MOUNTED_DISPLAY;

XrSystemId systemId;
CHK_XR(xrGetSystem(instance, &system_get_info, &systemId));

// create session
// create swapchains
// begin session

// main loop

// end session
// destroy session

// no access to hardware after this point

5.3. System Properties

The xrGetSystemProperties function is defined as:

```
XrResult xrGetSystemProperties(
    XrInstance instance,           // previously initialized
    XrSystemId systemId,           // previously initialized
    XrSystemProperties* properties); // previously initialized
```

**Parameter Descriptions**

- `instance` is the instance from which `systemId` was retrieved.
- `systemId` is the `XrSystemId` whose properties will be queried.
- `properties` points to an instance of the `XrSystemProperties` structure, that will be filled with returned information.

An application can call `xrGetSystemProperties` to retrieve information about the system such as vendor ID, system name, and graphics and tracking properties.
Valid Usage (Implicit)

- **instance** must be a valid `XrInstance` handle
- **properties** must be a pointer to an `XrSystemProperties` structure

Return Codes

**Success**

- XR_SUCCESS

**Failure**

- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_SYSTEM_INVALID

The `XrSystemProperties` structure is defined as:

```c
typedef struct XrSystemProperties {
    XrStructureType               type;
    void*                         next;
    XrSystemId                    systemId;
    uint32_t                      vendorId;
    char                          systemName[XR_MAX_SYSTEM_NAME_SIZE];
    XrSystemGraphicsProperties    graphicsProperties;
    XrSystemTrackingProperties    trackingProperties;
} XrSystemProperties;
```
Member Descriptions

- **type** is the `XrStructureType` of this structure.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **vendorId** is a unique identifier for the vendor of the system.
- **systemId** is the `XrSystemId` identifying the system.
- **systemName** is a string containing the name of the system.
- **graphicsProperties** is an `XrSystemGraphicsProperties` structure specifying the system graphics properties.
- **trackingProperties** is an `XrSystemTrackingProperties` structure specifying system tracking properties.

Valid Usage (Implicit)

- **type** must be `XR_TYPE_SYSTEM_PROPERTIES`
- **next** must be `NULL` or a valid pointer to the next structure in a structure chain

The `XrSystemGraphicsProperties` structure is defined as:

```c
typedef struct XrSystemGraphicsProperties {
    uint32_t    maxSwapchainImageHeight;
    uint32_t    maxSwapchainImageWidth;
    uint32_t    maxLayerCount;
} XrSystemGraphicsProperties;
```

Member Descriptions

- **maxSwapchainImageHeight** is the maximum swapchain image pixel height supported by this system.
- **maxSwapchainImageWidth** is the maximum swapchain image pixel width supported by this system.
- **maxLayerCount** is the maximum number of composition layers supported by this system. The runtime must support at least `XR_MIN_COMPOSITION_LAYERS_SUPPORTED` layers.


```c
#define XR_MIN_COMPOSITION_LAYERS_SUPPORTED 16
```

**XR_MIN_COMPOSITION_LAYERS_SUPPORTED** defines the minimum number of composition layers that a conformant runtime must support. A runtime **must** return the XrSystemGraphicsProperties::maxLayerCount at least the value of **XR_MIN_COMPOSITION_LAYERS_SUPPORTED**.

The **XrSystemTrackingProperties** structure is defined as:

```c
typedef struct XrSystemTrackingProperties {
    XrBool32 orientationTracking;
    XrBool32 positionTracking;
} XrSystemTrackingProperties;
```

**Member Descriptions**

- **orientationTracking** is set to XR_TRUE to indicate the system supports orientational tracking of the view pose(s), XR_FALSE otherwise.
- **positionTracking** is set to XR_TRUE to indicate the system supports positional tracking of the view pose(s), XR_FALSE otherwise.
Chapter 6. Path Tree and Semantic Paths

OpenXR incorporates an internal semantic path tree model, also known as the path tree, with entities associated with nodes organized in a logical tree and referenced by path name strings structured like a filesystem path or URL. The path tree unifies a number of concepts used in this specification and a runtime may add additional nodes as implementation details. As a general design principle, the most application-facing paths should have semantic and hierarchical meaning in their name. Thus, these paths are often referred to as semantic paths. However, path names in the path tree model may not all have the same level or kind of semantic meaning.

In regular use in an application, path name strings are converted to instance-specific XrPath values which are used in place of path strings. The mapping between XrPath values and their corresponding path name strings may be considered to be tracked by the runtime in a one-to-one mapping in addition to the natural tree structure of the referenced entities. Runtimes may use any internal implementation that satisfies the requirements.

Formally, the runtime maintains an instance-specific bijective mapping between well-formed path name strings and valid XrPath (uint64_t) values. These XrPath values are only valid within a single XrInstance, and applications must not share these values between instances. Applications must instead use the string representation of a path in their code and configuration, and obtain the correct corresponding XrPath at runtime in each XrInstance. The term path or semantic path may refer interchangeably to either the path name string or its associated XrPath value within an instance when context makes it clear which type is being discussed.

Given that path trees are a unifying model in this specification, the entities referenced by paths can be of diverse types. For example, they may be used to represent physical device or sensor components, which may be of various component types. They may also be used to represent frames of reference that are understood by the application and the runtime, as defined by an XrSpace. Additionally, to permit runtime re-configuration and support hardware-independent development, any syntactically-valid path string may be used to retrieve a corresponding XrPath without error given sufficient resources, even if no logical or hardware entity currently corresponds to that path at the time of the call. Later retrieval of the associated path string of such an XrPath using xrPathToString should succeed if the other requirements of that call are met. However, using such an XrPath in a later call to any other API function may result in an error if no entity of the type required by the call is available at the path at that later time. A runtime should permit the entity referenced by a path to vary over time to naturally reflect varying system configuration and hardware availability.

6.1. Path Atom Type

```
XR_DEFINE_ATOM(XrPath)
```
The XrPath is an atom that connects an application with a single path, within the context of a single instance. There is a bijective mapping between well-formed path strings and atoms in use. This atom is used—in place of the path name string it corresponds to—to retrieve state and perform other operations.

As an XrPath is only shorthand for a well-formed path string, they have no explicit life cycle.

Lifetime is implicitly managed by the XrInstance. An XrPath must not be used unless it is received at execution time from the runtime in the context of a particular XrInstance. Therefore, with the exception of XR_NULL_PATH, XrPath values must not be specified as constant values in applications: the corresponding path string should be used instead. During the lifetime of a given XrInstance, the XrPath associated with that instance with any given well-formed path must not vary, and similarly the well-formed path string that corresponds to a given XrPath in that instance must not vary. An XrPath that is received from one XrInstance may not be used with another. Such an invalid use may be detected and result in an error being returned, or it may result in undefined behavior.

Well-written applications should typically use a small, bounded set of paths in practice. However, the runtime should support looking up the XrPath for a large number of path strings for maximum compatibility. Runtime implementers should keep in mind that applications supporting diverse systems may look up path strings in a quantity exceeding the number of non-empty entities predicted or provided by any one runtime’s own path tree model, and this is not inherently an error. However, system resources are finite and thus runtimes may signal exhaustion of resources dedicated to these associations under certain conditions.

When discussing the behavior of runtimes at these limits, a new XrPath refers to an XrPath value that, as of some point in time, has neither been received by the application nor tracked internally by the runtime. In this case, since an application has not yet received the value of such an XrPath, the runtime has not yet made any assertions about its association with any path string. In this context, new only refers to the fact that the mapping has not necessarily been made constant for a given value/path string pair for the remaining life of the associated instance by being revealed to the application. It does not necessarily imply creation of the entity, if any, referred to by such a path. Similarly, it does not imply the absence of such an entity prior to that point. Entities in the path tree have varied lifetime that is independent from the duration of the mapping from path string to XrPath.

For flexibility, the runtime may internally track or otherwise make constant, in instance or larger scope, any mapping of a path string to an XrPath value even before an application would otherwise receive that value, thus making it no longer new by the above definition.

When the runtime’s resources to track the path string-XrPath mapping are exhausted, and the application makes an API call that would have otherwise retrieved a new XrPath as defined above, the runtime must return XR_ERROR_PATH_COUNT_EXCEEDED. This includes both explicit calls to xrStringToPath as well as other calls that retrieve an XrPath in any other way.

The runtime should support creating as many paths as memory will allow and must return XR_ERROR_PATH_COUNT_EXCEEDED from relevant functions when no more can be created.
The only `XrPath` value defined to be constant across all instances is the invalid path `XR_NULL_PATH`. No well-formed path string is associated with `XR_NULL_PATH`. Unless explicitly permitted, it should not be passed to API calls or used as a structure attribute when a valid `XrPath` is required.

### 6.2. Well-Formed Path Strings

Even though they look similar, semantic paths are not file paths. To avoid confusion with file path directory traversal conventions, many file path conventions are explicitly disallowed from well-formed path name strings.

A well-formed path name string must conform to the following rules:

- Path name strings must be constructed entirely from characters on the following list.
  - Lower case ASCII letters: a-z
  - Numeric digits: 0-9
  - Dash: -
  - Underscore: _
  - Period: .
  - Forward Slash: /
- Path name strings must start with a single forward slash character.
- Path name strings must not end with a forward slash character.
- Path name strings must not contain two or more adjacent forward slash characters.
- Path name strings must not contain two forward slash characters that are separated by only period characters.
- Path name strings must not contain only period characters following the final forward slash character in the string.
- The maximum string length for a path name string, including the terminating \0 character, is defined by `XR_MAX_PATH_LENGTH`.

### 6.2.1. `xrStringToPath`

The `xrStringToPath` function is defined as:
XrResult xrStringToPath(
    XrInstance instance,
    const char* pathString,
    XrPath* path);

Parameter Descriptions

• instance is an instance previously created.
• pathString is the path name string to retrieve the associated XrPath for.
• path is the output parameter, which must point to an XrPath. Given a well-formed path name string, this will be populated with an opaque value that is constant for that path string during the lifetime of that instance.

xrStringToPath retrieves the XrPath value for a well-formed path string. If such a value had not yet been assigned by the runtime to the provided path string in this XrInstance, one must be assigned at this point. All calls to this function with the same XrInstance and path string must retrieve the same XrPath value. Upon failure, xrStringToPath must return an appropriate XrResult, and may set the output parameter to XR_NULL_PATH. See Path Atom Type for the conditions under which an error may be returned when this function is given a valid XrInstance and a well-formed path string.

If the runtime's resources are exhausted and it cannot create the path, a return value of XR_ERROR_PATH_COUNT_EXCEEDED must be returned. If the application specifies a string that is not a well-formed path string, XR_ERROR_PATH_FORMAT_INVALID must be returned.

A return value of XR_SUCCESS from xrStringToPath may not necessarily imply that the runtime has a component or other source of data that will be accessible through that semantic path. It only means that the path string supplied was well-formed and that the retrieved XrPath maps to the given path string within and during the lifetime of the XrInstance given.

Valid Usage (Implicit)

• instance must be a valid XrInstance handle
• pathString must be a null-terminated UTF-8 string
• path must be a pointer to an XrPath value
Return Codes

Success
  • XR_SUCCESS

Failure
  • XR_ERROR_VALIDATION_FAILURE
  • XR_ERROR_RUNTIME_FAILURE
  • XR_ERROR_HANDLE_INVALID
  • XR_ERROR_INSTANCE_LOST
  • XR_ERROR_PATH_FORMAT_INVALID
  • XR_ERROR_PATH_COUNT_EXCEEDED

6.2.2. xrPathToString

XrResult xrPathToString(
  XrInstance instance,
  XrPath path,
  uint32_t bufferCapacityInput,
  uint32_t* bufferCountOutput,
  char* buffer);

Parameter Descriptions

• instance is an instance previously created.
• path is the valid XrPath value to retrieve the path string for.
• bufferCapacityInput is the capacity of the buffer, or 0 to indicate a request to retrieve the required capacity.
• bufferCountOutput is a pointer to the count of characters written (including the terminating '\0'), or a pointer to the required capacity in the case that bufferCapacityInput is insufficient.
• buffer is a pointer to an application-allocated buffer that will be filled with the semantic path string. It can be NULL if bufferCapacityInput is 0.
• See Buffer Size Parameters chapter for a detailed description of retrieving the required buffer size.

xrPathToString retrieves the path name string associated with an XrPath, in the context of a given
**Valid Usage (Implicit)**

- `instance` must be a valid `XrInstance` handle
- `bufferCountOutput` must be a pointer to a `uint32_t` value
- If `bufferCapacityInput` is not 0, `buffer` must be a pointer to an array of `bufferCapacityInput` char values

**Return Codes**

**Success**
- `XR_SUCCESS`

**Failure**
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SIZE_INSUFFICIENT`
- `XR_ERROR_PATH_INVALID`

### 6.3. Reserved Paths

In order for some uses of semantic paths to work consistently across runtimes, it is necessary to standardize several paths and require each runtime to use the same paths or patterns of paths for certain classes of usage. Those paths are as follows.

#### 6.3.1. /user paths

Some paths are used to refer to entities that are filling semantic roles in the system. These paths are all under the `/user` subtree.

The reserved user paths are:
Reserved Semantic Paths

- `/user/hand/left` represents the user’s left hand. It might be tracked using a controller or other device in the user’s left hand, or tracked without the user holding anything, e.g. using computer vision.

- `/user/hand/right` represents the user’s right hand in analog to the left hand.

- `/user/head` represents inputs on the user’s head, often from a device such as a head-mounted display. To reason about the user’s head, see the `XR_REFERENCE_SPACE_TYPE_VIEW` reference space.

- `/user/gamepad` is a two-handed gamepad device held by the user.

- `/user/treadmill` is a treadmill or other locomotion-targeted input device.

Runtimes are not required to provide interaction at all of these paths. For instance, in a system with no hand tracking, only `/user/head` would be active for interaction. In a system with only one controller, the runtime may provide access to that controller via either `/user/hand/left` or `/user/hand/right` as it deems appropriate.

The runtime may change the devices referred to by `/user/hand/left` and `/user/hand/right` at any time.

If more than two hand-held controllers or devices are active, the runtime must determine which two are accessible as `/user/hand/left` and `/user/hand/right`.

### 6.3.2. Input subpaths

Devices on the source side of the input system need to define paths for each component that can be bound to an action. This section describes the naming conventions for those input components. Runtimes must ignore input source paths that use identifiers and component names that do not appear in this specification or otherwise do not follow the pattern specified below.

Each input source path must match the following pattern:

- `.../input/<identifier>[_<location>][/<component>]`

Identifiers are often the label on the component or related to the type and location of the component.

When specifying a suggested binding there are several cases where the component part of the path can be determined automatically. See Suggested Bindings for more details.

See Interaction Profiles for examples of input subpaths.

### Standard identifiers

- trackpad - A 2D input source that usually includes click and touch component.

- thumbstick - A small 2D joystick that is meant to be used with the user’s thumb. These sometimes
include click and/or touch components.

- joystick - A 2D joystick that is meant to be used with the user’s entire hand, such as a flight stick. These generally do not have click component, but might have touch components.

- trigger - A 1D analog input component that returns to a rest state when the user stops interacting with it. These sometime include touch and/or click components.

- throttle - A 1D analog input component that remains in position when the user stops interacting with it.

- trackball - A 2D relative input source. These sometimes include click components.

- pedal - A 1D analog input component that is similar to a trigger but meant to be operated by a foot

- system - A button with the specialised meaning that it enables the user to access system-level functions and UI. Input data from system buttons is generally used internally by runtimes and may not be available to applications.

- dpad_up, dpad_down, dpad_left, and dpad_right - A set of buttons arranged in a plus shape.

- diamond_up, diamond_down, diamond_left, and diamond_right - Gamepads often have a set of four buttons arranged in a diamond shape. The labels on those buttons vary from gamepad to gamepad, but their arrangement is consistent. These names are used for the A/B/X/Y buttons on a Xbox controller, and the square/cross/circle/triangle button on a PlayStation controller.

- a, b, x, y, start, home, end, select - Standalone buttons are named for their physical labels. These are the standard identifiers for such buttons. Extensions may add new identifiers as detailed in the next section. Groups of four buttons in a diamond shape should use the diamond-prefix names above instead of using the labels on the buttons themselves.

- volume_up, volume_down, mute_mic, play_pause, menu, view, back - Some other standard controls are often identified by icons. These are their standard names.

- thumbrest - Some controllers have a place for the user to rest their thumb.

- shoulder - A button that is usually pressed with the index finger and is often positioned above a trigger.

- squeeze - An input source that indicates that the user is squeezing their fist closed. This could be a simple button or act more like a trigger. Sources with this identifier should either follow button or trigger conventions for their components.

- wheel - A steering wheel.

**Standard pose identifiers**

Input sources whose orientation and/or position are tracked also expose pose identifiers.

Standard pose identifiers for tracked hands or motion controllers as represented by `/user/hand/left` and `/user/hand/right` are:
Figure 2. Example grip and aim poses for generic motion controllers

- **grip** - A pose that allows applications to reliably render a virtual object held in the user’s hand, whether it is tracked directly or by a motion controller. The grip pose is defined as follows:
  - The grip position:
    - For tracked hands: The user’s palm centroid when closing the fist, at the surface of the palm.
    - For handheld motion controllers: A fixed position within the controller that generally lines up with the palm centroid when held by a hand in a neutral position. This position should be adjusted left or right to center the position within the controller’s grip.
  - The grip orientation’s +X axis: When you completely open your hand to form a flat 5-finger pose, the ray that is normal to the user’s palm (away from the palm in the left hand, into the palm in the right hand).
  - The grip orientation’s -Z axis: When you close your hand partially (as if holding the controller), the ray that goes through the center of the tube formed by your non-thumb fingers, in the direction of little finger to thumb.
  - The grip orientation’s +Y axis: orthogonal to +Z and +X using the right-hand rule.

- **aim** - A pose that allows applications to point in the world using the input source, according to the platform’s conventions for aiming with that kind of source. The aim pose is defined as follows:
  - For tracked hands: The ray that follows platform conventions for how the user aims at objects in the world with their entire hand, with +Y up, +X to the right, and -Z forward. The ray chosen will be runtime-dependent, for example, a ray emerging from the palm parallel to the forearm.
  - For handheld motion controllers: The ray that follows platform conventions for how the user targets objects in the world with the motion controller, with +Y up, +X to the right, and -Z
forward. This is usually for applications that are rendering a model matching the physical controller, as an application rendering a virtual object in the user's hand likely prefers to point based on the geometry of that virtual object. The ray chosen will be runtime-dependent, although this will often emerge from the frontmost tip of a motion controller.

**Standard locations**

When a single device contains multiple input sources that use the same identifier, a location suffix is added to create a unique identifier for that input source.

Standard locations are:

- left
- right
- left_upper
- left_lower
- right_upper
- right_lower
- upper
- lower

**Standard components**

Components are named for the specific boolean, scalar, or other value of the input source. Standard components are:

- click - A physical switch has been pressed by the user. This is valid for all buttons, and is common for trackpads, thumbsticks, triggers, and dpads. "click" components are always boolean.

- touch - The user has touched the input source. This is valid for all trackpads, and may be present for any other kind of input source if the device includes the necessary sensor. "touch" components are always boolean.

- force - A 1D scalar value that represents the user applying force to the input. It varies from 0 to 1, with 0 being the rest state. This is present for any input source with a force sensor.

- value - A 1D scalar value that varies from 0 to 1, with 0 being the rest state. This is present for triggers, throttles, and pedals. It may also be present for squeeze or other components.

- x, y - scalar components of 2D values. These vary in value from -1 to 1. These represent the 2D position of the input source with 0 being the rest state on each axis. -1 means all the way left for x axis or all the way down for y axis. +1 means all the way right for x axis or all the way up for y axis. x and y components are present for trackpads, thumbsticks, and joysticks.

- twist - Some sources, such as flight sticks, have a sensor that allows the user to twist the input left or right. For this component -1 means all the way left and 1 means all the way right.
• pose - The orientation and/or position of this input source. This component may exist for dedicated pose identifiers like grip and aim, or may be defined on other identifiers such as trackpad to let applications reason about the surface of that part.

**Output paths**

Many devices also have subpaths for output features such as haptics. The runtime must ignore output component paths that do not follow the pattern:

• .../output/<output_identifier>[_<location>]

Standard output identifiers are:

• haptic - A haptic element like an LRA (Linear Resonant Actuator) or vibration motor

Devices which contain multiple haptic elements with the same output identifier must use a location suffix as specified above.

### 6.3.3. Adding input sources via extensions

Extensions may enable input source path identifiers, output source path identifiers, and component names that are not included in the core specification, subject to the following conditions:

• EXT extensions must include the _ext suffix on any identifier or component name. E.g. ...
  /input/newidentifier_ext/newcomponent_ext

• Vendor extensions must include the vendor’s tag as a suffix on any identifier or component name. E.g. ...
  /input/newidentifier_vendor/newcomponent_vendor (where "vendor" is replaced with the vendor’s actual extension tag.)

• Khronos (KHR) extensions may add undecorated identifier or component names.

These rules are in place to prevent extensions from adding first class undecorated names that become de facto standards. Runtimes must ignore input source paths that do not follow the restrictions above.

Extensions may also add new location suffixes, and may do so by adding a new identifier and location combination using the appropriate suffix. E.g. ...
  /input/newidentifier_newlocation_ext

### 6.4. Interaction Profile Paths

An interaction profile path identifies a collection of buttons and other input sources in a physical arrangement to allow applications and runtimes to coordinate action bindings.

Interaction profile paths are of the form:

• /interaction_profiles/<vendor_name>/<type_name>
6.4.1. Khronos Simple Controller Profile

Path: /interaction_profiles/khr/simple_controller

Valid for user paths:

- /user/hand/left
- /user/hand/right

This interaction profile provides basic pose, button, and haptic support for applications with simple input needs. There is no hardware associated with the profile, and runtimes which support this profile should map the input paths provided to whatever the appropriate paths are on the actual hardware.

Supported component paths:

- .../input/select/click
- .../input/menu/click
- .../input/grip/pose
- .../input/aim/pose
- .../output/haptic

6.4.2. Google Daydream Controller Profile

Path: /interaction_profiles/google/daydream_controller

Valid for user paths:

- /user/hand/left
- /user/hand/right

This interaction profile represents the input sources on the Google Daydream Controller.

Supported component paths:

- .../input/select/click
- .../input/trackpad/x
- .../input/trackpad/y
- .../input/trackpad/click
- .../input/trackpad/touch
- .../input/grip/pose
- .../input/aim/pose
### 6.4.3. HTC Vive Controller Profile

Path: `/interaction_profiles/htc/vive_controller`

Valid for user paths:

- `/user/hand/left`
- `/user/hand/right`

This interaction profile represents the input sources and haptics on the Vive Controller.

Supported component paths:

- `.../input/system/click` *(may not be available for application use)*
- `.../input/squeeze/click`
- `.../input/menu/click`
- `.../input/trigger/click`
- `.../input/trigger/value`
- `.../input/trackpad/x`
- `.../input/trackpad/y`
- `.../input/trackpad/click`
- `.../input/trackpad/touch`
- `.../input/grip/pose`
- `.../input/aim/pose`
- `.../output/haptic`

### 6.4.4. HTC Vive Pro Profile

Path: `/interaction_profiles/htc/vive_pro`

Valid for user paths:

- `/user/head`

This interaction profile represents the input sources on the Vive Pro headset.

Supported component paths:

- `.../input/system/click` *(may not be available for application use)*
- `.../input/volume_up/click`
- `.../input/volume_down/click`
6.4.5. Microsoft Mixed Reality Motion Controller Profile

Path: /interaction_profiles/microsoft/motion_controller

Valid for user paths:

- /user/hand/left
- /user/hand/right

This interaction profile represents the input sources and haptics on the Microsoft Mixed Reality Controller.

Supported component paths:

- .../input/mute_mic/click
- .../input/menu/click
- .../input/squeeze/click
- .../input/trigger/value
- .../input/thumbstick/x
- .../input/thumbstick/y
- .../input/thumbstick/click
- .../input/trackpad/x
- .../input/trackpad/y
- .../input/trackpad/click
- .../input/trackpad/touch
- .../input/grip/pose
- .../input/aim/pose
- .../output/haptic

6.4.6. Microsoft Xbox Controller Profile

Path: /interaction_profiles/microsoft/xbox_controller

Valid for user paths:

- /user/gamepad

This interaction profile represents the input sources and haptics on the Microsoft Xbox Controller.

Supported component paths:
6.4.7. Oculus Go Controller Profile

Path: /interaction_profiles/oculus/go_controller

Valid for user paths:

- /user/hand/left
- /user/hand/right

This interaction profile represents the input sources on the Oculus Go controller.
Supported component paths:

- `/input/system/click` *(may not be available for application use)*
- `/input/trigger/click`
- `/input/back/click`
- `/input/trackpad/x`
- `/input/trackpad/y`
- `/input/trackpad/click`
- `/input/trackpad/touch`
- `/input/grip/pose`
- `/input/aim/pose`

### 6.4.8. Oculus Touch Controller Profile

Path: `/interaction_profiles/oculus/touch_controller`

Valid for user paths:

- `/user/hand/left`
- `/user/hand/right`

This interaction profile represents the input sources and haptics on the Oculus Touch controller.

Supported component paths:

- On `/user/hand/left` only:
  - `/input/x/click`
  - `/input/x/touch`
  - `/input/y/click`
  - `/input/y/touch`
  - `/input/menu/click`

- On `/user/hand/right` only:
  - `/input/a/click`
  - `/input/a/touch`
  - `/input/b/click`
  - `/input/b/touch`
  - `/input/system/click` *(may not be available for application use)*

- `/input/squeeze/value`
6.4.9. Valve Index Controller Profile

Path: /interaction_profiles/valve/index_controller

Valid for user paths:

- /user/hand/left
- /user/hand/right

This interaction profile represents the input sources and haptics on the Valve Index controller.

Supported component paths:

- .../input/system/click (may not be available for application use)
- .../input/system/touch (may not be available for application use)
- .../input/a/click
- .../input/a/touch
- .../input/b/click
- .../input/b/touch
- .../input/squeeze/value
- .../input/squeeze/force
- .../input/trigger/click
- .../input/trigger/value
- .../input/trigger/touch
- .../input/thumbstick/x
- .../input/thumbstick/y
- /input/thumbstick/click
- /input/thumbstick/touch
- /input/trackpad/x
- /input/trackpad/y
- /input/trackpad/force
- /input/trackpad/touch
- /input/grip/pose
- /input/aim/pose
- /output/haptic
Chapter 7. Spaces

Across both virtual reality and augmented reality, XR applications have a core need to map the location of virtual objects to the corresponding real-world locations where they will be rendered. **Spaces** allow applications to explicitly create and specify the frames of reference in which they choose to track the real world, and then determine how those frames of reference move relative to one another over time.

Spaces are represented by `XrSpace` handles, which the application creates and then uses in API calls. Whenever an application calls a function that returns coordinates, it provides an `XrSpace` to specify the frame of reference in which those coordinates will be expressed. Similarly, when providing coordinates to a function, the application specifies which `XrSpace` the runtime should use to interpret those coordinates.

OpenXR defines a set of well-known **reference spaces** that applications use to bootstrap their spatial reasoning. These reference spaces are: `VIEW`, `LOCAL` and `STAGE`. Each reference space has a well-defined meaning, which establishes where its origin is positioned and how its axes are oriented.

Runtimes whose tracking systems improve their understanding of the world over time may track spaces independently. For example, even though a `LOCAL` space and a `STAGE` space each map their origin to a static position in the world, a runtime with an inside-out tracking system may introduce slight adjustments to the origin of each space on a continuous basis to keep each origin in place.

Beyond well-known reference spaces, runtimes expose other independently-tracked spaces, such as a pose action space that tracks the pose of a motion controller over time.

When one or both spaces are tracking a dynamic object, passing in an updated time to `xrLocateSpace` each frame will result in an updated relative pose. For example, the location of the left hand’s pose action space in the `STAGE` reference space will change each frame as the user’s hand moves relative to the stage’s predefined origin on the floor. In other XR APIs, it is common to report the "pose" of an object relative to some presumed underlying global space. This API is careful to not explicitly define such an underlying global space, because it does not apply to all systems. Some systems will support no `STAGE` space, while others may support a `STAGE` space that switches between various physical stages with dynamic availability. To satisfy this wide variability, "poses" are always described as the relationship between two spaces.

Some devices improve their understanding of the world as the device is used. The location returned by `xrLocateSpace` in later frames may change over time, even for spaces that track static objects, as either the target space or base space adjusts its origin.

Composition layers submitted by the application include an `XrSpace` for the runtime to use to position that layer over time. Composition layers whose `XrSpace` is relative to the `VIEW` reference space are
implicitly "head-locked", even if they may not be "display-locked" for non-head-mounted form factors.

### 7.1. Reference Spaces

An XrSpace handle for a reference space is created using `xrCreateReferenceSpace`, by specifying the chosen reference space type and a pose within the natural reference frame defined for that reference space type.

Runtimes implement well-known reference spaces from `XrReferenceSpaceType` if they support tracking of that kind:

```c
typedef enum XrReferenceSpaceType {
    XR_REFERENCE_SPACE_TYPE_VIEW = 1,
    XR_REFERENCE_SPACE_TYPE_LOCAL = 2,
    XR_REFERENCE_SPACE_TYPE_STAGE = 3,
    XR_REFERENCE_SPACE_TYPE_MAX_ENUM = 0x7FFFFFFF
} XrReferenceSpaceType;
```

Available reference space types are indicated by `xrEnumerateReferenceSpaces`. Note that other spaces can be created as well, such as pose action spaces created by `xrCreateActionSpace`, which are not enumerated by that API.
Enumerant Descriptions

- **XR_REFERENCE_SPACE_TYPE_VIEW.** The **VIEW** space tracks the view origin used to generate view transforms for the primary viewer (or centroid of view origins if stereo), with +Y up, +X to the right, and -Z forward. This space points in the forward direction for the viewer without incorporating the user’s eye orientation, and is not gravity-aligned.

**VIEW** space is primarily useful when projecting from the user’s perspective into another space to obtain a targeting ray, or when rendering small head-locked content such as a reticle. Content rendered in **VIEW** space will stay at a fixed point on head-mounted displays and may be uncomfortable to view if too large. To obtain the ideal view and projection transforms to use each frame for rendering world content, applications should call `xrLocateViews` instead of using this space.

Runtimes **must** support this reference space.

- **XR_REFERENCE_SPACE_TYPE_LOCAL.** The **LOCAL** reference space establishes a world-locked origin, gravity-aligned to exclude pitch and roll, with +Y up, +X to the right, and -Z forward. This space locks in both its initial position and orientation, which the runtime **may** define to be either the initial position at application launch or some other calibrated zero position.

**LOCAL** space is useful when an application needs to render seated-scale content that is not positioned relative to the physical floor.

When a user needs to recenter **LOCAL** space, a runtime **may** offer some system-level recentering interaction that is transparent to the application, but which causes the current leveled head space to become the new **LOCAL** space. When such a recentering occurs, the runtime **must** queue the `XrEventDataReferenceSpaceChangePending` event, with the recentered **LOCAL** space origin only taking effect for `xrLocateSpace` or `xrLocateViews` calls whose `XrTime` parameter is greater than or equal to the `changeTime` provided in that event.

When views, controllers or other spaces experience tracking loss relative to the **LOCAL** space, runtimes **should** continue to provide inferred or last-known `position` and `orientation` values. These inferred poses can, for example, be based on neck model updates, inertial dead reckoning, or a last-known position, so long as it is still reasonable for the application to use that pose. While a runtime is providing position data, it **must** continue to set `XR_SPACE_LOCATION_POSITION_VALID_BIT` and `XR_VIEW_STATE_POSITION_VALID_BIT` but it **can** clear `XR_SPACE_LOCATION_POSITION_TRACKED_BIT` and `XR_VIEW_STATE_POSITION_TRACKED_BIT` to indicate that the position is inferred or last-known in this way.

When tracking is recovered, runtimes **should** snap the pose of other spaces back into position relative to the **LOCAL** space’s original origin.

Runtimes **must** support this reference space.

- **XR_REFERENCE_SPACE_TYPE_STAGE.** The **STAGE** reference space is a runtime-defined flat,
rectangular space that is empty and can be walked around on. The origin is on the floor at the center of the rectangle, with +Y up, and the X and Z axes aligned with the rectangle edges. The runtime may not be able to locate spaces relative to the STAGE reference space if the user has not yet defined one within the runtime-specific UI. Applications can use \texttt{xrGetReferenceSpaceBoundsRect} to determine the extents of the STAGE reference space's XZ bounds rectangle, if defined.

STAGE space is useful when an application needs to render standing-scale content (no bounds) or room-scale content (with bounds) that is relative to the physical floor.

When the user redefines the origin or bounds of the current STAGE space, or the runtime otherwise switches to a new STAGE definition, the runtime must queue the \texttt{XrEventDataReferenceSpaceChangePending} event, with the new STAGE space origin only taking effect for \texttt{xrLocateSpace} or \texttt{xrLocateViews} calls whose \texttt{XrTime} parameter is greater than or equal to the \texttt{changeTime} provided in that event.

When views, controllers or other spaces experience tracking loss relative to the STAGE space, runtimes should continue to provide inferred or last-known position and orientation values. These inferred poses can, for example, be based on neck model updates, inertial dead reckoning, or a last-known position, so long as it is still reasonable for the application to use that pose. While a runtime is providing position data, it must continue to set \texttt{XR_SPACE_LOCATION_POSITION_VALID_BIT} and \texttt{XR_VIEW_STATE_POSITION_VALID_BIT} but it can clear \texttt{XR_SPACE_LOCATION_POSITION_TRACKED_BIT} and \texttt{XR_VIEW_STATE_POSITION_TRACKED_BIT} to indicate that the position is inferred or last-known in this way.

When tracking is recovered, runtimes should snap the pose of other spaces back into position relative to the STAGE space's original origin.

XR systems may have limited real world spatial ranges in which users can freely move around while remaining tracked. Applications may wish to query these boundaries and alter application behavior or content placement to ensure the user can complete the experience while remaining within the boundary. Applications can query this information using \texttt{xrGetReferenceSpaceBoundsRect}.

When called, \texttt{xrGetReferenceSpaceBoundsRect} should return the extents of a rectangle that is clear of obstacles down to the floor, allowing where the user can freely move while remaining tracked, if available for that reference space. The returned extent represents the dimensions of an axis-aligned bounding box where the \texttt{XrExtent2Df::width} and \texttt{XrExtent2Df::height} fields correspond to the X and Z axes of the provided space, with the extents centered at the origin of the space. Not all systems or spaces may support boundaries. If a runtime is unable to provide bounds for a given space, \texttt{XR_SPACE_BOUNDS_UNAVAILABLE} will be returned and all fields of \texttt{bounds} will be set to 0.

The returned extents are expressed relative to the natural origin of the provided \texttt{XrReferenceSpaceType} and must not incorporate any origin offsets specified by the application during calls to \texttt{xrCreateReferenceSpace}. 
The runtime must return XR_ERROR_REFERENCE_SPACE_UNSUPPORTED if the XrReferenceSpaceType passed in createInfo is not supported by this session.

When a runtime will begin operating with updated space bounds, the runtime must queue a corresponding XrEventDataReferenceSpaceChangePending event.

```
XrResult xrGetReferenceSpaceBoundsRect(
    XrSession session,
    XrReferenceSpaceType referenceSpaceType,
    XrExtent2Df* bounds);
```

### Parameter Descriptions

- **type** is the XrStructureType of this structure.
- **session** is a handle to an XrSession previously created with xrCreateSession.
- **referenceSpaceType** is the reference space type whose bounds should be retrieved.
- **bounds** is the returned space extents.

### Valid Usage (Implicit)

- **session** must be a valid XrSession handle
- **referenceSpaceType** must be a valid XrReferenceSpaceType value
- **bounds** must be a pointer to an XrExtent2Df structure
Return Codes

Success
• XR_SUCCESS
• XR_SESSION_LOSS_PENDIND
• XR_SPACE_BOUNDS_UNAVAILABLE

Failure
• XR_ERROR_FUNCTION_UNSUPPORTED
• XR_ERROR_VALIDATION_FAILURE
• XR_ERROR_RUNTIME_FAILURE
• XR_ERROR_HANDLE_INVALID
• XR_ERROR_INSTANCE_LOST
• XR_ERROR_SESSION_LOST
• XR_ERROR_REFERENCE_SPACE_UNSUPPORTED

The XrEventDataReferenceSpaceChangePending event is sent to the application to notify it that the origin (and perhaps the bounds) of a reference space is changing. This may occur due to the user recentering the space explicitly, or the runtime otherwise switching to a different space definition.

The reference space change must only take effect for xrLocateSpace or xrLocateViews calls whose XrTime parameter is greater than or equal to the changeTime provided in that event. Runtimes should provide a changeTime to applications that allows for a deep render pipeline to present frames that are already in flight using the previous definition of the space. Runtimes should choose a changeTime that is midway between the displayTime of future frames to avoid threshold issues with applications that calculate future frame times using displayPeriod.

The pose provided here must only describe the change in the natural origin of the reference space and must not incorporate any origin offsets specified by the application during calls to xrCreateReferenceSpace. If the runtime does not know the location of the space's new origin relative to its previous origin, poseValid must be false, and the position and orientation of poseInPreviousSpace are undefined.
typedef struct XrEventDataReferenceSpaceChangePending {
    XrStructureType     type;
    const void*         next;
    XrSession           session;
    XrReferenceSpaceType referenceSpaceType;
    XrTime              changeTime;
    XrBool32            poseValid;
    XrPosef             poseInPreviousSpace;
} XrEventDataReferenceSpaceChangePending;

Member Descriptions

• type is the XrStructureType of this structure.

• next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

• session is the XrSession for which the reference space is changing.

• referenceSpaceType is the XrReferenceSpaceType that is changing.

• changeTime is the target XrTime after which xrLocateSpace or xrLocateViews will return values that respect this change.

• poseValid is true if the runtime can determine the pose of the new space in the previous space before the change.

• poseInPreviousSpace is an XrPosef defining the position and orientation of the new reference space's natural origin within the natural reference frame of its previous space.

Valid Usage (Implicit)

• type must be XR_TYPE_EVENT_DATA_REFERENCE_SPACE_CHANGE_PENDING

• next must be NULL or a valid pointer to the next structure in a structure chain

• session must be a valid XrSession handle

• referenceSpaceType must be a valid XrReferenceSpaceType value

7.2. Action Spaces

An XrSpace handle for a pose action is created using xrCreateActionSpace, by specifying the chosen pose action and a pose within the action's natural reference frame.

Runtimes support suggested pose action bindings to well-known user paths with ../pose subpaths if they support tracking for that particular identifier.
Some example well-known pose action paths:

- /user/hand/left/input/grip
- /user/hand/left/input/aim
- /user/hand/right/input/grip
- /user/hand/right/input/aim

For definitions of these well-known pose device paths, see the discussion of device input subpaths in the Semantic Paths chapter.

### 7.2.1. Action Spaces Lifetime

**XrSpace** handles created for a pose action **must** be unlocatable unless the action set that contains the corresponding pose action was set as active via the most recent **xrSyncActions** call. If the underlying device that is active for the action changes, the device this space is tracking **must** only change to track the new device when **xrSyncActions** is called.

If **xrLocateSpace** is called with an unlocatable action space, the implementation **must** return no position or orientation and both **XR_SPACE_LOCATION_POSITION_VALID_BIT** and **XR_SPACE_LOCATION_ORIENTATION_VALID_BIT** **must** be unset. If **XrSpaceVelocity** is also supplied, **XR_SPACE_VELOCITY_LINEAR_VALID_BIT** and **XR_SPACE_VELOCITY_ANGULAR_VALID_BIT** **must** be unset. If **xrLocateViews** is called with an unlocatable action space, the implementation **must** return no position or orientation and both **XR_VIEW_STATE_POSITION_VALID_BIT** and **XR_VIEW_STATE_ORIENTATION_VALID_BIT** **must** be unset.

### 7.3. Space Lifecycle

There are a small set of core APIs that allow applications to reason about reference spaces, action spaces, and their relative locations.

#### 7.3.1. xrEnumerateReferenceSpaces

The **xrEnumerateReferenceSpaces** function is defined as:

```c
XRResult xrEnumerateReferenceSpaces(  
    XrSession session,  
    uint32_t spaceCapacityInput,  
    uint32_t* spaceCountOutput,  
    XrReferenceSpaceType* spaces);  
```
Parameter Descriptions

- **session** is a handle to an `XrSession` previously created with `xrCreateSession`.
- **spaceCapacityInput** is the capacity of the spaces array, or 0 to indicate a request to retrieve the required capacity.
- **spaceCountOutput** is a pointer to the count of spaces written, or a pointer to the required capacity in the case that `spaceCapacityInput` is insufficient.
- **spaces** is a pointer to an application-allocated array that will be filled with the enumerant of each supported reference space. It can be `NULL` if `spaceCapacityInput` is 0.
- See **Buffer Size Parameters** chapter for a detailed description of retrieving the required `spaces` size.

Enumerates the set of reference space types that this runtime supports for a given session. Runtimes **must** always return identical buffer contents from this enumeration for the lifetime of the session.

If a session enumerates support for a given reference space type, calls to `xrCreateReferenceSpace` must succeed for that session, with any transient unavailability of poses expressed later during calls to `xrLocateSpace`.

Valid Usage (Implicit)

- **session** **must** be a valid `XrSession` handle
- **spaceCountOutput** **must** be a pointer to a `uint32_t` value
- If `spaceCapacityInput` is not 0, **spaces** **must** be a pointer to an array of `spaceCapacityInput` `XrReferenceSpaceType` values
Return Codes

Success

• XR_SUCCESS
• XR_SESSION_LOSS_PENDING

Failure

• XR_ERROR_VALIDATION_FAILURE
• XR_ERROR_RUNTIME_FAILURE
• XR_ERROR_HANDLE_INVALID
• XR_ERROR_INSTANCE_LOST
• XR_ERROR_SESSION_LOST
• XR_ERROR_SIZE_INSUFFICIENT

7.3.2. xrCreateReferenceSpace

The `xrCreateReferenceSpace` function is defined as:

```c
XrResult xrCreateReferenceSpace(
    XrSession session,
    const XrReferenceSpaceCreateInfo* createInfo,
    XrSpace* space);
```

Parameter Descriptions

• `session` is a handle to an `XrSession` previously created with `xrCreateSession`.
• `createInfo` is the `XrReferenceSpaceCreateInfo` used to specify the space.
• `space` is the returned space handle.

Creates an `XrSpace` handle based on a chosen reference space. Application can provide an `XrPosef` to define the position and orientation of the new space's origin within the natural reference frame of the reference space.

Multiple `XrSpace` handles may exist simultaneously, up to some limit imposed by the runtime. The `XrSpace` handle must be eventually freed via the `xrDestroySpace` function.

The runtime must return `XR_ERROR_REFERENCE_SPACE_UNSUPPORTED` if the given reference space type is not
supported by this session.

Valid Usage (Implicit)

- session must be a valid XrSession handle
- createInfo must be a pointer to a valid XrReferenceSpaceCreateInfo structure
- space must be a pointer to an XrSpace handle

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_LIMIT_REACHED
- XR_ERROR_REFERENCE_SPACE_UNSUPPORTED
- XR_ERROR_POSE_INVALID

The XrReferenceSpaceCreateInfo structure is defined as:

```c
typedef struct XrReferenceSpaceCreateInfo {
    XrStructureType         type;
    const void*             next;
    XrReferenceSpaceType    referenceSpaceType;
    XrPosef                 poseInReferenceSpace;
} XrReferenceSpaceCreateInfo;
```
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **referenceSpaceType** is the chosen XrReferenceSpaceType.
- **poseInReferenceSpace** is an XrPosef defining the position and orientation of the new space's origin within the natural reference frame of the reference space.

Valid Usage (Implicit)

- **type** must be XR_TYPE_REFERENCE_SPACE_CREATE_INFO
- **next** must be NULL or a valid pointer to the next structure in a structure chain
- **referenceSpaceType** must be a valid XrReferenceSpaceType value

7.3.3. xrCreateActionSpace

The xrCreateActionSpace function is defined as:

```c
XrResult xrCreateActionSpace(
    XrSession                                   session,
    const XrActionSpaceCreateInfo*              createInfo,
    XrSpace*                                    space);
```

Parameter Descriptions

- **session** is the XrSession to create the action space in.
- **createInfo** is the XrActionSpaceCreateInfo used to specify the space.
- **space** is the returned space handle.

Creates an XrSpace handle based on a chosen pose action. Application can provide an XrPosef to define the position and orientation of the new space’s origin within the natural reference frame of the action space.

Multiple XrSpace handles may exist simultaneously, up to some limit imposed by the runtime. The XrSpace handle must be eventually freed via the xrDestroySpace function or by destroying the parent
The runtime must return XR_ERROR_ACTION_TYPE_MISMATCH if the action provided in action is not of type XR_ACTION_TYPE_POSE_INPUT.

Valid Usage (Implicit)

- session must be a valid XrSession handle
- createInfo must be a pointer to a valid XrActionSpaceCreateInfo structure
- space must be a pointer to an XrSpace handle

Return Codes

Success
- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_LIMIT_REACHED
- XR_ERROR_POSE_INVALID
- XR_ERROR_PATH_UNSUPPORTED
- XR_ERROR_PATH_INVALID
- XR_ERROR_ACTION_TYPE_MISMATCH

The XrActionSpaceCreateInfo structure is defined as:
typedef struct XrActionSpaceCreateInfo {
    XrStructureType    type;
    const void*        next;
    XrAction           action;
    XrPath             subactionPath;
    XrPosef            poseInActionSpace;
} XrActionSpaceCreateInfo;

**Member Descriptions**

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **action** is a handle to a pose XrAction previously created with xrCreateAction.
- **subactionPath** is XR_NULL_PATH or an XrPath that was specified when the action was created. If subactionPath is a valid path not specified when the action was created the runtime must return XR_ERROR_PATH_UNSUPPORTED. If this parameter is set, the runtime must create a space that is relative to only that subaction’s pose binding.
- **poseInActionSpace** is an XrPosef defining the position and orientation of the new space’s origin within the natural reference frame of the pose action.

**Valid Usage (Implicit)**

- **type** must be XR_TYPE_ACTION_SPACE_CREATE_INFO
- **next** must be NULL or a valid pointer to the next structure in a structure chain
- **action** must be a valid XrAction handle

### 7.3.4. xrDestroySpace

The xrDestroySpace function is defined as:

```c
XrResult xrDestroySpace(
    XrSpace             space);
```
Parameter Descriptions

- `space` is a handle to an `XrSpace` previously created by a function such as `xrCreateReferenceSpace`.

`XrSpace` handles are destroyed using `xrDestroySpace`. The runtime may still use this space if there are active dependencies (e.g., compositions in progress).

Valid Usage (Implicit)

- `space` must be a valid `XrSpace` handle

Thread Safety

- Access to `space`, and any child handles, must be externally synchronized

Return Codes

Success
- `XR_SUCCESS`

Failure
- `XR_ERROR_HANDLE_INVALID`

7.4. Locating Spaces

Applications use the `xrLocateSpace` function to find the pose of an `XrSpace`'s origin within a base `XrSpace` at a given historical or predicted time. If an application wants to know the velocity of the space's origin, it can chain an `XrSpaceVelocity` structure to the `next` pointer of the `XrSpaceLocation` structure when calling the `xrLocateSpace` function. Applications should inspect the output `XrSpaceLocationFlagBits` and `XrSpaceVelocityFlagBits` to determine the validity and tracking status of the components of the location.

7.4.1. `xrLocateSpace`

`xrLocateSpace` provides the physical location of a space in a base space at a specified time, if currently known by the runtime.
XrResult xrLocateSpace(
    XrSpace                                     space,
    XrSpace                                     baseSpace,
    XrTime                                      time,
    XrSpaceLocation*                            location);

Parameter Descriptions

- **space** identifies the target space to locate.
- **baseSpace** identifies the underlying space in which to locate **space**.
- **time** is the time for which the location should be provided.
- **location** provides the location of **space** in **baseSpace**.

For a **time** in the past, the runtime **should** locate the spaces based on the runtime's most accurate current understanding of how the world was at that historical time.

For a **time** in the future, the runtime **should** locate the spaces based on the runtime's most up-to-date prediction of how the world will be at that future time.

The minimum valid range of values for **time** are described in Prediction Time Limits. For values of **time** outside this range, **xrLocateSpace** **may** return a location with no position and **XR_SPACE_LOCATION_POSITION_VALID_BIT** unset.

Some devices improve their understanding of the world as the device is used. The location returned by **xrLocateSpace** for a given **space**, **baseSpace** and **time** **may** change over time, even for spaces that track static objects, as one or both spaces adjust their origins.

During tracking loss of **space** relative to **baseSpace**, runtimes **should** continue to provide inferred or last-known position and orientation values. These inferred poses can, for example, be based on neck model updates, inertial dead reckoning, or a last-known position, so long as it is still reasonable for the application to use that pose. While a runtime is providing position data, it **must** continue to set **XR_SPACE_LOCATION_POSITION_VALID_BIT** but it **can** clear **XR_SPACE_LOCATION_POSITION_TRACKED_BIT** to indicate that the position is inferred or last-known in this way.

If the runtime has not yet observed even a last-known pose for how to locate **space** in **baseSpace** (e.g. one space is an action space bound to a motion controller that has not yet been detected, or the two spaces are in disconnected fragments of the runtime's tracked volume), the runtime **should** return a location with no position and **XR_SPACE_LOCATION_POSITION_VALID_BIT** unset.

The runtime **must** return a location with both **XR_SPACE_LOCATION_POSITION_VALID_BIT** and **XR_SPACE_LOCATION_POSITION_TRACKED_BIT** set when locating **space** and **baseSpace** if both spaces were created relative to the same entity (e.g. two action spaces for the same action), even if the entity is currently untracked. The location in this case is the difference in the two spaces' application-specified...
transforms relative to that common entity.

The runtime should return a location with `XR_SPACE_LOCATION_POSITION_VALID_BIT` set and `XR_SPACE_LOCATION_POSITION_TRACKED_BIT` unset for spaces tracking two static entities in the world when their relative pose is known to the runtime. This enables applications to make use of the runtime's latest knowledge of the world, even during tracking loss.

If an `XrSpaceVelocity` structure is chained to the next pointer of `XrSpaceLocation` and the velocity is observed or can be calculated by the runtime, the runtime must fill in the linear velocity of the origin of space within the reference frame of `baseSpace` and set the `XR_SPACE_VELOCITY_LINEAR_VALID_BIT`. Similarly, if an `XrSpaceVelocity` structure is chained to the next pointer of `XrSpaceLocation` and the angular velocity is observed or can be calculated by the runtime, the runtime must fill in the angular velocity of the origin of space within the reference frame of `baseSpace` and set the `XR_SPACE_VELOCITY_ANGULAR_VALID_BIT`.

The following example code shows how an application can get both the location and velocity of a space within a base space using the `xrLocateSpace` function by chaining an `XrSpaceVelocity` to the next pointer of `XrSpaceLocation` and calling `xrLocateSpace`.

```c
XrSpace space;      // previously initialized
XrSpace baseSpace;  // previously initialized
XrTime time;        // previously initialized

XrSpaceVelocity velocity {XR_TYPE_SPACE VELOCITY};
XrSpaceLocation location {XR_TYPE_SPACE_LOCATION, &velocity};
xrLocateSpace(space, baseSpace, time, &location);
```

**Valid Usage (Implicit)**

- `space` must be a valid `XrSpace` handle
- `baseSpace` must be a valid `XrSpace` handle
- `location` must be a pointer to an `XrSpaceLocation` structure
- Both of `baseSpace` and `space` must have been created, allocated, or retrieved from the same `XrSession`
Return Codes

Success
• XR_SUCCESS
• XR_SESSION_LOSS_PENDING

Failure
• XR_ERROR_VALIDATION_FAILURE
• XR_ERROR_RUNTIME_FAILURE
• XR_ERROR_HANDLE_INVALID
• XR_ERROR_INSTANCE_LOST
• XR_ERROR_SESSION_LOST
• XR_ERROR_TIME_INVALID

The XrSpaceLocation structure is defined as:

typedef struct XrSpaceLocation {
    XrStructureType         type;
    void*                   next;
    XrSpaceLocationFlags    locationFlags;
    XrPosef                 pose;
} XrSpaceLocation;

Member Descriptions

• type is the XrStructureType of this structure.

• next is NULL or a pointer to the next structure in a structure chain, such as XrSpaceVelocity.

• locationFlags is a bitfield, with bit masks defined in XrSpaceLocationFlagBits, to indicate which members contain valid data. If none of the bits are set, no other fields in this structure should be considered to be valid or meaningful.

• pose is an XrPosef defining the position and orientation of the origin of xrLocateSpace::space within the reference frame of xrLocateSpace::baseSpace.
Valid Usage (Implicit)

- **type** must be `XR_TYPE_SPACE_LOCATION`
- **next** must be `NULL` or a valid pointer to the next structure in a structure chain. See also: `XrSpaceVelocity`
- **locationFlags** must be `0` or a valid combination of `XrSpaceLocationFlagBits` values

The `locationFlags` member is a bitwise-OR of zero or more of the following flags:

```c
// Flag bits for XrSpaceLocationFlags
static const XrSpaceLocationFlags XR_SPACE_LOCATION_ORIENTATION_VALID_BIT = 0x00000001;
static const XrSpaceLocationFlags XR_SPACE_LOCATION_POSITION_VALID_BIT = 0x00000002;
static const XrSpaceLocationFlags XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT = 0x00000004;
static const XrSpaceLocationFlags XR_SPACE_LOCATION_POSITION_TRACKED_BIT = 0x00000008;
```

where the flags have the following meaning:
Flag Descriptions

• **XR_SPACE_LOCATION_ORIENTATION_VALID_BIT** indicates that the pose field's orientation field contains valid data. For a space location tracking a device with its own inertial tracking, **XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT** should remain set when this bit is set. Applications must not read the pose field's orientation if this flag is unset.

• **XR_SPACE_LOCATION_POSITION_VALID_BIT** indicates that the pose field's position field contains valid data. When a space location loses tracking, runtimes should continue to provide valid but untracked position values that are inferred or last-known, so long as it's still meaningful for the application to use that position, clearing **XR_SPACE_LOCATION_POSITION_TRACKED_BIT** until positional tracking is recovered. Applications must not read the pose field's position if this flag is unset.

• **XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT** indicates that the pose field's orientation field represents an actively tracked orientation. For a space location tracking a device with its own inertial tracking, this bit should remain set when **XR_SPACE_LOCATION_ORIENTATION_VALID_BIT** is set. For a space location tracking an object whose orientation is no longer known during tracking loss (e.g. an observed QR code), runtimes should continue to provide valid but untracked orientation values, so long as it's still meaningful for the application to use that orientation.

• **XR_SPACE_LOCATION_POSITION_TRACKED_BIT** indicates that the pose field's position field represents an actively tracked position. When a space location loses tracking, runtimes should continue to provide valid but untracked position values that are inferred or last-known, e.g. based on neck model updates, inertial dead reckoning, or a last-known position, so long as it's still meaningful for the application to use that position.

The **XrSpaceVelocity** structure is defined as:

typedef struct XrSpaceVelocity {
    XrStructureType         type;
    void*                   next;
    XrSpaceVelocityFlags    velocityFlags;
    XrVector3f              linearVelocity;
    XrVector3f              angularVelocity;
} XrSpaceVelocity;
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **velocityFlags** is a bitfield, with bit masks defined in XrSpaceVelocityFlagBits, to indicate which members contain valid data. If none of the bits are set, no other fields in this structure should be considered to be valid or meaningful.
- **linearVelocity** is the relative linear velocity of xrLocateSpace::space with respect to and expressed in the reference frame of xrLocateSpace::baseSpace, in units of meters per second.
- **angularVelocity** is the relative angular velocity of xrLocateSpace::space with respect to xrLocateSpace::baseSpace. The vector's direction is expressed in the reference frame of xrLocateSpace::baseSpace and is parallel to the rotational axis of xrLocateSpace::space. The vector's magnitude is the relative angular speed of xrLocateSpace::space in radians per second. The vector follows the right-hand rule for torque/rotation.

Valid Usage (Implicit)

- **type** must be XR_TYPE_SPACE_VELOCITY
- **next** must be NULL or a valid pointer to the next structure in a structure chain
- **velocityFlags** must be 0 or a valid combination of XrSpaceVelocityFlagBits values

The velocityFlags member is a bitwise-OR of zero or more of the following flags:

```c
// Flag bits for XrSpaceVelocityFlags
static const XrSpaceVelocityFlags XR_SPACE_VELOCITY_LINEAR_VALID_BIT = 0x00000001;
static const XrSpaceVelocityFlags XR_SPACE_VELOCITY_ANGULAR_VALID_BIT = 0x00000002;
```

where the flags have the following meaning:
Flag Descriptions

- **XR_SPACE_VELOCITY_LINEAR_VALID_BIT** — Indicates that the `linearVelocity` member contains valid data. Applications **must** not read the `linearVelocity` field if this flag is unset.

- **XR_SPACE_VELOCITY_ANGULAR_VALID BIT** — Indicates that the `angularVelocity` member contains valid data. Applications **must** not read the `angularVelocity` field if this flag is unset.
Chapter 8. View Configurations

A **view configuration** is a semantically meaningful set of one or more views for which an application can render images. A **primary view configuration** is a view configuration intended to be presented to the viewer interacting with the XR application. This distinction allows the later addition of additional views, for example views which are intended for spectators.

A typical head-mounted VR system has a view configuration with two views, while a typical phone-based AR system has a view configuration with a single view. A simple multi-wall projection-based (CAVE-like) VR system may have a view configuration with at least one view for each display surface (wall, floor, ceiling) in the room.

For any supported form factor, a system will support one or more primary view configurations. Supporting more than one primary view configuration can be useful if a system supports a special view configuration optimized for the hardware but also supports a more broadly used view configuration as a compatibility fallback.

View configurations are identified with an `XrViewConfigurationType`.

### 8.1. Primary View Configurations

```c
typedef enum XrViewConfigurationType {
    XR_VIEW_CONFIGURATION_TYPE_PRIMARY_MONO = 1,
    XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO = 2,
    XR_VIEW_CONFIGURATION_TYPE_MAX_ENUM = 0x7FFFFFFF
} XrViewConfigurationType;
```

The application selects its primary view configuration type when calling `xrBeginSession`, and that configuration remains constant for the lifetime of the session, until `xrEndSession` is called.

The number of views and the semantic meaning of each view index within a given view configuration is well-defined, specified below for all core view configurations. The predefined primary view configuration types are:
Enumerant Descriptions

- **XR_VIEW_CONFIGURATION_TYPE_PRIMARY_MONO.** One view representing the form factor's one primary display. For example, an AR phone's screen. This configuration requires one element in `XrViewConfigurationProperties` and one projection in each `XrCompositionLayerProjection` layer.

- **XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO.** Two views representing the form factor's two primary displays, which map to a left-eye and right-eye view. This configuration requires two views in `XrViewConfigurationProperties` and two views in each `XrCompositionLayerProjection` layer. View index 0 must represent the left eye and view index 1 must represent the right eye.

### 8.2. View Configuration API

First an application needs to select which primary view configuration it wants to use. If it supports multiple configurations, an application can call `xrEnumerateViewConfigurations` before creating an `XrSession` to get a list of the view configuration types supported for a given system.

The application can then call `xrGetViewConfigurationProperties` and `xrEnumerateViewConfigurationViews` to get detailed information about each view configuration type and its individual views.

#### 8.2.1. xrEnumerateViewConfigurations

The `xrEnumerateViewConfigurations` function is defined as:

```c
XrResult xrEnumerateViewConfigurations(
    XrInstance instance,
    XrSystemId systemId,
    uint32_t viewConfigurationTypeCapacityInput,
    uint32_t* viewConfigurationTypeCountOutput,
    XrViewConfigurationType* viewConfigurationTypes);
```
Parameter Descriptions

- **instance** is the instance from which **systemId** was retrieved.
- **systemId** is the **XrSystemId** whose view configurations will be enumerated.
- **viewConfigurationsTypeCapacityInput** is the capacity of the **viewConfigurations** array, or 0 to indicate a request to retrieve the required capacity.
- **viewConfigurationsTypeCountOutput** is a pointer to the count of **viewConfigurations** written, or a pointer to the required capacity in the case that **viewConfigurationsTypeCapacityInput** is insufficient.
- **viewConfigurationsTypes** is a pointer to an array of **XrViewConfigurationType** values, but can be **NULL** if **viewConfigurationsTypeCapacityInput** is 0.
- See **Buffer Size Parameters** chapter for a detailed description of retrieving the required **viewConfigurations** size.

**xrEnumerateViewConfigurations** enumerates the view configuration types supported by the **XrSystemId**. The supported set for that system **must** not change during the lifetime of its **XrInstance**. The returned list of primary view configurations **should** be in order from what the runtime considered highest to lowest user preference. Thus the first enumerated view configuration type **should** be the one the runtime prefers the application to use if possible.

Runtimes **must** always return identical buffer contents from this enumeration for the given **systemId** and for the lifetime of the instance.

Valid Usage (Implicit)

- **instance** **must** be a valid **XrInstance** handle
- **viewConfigurationTypeCountOutput** **must** be a pointer to a **uint32_t** value
- If **viewConfigurationTypeCapacityInput** is not 0, **viewConfigurationsTypes** **must** be a pointer to an array of **viewConfigurationTypeCapacityInput** **XrViewConfigurationType** values
### Return Codes

#### Success
- XR_SUCCESS

#### Failure
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SIZE_INSUFFICIENT
- XR_ERROR_SYSTEM_INVALID

### 8.2.2. xrGetViewConfigurationProperties

The `xrGetViewConfigurationProperties` function is defined as:

```c
XrResult xrGetViewConfigurationProperties(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    XrViewConfigurationType                     viewConfigurationType,
    XrViewConfigurationProperties*              configurationProperties);
```

#### Parameter Descriptions

- `instance` is the instance from which `systemId` was retrieved.
- `systemId` is the `XrSystemId` whose view configuration is being queried.
- `viewConfigurationType` is the `XrViewConfigurationType` of the configuration to get.
- `configurationProperties` is a pointer to view configuration properties to return.

`xrGetViewConfigurationProperties` queries properties of an individual view configuration. Applications **must** use one of the supported view configuration types returned by `xrEnumerateViewConfigurations`. If `viewConfigurationType` is not supported by this `XrInstance` the runtime **must** return `XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED`. 
Valid Usage (Implicit)

- `instance` must be a valid `XrInstance` handle
- `viewConfigurationType` must be a valid `XrViewConfigurationType` value
- `configurationProperties` must be a pointer to an `XrViewConfigurationProperties` structure

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED`
- `XR_ERROR_SYSTEM_INVALID`

8.2.3. `XrViewConfigurationProperties`

The `XrViewConfigurationProperties` structure is defined as:

```c
typedef struct XrViewConfigurationProperties {
    XrStructureType            type;
    void*                      next;
    XrViewConfigurationType    viewConfigurationType;
    XrBool32                   fovMutable;
} XrViewConfigurationProperties;
```
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **viewConfigurationType** is the XrViewConfigurationType of the configuration.
- **fovMutable** indicates if the view field of view can be modified by the application.

Valid Usage (Implicit)

- **type** must be XR_TYPE_VIEW_CONFIGURATION_PROPERTIES
- **next** must be NULL or a valid pointer to the next structure in a structure chain
- **viewConfigurationType** must be a valid XrViewConfigurationType value

8.2.4. xrEnumerateViewConfigurationViews

The **xrEnumerateViewConfigurationViews** function is defined as:

```c
XrResult xrEnumerateViewConfigurationViews(
    XrInstance instance, 
    XrSystemId systemId, 
    XrViewConfigurationType viewConfigurationType, 
    uint32_t viewCapacityInput, 
    uint32_t* viewCountOutput, 
    XrViewConfigurationView* views);
```
Parameter Descriptions

- **instance** is the instance from which **systemId** was retrieved.
- **systemId** is the XrSystemId whose view configuration is being queried.
- **viewConfigurationType** is the XrViewConfigurationType of the configuration to get.
- **viewCapacityInput** is the capacity of the views array, or 0 to indicate a request to retrieve the required capacity.
- **viewCountOutput** is a pointer to the count of views written, or a pointer to the required capacity in the case that **viewCapacityInput** is 0.
- **views** is a pointer to an array of XrViewConfigurationView values, but **can** be NULL if **viewCapacityInput** is 0.

Each XrViewConfigurationType defines the number of views associated with it. Applications can query more details of each view element using xrEnumerateViewConfigurationViews. If the supplied viewConfigurationType is not supported by this XrInstance and XrSystemId, the runtime **must** return XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED.

Runtimes **must** always return identical buffer contents from this enumeration for the given systemId and viewConfigurationType for the lifetime of the instance.

Valid Usage (Implicit)

- **instance** **must** be a valid XrInstance handle
- **viewConfigurationType** **must** be a valid XrViewConfigurationType value
- **viewCountOutput** **must** be a pointer to a uint32_t value
- If **viewCapacityInput** is not 0, **views** **must** be a pointer to an array of **viewCapacityInput** XrViewConfigurationView structures
Return Codes

Success
  • XR_SUCCESS

Failure
  • XR_ERROR_VALIDATION_FAILURE
  • XR_ERROR_RUNTIME_FAILURE
  • XR_ERROR_HANDLE_INVALID
  • XR_ERROR_INSTANCE_LOST
  • XR_ERROR_SIZE_INSUFFICIENT
  • XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED
  • XR_ERROR_SYSTEM_INVALID

8.2.5. XrViewConfigurationView

Each XrViewConfigurationView specifies properties related to rendering of an individual view within a view configuration.

The XrViewConfigurationView structure is defined as:

typedef struct XrViewConfigurationView {
    XrStructureType   type;
    void*             next;
    uint32_t           recommendedImageRectWidth;
    uint32_t           maxImageRectWidth;
    uint32_t           recommendedImageRectHeight;
    uint32_t           maxImageRectHeight;
    uint32_t           recommendedSwapchainSampleCount;
    uint32_t           maxSwapchainSampleCount;
} XrViewConfigurationView;
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **recommendedImageRectWidth** is the optimal width of imageRect to use when rendering this view into a swapchain.
- **maxImageRectWidth** is the maximum width of imageRect supported when rendering this view into a swapchain.
- **recommendedImageRectHeight** is the optimal height of imageRect to use when rendering this view into a swapchain.
- **maxImageRectHeight** is the maximum height of imageRect supported when rendering this view into a swapchain.
- **recommendedSwapchainSampleCount** is the recommended number of sub-data element samples to create for each swapchain image that will be rendered into for this view.
- **maxSwapchainSampleCount** is the maximum number of sub-data element samples supported for swapchain images that will be rendered into for this view.

See XrSwapchainSubImage for more information about imageRect values, and XrSwapchainCreateInfo for more information about creating swapchains appropriately sized to support those imageRect values.

The array of XrViewConfigurationView returned by the runtime must adhere to the rules defined in XrViewConfigurationType, such as the count and association to the left and right eyes.

Valid Usage (Implicit)

- **type** must be XR_TYPE_VIEW_CONFIGURATION_VIEW
- **next** must be NULL or a valid pointer to the next structure in a structure chain

8.3. Example View Configuration Code

```c
XrInstance instance; // previously initialized
XrSystemId system;   // previously initialized
XrSession session;   // previously initialized
XrSpace sceneSpace;  // previously initialized

// Enumerate the view configurations paths.
uint32_t configurationCount;
CHK_XR(xrEnumerateViewConfigurations(instance, system, 0, &configurationCount, nullptr));
```
std::vector<XrViewConfigurationType> configurationTypes(configurationCount);
CHK_XR(xrEnumerateViewConfigurations(instance, system, configurationCount,
&configurationCount, configurationTypes.data()));

bool configFound = false;
for(uint32_t i = 0; i < configurationCount; ++i)
{
    if (configurationTypes[i] == XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO)
    {
        configFound = true;
        break;  // Pick the first supported, i.e. preferred, view configuration.
    }
}

if (!configFound)
    return;   // Cannot support any view configuration of this system.

// Get detailed information of each view element.
uint32_t viewCount;
CHK_XR(xrEnumerateViewConfigurationViews(instance, system,
    XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO,
    0,
    &viewCount,
    nullptr));

std::vector<XrViewConfigurationView> configViews(viewCount,
    {XR_TYPE_VIEW_CONFIGURATION_VIEW});
CHK_XR(xrEnumerateViewConfigurationViews(instance, system,
    XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO,
    viewCount,
    &viewCount,
    configViews.data()));

// Set the primary view configuration for the session.
XrSessionBeginInfo beginInfo = {XR_TYPE_SESSION_BEGIN_INFO};
beginInfo.primaryViewConfigurationType = XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO;
CHK_XR(xrBeginSession(session, &beginInfo));

// Allocate a buffer according to viewCount.
std::vector<XrView> views(viewCount, {XR_TYPE_VIEW});

// Run a per-frame loop.
while (!quit)
{
    // Wait for a new frame.
    XrFrameWaitInfo frameWaitInfo{XR_TYPE_FRAME_WAIT_INFO};
    XrFrameState frameState{XR_TYPE_FRAME_STATE};
CHK_XR(xrWaitFrame(session, &frameWaitInfo, &frameState));

// Begin frame immediately before GPU work
XrFrameBeginInfo frameBeginInfo { XR_TYPE_FRAME_BEGIN_INFO };  
CHK_XR(xrBeginFrame(session, &frameBeginInfo));

std::vector<XrCompositionLayerBaseHeader*> layers;
XrCompositionLayerProjectionView projViews[2] = { /*...*/ };  
XrCompositionLayerProjection layerProj{ XR_TYPE_COMPOSITION_LAYER_PROJECTION};

if (frameState.shouldRender) {
    XrViewLocateInfo viewLocateInfo{XR_TYPE_VIEW_LOCATE_INFO};
    viewLocateInfo.displayTime = frameState.predictedDisplayTime;
    viewLocateInfo.space = sceneSpace;

    XrViewState viewState{XR_TYPE_VIEW_STATE};
    XrView views[2] = { {XR_TYPE_VIEW}, {XR_TYPE_VIEW}};
    uint32_t viewCountOutput;
    CHK_XR(xrLocateViews(session, &viewLocateInfo, &viewState, configViews.size(), &viewCountOutput, views));

    // ...
    // Use viewState and frameState for scene render, and fill in projViews[2]
    // ...

    // Assemble composition layers structure
    layerProj.layerFlags = XR_COMPOSITION_LAYER_BLEND_TEXTURE_SOURCE_ALPHA_BIT;
    layerProj.space = sceneSpace;
    layerProj.viewCount = 2;
    layerProj.views = projViews;
    layers.push_back(reinterpret_cast<XrCompositionLayerBaseHeader*>(&layerProj));
}

// End frame and submit layers, even if layers is empty due to shouldRender = false
XrFrameEndInfo frameEndInfo{ XR_TYPE_FRAME_END_INFO};
frameEndInfo.displayTime = frameState.predictedDisplayTime;
frameEndInfo.environmentBlendMode = XR_ENVIRONMENT_BLEND_MODE_OPAQUE;
frameEndInfo.layerCount = (uint32_t)layers.size();
frameEndInfo.layers = layers.data();
CHK_XR(xrEndFrame(session, &frameEndInfo));
Chapter 9. Session

A session represents an application's intention to display XR content to the user.

9.1. Session Lifecycle

A typical XR session coordinates the application and the runtime through session control functions and session state events.

1. The application creates a session by choosing a system and a graphics API and passing them into xrCreateSession. The newly created session is in the XR_SESSION_STATE_IDLE state.

2. The application then monitors for session state changes via XrEventDataSessionStateChanged events.

3. When the runtime determines that the system is ready to start transitioning to this session's XR content, the application receives a notification of session state change.
to \texttt{XR\_SESSION\_STATE\_READY}. Once the application is also ready to proceed and display its XR content, it calls \texttt{xrBeginSession} and starts its frame loop, which begins a running session.

4. While the session is running, the application is expected to continuously execute its frame loop by calling \texttt{xrWaitFrame}, \texttt{xrBeginFrame} and \texttt{xrEndFrame} each frame, establishing synchronization with the runtime. Once the runtime is synchronized with the application's frame loop and ready to display application's frames, the session moves into the \texttt{XR\_SESSION\_STATE\_SYNCHRONIZED} state. In this state, the submitted frames will not be displayed or visible to the user yet.

5. When the runtime intends to display frames from the application, it notifies with \texttt{XR\_SESSION\_STATE\_VISIBLE} state, and sets \texttt{XrFrameState::shouldRender} to \texttt{true} in \texttt{xrWaitFrame}. The application should render XR content and submit the composition layers to \texttt{xrEndFrame}.

6. When the runtime determines the application is eligible to receive XR inputs, e.g. motion controller or hand tracking inputs, it notifies with \texttt{XR\_SESSION\_STATE\_FOCUSED} state. The application can expect to receive active action inputs.

7. When the runtime determines the application has lost XR input focus, it moves the session state from \texttt{XR\_SESSION\_STATE\_FOCUSED} to \texttt{XR\_SESSION\_STATE\_VISIBLE} state. The application may need to change its own internal state while input is unavailable. Since the session is still visible, the application needs to render and submit frames at full frame rate, but may wish to change visually to indicate its input suspended state. When the runtime returns XR focus back to the application, it moves the session state back to \texttt{XR\_SESSION\_STATE\_FOCUSED}.

8. When the runtime needs to end a running session due to the user closing or switching the application, the runtime will change the session state through appropriate intermediate ones and finally to \texttt{XR\_SESSION\_STATE\_STOPPING}. When the application receives the \texttt{XR\_SESSION\_STATE\_STOPPING} event, it should stop its frame loop and then call \texttt{xrEndSession} to tell the runtime to stop the running session.

9. After \texttt{xrEndSession}, the runtime transitions the session state to \texttt{XR\_SESSION\_STATE\_IDLE}. If the XR session is temporarily paused in the background, the runtime will keep the session state at \texttt{XR\_SESSION\_STATE\_IDLE} and later transition the session state back to \texttt{XR\_SESSION\_STATE\_READY} when the XR session is resumed. If the runtime determines that its use of this XR session has concluded, it will transition the session state from \texttt{XR\_SESSION\_STATE\_IDLE} to \texttt{XR\_SESSION\_STATE\_EXITING}.

10. When the application receives the \texttt{XR\_SESSION\_STATE\_EXITING} event, it releases the resources related to the session and calls \texttt{xrDestroySession}.

A session is considered \textbf{running} after a successful call to \texttt{xrBeginSession} and remains running until any call is made to \texttt{xrEndSession}. Certain functions are only valid to call when a session is running, such as \texttt{xrWaitFrame}, or else the \texttt{XR\_ERROR\_SESSION\_NOT\_RUNNING} error must be returned by the runtime.
A session is considered **not running** before a successful call to `xrBeginSession` and becomes not running again after any call is made to `xrEndSession`. Certain functions are only valid to call when a session is not running, such as `xrBeginSession`, or else the `XR_ERROR_SESSION_RUNNING` error **must** be returned by the runtime.

If an error is returned from `xrBeginSession`, the session remains in its current running or not running state. Calling `xrEndSession` always transitions a session to the not running state, regardless of any errors returned.

Only running sessions may become focused sessions that receive XR input. When a session is **not running**, the application **must** not submit frames. This is important because without a running session, the runtime no longer has to spend resources on sub-systems (tracking etc.) that are no longer needed by the application.

An application **must** call `xrBeginSession` when the session is in the `XR_SESSION_STATE_READY` state, or `XR_ERROR_SESSION_NOT_READY` will be returned; it **must** call `xrEndSession` when the session is in the `XR_SESSION_STATE_STOPPING` state, otherwise `XR_ERROR_SESSION_NOT_STOPPING` will be returned. This is to allow the runtimes to seamlessly transition from one application's session to another.

The application **can** call `xrDestroySession` at any time during the session life cycle, however, it **must** stop using the `XrSession` handle immediately in all threads and stop using any related resources. Therefore, it's typically undesirable to destroy a running session and instead it's recommended to wait for `XR_SESSION_STATE_EXITING` to destroy a session.

### 9.2. Session Creation

To present graphical content on an output device, OpenXR applications need to pick a graphics API which is supported by the runtime. Unextended OpenXR does not support any graphics APIs natively but provides a number of extensions of which each runtime can support any subset. These extensions can be activated during `XrInstance` create time.

During `XrSession` creation the application **must** provide information about which graphics API it intends to use by adding an `XrGraphicsBinding*` struct of one (and only one) of the enabled graphics API extensions to the next chain of `XrSessionCreateInfo`. The application **must** call the `xrGet*GraphicsRequirements` method (where * is a placeholder) provided by the chosen graphics API extension before attempting to create the session (for example, `xrGetD3D11GraphicsRequirementsKHR` `xrGetD3D12GraphicsRequirementsKHR` `xrGetOpenGLGraphicsRequirementsKHR` `xrGetVulkanGraphicsRequirementsKHR` `xrGetVulkanGraphicsRequirements2KHR`).

Unless specified differently in the graphics API extension, the application is responsible for creating a valid graphics device binding based on the requirements returned by `xrGet*GraphicsRequirements` methods (for details refer to the extension specification of the graphics API).

The `xrCreateSession` function is defined as:
XrResult xrCreateSession(  
    XrInstance instance,  
    const XrSessionCreateInfo* createInfo,  
    XrSession* session);  

Parameter Descriptions

- **instance** is the instance from which `systemId` was retrieved.
- **createInfo** is a pointer to an `XrSessionCreateInfo` structure containing information about how to create the session.
- **session** is a pointer to a handle in which the created `XrSession` is returned.

Creates a session using the provided `createInfo` and returns a handle to that session. This session is created in the `XR_SESSION_STATE_IDLE` state, and a corresponding `XrEventDataSessionStateChanged` event to the `XR_SESSION_STATE_IDLE` state **must** be generated as the first such event for the new session.

The runtime **must** return `XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING` (**XR_ERROR_VALIDATION_FAILURE** may be returned due to legacy behavior) on calls to `xrCreateSession` if a function named like `xrGet*GraphicsRequirements` has not been called for the same `instance` and `XrSessionCreateInfo::systemId`. (See graphics binding extensions for details.)

Valid Usage (Implicit)

- **instance** **must** be a valid `XrInstance` handle
- **createInfo** **must** be a pointer to a valid `XrSessionCreateInfo` structure
- **session** **must** be a pointer to an `XrSession` handle
Return Codes

Success
- XR_SUCCESS

Failure
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_LIMIT_REACHED
- XR_ERROR_SYSTEM_INVALID
- XR_ERROR_INITIALIZATION_FAILED
- XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING
- XR_ERROR_GRAPHICS_DEVICE_INVALID

The XrSessionCreateInfo structure is defined as:

```c
typedef struct XrSessionCreateInfo {
    XrStructureType         type;
    const void*             next;
    XrSessionCreateFlags    createFlags;
    XrSystemId              systemId;
} XrSessionCreateInfo;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR. Note that in most cases one graphics API extension specific struct needs to be in this next chain.
- `createFlags` identifies XrSessionCreateFlags that apply to the creation.
- `systemId` is the XrSystemId representing the system of devices to be used by this session.
Valid Usage

- `systemId` must be a valid `XrSystemId` or `XR_ERROR_SYSTEM_INVALID` must be returned.
- `next`, unless otherwise specified via an extension, must contain exactly one graphics API binding structure (a structure whose name begins with “XrGraphicsBinding”) or `XR_ERROR_GRAPHICS_DEVICE_INVALID` must be returned.

Valid Usage (Implicit)

- `type` must be `XR_TYPE_SESSION_CREATE_INFO`
- `next` must be `NULL` or a valid pointer to the next structure in a structure chain. See also: `XrGraphicsBindingD3D11KHR`, `XrGraphicsBindingD3D12KHR`, `XrGraphicsBindingOpenGLESAndroidKHR`, `XrGraphicsBindingOpenGLWaylandKHR`, `XrGraphicsBindingOpenGLWin32KHR`, `XrGraphicsBindingOpenGLXcbKHR`, `XrGraphicsBindingOpenGLXlibKHR`, `XrGraphicsBindingVulkanKHR`
- `createFlags` must be 0

The `XrSessionCreateFlags` include:

```c
// Flag bits for XrSessionCreateFlags
```

There are currently no session creation flags. This is reserved for future use.

The `xrDestroySession` function is defined as:

```c
XrResult xrDestroySession( 
    XrSession session); 
```

Parameter Descriptions

- `session` is the session to destroy.

`XrSession` handles are destroyed using `xrDestroySession`. When an `XrSession` is destroyed, all handles that are children of that `XrSession` are also destroyed.
The application is responsible for ensuring that it has no calls using `session` in progress when the session is destroyed.

`xrDestroySession` can be called when the session is in any session state.

### Valid Usage (Implicit)
- `session` must be a valid `XrSession` handle

### Thread Safety
- Access to `session`, and any child handles, must be externally synchronized

### Return Codes

#### Success
- `XR_SUCCESS`

#### Failure
- `XR_ERROR_HANDLE_INVALID`

## 9.3. Session Control

The `xrBeginSession` function is defined as:

```c
XrResult xrBeginSession(
    XrSession session,
    const XrSessionBeginInfo* beginInfo);
```

### Parameter Descriptions
- `session` is a valid `XrSession` handle.
- `beginInfo` is a pointer to an `XrSessionBeginInfo` structure.

When the application receives `XrEventDataSessionStateChanged` event with the `XR_SESSION_STATE_READY` state, the application should then call `xrBeginSession` to start rendering frames for display to the user.
After this function successfully returns, the session is considered to be running. The application should then start its frame loop consisting of some sequence of `xrWaitFrame/xrBeginFrame/xrEndFrame` calls.

If the session is already running when the application calls `xrBeginSession`, the runtime must return error `XR_ERROR_SESSION_RUNNING`. If the session is not running when the application calls `xrBeginSession`, but the session is not yet in the `XR_SESSION_STATE_READY` state, the runtime must return error `XR_ERROR_SESSION_NOT_READY`.

Note that a runtime may decide not to show the user any given frame from a session at any time, for example if the user has switched to a different application’s running session. The application should check whether `xrWaitFrame` returns an `XrFrameState` with `shouldRender` set to true before rendering a given frame to determine whether that frame will be visible to the user.

Runtime session frame state must start in a reset state when a session transitions to running so that no state is carried over from when the same session was previously running.

If `primaryViewConfigurationType` in `beginInfo` is not supported by the `XrSystemId` used to create the session, the runtime must return `XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED`.

---

### Valid Usage (Implicit)

- `session` must be a valid `XrSession` handle
- `beginInfo` must be a pointer to a valid `XrSessionBeginInfo` structure

---

### Return Codes

#### Success
- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

#### Failure
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED`
- `XR_ERROR_SESSION_RUNNING`
- `XR_ERROR_SESSION_NOT_READY`
The **XrSessionBeginInfo** structure is defined as:

```c
typedef struct XrSessionBeginInfo {
    XrStructureType            type;
    const void*                next;
    XrViewConfigurationType    primaryViewConfigurationType;
} XrSessionBeginInfo;
```

**Member Descriptions**

- **type** is the **XrStructureType** of this structure.
- **next** is **NULL** or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **primaryViewConfigurationType** is the **XrViewConfigurationType** to use during this session to provide images for the form factor’s primary displays.

**Valid Usage (Implicit)**

- **type** must be **XR_TYPE_SESSION_BEGIN_INFO**
- **next** must be **NULL** or a valid pointer to the next structure in a structure chain
- **primaryViewConfigurationType** must be a valid **XrViewConfigurationType** value

The **xrEndSession** function is defined as:

```c
XrResult xrEndSession(
    XrSession session);
```

**Parameter Descriptions**

- **session** is a handle to a running **XrSession**.

When the application receives **XrEventDataSessionStateChanged** event with the **XR_SESSION_STATE_STOPPING** state, the application should stop its frame loop and then call **xrEndSession** to end the running session. This function signals to the runtime that the application will no longer call **xrWaitFrame**, **xrBeginFrame** or **xrEndFrame** from any thread allowing the runtime to safely transition
the session to XR_SESSION_STATE_IDLE. The application must also avoid reading input state or sending haptic output after calling xrEndSession.

If the session is not running when the application calls xrEndSession, the runtime must return error XR_ERROR_SESSION_NOT_RUNNING. If the session is still running when the application calls xrEndSession, but the session is not yet in the XR_SESSION_STATE_STOPPING state, the runtime must return error XR_ERROR_SESSION_NOT_STOPPING.

If the application wishes to exit a running session, the application can call xrRequestExitSession so that the session transitions from XR_SESSION_STATE_IDLE to XR_SESSION_STATE_EXITING.

Valid Usage (Implicit)

- session must be a valid XrSession handle

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_SESSION_NOT_STOPPING
- XR_ERROR_SESSION_NOT_RUNNING

When an application wishes to exit a running session, it can call xrRequestExitSession, requesting that the runtime transition through the various intermediate session states including XR_SESSION_STATE_STOPPING to XR_SESSION_STATE_EXITING.

On platforms where an application’s lifecycle is managed by the system, session state changes may be implicitly triggered by application lifecycle state changes. On such platforms, using platform-specific methods to alter application lifecycle state may be the preferred method of provoking session state changes. The behavior of xrRequestExitSession is not altered, however explicit session exit may not interact with the platform-specific application lifecycle.

The xrRequestExitSession function is defined as:
XrResult xrRequestExitSession(
    XrSession session);

**Parameter Descriptions**

- `session` is a handle to a running `XrSession`.

If `session` is not running when `xrRequestExitSession` is called, `XR_ERROR_SESSION_NOT_RUNNING` must be returned.

**Valid Usage (Implicit)**

- `session` must be a valid `XrSession` handle

**Return Codes**

**Success**

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

**Failure**

- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_SESSION_NOT_RUNNING`

**9.4. Session States**

While events can be expanded upon, there are a minimum set of lifecycle events which can occur which all OpenXR applications must be aware of. These events are detailed below.
### 9.4.1. XrEventDataSessionStateChanged

The `XrEventDataSessionStateChanged` structure is defined as:

```c
typedef struct XrEventDataSessionStateChanged {
    XrStructureType type;
    const void*     next;
    XrSession       session;
    XrSessionState  state;
    XrTime          time;
} XrEventDataSessionStateChanged;
```

### Member Descriptions

- **type** is the `XrStructureType` of this structure.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **session** is the `XrSession` which has changed state.
- **state** is the current `XrSessionState` of the `session`.
- **time** is an `XrTime` which indicates the time of the state change.

Receiving the `XrEventDataSessionStateChanged` event structure indicates that the application has changed lifecycle state.

### Valid Usage (Implicit)

- **type** must be `XR_TYPE_EVENT_DATA_SESSION_STATE_CHANGED`
- **next** must be `NULL` or a valid pointer to the next structure in a structure chain
- **session** must be a valid `XrSession` handle
- **state** must be a valid `XrSessionState` value

The `XrSessionState` enumerates the possible session lifecycle states:
typedef enum XrSessionState {
    XR_SESSION_STATE_UNKNOWN = 0,
    XR_SESSION_STATE_IDLE = 1,
    XR_SESSION_STATE_READY = 2,
    XR_SESSION_STATE_SYNCHRONIZED = 3,
    XR_SESSION_STATE_VISIBLE = 4,
    XR_SESSION_STATE_FOCUSED = 5,
    XR_SESSION_STATE_STOPPING = 6,
    XR_SESSION_STATE_LOSS_PENDING = 7,
    XR_SESSION_STATE_EXITING = 8,
    XR_SESSION_STATE_MAX_ENUM = 0x7FFFFFFF
} XrSessionState;

**Enumerant Descriptions**

- **XR_SESSION_STATE_UNKNOWN**: An unknown state. The runtime **must** not return this value in an XrEventDataSessionStateChanged event.
- **XR_SESSION_STATE_IDLE**: The initial state after calling xrCreateSession or returned to after calling xrEndSession.
- **XR_SESSION_STATE_READY**: The application is ready to call xrBeginSession and sync its frame loop with the runtime.
- **XR_SESSION_STATE_SYNCHRONIZED**: The application has synced its frame loop with the runtime but is not visible to the user.
- **XR_SESSION_STATE_VISIBLE**: The application has synced its frame loop with the runtime and is visible to the user but cannot receive XR input.
- **XR_SESSION_STATE_FOCUSED**: The application has synced its frame loop with the runtime, is visible to the user and can receive XR input.
- **XR_SESSION_STATE_STOPPING**: The application should exit its frame loop and call xrEndSession.
- **XR_SESSION_STATE_LOSS_PENDING**: The session is in the process of being lost. The application should destroy the current session and can optionally recreate it.
- **XR_SESSION_STATE_EXITING**: The application should end its XR experience and not automatically restart it.

The **XR_SESSION_STATE_UNKNOWN** state **must** not be returned by the runtime, and is only defined to avoid 0 being a valid state.

Receiving the **XR_SESSION_STATE_IDLE** state indicates that the runtime considers the session is idle. Applications in this state **should** minimize resource consumption but continue to call xrPollEvent at some reasonable cadence.
Receiving the `XR_SESSION_STATE_READY` state indicates that the runtime desires the application to prepare rendering resources, begin its session and synchronize its frame loop with the runtime. The application does this by successfully calling `xrBeginSession` and then running its frame loop by calling `xrWaitFrame`, `xBeginFrame` and `xEndFrame` in a loop. If the runtime wishes to return the session to the `XR_SESSION_STATE_IDLE` state, it must wait until the application calls `xrBeginSession`. After returning from the `xrBeginSession` call, the runtime may then immediately transition forward through the `XR_SESSION_STATE_SYNCHRONIZED` state to the `XR_SESSION_STATE_STOPPING` state, to request that the application end this session. If the system supports a user engagement sensor and runtime is in `XR_SESSION_STATE_IDLE` state, the runtime should not transition to the `XR_SESSION_STATE_READY` state until the user starts engaging with the device.

Receiving the `XR_SESSION_STATE_SYNCHRONIZED` state indicates that the application has synchronized its frame loop with the runtime, but its frames are not visible to the user. The application should continue running its frame loop by calling `xrWaitFrame`, `xBeginFrame` and `xEndFrame`, although it should avoid heavy GPU work so that other visible applications can take CPU and GPU precedence. The application can save resources here by skipping rendering and not submitting any composition layers until `xrWaitFrame` returns an `XrFrameState` with `shouldRender` set to true. A runtime may use this frame synchronization to facilitate seamless switching from a previous XR application to this application on a frame boundary.

Receiving the `XR_SESSION_STATE_VISIBLE` state indicates that the application has synchronized its frame loop with the runtime, and the session’s frames will be visible to the user, but the session is not eligible to receive XR input. An application may be visible but not have focus, for example when the runtime is composing a modal pop-up on top of the application’s rendered frames. The application should continue running its frame loop, rendering and submitting its composition layers, although it may wish to pause its experience, as users cannot interact with the application at this time. It is important for applications to continue rendering when visible, even when they do not have focus, so the user continues to see something reasonable underneath modal pop-ups. Runtimes should make input actions inactive while the application is unfocused, and applications should react to an inactive input action by skipping rendering of that action’s input avatar (depictions of hands or other tracked objects controlled by the user).

Receiving the `XR_SESSION_STATE_FOCUSED` state indicates that the application has synchronized its frame loop with the runtime, the session’s frames will be visible to the user, and the session is eligible to receive XR input. The runtime should only give one session XR input focus at any given time. The application should be running its frame loop, rendering and submitting composition layers, including input avatars (depictions of hands or other tracked objects controlled by the user) for any input actions that are active. The runtime should avoid rendering its own input avatars when an application is focused, unless input from a given source is being captured by the runtime at the moment.

Receiving the `XR_SESSION_STATE_STOPPING` state indicates that the runtime has determined that the application should halt its rendering loop. Applications should exit their rendering loop and call `xEndSession` when in this state. A possible reason for this would be to minimize contention between multiple applications. If the system supports a user engagement sensor and the session is running, the runtime should transition to the `XR_SESSION_STATE_STOPPING` state when the user stops engaging with
Receiving the `XR_SESSION_STATE_EXITING` state indicates the runtime wishes the application to terminate its XR experience, typically due to a user request via a runtime user interface. Applications should gracefully end their process when in this state if they do not have a non-XR user experience.

Receiving the `XR_SESSION_STATE_LOSS_PENDING` state indicates the runtime is no longer able to operate with the current session, for example due to the loss of a display hardware connection. An application should call `xrDestroySession` and may end its process or decide to poll `xrGetSystem` at some reasonable cadence to get a new `XrSystemId`, and re-initialize all graphics resources related to the new system, and then create a new session using `xrCreateSession`. After the event is queued, subsequent calls to functions that accept `XrSession` parameters must no longer return any success code other than `XR_SESSION_LOSS_PENDING` for the given `XrSession` handle. The `XR_SESSION_LOSS_PENDING` success result is returned for an unspecified grace period of time, and the functions that return it simulate success in their behavior. If the runtime has no reasonable way to successfully complete a given function (e.g. `xrCreateSwapchain`) when a lost session is pending, or if the runtime is not able to provide the application a grace period, the runtime may return `XR_ERROR_SESSION_LOST`. Thereafter, functions which accept `XrSession` parameters for the lost session may return `XR_ERROR_SESSION_LOST` to indicate that the function failed and the given session was lost. The `XrSession` handle and child handles are henceforth unusable and should be destroyed by the application in order to immediately free up resources associated with those handles.
Chapter 10. Rendering

10.1. Swapchain Image Management

XR_DEFINE_HANDLE(XrSwapchain)

Normal XR applications will want to present rendered images to the user. To allow this, the runtime provides images organized in swapchains for the application to render into. The runtime must allow applications to create multiple swapchains.

Swapchain image format support by the runtime is specified by the `xrEnumerateSwapchainFormats` function. Runtimes should support R8G8B8A8 and R8G8B8A8 sRGB formats if possible.

Swapchain images can be 2D or 2D Array.

Rendering operations involving composition of submitted layers should be assumed to be internally performed by the runtime in linear color space. Images submitted in sRGB color space must be created using an API-specific sRGB format (e.g. `DXGI_FORMAT_R8G8B8A8_UNORM_SRGB`, `GL_SRGB8_ALPHA8`, `VK_FORMAT_R8G8B8A8_SRGB`) to apply automatic sRGB-to-linear conversion when read by the runtime. All other formats will be treated as linear values.

**Note**

OpenXR applications should avoid submitting linear encoded 8 bit color data (e.g. `DXGI_FORMAT_R8G8B8A8_UNORM`) whenever possible as it may result in color banding.


**Note**

DXGI resources will be created with their associated TYPELESS format, but the runtime will use the application-specified format for reading the data.

The `xrEnumerateSwapchainFormats` function is defined as:

The `xrEnumerateSwapchainFormats` function is defined as:
XrResult xrEnumerateSwapchainFormats(
    XrSession session,
    uint32_t formatCapacityInput,
    uint32_t* formatCountOutput,
    int64_t* formats);

Parameter Descriptions

• **session** is the session that enumerates the supported formats.

• **formatCapacityInput** is the capacity of the formats, or 0 to retrieve the required capacity.

• **formatCountOutput** is a pointer to the count of uint64_t formats written, or a pointer to the required capacity in the case that **formatCapacityInput** is insufficient.

• **formats** is a pointer to an array of int64_t format ids, but can be NULL if **formatCapacityInput** is 0. The format ids are specific to the specified graphics API.

• See **Buffer Size Parameters** chapter for a detailed description of retrieving the required formats size.

xrEnumerateSwapchainFormats enumerates the texture formats supported by the current session. The type of formats returned are dependent on the graphics API specified in **xrCreateSession**. For example, if a DirectX graphics API was specified, then the enumerated formats correspond to the DXGI formats, such as **DXGI_FORMAT_R8G8B8A8_UNORM_SRGB**. Texture formats should be in order from highest to lowest runtime preference. The application should use the highest preference format that it supports for optimal performance and quality.

With an OpenGL-based graphics API, the texture formats correspond to OpenGL internal formats.

With a Direct3D-based graphics API, xrEnumerateSwapchainFormats never returns typeless formats (e.g. **DXGI_FORMAT_R8G8B8A8_TYPELESS**). Only concrete formats are returned, and only concrete formats may be specified by applications for swapchain creation.

Runtimes must always return identical buffer contents from this enumeration for the lifetime of the session.

Valid Usage (Implicit)

• **session** must be a valid XrSession handle

• **formatCountOutput** must be a pointer to a uint32_t value

• If **formatCapacityInput** is not 0, **formats** must be a pointer to an array of **formatCapacityInput** int64_t values
Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_SIZE_INSUFFICIENT

XrSwapchainUsageFlags specify the intended usage of the swapchain images. When images are created, the runtime needs to know how the images are used in a way that requires more information than simply the image format. The XrSwapchainCreateInfo passed to xrCreateSwapchain should match the intended usage or else undefined behavior may result when the application works with the images.

Flags include:

```c
// Flag bits for XrSwapchainUsageFlags
static const XrSwapchainUsageFlags XR_SWAPCHAIN_USAGE_COLOR_ATTACHMENT_BIT = 0x00000001;
static const XrSwapchainUsageFlags XR_SWAPCHAIN_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT = 0x00000002;
static const XrSwapchainUsageFlags XR_SWAPCHAIN_USAGE_UNORDERED_ACCESS_BIT = 0x00000004;
static const XrSwapchainUsageFlags XR_SWAPCHAIN_USAGE_TRANSFER_SRC_BIT = 0x00000008;
static const XrSwapchainUsageFlags XR_SWAPCHAIN_USAGE_TRANSFER_DST_BIT = 0x00000010;
static const XrSwapchainUsageFlags XR_SWAPCHAIN_USAGE_SAMPLED_BIT = 0x00000020;
static const XrSwapchainUsageFlags XR_SWAPCHAIN_USAGE_MUTABLE_FORMAT_BIT = 0x00000040;
static const XrSwapchainUsageFlags XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_MND = 0x00000080;
static const XrSwapchainUsageFlags XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_KHR = XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_MND; // alias of XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_MND
```
Flag Descriptions

- **XR_SWAPCHAIN_USAGE_COLOR_ATTACHMENT_BIT** — Specifies that the image **may** be a color rendering target.
- **XR_SWAPCHAIN_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT** — Specifies that the image **may** be a depth/stencil rendering target.
- **XR_SWAPCHAIN_USAGE_UNORDERED_ACCESS_BIT** — Specifies that the image **may** be accessed out of order and that access **may** be via atomic operations.
- **XR_SWAPCHAIN_USAGE_TRANSFER_SRC_BIT** — Specifies that the image **may** be used as the source of a transfer operation.
- **XR_SWAPCHAIN_USAGE_TRANSFER_DST_BIT** — Specifies that the image **may** be used as the destination of a transfer operation.
- **XR_SWAPCHAIN_USAGE_SAMPLED_BIT** — Specifies that the image **may** be sampled by a shader.
- **XR_SWAPCHAIN_USAGE_MUTABLE_FORMAT_BIT** — Specifies that the image **may** be reinterpreted as another image format.
- **XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_MND** — Specifies that the image **may** be used as an input attachment. (Added by the [XR_MND_swapchain_usage_input_attachment_bit] extension)
- **XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_KHR** — Specifies that the image **may** be used as an input attachment. (Added by the XR_KHR_swapchain_usage_input_attachment_bit extension)

The `xrCreateSwapchain` function is defined as:

```c
XrResult xrCreateSwapchain(
    XrSession session,
    const XrSwapchainCreateInfo* createInfo,
    XrSwapchain* swapchain);
```

Parameter Descriptions

- `session` is the session that creates the image.
- `createInfo` is a pointer to an `XrSwapchainCreateInfo` structure containing parameters to be used to create the image.
- `swapchain` is a pointer to a handle in which the created `XrSwapchain` is returned.

Creates an `XrSwapchain` handle. The returned swapchain handle **may** be subsequently used in API calls. Multiple `XrSwapchain` handles may exist simultaneously, up to some limit imposed by the
The `XrSwapchain` handle must be eventually freed via the `xrDestroySwapchain` function. The runtime must return `XR_ERROR_SWAPCHAIN_FORMAT_UNSUPPORTED` if the image format specified in the `XrSwapchainCreateInfo` is unsupported. The runtime must return `XR_ERROR_FEATURE_UNSUPPORTED` if any bit of the create flags specified in the `XrSwapchainCreateInfo` is unsupported.

**Valid Usage (Implicit)**

- `session` must be a valid `XrSession` handle
- `createInfo` must be a pointer to a valid `XrSwapchainCreateInfo` structure
- `swapchain` must be a pointer to an `XrSwapchain` handle

**Return Codes**

**Success**

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

**Failure**

- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_Runtime_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_OUT_OF_MEMORY`
- `XR_ERROR_LIMIT_REACHED`
- `XR_ERROR_SWAPCHAIN_FORMAT_UNSUPPORTED`
- `XR_ERROR_FEATURE_UNSUPPORTED`

The `XrSwapchainCreateInfo` structure is defined as:
typedef struct XrSwapchainCreateInfo {
    XrStructureType      type;
    const void*           next;
    XrSwapchainCreateFlags createFlags;
    XrSwapchainUsageFlags usageFlags;
    int64_t               format;
    uint32_t              sampleCount;
    uint32_t              width;
    uint32_t              height;
    uint32_t              faceCount;
    uint32_t              arraySize;
    uint32_t              mipCount;
} XrSwapchainCreateInfo;
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **createFlags** is a bitmask of XrSwapchainCreateFlagBits describing additional properties of the swapchain.
- **usageFlags** is a bitmask of XrSwapchainUsageFlagBits describing the intended usage of the swapchain's images. The usage flags define how the corresponding graphics API objects are created. A mismatch may result in swapchain images that do not support the application's usage.
- **format** is a graphics API-specific texture format identifier. For example, if the graphics API specified in xrCreateSession is Vulkan, then this format is a Vulkan format such as VK_FORMAT_R8G8B8A8_SRGB. The format identifies the format that the runtime will interpret the texture as upon submission. Valid formats are indicated by xrEnumerateSwapchainFormats.
- **sampleCount** is the number of sub-data element samples in the image, must not be 0 or greater than the graphics API's maximum limit.
- **width** is the width of the image, must not be 0 or greater than the graphics API's maximum limit.
- **height** is the height of the image, must not be 0 or greater than the graphics API's maximum limit.
- **faceCount** is the number of faces, which can be either 6 (for cubemaps) or 1.
- **arraySize** is the number of array layers in the image or 1 for a 2D image, must not be 0 or greater than the graphics API's maximum limit.
- **mipCount** describes the number of levels of detail available for minified sampling of the image, must not be 0 or greater than the graphics API's maximum limit.

Valid Usage (Implicit)

- **type** must be XR_TYPE_SWAPCHAIN_CREATE_INFO
- **next** must be NULL or a valid pointer to the next structure in a structure chain
- **createFlags** must be 0 or a valid combination of XrSwapchainCreateFlagBits values
- **usageFlags** must be 0 or a valid combination of XrSwapchainUsageFlagBits values

The createFlags are a combination of the following:
// Flag bits for XrSwapchainCreateFlags
static const XrSwapchainCreateFlags XR_SWAPCHAIN_CREATE_PROTECTED_CONTENT_BIT = 0x00000001;
static const XrSwapchainCreateFlags XR_SWAPCHAIN_CREATE_STATIC_IMAGE_BIT = 0x00000002;

Flag Descriptions

- **XR_SWAPCHAIN_CREATE_PROTECTED_CONTENT_BIT** indicates that the swapchain's images will be protected from CPU access, using a mechanism such as Vulkan protected memory.

- **XR_SWAPCHAIN_CREATE_STATIC_IMAGE_BIT** indicates that the application will acquire and release only one image to this swapchain over its entire lifetime. The runtime **must** allocate only one swapchain image.

A runtime **may** implement any of these, but is not required to. A runtime **must** return **XR_ERROR_FEATURE_UNSUPPORTED** from **xrCreateSwapchain** if an **XrSwapchainCreateFlags** bit is requested but not implemented.

The number of images in each swapchain is implementation-defined except in the case of a static swapchain. To obtain the number of images actually allocated, call **xrEnumerateSwapchainImages**.

With a Direct3D-based graphics API, the swapchain returned by **xrCreateSwapchain** will be a typeless format if the requested format has a typeless analogue. Applications are required to reinterpret the swapchain as a compatible non-typeless type. Upon submitting such swapchains to the runtime, they are interpreted as the format specified by the application in the **XrSwapchainCreateInfo**.

Swapchains will be created with graphics API-specific flags appropriate to the type of underlying image and its usage. Extensions may exist to further assist the runtime in choosing how to create swapchains.

Runtimes **must** honor underlying graphics API limits when creating resources.

**xrEnumerateSwapchainFormats** never returns typeless formats (e.g. **DXGI_FORMAT_R8G8B8A8_TYPELESS**). Only concrete formats are returned, and only concrete formats may be specified by applications for swapchain creation.

The **xrDestroySwapchain** function is defined as:

```
XrResult xrDestroySwapchain(
    XrSwapchain swapchain);
```
Parameter Descriptions

- **swapchain** is the swapchain to destroy.

All submitted graphics API commands that refer to **swapchain** must have completed execution. Runtimes **may** continue to utilize swapchain images after `xrDestroySwapchain` is called.

Valid Usage (Implicit)

- **swapchain** must be a valid XrSwapchain handle

Thread Safety

- Access to **swapchain**, and any child handles, **must** be externally synchronized

Return Codes

**Success**

- XR_SUCCESS

**Failure**

- XR_ERROR_HANDLE_INVALID

Swapchain images are acquired, waited on, and released by index, but the number of images in a swapchain is implementation-defined. Additionally, rendering to images requires access to the underlying image primitive of the graphics API being used. Applications may query and cache the images at any time after swapchain creation.

The `xrEnumerateSwapchainImages` function is defined as:

```c
XrResult xrEnumerateSwapchainImages(
    XrSwapchain swapchain,
    uint32_t imageCapacityInput,
    uint32_t* imageCountOutput,
    XrSwapchainImageBaseHeader* images);
```
Parameter Descriptions

- `swapchain` is the `XrSwapchain` to get images from.

- `imageCapacityInput` is the capacity of the `images` array, or 0 to indicate a request to retrieve the required capacity.

- `imageCountOutput` is a pointer to the count of `images` written, or a pointer to the required capacity in the case that `imageCapacityInput` is insufficient.

- `images` is a pointer to an array of graphics API-specific `XrSwapchainImage` structures, all of the same type, based on `XrSwapchainImageBaseHeader`. It can be `NULL` if `imageCapacityInput` is 0.

- See Buffer Size Parameters chapter for a detailed description of retrieving the required `images` size.

Fills an array of graphics API-specific `XrSwapchainImage` structures. The resources must be constant and valid for the lifetime of the `XrSwapchain`.

Runtimes must always return identical buffer contents from this enumeration for the lifetime of the swapchain.

Note: `images` is a pointer to an array of structures of graphics API-specific type, not an array of structure pointers.

The pointer submitted as `images` will be treated as an array of the expected graphics API-specific type based on the graphics API used at session creation time. If the `type` member of any array element accessed in this way does not match the expected value, the runtime must return `XR_ERROR_VALIDATION_FAILURE`.

Note

Under a typical memory model, a runtime must treat the supplied pointer as an opaque blob beginning with `XrSwapchainImageBaseHeader`, until after it has verified the `type`.

Valid Usage (Implicit)

- `swapchain` must be a valid `XrSwapchain` handle

- `imageCountOutput` must be a pointer to a `uint32_t` value

- If `imageCapacityInput` is not 0, `images` must be a pointer to an array of `imageCapacityInput` `XrSwapchainImageBaseHeader`-based structures. See also: `XrSwapchainImageD3D11KHR`, `XrSwapchainImageD3D12KHR`, `XrSwapchainImageOpenGLKLR`, `XrSwapchainImageOpenGLESKHR`, `XrSwapchainImageVulkanKHR`
Return Codes

Success

• XR_SUCCESS
• XR_SESSION_LOSS_PENDING

Failure

• XR_ERROR_VALIDATION_FAILURE
• XR_ERROR_RUNTIME_FAILURE
• XR_ERROR_HANDLE_INVALID
• XR_ERROR_INSTANCE_LOST
• XR_ERROR_SESSION_LOST
• XR_ERROR_SIZE_INSUFFICIENT

The XrSwapchainImageBaseHeader structure is defined as:

typedef struct XrSwapchainImageBaseHeader {
    XrStructureType type;
    void* next;
} XrSwapchainImageBaseHeader;

Member Descriptions

• type is the XrStructureType of this structure. This base structure itself has no associated XrStructureType value.
• next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

The XrSwapchainImageBaseHeader is a base structure that can be overridden by a graphics API-specific XrSwapchainImage* child structure.
Valid Usage (Implicit)

- **type** must be one of the following XrStructureType values:
  - XR_TYPE_SWAPCHAIN_IMAGE_D3D11_KHR
  - XR_TYPE_SWAPCHAIN_IMAGE_D3D12_KHR
  - XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_ES_KHR
  - XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_KHR
  - XR_TYPE_SWAPCHAIN_IMAGE_VULKAN_KHR
- **next** must be NULL or a valid pointer to the next structure in a structure chain

Before an application can start building graphics API command buffers that refer to an image in a swapchain, it must acquire the image from the swapchain. The acquire operation determines the index of the next image that will be used in the swapchain. The order in which images are acquired is undefined. The runtime must allow the application to acquire more than one image from a single swapchain at a time, for example if the application implements a multiple frame deep rendering pipeline.

The **xrAcquireSwapchainImage** function is defined as:

```c
XrResult xrAcquireSwapchainImage(
    XrSwapchain swapchain,  // swapchain is the swapchain from which to acquire an image.
    const XrSwapchainImageAcquireInfo* acquireInfo,  // acquireInfo exists for extensibility purposes, it is NULL or a pointer to a valid XrSwapchainImageAcquireInfo.
    uint32_t* index)  // index is the returned image index that has been acquired.
```

Acquires the image corresponding to the **index** position in the array returned by **xrEnumerateSwapchainImages**. The runtime must return **XR_ERROR_CALL_ORDER_INVALID** if the next available index has already been acquired and not yet released with **xrReleaseSwapchainImage**. If the **swapchain** was created with the **XR_SWAPCHAIN_CREATE_STATIC_IMAGE_BIT** set in **XrSwapchainCreateInfo::createFlags**, this function must not have been previously called for this swapchain. The runtime must return **XR_ERROR_CALL_ORDER_INVALID** if a **swapchain** created with the **XR_SWAPCHAIN_CREATE_STATIC_IMAGE_BIT** set in **XrSwapchainCreateInfo::createFlags** and this function has been successfully called previously for this swapchain.
Valid Usage (Implicit)

- **swapchain** must be a valid `XrSwapchain` handle
- If `acquireInfo` is not `NULL`, `acquireInfo` must be a pointer to a valid `XrSwapchainImageAcquireInfo` structure
- `index` must be a pointer to a `uint32_t` value

Return Codes

**Success**
- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

**Failure**
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_CALL_ORDER_INVALID`

The `XrSwapchainImageAcquireInfo` structure is defined as:

```c
typedef struct XrSwapchainImageAcquireInfo {
    XrStructureType    type;
    const void*        next;
} XrSwapchainImageAcquireInfo;
```

Member Descriptions

- `type` is the `XrStructureType` of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

Because this structure only exists to support extension-specific structures, `xrAcquireSwapchainImage`
will accept a **NULL** argument for acquireInfo for applications that are not using any relevant extensions.

### Valid Usage (Implicit)

- **type must** be `XR_TYPE_SWAPCHAIN_IMAGE_ACQUIRE_INFO`
- **next must** be **NULL** or a valid pointer to the next structure in a structure chain

The `xrWaitSwapchainImage` function is defined as:

```c
XrResult xrWaitSwapchainImage(
    XrSwapchain swapchain,
    const XrSwapchainImageWaitInfo* waitInfo);
```

### Parameter Descriptions

- **swapchain** is the swapchain from which to wait for an image.
- **waitInfo** is a pointer to an `XrSwapchainImageWaitInfo` structure.

Before an application can begin writing to a swapchain image, it must first wait on the image to avoid writing to it before the compositor has finished reading from it. `xrWaitSwapchainImage` will implicitly wait on the oldest acquired swapchain image which has not yet been successfully waited on. Once a swapchain image has been successfully waited on without timeout, the app **must** release before waiting on the next acquired swapchain image.

This function may block for longer than the timeout specified in `XrSwapchainImageWaitInfo` due to scheduling or contention.

If the timeout expires without the image becoming available for writing, `XR_TIMEOUT_EXPIRED` **must** be returned. If `xrWaitSwapchainImage` returns `XR_TIMEOUT_EXPIRED`, the next call to `xrWaitSwapchainImage` will wait on the same image index again until the function succeeds with `XR_SUCCESS`. Note that this is not an error code; `XR_SUCCEEDED(XR_TIMEOUT_EXPIRED)` is **true**.

The runtime **must** eventually relinquish ownership of a swapchain image to the application and **must** not block indefinitely.

The runtime **must** return `XR_ERROR_CALL_ORDER_INVALID` if no image has been acquired by calling `xrAcquireSwapchainImage`. 

---

Chapter 10. Rendering | 149
Valid Usage (Implicit)

- `swapchain` must be a valid `XrSwapchain` handle
- `waitInfo` must be a pointer to a valid `XrSwapchainImageWaitInfo` structure

Return Codes

**Success**
- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`
- `XR_TIMEOUT_EXPIRED`

**Failure**
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_CALL_ORDER_INVALID`

The `XrSwapchainImageWaitInfo` structure describes a swapchain image wait operation. It is defined as:

```c
typedef struct XrSwapchainImageWaitInfo {
    XrStructureType    type;
    const void*        next;
    XrDuration         timeout;
} XrSwapchainImageWaitInfo;
```
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **timeout** indicates how many nanoseconds the call should block waiting for the image to become available for writing.

Valid Usage (Implicit)

- **type** must be XR_TYPE_SWAPCHAIN_IMAGE_WAIT_INFO
- **next** must be NULL or a valid pointer to the next structure in a structure chain

Once an application is done submitting commands that reference the swapchain image, the application **must** release the swapchain image. xrReleaseSwapchainImage will implicitly release the oldest swapchain image which has been acquired. The swapchain image **must** have been successfully waited on without timeout before it is released. xrEndFrame will use the most recently released swapchain image. In each frame submitted to the compositor only one image index from each swapchain will be used. Note that in case the swapchain contains 2D image arrays, one array is referenced per swapchain index and thus the whole image array can be used in one frame.

The **xrReleaseSwapchainImage** function is defined as:

```c
XrResult xrReleaseSwapchainImage(
    XrSwapchain swapchain,
    const XrSwapchainImageReleaseInfo* releaseInfo);
```

Parameter Descriptions

- **swapchain** is the XrSwapchain from which to release an image.
- **releaseInfo** exists for extensibility purposes, it is NULL or a pointer to a valid XrSwapchainImageReleaseInfo.

If the **swapchain** was created with the **XR_SWAPCHAIN_CREATE_STATIC_IMAGE_BIT** set in XrSwapchainCreateInfo::createFlags structure, this function **must** not have been previously called for this swapchain.

The runtime **must** return XR_ERROR_CALL_ORDER_INVALID if no image has been waited on by calling
xrWaitSwapchainImage.

Valid Usage (Implicit)

- `swapchain` must be a valid `XrSwapchain` handle
- If `releaseInfo` is not `NULL`, `releaseInfo` must be a pointer to a valid `XrSwapchainImageReleaseInfo` structure

Return Codes

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_CALL_ORDER_INVALID`

The `XrSwapchainImageReleaseInfo` structure is defined as:

```c
typedef struct XrSwapchainImageReleaseInfo {
    XrStructureType   type;
    const void*       next;
} XrSwapchainImageReleaseInfo;
```

Member Descriptions

- `type` is the `XrStructureType` of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

Because this structure only exists to support extension-specific structures, `xrReleaseSwapchainImage`
will accept a NULL argument for releaseInfo for applications that are not using any relevant extensions.

<table>
<thead>
<tr>
<th>Valid Usage (Implicit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• type <strong>must</strong> be XR_TYPE_SWAPCHAIN_IMAGE_RELEASE_INFO</td>
</tr>
<tr>
<td>• next <strong>must</strong> be NULL or a valid pointer to the next structure in a structure chain</td>
</tr>
</tbody>
</table>

# 10.2. View and Projection State

An application usesxrLocateViewsto retrieve the viewer pose and projection parameters needed to render each view for use in a composition projection layer.

The `xrLocateViews` function is defined as:

```c
XrResult xrLocateViews(
    XrSession session,  
    const XrViewLocateInfo* viewLocateInfo,  
    XrViewState* viewState,  
    uint32_t viewCapacityInput,  
    uint32_t* viewCountOutput,  
    XrView* views);
```

<table>
<thead>
<tr>
<th>Parameter Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <strong>session</strong> is a handle to the provided XrSession.</td>
</tr>
<tr>
<td>• <strong>viewLocateInfo</strong> is a pointer to a valid XrViewLocateInfo structure.</td>
</tr>
<tr>
<td>• <strong>viewState</strong> is the output structure with the viewer state information.</td>
</tr>
<tr>
<td>• <strong>viewCapacityInput</strong> is an input parameter which specifies the capacity of the <strong>views</strong> array. The required capacity <strong>must</strong> be same as defined by the corresponding XrViewConfigurationType.</td>
</tr>
<tr>
<td>• <strong>viewCountOutput</strong> is an output parameter which identifies the valid count of <strong>views</strong>.</td>
</tr>
<tr>
<td>• <strong>views</strong> is an array of XrView.</td>
</tr>
<tr>
<td>• See Buffer Size Parameters chapter for a detailed description of retrieving the required views size.</td>
</tr>
</tbody>
</table>

The `xrLocateViews` function returns the view and projection info for a particular display time. This time is typically the target display time for a given frame. Repeatedly calling `xrLocateViews` with the same time may not necessarily return the same result. Instead the prediction gets increasingly...
accurate as the function is called closer to the given time for which a prediction is made. This allows an application to get the predicted views as late as possible in its pipeline to get the least amount of latency and prediction error.

\texttt{xrLocateViews} returns an array of \texttt{XrView} elements, one for each view of the specified view configuration type, along with an \texttt{XrViewState} containing additional state data shared across all views. The eye each view corresponds to is statically defined in \texttt{XrViewConfigurationType} in case the application wants to apply eye-specific rendering traits. The \texttt{XrViewState} and \texttt{XrView} member data may change on subsequent calls to \texttt{xrLocateViews}, and so applications must not assume it to be constant.

### Valid Usage (Implicit)

- \texttt{session} must be a valid \texttt{XrSession} handle
- \texttt{viewLocateInfo} must be a pointer to a valid \texttt{XrViewLocateInfo} structure
- \texttt{viewState} must be a pointer to an \texttt{XrViewState} structure
- \texttt{viewCountOutput} must be a pointer to a \texttt{uint32_t} value
- If \texttt{viewCapacityInput} is not 0, \texttt{views} must be a pointer to an array of \texttt{viewCapacityInput} \texttt{XrView} structures

### Return Codes

**Success**

- \texttt{XR_SUCCESS}
- \texttt{XR_SESSION_LOSS_PENDING}

**Failure**

- \texttt{XR_ERROR_VALIDATION_FAILURE}
- \texttt{XR_ERROR_RUNTIME_FAILURE}
- \texttt{XR_ERROR_HANDLE_INVALID}
- \texttt{XR_ERROR_INSTANCE_LOST}
- \texttt{XR_ERROR_SESSION_LOST}
- \texttt{XR_ERROR_SIZE_INSUFFICIENT}
- \texttt{XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED}
- \texttt{XR_ERROR_TIME_INVALID}

The \texttt{XrViewLocateInfo} structure is defined as:

---

154 | Chapter 10. Rendering
typedef struct XrViewLocateInfo {
    XrStructureType            type;
    const void*                next;
    XrViewConfigurationType    viewConfigurationType;
    XrTime                     displayTime;
    XrSpace                    space;
} XrViewLocateInfo;

**Member Descriptions**

- `viewConfigurationType` is `XrViewConfigurationType` to query for.
- `displayTime` is the time for which the view poses are predicted.
- `space` is the `XrSpace` in which the `pose` in each `XrView` is expressed.

The `XrViewLocateInfo` structure contains the display time and space used to locate the view `XrView` structures.

The runtime **must** return error `XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED` if the given `viewConfigurationType` is not one of the supported type reported by `xrEnumerateViewConfigurations`.

**Valid Usage (Implicit)**

- `type` **must** be `XR_TYPE_VIEW_LOCATE_INFO`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `viewConfigurationType` **must** be a valid `XrViewConfigurationType` value
- `space` **must** be a valid `XrSpace` handle

The `XrView` structure is defined as:

typedef struct XrView {
    XrStructureType    type;
    void*              next;
    XrPosef            pose;
    XrFovf             fov;
} XrView;
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **pose** is an XrPosef defining the location and orientation of the view in the space specified by the xrLocateViews function.
- **fov** is the XrFovf for the four sides of the projection.

The XrView structure contains view pose and projection state necessary to render a single projection view in the view configuration.

Valid Usage (Implicit)

- **type** must be XR_TYPE_VIEW
- **next** must be NULL or a valid pointer to the next structure in a structure chain

The XrViewState structure is defined as:

```c
typedef struct XrViewState {
    XrStructureType     type;
    void*               next;
    XrViewStateFlags    viewStateFlags;
} XrViewState;
```

Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **viewStateFlags** is a bitmask of XrViewStateFlagBits indicating state for all views.

The XrViewState contains additional view state from xrLocateViews common to all views of the active view configuration.
Valid Usage (Implicit)

- **type** must be `XR_TYPE_VIEW_STATE`
- **next** must be `NULL` or a valid pointer to the next structure in a structure chain
- **viewStateFlags** must be `0` or a valid combination of `XrViewStateFlagBits` values

The `XrViewStateFlags` specifies the validity and quality of the corresponding `XrView` array returned by `xrLocateViews`.

Flags include:

```cpp
// Flag bits for XrViewStateFlags
static const XrViewStateFlags XR_VIEW_STATE_ORIENTATION_VALID_BIT = 0x00000001;
static const XrViewStateFlags XR_VIEW_STATE_POSITION_VALID_BIT = 0x00000002;
static const XrViewStateFlags XR_VIEW_STATE_ORIENTATION_TRACKED_BIT = 0x00000004;
static const XrViewStateFlags XR_VIEW_STATE_POSITION_TRACKED_BIT = 0x00000008;
```

**Flag Descriptions**

- **XR_VIEW_STATE_ORIENTATION_VALID_BIT** indicates whether all `XrView` orientations contain valid data. Applications **must not** read any of the `XrView::pose orientation` fields if this flag is unset. `XR_VIEW_STATE_ORIENTATION_TRACKED_BIT` **should** generally remain set when this bit is set for views on a tracked headset or handheld device.

- **XR_VIEW_STATE_POSITION_VALID_BIT** indicates whether all `XrView` positions contain valid data. Applications **must not** read any of the `XrView::pose position` fields if this flag is unset. When a view loses tracking, runtimes **should** continue to provide valid but untracked view `position` values that are inferred or last-known, so long as it's still meaningful for the application to render content using that position, clearing `XR_VIEW_STATE_POSITION_TRACKED_BIT` until tracking is recovered.

- **XR_VIEW_STATE_ORIENTATION_TRACKED_BIT** indicates whether all `XrView` orientations represent an actively tracked orientation. This bit **should** generally remain set when `XR_VIEW_STATE_ORIENTATION_VALID_BIT` is set for views on a tracked headset or handheld device.

- **XR_VIEW_STATE_POSITION_TRACKED_BIT** indicates whether all `XrView` positions represent an actively tracked position. When a view loses tracking, runtimes **should** continue to provide valid but untracked view `position` values that are inferred or last-known, e.g. based on neck model updates, inertial dead reckoning, or a last-known position, so long as it's still meaningful for the application to render content using that position.
10.3. Frame Synchronization

An application synchronizes its rendering loop to the runtime by calling `xrWaitFrame`.

The `xrWaitFrame` function is defined as:

```c
XrResult xrWaitFrame(
    XrSession session,
    const XrFrameWaitInfo* frameWaitInfo,
    XrFrameState* frameState);
```

**Parameter Descriptions**

- `session` is a valid `XrSession` handle.
- `frameWaitInfo` exists for extensibility purposes, it is `NULL` or a pointer to a valid `XrFrameWaitInfo`.
- `frameState` is a pointer to a valid `XrFrameState`, an output parameter.

`xrWaitFrame` throttles the application frame loop in order to synchronize application frame submissions with the display. `xrWaitFrame` returns a predicted display time for the next time that the runtime predicts a composited frame will be displayed. The runtime may affect this computation by changing the return values and throttling of `xrWaitFrame` in response to feedback from frame submission and completion times in `xrEndFrame`. An application must eventually match each `xrWaitFrame` call with one call to `xrBeginFrame`. A subsequent `xrWaitFrame` call must block until the previous frame has been begun with `xrBeginFrame` and must unblock independently of the corresponding call to `xrEndFrame`. When less than one frame interval has passed since the previous return from `xrWaitFrame`, the runtime should block until the beginning of the next frame interval. If more than one frame interval has passed since the last return from `xrWaitFrame`, the runtime may return immediately or block until the beginning of the next frame interval.

In the case that an application has pipelined frame submissions, the application should compute the appropriate target display time using both the predicted display time and predicted display interval. The application should use the computed target display time when requesting space and view locations for rendering.

The `XrFrameState::predictedDisplayTime` returned by `xrWaitFrame` must be monotonically increasing.

The runtime may dynamically adjust the start time of the frame interval relative to the display hardware's refresh cycle to minimize graphics processor contention between the application and the compositor.
xrWaitFrame must be callable from any thread, including a different thread than xrBeginFrame/xrEndFrame are being called from.

Calling xrWaitFrame must be externally synchronized by the application, concurrent calls may result in undefined behavior.

The runtime must return XR_ERROR_SESSION_NOT_RUNNING if the session is not running.

Note

The engine simulation should advance based on the display time. Every stage in the engine pipeline should use the exact same display time for one particular application-generated frame. An accurate and consistent display time across all stages and threads in the engine pipeline is important to avoid object motion judder. If the application has multiple pipeline stages, the application should pass its computed display time through its pipeline, as xrWaitFrame must be called only once per frame.

Valid Usage (Implicit)

- session must be a valid XrSession handle
- If frameWaitInfo is not NULL, frameWaitInfo must be a pointer to a valid XrFrameWaitInfo structure
- frameState must be a pointer to an XrFrameState structure

Thread Safety

- Access to the session parameter by any other xrWaitFrame call must be externally synchronized
Return Codes

Success
- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_SESSION_NOT_RUNNING

The XrFrameWaitInfo structure is defined as:

typedef struct XrFrameWaitInfo {
    XrStructureType type;
    const void*    next;
} XrFrameWaitInfo;

Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

Because this structure only exists to support extension-specific structures, xrWaitFrame must accept a NULL argument for frameWaitInfo for applications that are not using any relevant extensions.

Valid Usage (Implicit)

- **type** must be XR_TYPE_FRAME_WAIT_INFO
- **next** must be NULL or a valid pointer to the next structure in a structure chain

The XrFrameState structure is defined as:
typedef struct XrFrameState {
    XrStructureType    type;
    void*              next;
    XrTime             predictedDisplayTime;
    XrDuration         predictedDisplayPeriod;
    XrBool32           shouldRender;
} XrFrameState;

Member Descriptions

- **type** is the `XrStructureType` of this structure.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **predictedDisplayTime** is the anticipated display `XrTime` for the next application-generated frame.
- **predictedDisplayPeriod** is the `XrDuration` of the display period for the next application-generated frame, for use in predicting display times beyond the next one.
- **shouldRender** is `XR_TRUE` if the application **should** render its layers as normal and submit them to `xrEndFrame`. When this value is `XR_FALSE`, the application **should** avoid heavy GPU work where possible, for example by skipping layer rendering and then omitting those layers when calling `xrEndFrame`.

`XrFrameState` describes the time at which the next frame will be displayed to the user. **predictedDisplayTime** must refer to the midpoint of the interval during which the frame is displayed. The runtime **may** report a different **predictedDisplayPeriod** from the hardware's refresh cycle.

For any frame where **shouldRender** is `XR_FALSE`, the application **should** avoid heavy GPU work for that frame, for example by not rendering its layers. This typically happens when the application is transitioning into or out of a running session, or when some system UI is fully covering the application at the moment. As long as the session is **running**, the application **should** keep running the frame loop to maintain the frame synchronization to the runtime, even if this requires calling `xrEndFrame` with all layers omitted.

Valid Usage (Implicit)

- **type** must be `XR_TYPE_FRAME_STATE`
- **next** must be `NULL` or a valid pointer to the next structure in a structure chain
10.4. Frame Submission

Every application must call `xrBeginFrame` before calling `xrEndFrame`, and should call `xrEndFrame` before calling `xrBeginFrame` again. Calling `xrEndFrame` again without a prior call to `xrBeginFrame` must result in `XR_ERROR_CALL_ORDER_INVALID` being returned by `xrEndFrame`. An application may call `xrBeginFrame` again if the prior `xrEndFrame` fails or if the application wishes to discard an in-progress frame. A successful call to `xrBeginFrame` again with no intervening `xrEndFrame` call must result in the success code `XR_FRAME_DISCARDED` being returned from `xrBeginFrame`. In this case it is assumed that the `xrBeginFrame` refers to the next frame and the previously begun frame is forfeited by the application. An application may call `xrEndFrame` without having called `xrReleaseSwapchainImage` since the previous call to `xrEndFrame` for any swapchain passed to `xrEndFrame`. Applications should call `xrBeginFrame` right before executing any graphics device work for a given frame, as opposed to calling it afterwards. The runtime must only compose frames whose `xrBeginFrame` and `xrEndFrame` both return success codes. While `xrBeginFrame` and `xrEndFrame` do not need to be called on the same thread, the application must handle synchronization if they are called on separate threads.

The `xrBeginFrame` function is defined as:

```c
XrResult xrBeginFrame(
    XrSession session,
    const XrFrameBeginInfo* frameBeginInfo);
```

### Parameter Descriptions

- **session** is a valid `XrSession` handle.
- **frameBeginInfo** exists for extensibility purposes, it is NULL or a pointer to a valid `XrFrameBeginInfo`.

`xrBeginFrame` is called prior to the start of frame rendering. The application should still call `xrBeginFrame` but omit rendering work for the frame if `XrFrameState::shouldRender` is `XR_FALSE`.

Runtimes must not perform frame synchronization or throttling through the `xrBeginFrame` function and should instead do so through `xrWaitFrame`.

The runtime must return the error code `XR_ERROR_CALL_ORDER_INVALID` if there was no corresponding successful call to `xrWaitFrame`.

The runtime must return the success code `XR_FRAME_DISCARDED` if a prior `xrBeginFrame` has been called without an intervening call to `xrEndFrame`.

The runtime must return `XR_ERROR_SESSION_NOT_RUNNING` if the `session` is not running.
Valid Usage (Implicit)

- `session` must be a valid `XrSession` handle
- If `frameBeginInfo` is not `NULL`, `frameBeginInfo` must be a pointer to a valid `XrFrameBeginInfo` structure

Thread Safety

- Access to the `session` parameter by any other `xrBeginFrame` or `xrEndFrame` call must be externally synchronized

Return Codes

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`
- `XR_FRAME_DISCARDED`

Failure

- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_SESSION_NOT_RUNNING`
- `XR_ERROR_CALL_ORDER_INVALID`

The `XrFrameBeginInfo` structure is defined as:

typedef struct XrFrameBeginInfo {
    XrStructureType    type;
    const void*        next;
} XrFrameBeginInfo;
Member Descriptions

- **type** is the `XrStructureType` of this structure.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

Because this structure only exists to support extension-specific structures, `xBeginFrame` will accept a `NULL` argument for `frameBeginInfo` for applications that are not using any relevant extensions.

Valid Usage (Implicit)

- **type** must be `XR_TYPE_FRAME_BEGIN_INFO`
- **next** must be `NULL` or a valid pointer to the next structure in a structure chain

The `xEndFrame` function is defined as:

```c
XrResult xEndFrame(
    XrSession session,
    const XrFrameEndInfo* frameEndInfo);
```

Parameter Descriptions

- **session** is a valid `XrSession` handle.
- **frameEndInfo** is a pointer to a valid `XrFrameEndInfo`.

`xEndFrame` may return immediately to the application. `XrFrameEndInfo::displayTime` should be computed using values returned by `xWaitFrame`. The runtime should be robust against variations in the timing of calls to `xWaitFrame`, since a pipelined system may call `xWaitFrame` on a separate thread from `xBeginFrame` and `xEndFrame` without any synchronization guarantees.

**Note**

An accurate predicted display time is very important to avoid black pull-in by reprojection and to reduce motion judder in case the runtime does not implement a translational reprojection. Reprojection should never display images before the display refresh period they were predicted for, even if they are completed early, because this will cause motion judder just the same. In other words, the better the predicted display time, the less latency experienced by the user.
Every call to `xrEndFrame` must be preceded by a successful call to `xrBeginFrame`. Failure to do so must result in `XR_ERROR_CALL_ORDER_INVALID` being returned by `xrEndFrame`. `XrFrameEndInfo` may reference swapchains into which the application has rendered for this frame. From each `XrSwapchain` only one image index is implicitly referenced per frame, the one corresponding to the last call to `xrReleaseSwapchainImage`. However, a specific swapchain (and by extension a specific swapchain image index) may be referenced in `XrFrameEndInfo` multiple times. This can be used for example to render a side by side image into a single swapchain image and referencing it twice with differing image rectangles in different layers.

If no layers are provided then the display must be cleared.

`XR_ERROR_LAYER_INVALID` must be returned if an unknown, unsupported layer type, or `NULL` pointer is passed as one of the `XrFrameEndInfo::layers`.

`XR_ERROR_LAYER_INVALID` must be returned if a layer references a swapchain that has no released swapchain image.

`XR_ERROR_LAYER_LIMIT_EXCEEDED` must be returned if `XrFrameEndInfo::layerCount` exceeds `XrSystemGraphicsProperties::maxLayerCount` or if the runtime is unable to composite the specified layers due to resource constraints.

`XR_ERROR_SWAPCHAIN_RECT_INVALID` must be returned if `XrFrameEndInfo::layers` contains a composition layer which references pixels outside of the associated swapchain image or if negatively sized.

`XR_ERROR_ENVIRONMENT_BLEND_MODE_UNSUPPORTED` must be returned if `XrFrameEndInfo::environmentBlendMode` is not supported.

`XR_ERROR_SESSION_NOT_RUNNING` must be returned if the session is not running.

**Note**

Applications should discard frames for which `xrEndFrame` returns a recoverable error over attempting to resubmit the frame with different frame parameters to provide a more consistent experience across different runtime implementations.

**Valid Usage (Implicit)**

- `session` must be a valid `XrSession` handle
- `frameEndInfo` must be a pointer to a valid `XrFrameEndInfo` structure

**Thread Safety**

- Access to the `session` parameter by any other `xrBeginFrame` or `xrEndFrame` call must be externally synchronized
Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_TIME_INVALID
- XR_ERROR_SWAPCHAIN_RECT_INVALID
- XR_ERROR_SESSION_NOT_RUNNING
- XR_ERROR_POSE_INVALID
- XR_ERROR_LAYER_LIMIT_EXCEEDED
- XR_ERROR_LAYER_INVALID
- XR_ERROR_ENVIRONMENT_BLEND_MODE_UNSUPPORTED
- XR_ERROR_CALL_ORDER_INVALID

The **XrFrameEndInfo** structure is defined as:

```c
typedef struct XrFrameEndInfo {
    XrStructureType                               type;
    const void*                                   next;
    XrTime                                        displayTime;
    XrEnvironmentBlendMode                        environmentBlendMode;
    uint32_t                                      layerCount;
    const XrCompositionLayerBaseHeader* const*    layers;
} XrFrameEndInfo;
```
Member Descriptions

- **type** is the `XrStructureType` of this structure.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **displayTime** is the `XrTime` at which this frame should be displayed.
- **environmentBlendMode** is the `XrEnvironmentBlendMode` value representing the desired environment blend mode for this frame.
- **layerCount** is the number of composition layers in this frame. The maximum supported layer count is identified by `XrSystemGraphicsProperties::maxLayerCount`. If layerCount is greater than the maximum supported layer count then `XR_ERROR_LAYER_LIMIT_EXCEEDED` must be returned.
- **layers** is a pointer to an array of `XrCompositionLayerBaseHeader` pointers.

Valid Usage (Implicit)

- **type** must be `XR_TYPE_FRAME_END_INFO`
- **next** must be `NULL` or a valid pointer to the next structure in a structure chain
- **environmentBlendMode** must be a valid `XrEnvironmentBlendMode` value
- If **layerCount** is not 0, **layers** must be a pointer to an array of layerCount valid `XrCompositionLayerBaseHeader`-based structures. See also: `XrCompositionLayerCubeKHR`, `XrCompositionLayerCylinderKHR`, `XrCompositionLayerEquirect2KHR`, `XrCompositionLayerEquirectKHR`, `XrCompositionLayerProjection`, `XrCompositionLayerQuad` All layers submitted to `xrEndFrame` will be presented to the primary view configuration of the running session.

### 10.4.1. Frame Rate

For every application-generated frame, the application may call `xrEndFrame` to submit the application-generated composition layers. In addition, the application must call `xrWaitFrame` when the application is ready to begin preparing the next set of frame layers. `xrEndFrame` may return immediately to the application, but `xrWaitFrame` must block for an amount of time that depends on throttling of the application by the runtime. The earliest the runtime will return from `xrWaitFrame` is when it determines that the application should start drawing the next frame.

### 10.4.2. Compositing

Composition layers are submitted by the application via the `xrEndFrame` call. All composition layers to
be drawn must be submitted with every xrEndFrame call. A layer that is omitted in this call will not be
drawn by the runtime layer compositor. All views associated with projection layers must be supplied,
or XR_ERROR_VALIDATION_FAILURE must be returned by xrEndFrame.

Composition layers must be drawn in the same order as they are specified in via XrFrameEndInfo,
with the 0th layer drawn first. Layers must be drawn with a "painter's algorithm," with each
successive layer potentially overwriting the destination layers whether or not the new layers are
virtually closer to the viewer.

10.4.3. Composition Layer Flags

The XrCompositionLayerFlagBits bitfield is specified as:

```c
// Flag bits for XrCompositionLayerFlags
static const XrCompositionLayerFlags XR_COMPOSITION_LAYER_CORRECT_CHROMATIC_ABERRATION_BIT = 0x00000001;
static const XrCompositionLayerFlags XR_COMPOSITION_LAYER_BLEND_TEXTURE_SOURCE_ALPHA_BIT
= 0x00000002;
static const XrCompositionLayerFlags XR_COMPOSITION_LAYER_UNPREMULTIPLIED_ALPHA_BIT =
0x00000004;
```

XrCompositionLayerFlags specify options for individual composition layers.

**Flag Descriptions**

- **XR_COMPOSITION_LAYER_CORRECT_CHROMATIC_ABERRATION_BIT** — Enables chromatic aberration
correction when not done by default.
- **XR_COMPOSITION_LAYER_BLEND_TEXTURE_SOURCE_ALPHA_BIT** — Enables the layer texture alpha
  channel.
- **XR_COMPOSITION_LAYER_UNPREMULTIPLIED_ALPHA_BIT** — Indicates the texture color channels have
  not been premultiplied by the texture alpha channel.

10.4.4. Composition Layer Blending

All types of composition layers are subject to blending with other layers. Blending of layers can be
controlled by layer per-texel source alpha. Layer swapchain textures may contain an alpha channel,
depending on the image format. If a submitted swapchain’s texture format does not include an alpha
channel or if the **XR_COMPOSITION_LAYER_BLEND_TEXTURE_SOURCE_ALPHA_BIT** is unset, then the layer alpha is
initialized to one.

If the swapchain texture format color encoding is other than RGBA, it is converted to RGBA.
If the texture color channels are encoded without premultiplying by alpha, the `XR_COMPOSITION_LAYER_UNPREMULTIPLIED_ALPHA_BIT` should be set. The effect of this bit alters the layer color as follows:

\[
\text{LayerColor.RGB} *\text{LayerColor.A}
\]

LayerColor is then clamped to a range of [0.0, 1.0].

The layer blending operation is defined as:

\[
\text{CompositeColor} = \text{LayerColor} + \text{CompositeColor} * (1 - \text{LayerColor.A})
\]

Before the first layer is composited, all components of CompositeColor are initialized to zero.

### 10.4.5. Composition Layer Types

Composition layers allow an application to offload the composition of the final image to a runtime-supplied compositor. This reduces the application’s rendering complexity since details such as frame-rate interpolation and distortion correction can be performed by the runtime. The core specification defines `XrCompositionLayerProjection` and `XrCompositionLayerQuad` layer types.

The projection layer type represents planar projected images rendered from the eye point of each eye using a perspective projection. This layer type is typically used to render the virtual world from the user’s perspective.

The quad layer type describes a posable planar rectangle in the virtual world for displaying two-dimensional content. Quad layers can subtend a smaller portion of the display’s field of view, allowing a better match between the resolutions of the `XrSwapchain` image and footprint of that image in the final composition. This improves legibility for user interface elements or heads-up displays and allows optimal sampling during any composition distortion corrections the runtime might employ.

The classes below describe the layer types in the layer composition system.

The `XrCompositionLayerBaseHeader` structure is defined as:

```c
typedef struct XrCompositionLayerBaseHeader {
    XrStructureType type;
    const void* next;
    XrCompositionLayerFlags layerFlags;
    XrSpace space;
} XrCompositionLayerBaseHeader;
```
Member Descriptions

- **type** is the `XrStructureType` of this structure. This base structure itself has no associated `XrStructureType` value.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **layerFlags** is a bitmask of `XrCompositionLayerFlagBits` describing flags to apply to the layer.
- **space** is the `XrSpace` in which the layer will be kept stable over time.

All composition layer structures begin with the elements described in the `XrCompositionLayerBaseHeader`. The `XrCompositionLayerBaseHeader` structure is not intended to be directly used, but forms a basis for defining current and future structures containing composition layer information. The `XrFrameEndInfo` structure contains an array of pointers to these polymorphic header structures. All composition layer type pointers must be type-castable as an `XrCompositionLayerBaseHeader` pointer.

Valid Usage (Implicit)

- **type** must be one of the following `XrStructureType` values:
  - `XR_TYPE_COMPOSITION_LAYER_CUBE_KHR`,
  - `XR_TYPE_COMPOSITION_LAYER_CYLINDER_KHR`,
  - `XR_TYPE_COMPOSITION_LAYER_EQUIRECT2_KHR`,
  - `XR_TYPE_COMPOSITION_LAYER_EQUIRECT_KHR`,
  - `XR_TYPE_COMPOSITION_LAYER_PROJECTION`,
  - `XR_TYPE_COMPOSITION_LAYER_QUAD`
- **next** must be `NULL` or a valid pointer to the next structure in a structure chain. See also: `XrCompositionLayerColorScaleBiasKHR`
- **layerFlags** must be `0` or a valid combination of `XrCompositionLayerFlagBits` values
- **space** must be a valid `XrSpace` handle

Many composition layer structures also contain one or more references to generic layer data stored in an `XrSwapchainSubImage` structure.

The `XrSwapchainSubImage` structure is defined as:

```c
typedef struct XrSwapchainSubImage {
    XrSwapchain swapchain;
    XrRect2Di imageRect;
    uint32_t imageArrayIndex;
} XrSwapchainSubImage;
```
Member Descriptions

- **swapchain** is the XrSwapchain to be displayed.
- **imageRect** is an XrRect2Di representing the valid portion of the image to use, in pixels. It also implicitly defines the transform from normalized image coordinates into pixel coordinates. The coordinate origin depends on which graphics API is being used. See the graphics API extension details for more information on the coordinate origin definition. Note that the compositor may bleed in pixels from outside the bounds in some cases, for instance due to mipmapping.
- **imageArrayIndex** is the image array index, with 0 meaning the first or only array element.

Valid Usage (Implicit)

- **swapchain must** be a valid XrSwapchain handle

Runtimes must return XR_ERROR_VALIDATION_FAILURE if the XrSwapchainSubImage::imageArrayIndex is equal to or greater than the XrSwapchainCreateInfo::arraySize that the XrSwapchainSubImage::swapchain was created with.

Projection Composition

The XrCompositionLayerProjection layer represents planar projected images rendered from the eye point of each eye using a standard perspective projection.

The XrCompositionLayerProjection structure is defined as:

```c
typedef struct XrCompositionLayerProjection {
    XrStructureType                            type;
    const void*                                next;
    XrCompositionLayerFlags                    layerFlags;
    XrSpace                                    space;
    uint32_t                                   viewCount;
    const XrCompositionLayerProjectionView*    views;
} XrCompositionLayerProjection;
```
**Member Descriptions**

- **type** is the `XrStructureType` of this structure.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **layerFlags** is a bitmask of `XrCompositionLayerFlagBits` describing flags to apply to the layer.
- **space** is the `XrSpace` in which the pose of each `XrCompositionLayerProjectionView` is evaluated over time by the compositor.
- **viewCount** is the count of views in the `views` array. This **must** be equal to the number of view poses returned by `xrLocateViews`.
- **views** is the array of type `XrCompositionLayerProjectionView` containing each projection layer view.

**Note**

Because a runtime may reproject the layer over time, a projection layer should specify an `XrSpace` in which to maximize stability of the layer content. For example, a projection layer containing world-locked content should use an `XrSpace` which is also world-locked, such as the `LOCAL` or `STAGE` reference spaces. In the case that the projection layer should be head-locked, such as a heads up display, the `VIEW` reference space would provide the highest quality layer reprojection.

**Valid Usage (Implicit)**

- **type** must be `XR_TYPE_COMPOSITION_LAYER_PROJECTION`  
- **next** must be `NULL` or a valid pointer to the next structure in a structure chain  
- **layerFlags** must be `0` or a valid combination of `XrCompositionLayerFlagBits` values  
- **space** must be a valid `XrSpace` handle  
- **views** must be a pointer to an array of `viewCount` valid `XrCompositionLayerProjectionView` structures  
- The `viewCount` parameter must be greater than `0`  

The `XrCompositionLayerProjectionView` structure is defined as:
typedef struct XrCompositionLayerProjectionView {
    XrStructureType        type;
    const void*            next;
    XrPosef                pose;
    XrFovf                 fov;
    XrSwapchainSubImage    subImage;
} XrCompositionLayerProjectionView;

**Member Descriptions**

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **pose** is an XrPosef defining the location and orientation of this projection element in the space of the corresponding XrCompositionLayerProjectionView.
- **fov** is the XrFovf for this projection element.
- **subImage** is the image layer XrSwapchainSubImage to use.

The count and order of view poses submitted with XrCompositionLayerProjection must be the same order as that returned by xrLocateViews. The XrCompositionLayerProjectionView::pose and XrCompositionLayerProjectionView::fov should almost always derive from XrView::pose and XrView::fov as found in the xrLocateViews::views array. However, applications may submit an XrCompositionLayerProjectionView which has a different view or FOV than that from xrLocateViews. In this case, the runtime will map the view and FOV to the system display appropriately. In the case that two submitted views within a single layer overlap, they must be composited in view array order.

**Valid Usage (Implicit)**

- **type** must be XR_TYPE_COMPOSITION_LAYER_PROJECTION_VIEW
- **next** must be NULL or a valid pointer to the next structure in a structure chain. See also: XrCompositionLayerDepthInfoKHR
- **subImage** must be a valid XrSwapchainSubImage structure

**Quad Layer Composition**

The XrCompositionLayerQuad structure defined as:
typedef struct XrCompositionLayerQuad {
    XrStructureType type;
    const void* next;
    XrCompositionLayerFlags layerFlags;
    XrSpace space;
    XrEyeVisibility eyeVisibility;
    XrSwapchainSubImage subImage;
    XrPosef pose;
    XrExtent2Df size;
} XrCompositionLayerQuad;

**Member Descriptions**

- **type** is the `XrStructureType` of this structure.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **layerFlags** is a bitmask of `XrCompositionLayerFlagBits` describing flags to apply to the layer.
- **space** is the `XrSpace` in which the `pose` of the quad layer is evaluated over time.
- **eyeVisibility** is the `XrEyeVisibility` for this layer.
- **subImage** is the image layer `XrSwapchainSubImage` to use.
- **pose** is an `XrPosef` defining the position and orientation of the quad in the reference frame of the `space`.
- **size** is the width and height of the quad in meters.

The `XrCompositionLayerQuad` layer is useful for user interface elements or 2D content rendered into the virtual world. The layer’s `XrSwapchainSubImage::swapchain` image is applied to a quad in the virtual world space. Only front face of the quad surface is visible; the back face is not visible and **must** not be drawn by the runtime. A quad layer has no thickness; it is a two-dimensional object positioned and oriented in 3D space. The position of a quad refers to the center of the quad within the given `XrSpace`. The orientation of the quad refers to the orientation of the normal vector from the front face. The size of a quad refers to the quad’s size in the x-y plane of the given `XrSpace`’s coordinate system. A quad with a position of `{0,0,0}`, rotation of `{0,0,0,1}` (no rotation), and a size of `{1,1}` refers to a 1 meter x 1 meter quad centered at `{0,0,0}` with its front face normal vector coinciding with the +z axis.
Valid Usage (Implicit)

- **type** must be `XR_TYPE_COMPOSITION_LAYER_QUAD`
- **next** must be `NULL` or a valid pointer to the next structure in a structure chain
- **layerFlags** must be `0` or a valid combination of `XrCompositionLayerFlagBits` values
- **space** must be a valid `XrSpace` handle
- **eyeVisibility** must be a valid `XrEyeVisibility` value
- **subImage** must be a valid `XrSwapchainSubImage` structure

The `XrEyeVisibility` enum selects which of the viewer’s eyes to display a layer to:

```c
typedef enum XrEyeVisibility {
    XR_EYE_VISIBILITY_BOTH = 0,
    XR_EYE_VISIBILITY_LEFT = 1,
    XR_EYE_VISIBILITY_RIGHT = 2,
    XR_EYE_VISIBILITY_MAX_ENUM = 0x7FFFFFFF
} XrEyeVisibility;
```

Enumerant Descriptions

- **XR_EYE_VISIBILITY_BOTH** displays the layer to both eyes.
- **XR_EYE_VISIBILITY_LEFT** displays the layer to the viewer’s physical left eye.
- **XR_EYE_VISIBILITY_RIGHT** displays the layer to the viewer’s physical right eye.

10.4.6. Environment Blend Mode

After the compositor has blended and flattened all layers (including any layers added by the runtime itself), it will then present this image to the system's display. The composited image will then blend with the user’s view of the physical world behind the displays in one of three modes, based on the application’s chosen environment blend mode. VR applications will generally choose the `XR_ENVIRONMENT_BLEND_MODE_OPAQUE` blend mode, while AR applications will generally choose either the `XR_ENVIRONMENT_BLEND_MODE_ADDITIVE` or `XR_ENVIRONMENT_BLEND_MODE_ALPHA_BLEND` mode.

Applications select their environment blend mode each frame as part of their call to `xrEndFrame`. The application can inspect the set of supported environment blend modes for a given system using `xrEnumerateEnvironmentBlendModes`, and prepare their assets and rendering techniques differently based on the blend mode they choose. For example, a black shadow rendered using the
XR_ENVIRONMENT_BLEND_MODE_ADDITIVE blend mode will appear transparent, and so an application in that mode may render a glow as a grounding effect around the black shadow to ensure the shadow can be seen. Similarly, an application designed for XR_ENVIRONMENT_BLEND_MODE_OPAQUE or XR_ENVIRONMENT_BLEND_MODE_ADDITIVE rendering may choose to leave garbage in their alpha channel as a side effect of a rendering optimization, but this garbage would appear as visible display artifacts if the environment blend mode was instead XR_ENVIRONMENT_BLEND_MODE_ALPHA_BLEND.

Not all systems will support all environment blend modes. For example, a VR headset may not support the XR_ENVIRONMENT_BLEND_MODE_ADDITIVE or XR_ENVIRONMENT_BLEND_MODE_ALPHA_BLEND modes unless it has video passthrough, while an AR headset with an additive display may not support the XR_ENVIRONMENT_BLEND_MODE_OPAQUE or XR_ENVIRONMENT_BLEND_MODE_ALPHA_BLEND modes.

For devices that can support multiple environment blend modes, such as AR phones with video passthrough, the runtime may optimize power consumption on the device in response to the environment blend mode that the application chooses each frame. For example, if an application on a video passthrough phone knows that it is currently rendering a 360-degree background covering all screen pixels, it can submit frames with an environment blend mode of XR_ENVIRONMENT_BLEND_MODE_OPAQUE, saving the runtime the cost of compositing a camera-based underlay of the physical world behind the application’s layers.

The xrEnumerateEnvironmentBlendModes function is defined as:

```c
XrResult xrEnumerateEnvironmentBlendModes(
    XrInstance                                  instance,                      
    XrSystemId                                  systemId,                     
    XrViewConfigurationType                     viewConfigurationType,        
    uint32_t                                    environmentBlendModeCapacityInput,  
    uint32_t*                                   environmentBlendModeCountOutput, 
    XrEnvironmentBlendMode*                     environmentBlendModes);        
```
Parameter Descriptions

- **instance** is the instance from which **systemId** was retrieved.
- **systemId** is the XrSystemId whose environment blend modes will be enumerated.
- **viewConfigurationType** is the XrViewConfigurationType to enumerate.
- **environmentBlendModeCapacityInput** is the capacity of the **environmentBlendModes** array, or 0 to indicate a request to retrieve the required capacity.
- **environmentBlendModeCountOutput** is a pointer to the count of **environmentBlendModes** written, or a pointer to the required capacity in the case that **environmentBlendModeCapacityInput** is insufficient.
- **environmentBlendModes** is a pointer to an array of XrEnvironmentBlendMode values, but can be NULL if **environmentBlendModeCapacityInput** is 0.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required **environmentBlendModes** size.

Enumerates the set of environment blend modes that this runtime supports for a given view configuration of the system. Environment blend modes should be in order from highest to lowest runtime preference.

Runtimes must always return identical buffer contents from this enumeration for the given **systemId** and **viewConfigurationType** for the lifetime of the instance.

Valid Usage (Implicit)

- **instance** must be a valid XrInstance handle
- **viewConfigurationType** must be a valid XrViewConfigurationType value
- **environmentBlendModeCountOutput** must be a pointer to a uint32_t value
- If **environmentBlendModeCapacityInput** is not 0, **environmentBlendModes** must be a pointer to an array of **environmentBlendModeCapacityInput** XrEnvironmentBlendMode values
Return Codes

Success
- XR_SUCCESS

Failure
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SIZE_INSUFFICIENT
- XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED
- XR_ERROR_SYSTEM_INVALID

The possible blend modes are specified by the XrEnvironmentBlendMode enumeration:

typedef enum XrEnvironmentBlendMode {
    XR_ENVIRONMENT_BLEND_MODE_OPAQUE = 1,
    XR_ENVIRONMENT_BLEND_MODE_ADDITIVE = 2,
    XR_ENVIRONMENT_BLEND_MODE_ALPHA_BLEND = 3,
    XR_ENVIRONMENT_BLEND_MODE_MAX_ENUM = 0x7FFFFFFF
} XrEnvironmentBlendMode;
Enumerant Descriptions

- **XR_ENVIRONMENT_BLEND_MODE_OPAQUE.** The composition layers will be displayed with no view of the physical world behind them. The composited image will be interpreted as an RGB image, ignoring the composited alpha channel. This is the typical mode for VR experiences, although this mode can also be supported on devices that support video passthrough.

- **XR_ENVIRONMENT_BLEND_MODE_ADDITIVE.** The composition layers will be additively blended with the real world behind the display. The composited image will be interpreted as an RGB image, ignoring the composited alpha channel during the additive blending. This will cause black composited pixels to appear transparent. This is the typical mode for an AR experience on a see-through headset with an additive display, although this mode can also be supported on devices that support video passthrough.

- **XR_ENVIRONMENT_BLEND_MODE_ALPHA_BLEND.** The composition layers will be alpha-blended with the real world behind the display. The composited image will be interpreted as an RGBA image, with the composited alpha channel determining each pixel's level of blending with the real world behind the display. This is the typical mode for an AR experience on a phone or headset that supports video passthrough.
Chapter 11. Input and Haptics

11.1. Action Overview

OpenXR applications communicate with input devices using XrActions. Actions are created at initialization time and later used to request input device state, create action spaces, or control haptic events. Input action handles represent 'actions' that the application is interested in obtaining the state of, not direct input device hardware. For example, instead of the application directly querying the state of the A button when interacting with a menu, an OpenXR application instead creates a `menu_select` action at startup then asks OpenXR for the state of the action.

The application recommends that the action be assigned to a specific input source on the input device for a known interaction profile, but runtimes have the ability to choose a different control depending on user preference, input device availability, or any other reason. This abstraction ensures that applications can run on a wide variety of input hardware and maximize user accessibility.

Example usage:

```c
XrInstance instance; // previously initialized
XrSession session; // previously initialized

// Create an action set
XrActionSetCreateInfo actionSetInfo{XR_TYPE_ACTION_SET_CREATE_INFO};
strncpy(actionSetInfo.actionSetName, "gameplay");
strncpy(actionSetInfo.localizedActionSetName, "Gameplay");
actionSetInfo.priority = 0;
XrActionSet inGameActionSet;
CHK_XR(xrCreateActionSet(instance, &actionSetInfo, &inGameActionSet));

// create a "teleport" input action
XrActionCreateInfo actioninfo{XR_TYPE_ACTION_CREATE_INFO};
strncpy(actioninfo.actionName, "teleport");
actioninfo.actionType = XR_ACTION_TYPE_BOOLEAN_INPUT;
strncpy(actioninfo.localizedDescription, "Teleport");
XrAction teleportAction;
CHK_XR(xrCreateAction(inGameActionSet, &actioninfo, &teleportAction));

// create a "player_hit" output action
XrActionCreateInfo hapticsactioninfo{XR_TYPE_ACTION_CREATE_INFO};
strncpy(hapticsactioninfo.actionName, "player_hit");
hapticsactioninfo.actionType = XR_ACTION_TYPE_VIBRATION_OUTPUT;
strncpy(hapticsactioninfo.localizedDescription, "Player hit");
XrAction hapticsAction;
CHK_XR(xrCreateAction(inGameActionSet, &hapticsactioninfo, &hapticsAction));
```
XrPath triggerClickPath, hapticPath;
CHK_XR(xrStringToPath(instance, "/user/hand/right/input/trigger/click",
   &triggerClickPath));
CHK_XR(xrStringToPath(instance, "/user/hand/right/output/haptic",
   &hapticPath));

XrPath interactionProfilePath;
CHK_XR(xrStringToPath(instance, "/interaction_profiles/vendor_x/profile_x",
   &interactionProfilePath));

XrActionSuggestedBinding bindings[2];
bindings[0].action = teleportAction;
bindings[0].binding = triggerClickPath;
bindings[1].action = hapticsAction;
bindings[1].binding = hapticPath;

XrInteractionProfileSuggestedBinding
suggestedBindings[XR_TYPE_INTERACTION_PROFILE_SUGGESTED_BINDING];
suggestedBindings.interactionProfile = interactionProfilePath;
suggestedBindings.suggestedBindings = bindings;
suggestedBindings.countSuggestedBindings = 2;
CHK_XR(xrSuggestInteractionProfileBindings(instance, &suggestedBindings));

XrSessionActionSetsAttachInfo attachInfo[XR_TYPE_SESSION_ACTION_SETS_ATTACH_INFO];
attachInfo.countActionSets = 1;
attachInfo.actionSets = &inGameActionSet;
CHK_XR(xrAttachSessionActionSets(session, &attachInfo));

// application main loop
while (1)
{
   // sync action data
   XrActiveActionSet activeActionSet[inGameActionSet, XR_NULL_PATH];
   XrActionsSyncInfo syncInfo[XR_TYPE_ACTIONS_SYNC_INFO];
syncInfo.countActiveActionSets = 1;
syncInfo.activeActionSets = &activeActionSet;
   CHK_XR(xrSyncActions(session, &syncInfo));

   // query input action state
   XrActionStateBoolean teleportState[XR_TYPE_ACTION_STATE_BOOLEAN];
   XrActionStateGetInfo getInfo[XR_TYPE_ACTION_STATE_GET_INFO];
   getInfo.action = teleportAction;
   CHK_XR(xrGetActionStateBoolean(session, &getInfo, &teleportState));

   if (teleportState.changedSinceLastSync & teleportState.currentState)
   {
      // fire haptics using output action
      XrHapticVibration vibration[XR_TYPE_HAPTIC_VIBRATION];
      vibration.amplitude = 0.5;
vibration.duration = 300;
vibration.frequency = 3000;
XrHapticActionInfo hapticActionInfo{XR_TYPE_HAPTIC_ACTION_INFO};
hapticActionInfo.action = hapticsAction;
CHK_XR(xrApplyHapticFeedback(session, &hapticActionInfo, (const
XrHapticBaseHeader**)&vibration));
}

## 11.2. Action Sets

Action sets are application-defined collections of actions. They are attached to a given XrSession with a xrAttachSessionActionSets call. They are enabled or disabled by the application via xrSyncActions depending on the current application context. For example, a game may have one set of actions that apply to controlling a character and another set for navigating a menu system. When these actions are grouped into two XrActionSet handles they can be selectively enabled and disabled using a single function call.

Actions are passed a handle to their XrActionSet when they are created.

Action sets are created by calling xrCreateActionSet:

The xrCreateActionSet function is defined as:

```c
XrResult xrCreateActionSet(
    XrInstance instance,
    const XrActionSetCreateInfo* createInfo,
    XrActionSet* actionSet);
```

### Parameter Descriptions

- **instance** is a handle to an XrInstance.
- **createInfo** is a pointer to a valid XrActionSetCreateInfo structure that defines the action set being created.
- **actionSet** is a pointer to an XrActionSet where the created action set is returned.
The `xrCreateActionSet` function creates an action set and returns a handle to the created action set.

### Valid Usage (Implicit)

- **instance** must be a valid `XrInstance` handle
- **createInfo** must be a pointer to a valid `XrActionSetCreateInfo` structure
- **actionSet** must be a pointer to an `XrActionSet` handle

### Return Codes

**Success**

- `XR_SUCCESS`

**Failure**

- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_OUT_OF_MEMORY`
- `XR_ERROR_LIMIT_REACHED`
- `XR_ERROR_PATH_FORMAT_INVALID`
- `XR_ERROR_NAME_INVALID`
- `XR_ERROR_NAME_DUPLICATED`
- `XR_ERROR_LOCALIZED_NAME_INVALID`
- `XR_ERROR_LOCALIZED_NAME_DUPLICATED`

The `XrActionSetCreateInfo` structure is defined as:

```c
typedef struct XrActionSetCreateInfo {
    XrStructureType    type;
    const void*        next;
    char               actionSetName[XR_MAX_ACTION_SET_NAME_SIZE];
    char               localizedActionSetName[XR_MAX_LOCALIZED_ACTION_SET_NAME_SIZE];
    uint32_t           priority;
} XrActionSetCreateInfo;
```
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **actionSetName** is an array containing a NULL terminated non-empty string with the name of this action set.
- **localizedActionSetName** is an array containing a NULL terminated UTF-8 string that can be presented to the user as a description of the action set. This string should be presented in the system's current active locale.
- **priority** defines which action sets' actions are active on a given input source when actions on multiple active action sets are bound to the same input source. Larger priority numbers take precedence over smaller priority numbers.

When multiple actions are bound to the same input source, the **priority** of each action set determines which bindings are suppressed. Runtimes **must** ignore input sources from action sets with a lower priority number if those specific input sources are also present in active actions within a higher priority action set. If multiple action sets with the same priority are bound to the same input source and that is the highest priority number, runtimes **must** process all those bindings at the same time.

Two actions are considered to be bound to the same input source if they use the same **identifier** and **optional location** path segments, even if they have different component segments.

When runtimes are ignoring bindings because of priority, they **must** treat the binding to that input source as though they do not exist. That means the **isActive** field **must** be XR_FALSE when retrieving action data, and that the runtime **must** not provide any visual, haptic, or other feedback related to the binding of that action to that input source. Other actions in the same action set which are bound to input sources that do not collide are not affected and are processed as normal.

If **actionSetName** or **localizedActionSetName** are empty strings, the runtime **must** return XR_ERROR_NAME_INVALID or XR_ERROR_LOCALIZED_NAME_INVALID respectively. If **actionSetName** or **localizedActionSetName** are duplicates of the corresponding field for any existing action set in the specified instance, the runtime **must** return XR_ERROR_NAME_DUPLICATED or XR_ERROR_LOCALIZED_NAME_DUPLICATED respectively. If the conflicting action set is destroyed, the conflicting field is no longer considered duplicated. If **actionSetName** contains characters which are not allowed in a single level of a well-formed path string, the runtime **must** return XR_ERROR_PATH_FORMAT_INVALID.
Valid Usage (Implicit)

- **type** must be `XR_TYPE_ACTION_SET_CREATE_INFO`
- **next** must be `NULL` or a valid pointer to the next structure in a structure chain
- **actionSetName** must be a null-terminated UTF-8 string whose length is less than or equal to `XR_MAX_ACTION_SET_NAME_SIZE`
- **localizedActionSetName** must be a null-terminated UTF-8 string whose length is less than or equal to `XR_MAX_LOCALIZED_ACTION_SET_NAME_SIZE`

The `xrDestroyActionSet` function is defined as:

```
XrResult xrDestroyActionSet(
  XrActionSet actionSet);
```

Parameter Descriptions

- **actionSet** is the action set to destroy.

Action set handles can be destroyed by calling `xrDestroyActionSet`. When an action set handle is destroyed, all handles of actions in that action set are also destroyed.

The implementation must not free underlying resources for the action set while there are other valid handles that refer to those resources. The implementation may release resources for an action set when all of the action spaces for actions in that action set have been destroyed. See Action Spaces Lifetime for details.

Resources for all action sets in an instance must be freed when the instance containing those actions sets is destroyed.

Valid Usage (Implicit)

- **actionSet** must be a valid `XrActionSet` handle

Thread Safety

- Access to **actionSet**, and any child handles, must be externally synchronized
### Return Codes

**Success**
- XR_SUCCESS

**Failure**
- XR_ERROR_HANDLE_INVALID

### 11.3. Creating Actions

XR_DEFINE_HANDLE(XrAction)

Action handles are used to refer to individual actions when retrieving action data, creating action spaces, or sending haptic events.

The `xrCreateAction` function is defined as:

```c
XrResult xrCreateAction(
    XrActionSet actionSet,          // set to attach this action to
    const XrActionCreateInfo* createInfo,  // structure that defines the action
    XrAction* action);              // action to create
```

**Parameter Descriptions**

- `actionSet` is a handle to an `XrActionSet`.
- `createInfo` is a pointer to a valid `XrActionCreateInfo` structure that defines the action being created.
- `action` is a pointer to an `XrAction` where the created action is returned.

`xrCreateAction` creates an action and returns its handle.

**Note**: If `actionSet` has been included in a call to `xrAttachSessionActionSets`, the implementation **must** return `XR_ERROR_ACTIONSETS_ALREADY_ATTACHED`.

---

186 | Chapter 11. Input and Haptics
Valid Usage (Implicit)

- **actionSet** must be a valid XrActionSet handle
- **createInfo** must be a pointer to a valid XrActionCreateInfo structure
- **action** must be a pointer to an XrAction handle

Return Codes

**Success**
- XR_SUCCESS

**Failure**
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_LIMIT_REACHED
- XR_ERROR_PATH_UNSUPPORTED
- XR_ERROR_PATH_INVALID
- XR_ERROR_PATH_FORMAT_INVALID
- XR_ERROR_NAME_INVALID
- XR_ERROR_NAME_DUPLICATED
- XR_ERROR_LOCALIZED_NAME_INVALID
- XR_ERROR_LOCALIZED_NAME_DUPLICATED
- XR_ERROR_ACTIONSETS_ALREADY_ATTACHED

The XrActionCreateInfo structure is defined as:
typedef struct XrActionCreateInfo {
    XrStructureType    type;
    const void*        next;
    char               actionName[XR_MAX_ACTION_NAME_SIZE];
    XrActionType       actionType;
    uint32_t           countSubactionPaths;
    const XrPath*      subactionPaths;
    char               localizedActionName[XR_MAX_LOCALIZED_ACTION_NAME_SIZE];
} XrActionCreateInfo;

Member Descriptions

• **type** is the XrStructureType of this structure.

• **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

• **actionName** is an array containing a NULL terminated string with the name of this action.

• **actionType** is the XrActionType of the action to be created.

• **countSubactionPaths** is the number of elements in the subactionPaths array. If subactionPaths is NULL, this parameter must be 0.

• **subactionPaths** is an array of XrPath or NULL. If this array is specified, it contains one or more subaction paths that the application intends to query action state for.

• **localizedActionName** is an array containing a NULL terminated UTF-8 string that can be presented to the user as a description of the action. This string should be in the system’s current active locale.

Subaction paths are a mechanism that enables applications to use the same action name and handle on multiple devices. Applications can query action state using subaction paths that differentiate data coming from each device. This allows the runtime to group logically equivalent actions together in system UI. For instance, an application could create a single pick_up action with the /user/hand/left and /user/hand/right subaction paths and use the subaction paths to independently query the state of pick_up_with_left_hand and pick_up_with_right_hand.

Applications can create actions with or without the subactionPaths set to a list of paths. If this list of paths is omitted (i.e. subactionPaths is set to NULL, and countSubactionPaths is set to 0), the application is opting out of filtering action results by subaction paths and any call to get action data must also omit subaction paths.

If subactionPaths is specified and any of the following conditions are not satisfied, the runtime must return XR_ERROR_PATH_UNSUPPORTED:

• Each path provided is one of:
- /user/head
- /user/hand/left
- /user/hand/right
- /user/gamepad

- No path appears in the list more than once

Extensions **may** append additional top level user paths to the above list.

**Note**

Earlier revisions of the spec mentioned /user but it could not be implemented as specified and was removed as errata.

The runtime **must** return `XR_ERROR_PATH_UNSUPPORTED` in the following circumstances:

- The application specified subaction paths at action creation and the application called `xrGetActionState*` or a haptic function with an empty subaction path array.
- The application called `xrGetActionState*` or a haptic function with a subaction path that was not specified when the action was created.

If `actionName` or `localizedActionName` are empty strings, the runtime **must** return `XR_ERROR_NAME_INVALID` or `XR_ERROR_LOCALIZED_NAME_INVALID` respectively. If `actionName` or `localizedActionName` are duplicates of the corresponding field for any existing action in the specified action set, the runtime **must** return `XR_ERROR_NAME_DUPLICATED` or `XR_ERROR_LOCALIZED_NAME_DUPLICATED` respectively. If the conflicting action is destroyed, the conflicting field is no longer considered duplicated. If `actionName` contains characters which are not allowed in a single level of a well-formed path string, the runtime **must** return `XR_ERROR_PATH_FORMAT_INVALID`.

### Valid Usage (Implicit)

- **type** **must** be `XR_TYPE_ACTION_CREATE_INFO`
- **next** **must** be `NULL` or a valid pointer to the next structure in a structure chain
- **actionName** **must** be a null-terminated UTF-8 string whose length is less than or equal to `XR_MAX_ACTION_NAME_SIZE`
- **actionType** **must** be a valid `XrActionType` value
- If `countSubactionPaths` is not 0, `subactionPaths` **must** be a pointer to an array of `countSubactionPaths` valid `XrPath` values
- **localizedActionName** **must** be a null-terminated UTF-8 string whose length is less than or equal to `XR_MAX_LOCALIZED_ACTION_NAME_SIZE`

The `XrActionType` parameter takes one of the following values:
typedef enum XrActionType {
    XR_ACTION_TYPE_BOOLEAN_INPUT = 1,
    XR_ACTION_TYPE_FLOAT_INPUT = 2,
    XR_ACTION_TYPE_VECTOR2F_INPUT = 3,
    XR_ACTION_TYPE_POSE_INPUT = 4,
    XR_ACTION_TYPE_VIBRATION_OUTPUT = 100,
    XR_ACTION_TYPE_MAX_ENUM = 0x7FFFFFFF
} XrActionType;

### Enumerant Descriptions

- **XR_ACTION_TYPE_BOOLEAN_INPUT.** The action can be passed to `xrGetActionStateBoolean` to retrieve a boolean value.

- **XR_ACTION_TYPE_FLOAT_INPUT.** The action can be passed to `xrGetActionStateFloat` to retrieve a float value.

- **XR_ACTION_TYPE_VECTOR2F_INPUT.** The action can be passed to `xrGetActionStateVector2f` to retrieve a 2D float vector.

- **XR_ACTION_TYPE_POSE_INPUT.** The action can be passed to `xrCreateActionSpace` to create a space.

- **XR_ACTION_TYPE_VIBRATION_OUTPUT.** The action can be passed to `xrApplyHapticFeedback` to send a haptic event to the runtime.

The `xrDestroyAction` function is defined as:

```
XrResult xrDestroyAction(
    XrAction action);
```

### Parameter Descriptions

- **action** is the action to destroy.

Action handles can be destroyed by calling `xrDestroyAction`. Handles for actions that are part of an action set are automatically destroyed when the action set's handle is destroyed.

The implementation must not destroy the underlying resources for an action when `xrDestroyAction` is called. Those resources are still used to make action spaces locatable and when processing action
priority in `xrSyncActions`. Destroying the action handle removes the application’s access to these resources, but has no other change on actions.

Resources for all actions in an instance **must** be freed when the instance containing those actions sets is destroyed.

### Valid Usage (Implicit)

- **action** must be a valid `XrAction` handle

### Thread Safety

- Access to **action**, and any child handles, **must** be externally synchronized

### Return Codes

**Success**
- `XR_SUCCESS`

**Failure**
- `XR_ERROR_HANDLE_INVALID`

### 11.3.1. Input Actions & Output Actions

Input actions are used to read sensors like buttons or joysticks while output actions are used for triggering haptics or motion platforms. The type of action created by `xrCreateAction` depends on the value of the `XrActionType` argument.

A given action can either be used for either input or output, but not both. Input actions are queried using one of the `xrGetActionState*` function calls, while output actions are set using the haptics calls. If either call is used with an action of the wrong type `XR_ERROR_ACTION_TYPE_MISMATCH` must be returned.

### 11.4. Suggested Bindings

Applications usually need to provide default bindings for their actions to runtimes so that input data can be mapped appropriately to the application's actions. Applications can do this by calling `xrSuggestInteractionProfileBindings` for each interaction profile that the applications have default bindings for. If bindings are provided for an appropriate interaction profile, the runtime may select one and input will begin to flow. Interaction profile selection changes **must** only happen when `xrSyncActions` is called. Applications can call `xrGetCurrentInteractionProfile` during on a running session to learn what the active interaction profile are for a top level user path. If this value ever
changes, the runtime must send an `XR_TYPE_EVENT_DATA_INTERACTION_PROFILE_CHANGED` event to the application to indicate that the value should be queried again.

The bindings suggested by this system are only a hint to the runtime. Some runtimes may choose to use a different device binding depending on user preference, accessibility settings, or for any other reason. If the runtime is using the values provided by suggested bindings, it must make a best effort to convert the input value to the created action and apply certain rules to that use so that suggested bindings function in the same way across runtimes. If an input value cannot be converted to the type of the action, the value must be ignored and not contribute to the state of the action.

For actions created with `XR_ACTION_TYPE_BOOLEAN_INPUT` when the runtime is obeying suggested bindings: Boolean input sources must be bound directly to the action. If the path is to a scalar value, a threshold must be applied to the value and values over that threshold will be `XR_TRUE`. The runtime should use hysteresis when applying this threshold. The threshold and hysteresis range may vary from device to device or component to component and are left as an implementation detail. If the path refers to the parent of input values instead of to an input value itself, the runtime must use `.../example/path/click` instead of `.../example/path` if it is available. If a parent path does not have a `.../click` subpath, the runtime must use `.../value` and apply the same thresholding that would be applied to any scalar input. In any other situation the runtime may provide an alternate binding for the action or it will be unbound.

For actions created with `XR_ACTION_TYPE_FLOAT_INPUT` when the runtime is obeying suggested bindings: If the input value specified by the path is scalar, the input value must be bound directly to the float. If the path refers to the parent of input values instead of to an input value itself, the runtime must use `/example/path/value` instead of `.../example/path` as the source of the value. If a parent path does not have a `.../value` subpath, the runtime must use `.../click`. If the input value is boolean, the runtime must supply 0.0 or 1.0 as a conversion of the boolean value. In any other situation, the runtime may provide an alternate binding for the action or it will be unbound.

For actions created with `XR_ACTION_TYPE_VECTOR2F_INPUT` when the runtime is obeying suggested bindings: The suggested binding path must refer to the parent of input values instead of to the input values themselves, and that parent path must contain subpaths `.../x` and `.../y`. `.../x` and `.../y` must be bound to 'x' and 'y' of the vector, respectively. In any other situation, the runtime may provide an alternate binding for the action or it will be unbound.

For actions created with `XR_ACTION_TYPE_POSE_INPUT` when the runtime is obeying suggested bindings: Pose input sources must be bound directly to the action. If the path refers to the parent of input values instead of to an input value itself, the runtime must use `.../example/path/pose` instead of `.../example/path` if it is available. In any other situation the runtime may provide an alternate binding for the action or it will be unbound.

The `XrEventDataInteractionProfileChanged` structure is defined as:
typedef struct XrEventDataInteractionProfileChanged {
    XrStructureType    type;
    const void*        next;
    XrSession          session;
} XrEventDataInteractionProfileChanged;

**Member Descriptions**

- **type** is the *XrStructureType* of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **session** is the *XrSession* for which at least one of the interaction profiles for a top level path has changed.

The *XrEventDataInteractionProfileChanged* event is sent to the application to notify it that the active input form factor for one or more top level user paths has changed. This event must only be sent for interaction profiles that the application indicated its support for via *xrSuggestInteractionProfileBindings*. This event must only be sent for running sessions.

The application can call *xrGetCurrentInteractionProfile* if it wants to change its own behavior based on the active hardware.

**Valid Usage (Implicit)**

- **type** must be *XR_TYPE_EVENT_DATA_INTERACTION_PROFILE_CHANGED*
- **next** must be NULL or a valid pointer to the next structure in a structure chain
- **session** must be a valid *XrSession* handle

The *xrSuggestInteractionProfileBindings* function is defined as:

```c
XrResult xrSuggestInteractionProfileBindings(
    XrInstance                                  instance,
    const XrInteractionProfileSuggestedBinding* suggestedBindings);
```
Parameter Descriptions

- **instance** is the XrInstance for which the application would like to set suggested bindings
- **suggestedBindings** is the XrInteractionProfileSuggestedBinding that the application would like to set

xrSuggestInteractionProfileBindings sets an interaction profile for which the application can provide default bindings. The application can call xrSuggestInteractionProfileBindings once per interaction profile that it supports.

The application can provide any number of bindings for each action.

If the application successfully calls xrSuggestInteractionProfileBindings more than once for an interaction profile, the runtime must discard the previous suggested bindings and replace them with the new suggested bindings for that profile.

If the interaction profile path does not follow the structure defined in Interaction Profiles or suggested bindings contain paths that do not follow the format defined in Device input subpaths, the runtime must return XR_ERROR_PATH_UNSUPPORTED. If the interaction profile or input source for any of the suggested bindings does not exist in the allowlist defined in Interaction Profile Paths, the runtime must return XR_ERROR_PATH_UNSUPPORTED. A runtime must accept every valid binding in the allowlist though it is free to ignore any of them.

If the action set for any action referenced in the suggestedBindings parameter has been included in a call to xrAttachSessionActionSets, the implementation must return XR_ERROR_ACTIONSETS_ALREADY_ATTACHED.

Valid Usage (Implicit)

- **instance** must be a valid XrInstance handle
- **suggestedBindings** must be a pointer to a valid XrInteractionProfileSuggestedBinding structure
Return Codes

**Success**
- XR_SUCCESS

**Failure**
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_PATH_UNSUPPORTED
- XR_ERROR_PATH_INVALID
- XR_ERROR_ACTIONSETS_ALREADY_ATTACHED

The `XrInteractionProfileSuggestedBinding` structure is defined as:

```c
typedef struct XrInteractionProfileSuggestedBinding {
    XrStructureType type;
    const void* next;
    XrPath interactionProfile;
    uint32_t countSuggestedBindings;
    const XrActionSuggestedBinding* suggestedBindings;
} XrInteractionProfileSuggestedBinding;
```

**Member Descriptions**

- `type` is the `XrStructureType` of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `interactionProfile` is the `XrPath` of an interaction profile.
- `countSuggestedBindings` is the number of suggested bindings in the array pointed to by `suggestedBindings`.
- `suggestedBindings` is a pointer to an array of `XrActionSuggestedBinding` structures that define all of the application's suggested bindings for the specified interaction profile.
**Valid Usage (Implicit)**

- **type** must be `XR_TYPE_INTERACTION_PROFILE_SUGGESTED_BINDING`
- **next** must be `NULL` or a valid pointer to the next structure in a structure chain. See also: `XrBindingModificationsKHR`
- **suggestedBindings** must be a pointer to an array of `countSuggestedBindings` valid `XrActionSuggestedBinding` structures
- The `countSuggestedBindings` parameter must be greater than 0

The `XrActionSuggestedBinding` structure is defined as:

```c
typedef struct XrActionSuggestedBinding {
    XrAction    action;
    XrPath      binding;
} XrActionSuggestedBinding;
```

**Member Descriptions**

- **action** is the `XrAction` handle for an action
- **binding** is the `XrPath` of a binding for the action specified in `action`. This path is any top level user path plus input source path, for example `/user/hand/right/input/trigger/click`. See suggested bindings for more details.

**Valid Usage (Implicit)**

- **action** must be a valid `XrAction` handle

The `xrAttachSessionActionSets` function is defined as:

```c
XrResult xrAttachSessionActionSets(
    XrSession                                   session,
    const XrSessionActionSetsAttachInfo*        attachInfo);
```
Parameter Descriptions

- **session** is the XrSession to attach the action sets to.
- **attachInfo** is the XrSessionActionSetsAttachInfo to provide information to attach action sets to the session.

xrAttachSessionActionSets attaches the XrActionSet handles in attachInfo.actionSets to the session. Action sets **must** be attached in order to be synchronized with xrSyncActions.

When an action set is attached to a session, that action set becomes immutable. See xrCreateAction and xrSuggestInteractionProfileBindings for details.

After action sets are attached to a session, if any unattached actions are passed to functions for the same session, then for those functions the runtime **must** return XR_ERROR_ACTIONSET_NOT_ATTACHED.

The runtime **must** return **XR_ERROR_ACTIONSETS_ALREADY_ATTACHED** if xrAttachSessionActionSets is called more than once for a given session.

Valid Usage (Implicit)

- **session** **must** be a valid XrSession handle
- **attachInfo** **must** be a pointer to a valid XrSessionActionSetsAttachInfo structure

Return Codes

**Success**
- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

**Failure**
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_ACTIONSETS_ALREADY_ATTACHED

The XrSessionActionSetsAttachInfo structure is defined as:
typedef struct XrSessionActionSetsAttachInfo {
    XrStructureType       type;
    const void*           next;
    uint32_t              countActionSets;
    const XrActionSet*    actionSets;
} XrSessionActionSetsAttachInfo;

Member Descriptions

• **type** is the XrStructureType of this structure.

• **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.

• **countActionSets** is an integer specifying the number of valid elements in the **actionSets** array.

• **actionSets** is a pointer to an array of one or more XrActionSet handles to be attached to the session.

Valid Usage (Implicit)

• **type** must be XR_TYPE_SESSION_ACTION_SETS_ATTACH_INFO

• **next** must be NULL or a valid pointer to the **next** structure in a structure chain

• **actionSets** must be a pointer to an array of **countActionSets** valid XrActionSet handles

• The **countActionSets** parameter must be greater than 0

The xrGetCurrentInteractionProfile function is defined as:

XrResult xrGetCurrentInteractionProfile(
    XrSession session,
    XrPath topLevelUserPath,
    XrInteractionProfileState* interactionProfile);
Parameter Descriptions

- `session` is the `XrSession` for which the application would like to retrieve the current interaction profile.
- `topLevelUserPath` is the top level user path the application would like to retrieve the interaction profile for.
- `interactionProfile` is a pointer to an `XrInteractionProfileState` structure to receive the current interaction profile.

`xrGetCurrentInteractionProfile` asks the runtime for the active interaction profiles for a top level user path.

The runtime **must** return only interaction profiles for which the application has provided bindings with `xrSuggestInteractionProfileBindings` or `XR_NULL_PATH`. The runtime **may** return interaction profiles that do not represent physically present hardware, for example if the runtime is using a known interaction profile to bind to hardware that the application is not aware of. The runtime **may** return the last-known interaction profile in the event that no controllers are active.

If `xrAttachSessionActionSets` has not yet been called for the `session`, the runtime **must** return `XR_ERROR_ACTIONSET_NOT_ATTACHED`. If `topLevelUserPath` is not one of the device input subpaths described in section /user paths, the runtime **must** return `XR_ERROR_PATH_UNSUPPORTED`.

Valid Usage (Implicit)

- `session` **must** be a valid `XrSession` handle
- `interactionProfile` **must** be a pointer to an `XrInteractionProfileState` structure
Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_PATH_UNSUPPORTED
- XR_ERROR_PATH_INVALID
- XR_ERROR_ACTIONSET_NOT_ATTACHED

The XrInteractionProfileState structure is defined as:

typedef struct XrInteractionProfileState {
    XrStructureType    type;
    void*              next;
    XrPath             interactionProfile;
} XrInteractionProfileState;

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- interactionProfile is the XrPath of the interaction profile path for the topLevelUserPath used to retrieve this state, or XR_NULL_PATH if there is no active interaction profile at that top level user path.

The runtime must only include interaction profiles that the application has provided bindings for via xrSuggestInteractionProfileBindings or XR_NULL_PATH. If the runtime is rebinding an interaction profile provided by the application to a device that the application did not provide bindings for, it
must return the interaction profile path that it is emulating. If the runtime is unable to provide input because it cannot emulate any of the application-provided interaction profiles, it must return XR_NULL_PATH.

Valid Usage (Implicit)

- type must be XR_TYPE_INTERACTION_PROFILE_STATE
- next must be NULL or a valid pointer to the next structure in a structure chain

11.5. Reading Input Action State

The current state of an input action can be obtained by calling the xrGetActionState* function call that matches the XrActionType provided when the action was created. If a mismatched call is used to retrieve the state XR_ERROR_ACTION_TYPE_MISMATCH must be returned. xrGetActionState* calls for an action in an action set never bound to the session with xrAttachSessionActionSets must return XR_ERROR_ACTIONSET_NOT_ATTACHED.

The result of calls to xrGetActionState* for an XrAction and subaction path must not change between calls to xrSyncActions. When the combination of the parent XrActionSet and subaction path for an action is passed to xrSyncActions, the runtime must update the results from xrGetActionState* after this call with any changes to the state of the underlying hardware. When the parent action set and subaction path for an action is removed from or added to the list of active action sets passed to xrSyncActions, the runtime must update isActive to reflect the new active state after this call. In all cases the runtime must not change the results of xrGetActionState* calls between calls to xrSyncActions.

When xrGetActionState* or haptic output functions are called while the session is not focused, the runtime must set the isActive value to XR_FALSE and suppress all haptic output. Furthermore, the runtime should stop all in-progress haptic events when a session loses focus.

When retrieving action state, lastChangeTime must be set to the runtime's best estimate of when the physical state of the part of the device bound to that action last changed.

The currentState value is computed based on the current sync, combining the underlying input sources bound to the provided subactionPaths within this action.

The changedSinceLastSync value must be XR_TRUE if the computed currentState value differs from the currentState value that would have been computed as of the previous sync for the same subactionPaths. If there is no previous sync, or the action was not active for the previous sync, the changedSinceLastSync value must be set to XR_FALSE.

The isActive value must be XR_TRUE whenever an action is bound and a source is providing state data for the current sync. If the action is unbound or no source is present, the isActive value must be XR_FALSE. For any action which is inactive, the runtime must return zero (or XR_FALSE) for state,
XR_FALSE for changedSinceLastSync, and 0 for lastChangeTime.

11.5.1. Resolving a single action bound to multiple inputs or outputs

It is often the case that a single action will be bound to multiple physical inputs simultaneously. In these circumstances, the runtime must resolve the ambiguity in that multiple binding as follows:

The current state value is selected based on the type of the action:

- **Boolean actions** - The current state must be the result of a boolean OR of all bound inputs
- **Float actions** - The current state must be the state of the input with the largest absolute value
- **Vector2 actions** - The current state must be the state of the input with the longest length
- **Pose actions** - The runtime must select a single pose source when the action is created or bound and use that value consistently. The runtime should use subaction paths specified by the application to make this choice where possible.
- **Haptic actions** - The runtime must send output events to all bound haptic devices

11.5.2. Structs to describe action and subaction paths

The XrActionStateGetInfo structure is used to provide action and subaction paths when calling xrGetActionState* function. It is defined as:

```c
typedef struct XrActionStateGetInfo {
    XrStructureType    type;
    const void*        next;
    XrAction           action;
    XrPath             subactionPath;
} XrActionStateGetInfo;
```
Member Descriptions

- **type** is the `XrStructureType` of this structure.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **action** is the `XrAction` being queried.
- **subactionPath** is the subaction path `XrPath` to query data from, or `XR_NULL_PATH` to specify all subaction paths. If the subaction path is specified, it is one of the subaction paths that were specified when the action was created. If the subaction path was not specified when the action was created, the runtime must return `XR_ERROR_PATH_UNSUPPORTED`. If this parameter is specified, the runtime must return data that originates only from the subaction paths specified.

See `XrActionCreateInfo` for a description of subaction paths, and the restrictions on their use.

Valid Usage (Implicit)

- **type** must be `XR_TYPE_ACTION_STATE_GET_INFO`
- **next** must be `NULL` or a valid pointer to the next structure in a structure chain
- **action** must be a valid `XrAction` handle

The `XrHapticActionInfo` structure is used to provide action and subaction paths when calling `xr*HapticFeedback` function. It is defined as:

```c
typedef struct XrHapticActionInfo {
    XrStructureType    type;
    const void*        next;
    XrAction           action;
    XrPath             subactionPath;
} XrHapticActionInfo;
```
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **action** is the XrAction handle for the desired output haptic action.
- **subactionPath** is the subaction path XrPath of the device to send the haptic event to, or XR_NULL_PATH to specify all subaction paths. If the subaction path is specified, it is one of the subaction paths that were specified when the action was created. If the subaction path was not specified when the action was created, the runtime **must** return XR_ERROR_PATH_UNSUPPORTED. If this parameter is specified, the runtime **must** trigger the haptic events only on the device from the subaction path.

See XrActionCreateInfo for a description of subaction paths, and the restrictions on their use.

Valid Usage (Implicit)

- **type** **must** be XR_TYPE_HAPTIC_ACTION_INFO
- **next** **must** be NULL or a valid pointer to the next structure in a structure chain
- **action** **must** be a valid XrAction handle

11.5.3. Boolean Actions

xrGetActionStateBoolean retrieves the current state of a boolean action. It is defined as:

```c
XrResult xrGetActionStateBoolean(
    XrSession session,
    const XrActionStateGetInfo* getInfo,
    XrActionStateBoolean* state);
```

Parameter Descriptions

- **session** is the XrSession to query.
- **getInfo** is a pointer to XrActionStateGetInfo to provide action and subaction paths information.
- **state** is a pointer to a valid XrActionStateBoolean into which the state will be placed.
Valid Usage (Implicit)

- `session` must be a valid `XrSession` handle
- `getInfo` must be a pointer to a valid `XrActionStateGetInfo` structure
- `state` must be a pointer to an `XrActionStateBoolean` structure

Return Codes

**Success**

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

**Failure**

- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_PATH_UNSUPPORTED`
- `XR_ERROR_PATH_INVALID`
- `XR_ERROR_ACTION_TYPE_MISMATCH`
- `XR_ERROR_ACTIONSET_NOT_ATTACHED`

The `XrActionStateBoolean` structure is defined as:

```c
typedef struct XrActionStateBoolean {
    XrStructureType type;
    void* next;
    XrBool32 currentState;
    XrBool32 changedSinceLastSync;
    XrTime lastChangeTime;
    XrBool32 isActive;
} XrActionStateBoolean;
```
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **currentState** is the current state of the action.
- **changedSinceLastSync** is XR_TRUE if the value of currentState is different than it was before the most recent call to xrSyncActions. This parameter can be combined with currentState to detect rising and falling edges since the previous call to xrSyncActions. E.g. if both changedSinceLastSync and currentState are XR_TRUE then a rising edge (XR_FALSE to XR_TRUE) has taken place.
- **lastChangeTime** is the XrTime when this action’s value last changed.
- **isActive** is XR_TRUE if and only if there exists an input source that is contributing to the current state of this action.

When multiple input sources are bound to this action, the currentState follows the previously defined rule to resolve ambiguity.

Valid Usage (Implicit)

- **type** must be XR_TYPE_ACTION_STATE_BOOLEAN
- **next** must be NULL or a valid pointer to the next structure in a structure chain

11.5.4. Scalar and Vector Actions

xrGetActionStateFloat retrieves the current state of a floating-point action. It is defined as:

```c
XrResult xrGetActionStateFloat(
    XrSession session,
    const XrActionStateGetInfo* getInfo,
    XrActionStateFloat* state);
```
Parameter Descriptions

- **session** is the XrSession to query.
- **getInfo** is a pointer to XrActionStateGetInfo to provide action and subaction paths information.
- **state** is a pointer to a valid XrActionStateFloat into which the state will be placed.

Valid Usage (Implicit)

- **session** must be a valid XrSession handle
- **getInfo** must be a pointer to a valid XrActionStateGetInfo structure
- **state** must be a pointer to an XrActionStateFloat structure

Return Codes

**Success**

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

**Failure**

- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_PATH_UNSUPPORTED
- XR_ERROR_PATH_INVALID
- XR_ERROR_ACTION_TYPE_MISMATCH
- XR_ERROR_ACTIONSET_NOT_ATTACHED

The XrActionStateFloat structure is defined as:
typedef struct XrActionStateFloat {
    XrStructureType         type;
    void*                   next;
    float                   currentState;
    XrBool32                changedSinceLastSync;
    XrTime                  lastChangeTime;
    XrBool32                isActive;
} XrActionStateFloat;

**Member Descriptions**

- **type** is the **XrStructureType** of this structure.
- **next** is **NULL** or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **currentState** is the current state of the Action.
- **changedSinceLastSync** is **XR_TRUE** if the value of **currentState** is different than it was before the most recent call to **xrSyncActions**.
- **lastChangeTime** is the **XrTime** in nanoseconds since this action’s value last changed.
- **isActive** is **XR_TRUE** if and only if there exists an input source that is contributing to the current state of this action.

When multiple input sources are bound to this action, the **currentState** follows the previously defined rule to resolve ambiguity.

**Valid Usage (Implicit)**

- **type** must be **XR_TYPE_ACTION_STATE_FLOAT**
- **next** must be **NULL** or a valid pointer to the next structure in a structure chain

**xrGetActionStateVector2f** retrieves the current state of a two-dimensional vector action. It is defined as:

XrResult xrGetActionStateVector2f(
    XrSession session,
    const XrActionStateGetInfo* getInfo,
    XrActionStateVector2f* state);
Parameter Descriptions

- **session** is the XrSession to query.
- **getInfo** is a pointer to XrActionStateGetInfo to provide action and subaction paths information.
- **state** is a pointer to a valid XrActionStateVector2f into which the state will be placed.

Valid Usage (Implicit)

- **session** must be a valid XrSession handle
- **getInfo** must be a pointer to a valid XrActionStateGetInfo structure
- **state** must be a pointer to an XrActionStateVector2f structure

Return Codes

Success

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure

- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_PATH_UNSUPPORTED
- XR_ERROR_ACTION_TYPE_MISMATCH
- XR_ERROR_ACTIONSET_NOT_ATTACHED

The XrActionStateVector2f structure is defined as:
typedef struct XrActionStateVector2f {
    XrStructureType type;
    void* next;
    XrVector2f currentState;
    XrBool32 changedSinceLastSync;
    XrTime lastChangeTime;
    XrBool32 isActive;
} XrActionStateVector2f;

**Member Descriptions**

- **type** is the *XrStructureType* of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **currentState** is the current *XrVector2f* state of the Action.
- **changedSinceLastSync** is XR_TRUE if the value of **currentState** is different than it was before the most recent call to *xrSyncActions*.
- **lastChangeTime** is the *XrTime* in nanoseconds since this action’s value last changed.
- **isActive** is XR_TRUE if and only if there exists an input source that is contributing to the current state of this action.

When multiple input sources are bound to this action, the **currentState** follows the previously defined rule to resolve ambiguity.

**Valid Usage (Implicit)**

- **type** must be XR_TYPE_ACTION_STATE_VECTOR2F
- **next** must be NULL or a valid pointer to the next structure in a structure chain

### 11.5.5. Pose Actions

The *xrGetActionStatePose* function is defined as:

```c
XrResult xrGetActionStatePose(
    XrSession session,
    const XrActionStateGetInfo* getInfo,
    XrActionStatePose* state);
```
**Parameter Descriptions**

- **session** is the XrSession to query.
- **getInfo** is a pointer to XrActionStateGetInfo to provide action and subaction paths information.
- **state** is a pointer to a valid XrActionStatePose into which the state will be placed.

xrGetActionStatePose returns information about the binding and active state for the specified action. To determine the pose of this action at a historical or predicted time, the application can create an action space using xrCreateActionSpace. Then, after each sync, the application can locate the pose of this action space within a base space using xrLocateSpace.

**Valid Usage (Implicit)**

- **session** must be a valid XrSession handle
- **getInfo** must be a pointer to a valid XrActionStateGetInfo structure
- **state** must be a pointer to an XrActionStatePose structure

**Return Codes**

**Success**

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

**Failure**

- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_PATH_UNSUPPORTED
- XR_ERROR_PATH_INVALID
- XR_ERROR_ACTION_TYPE_MISMATCH
- XR_ERROR_ACTIONSET_NOT_ATTACHED

The XrActionStatePose structure is defined as:
typedef struct XrActionStatePose {
    XrStructureType    type;
    void*              next;
    XrBool32           isActive;
} XrActionStatePose;

**Member Descriptions**

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **isActive** is XR_TRUE if and only if there exists an input source that is being tracked by this pose action.

A pose action **must** not be bound to multiple input sources, according to the previously defined rule.

**Valid Usage (Implicit)**

- **type** **must** be XR_TYPE_ACTION_STATE_POSE
- **next** **must** be NULL or a valid pointer to the next structure in a structure chain

### 11.6. Output Actions and Haptics

Haptic feedback is sent to a device using the xrApplyHapticFeedback function. The hapticEvent points to a supported event structure. All event structures have in common that the first element is an XrHapticBaseHeader which can be used to determine the type of the haptic event.

Haptic feedback may be immediately halted for a haptic action using the xrStopHapticFeedback function.

Output action requests activate immediately and **must** not wait for the next call to xrSyncActions.

If a haptic event is sent to an action before a previous haptic event completes, the latest event will take precedence and the runtime **must** cancel all preceding incomplete haptic events on that action.

Output action requests **must** be discarded and have no effect on hardware if the application’s session is not focused.

Output action requests for an action in an action set never attached to the session with...
xrAttachSessionActionSets must return XR_ERROR_ACTIONSET_NOT_ATTACHED.

The only haptics type supported by unextended OpenXR is XrHapticVibration.

The xrApplyHapticFeedback function is defined as:

```c
XrResult xrApplyHapticFeedback(
    XrSession session,
    const XrHapticActionInfo* hapticActionInfo,
    const XrHapticBaseHeader* hapticFeedback);
```

### Parameter Descriptions

- **session** is the XrSession to start outputting to.
- **hapticActionInfo** is a pointer to XrHapticActionInfo to provide action and subaction paths information.
- **hapticFeedback** is a pointer to a haptic event structure which starts with an XrHapticBaseHeader.

Triggers a haptic event through the specified action of type XR_ACTION_TYPE_VIBRATION_OUTPUT. The runtime should deliver this request to the appropriate device, but exactly which device, if any, this event is sent to is up to the runtime to decide. If an appropriate device is unavailable the runtime may ignore this request for haptic feedback.

If session is not focused, the runtime must return XR_SESSION_NOT_FOCUSED, and not trigger a haptic event.

If another haptic event from this session is currently happening on the device bound to this action, the runtime must interrupt that other event and replace it with the new one.

### Valid Usage (Implicit)

- **session** must be a valid XrSession handle
- **hapticActionInfo** must be a pointer to a valid XrHapticActionInfo structure
- **hapticFeedback** must be a pointer to a valid XrHapticBaseHeader-based structure. See also: XrHapticVibration
Return Codes

**Success**
- XR_SUCCESS
- XR_SESSION_LOSS_PENDING
- XR_SESSION_NOT_FOCUSED

**Failure**
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_PATH_UNSUPPORTED
- XR_ERROR_PATH_INVALID
- XR_ERROR_ACTION_TYPE_MISMATCH
- XR_ERROR_ACTIONSET_NOT_ATTACHED

The `XrHapticBaseHeader` structure is defined as:

typedef struct XrHapticBaseHeader {
    XrStructureType type;
    const void* next;
} XrHapticBaseHeader;

**Member Descriptions**
- *type* is the `XrStructureType` of this structure. This base structure itself has no associated `XrStructureType` value.
- *next* is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
Valid Usage (Implicit)

- `type` must be `XR_TYPE_HAPTIC_VIBRATION`
- `next` must be `NULL` or a valid pointer to the next structure in a structure chain

The `XrHapticVibration` structure is defined as:

```c
typedef struct XrHapticVibration {
    XrStructureType    type;
    const void*        next;
    XrDuration         duration;
    float              frequency;
    float              amplitude;
} XrHapticVibration;
```

Member Descriptions

- `type` is the `XrStructureType` of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- `duration` is the number of nanoseconds the vibration should last. If `XR_MIN_HAPTIC_DURATION` is specified, the runtime must produce a short haptics pulse of minimal supported duration for the haptic device.
- `frequency` is the frequency of the vibration in Hz. If `XR_FREQUENCY_UNSPECIFIED` is specified, it is left to the runtime to decide the optimal frequency value to use.
- `amplitude` is the amplitude of the vibration between 0.0 and 1.0.

The `XrHapticVibration` is used in calls to `xrApplyHapticFeedback` that trigger vibration output actions.

The `duration`, and `frequency` parameters may be clamped to implementation-dependent ranges.

Valid Usage (Implicit)

- `type` must be `XR_TYPE_HAPTIC_VIBRATION`
- `next` must be `NULL` or a valid pointer to the next structure in a structure chain

`XR_MIN_HAPTIC_DURATION` is used to indicate to the runtime that a short haptic pulse of the minimal
supported duration for the haptic device.

#define XR_MIN_HAPTIC_DURATION -1

XR_FREQUENCY_UNSPECIFIED is used to indicate that the application wants the runtime to decide what the optimal frequency is for the haptic pulse.

#define XR_FREQUENCY_UNSPECIFIED 0

The xrStopHapticFeedback function is defined as:

```c
XrResult xrStopHapticFeedback(  
    XrSession session,  
    const XrHapticActionInfo* hapticActionInfo);
```

**Parameter Descriptions**

- **session** is the XrSession to stop outputting to.
- **hapticActionInfo** is a pointer to an XrHapticActionInfo to provide action and subaction path information.

If a haptic event from this XrAction is in progress, when this function is called the runtime **must** stop that event.

If **session** is not focused, the runtime **must** return XR_SESSION_NOT_FOCUSED.

**Valid Usage (Implicit)**

- **session must** be a valid XrSession handle
- **hapticActionInfo must** be a pointer to a valid XrHapticActionInfo structure
**Return Codes**

**Success**
- XR_SUCCESS
- XR_SESSION_LOSS_PENDING
- XR_SESSION_NOT_FOCUSED

**Failure**
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_PATH_UNSUPPORTED
- XR_ERROR_PATH_INVALID
- XR_ERROR_ACTION_TYPE_MISMATCH
- XR_ERROR_ACTIONSET_NOT_ATTACHED

### 11.7. Input Action State Synchronization

The `xrSyncActions` function is defined as:

```c
XrResult xrSyncActions(
    XrSession                        session,
    const XrActionsSyncInfo*        syncInfo);
```

**Parameter Descriptions**

- `session` is a handle to the `XrSession` that all provided action set handles belong to.
- `syncInfo` is an `XrActionsSyncInfo` providing information to synchronize action states.

`xrSyncActions` updates the current state of input actions. Repeated input action state queries between subsequent synchronization calls must return the same values. The `XrActionSet` structures referenced in the `syncInfo.activeActionSets` must have been previously attached to the session via `xrAttachSessionActionSets`. If any action sets not attached to this session are passed to `xrSyncActions` it...
must return XR_ERROR_ACTIONSET_NOT_ATTACHED. Subsets of the bound action sets can be synchronized in order to control which actions are seen as active.

If session is not focused, the runtime must return XR_SESSION_NOT_FOCUSED, and all action states in the session must be inactive.

Valid Usage (Implicit)

- session must be a valid XrSession handle
- syncInfo must be a pointer to a valid XrActionsSyncInfo structure

Return Codes

Success
- XR_SUCCESS
- XR_SESSION_LOSS_PENDING
- XR_SESSION_NOT_FOCUSED

Failure
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_PATH_UNSUPPORTED
- XR_ERROR_PATH_INVALID
- XR_ERROR_ACTIONSET_NOT_ATTACHED

The XrActionsSyncInfo structure is defined as:

typedef struct XrActionsSyncInfo {
    XrStructureType             type;
    const void*                 next;
    uint32_t                    countActiveActionSets;
    const XrActiveActionSet*    activeActionSets;
} XrActionsSyncInfo;
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **countActiveActionSets** is an integer specifying the number of valid elements in the activeActionSets array.
- **activeActionSets** is NULL or a pointer to an array of one or more XrActiveActionSet structures that should be synchronized.

Valid Usage (Implicit)

- **type** must be XR_TYPE_ACTIONS_SYNC_INFO
- **next** must be NULL or a valid pointer to the next structure in a structure chain
- If **countActiveActionSets** is not 0, **activeActionSets** must be a pointer to an array of **countActiveActionSets** valid XrActiveActionSet structures

The XrActiveActionSet structure is defined as:

```c
typedef struct XrActiveActionSet {
    XrActionSet   actionSet;
    XrPath        subactionPath;
} XrActiveActionSet;
```

Member Descriptions

- **actionSet** is the handle of the action set to activate.
- **subactionPath** is a subaction path that was declared when one or more actions in the action set was created or XR_NULL_PATH. If the application wants to activate the action set on more than one subaction path, it can include additional XrActiveActionSet structs with the other subactionPath values. Using XR_NULL_PATH as the value for subactionPath, acts as a wildcard for all subaction paths on the actions in the action set. If the subaction path was not specified on any of the actions in the actionSet when that action was created, the runtime must return XR_ERROR_PATH_UNSUPPORTED.

This structure defines a single active action set and subaction path combination. Applications can
provide a list of these structures to the `xrSyncActions` function.

### Valid Usage (Implicit)

- `actionSet` must be a valid `XrActionSet` handle

## 11.8. Action Sources

An application can use the `xrEnumerateBoundSourcesForAction` and `xrGetInputSourceLocalizedName` calls to prompt the user which physical inputs to use in order to perform an action. A source is the physical control that the action is bound to within the current interaction profile as returned by `xrGetCurrentInteractionProfile`. An action may be bound to multiple sources at one time, for example an action named `hold` could be bound to both the X and A buttons.

Once the semantic paths for the action’s source are obtained, the application can gather additional information about the source. `xrGetInputSourceLocalizedName` returns a localized human-readable string describing the source, e.g. 'A Button'.

The `xrEnumerateBoundSourcesForAction` function is defined as:

```c
XrResult xrEnumerateBoundSourcesForAction(
    XrSession session,
    const XrBoundSourcesForActionEnumerateInfo* enumerateInfo,
    uint32_t sourceCapacityInput,
    uint32_t* sourceCountOutput,
    XrPath* sources);
```
Parameter Descriptions

- **session** is the XrSession being queried.
- **enumerateInfo** is an XrBoundSourcesForActionEnumerateInfo providing the query information.
- **sourceCapacityInput** is the capacity of the array, or 0 to indicate a request to retrieve the required capacity.
- **sourceCountOutput** is a pointer to the count of sources, or a pointer to the required capacity in the case that **sourceCapacityInput** is insufficient.
- **sources** is a pointer to an application-allocated array that will be filled with the XrPath values for all sources. It can be NULL if **sourceCapacityInput** is 0.
- See **Buffer Size Parameters** chapter for a detailed description of retrieving the required sources size.

If an action is unbound, **xrEnumerateBoundSourcesForAction** must assign 0 to the value pointed to by **sourceCountOutput** and not modify the array.

**xrEnumerateBoundSourcesForAction** must return XR_ERROR_ACTIONSET_NOT_ATTACHED if passed an action in an action set never attached to the session with **xrAttachSessionActionSets**.

As bindings for actions do not change between calls to **xrSyncActions**, **xrEnumerateBoundSourcesForAction** must enumerate the same set of bound sources, or absence of bound sources, for a given query (defined by the **enumerateInfo** parameter) between any two calls to **xrSyncActions**.

Valid Usage (Implicit)

- **session** must be a valid XrSession handle
- **enumerateInfo** must be a pointer to a valid XrBoundSourcesForActionEnumerateInfo structure
- **sourceCountOutput** must be a pointer to a uint32_t value
- If **sourceCapacityInput** is not 0, **sources** must be a pointer to an array of **sourceCapacityInput** XrPath values
Return Codes

Success
- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_SIZE_INSUFFICIENT
- XR_ERROR_PATH_INVALID
- XR_ERROR_ACTIONSET_NOT_ATTACHED

The `XrBoundSourcesForActionEnumerateInfo` structure is defined as:

typedef struct XrBoundSourcesForActionEnumerateInfo {
    XrStructureType type;
    const void* next;
    XrAction action;
} XrBoundSourcesForActionEnumerateInfo;

Member Descriptions

- **type** is the `XrStructureType` of this structure.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **action** is the handle of the action to query.
Valid Usage (Implicit)

- **type** must be XR_TYPE_BOUND_SOURCES_FOR_ACTION_ENUMERATE_INFO
- **next** must be NULL or a valid pointer to the next structure in a structure chain
- **action** must be a valid XrAction handle

The `xrGetInputSourceLocalizedName` function is defined as:

```c
XrResult xrGetInputSourceLocalizedName(
    XrSession session,
    const XrInputSourceLocalizedNameGetInfo* getInfo,
    uint32_t bufferCapacityInput,
    uint32_t* bufferCountOutput,
    char* buffer);
```

**Parameter Descriptions**

- **session** is a handle to the XrSession associated with the action that reported this source.
- **getInfo** is an XrInputSourceLocalizedNameGetInfo providing the query information.
- **bufferCapacityInput** is the capacity of the buffer, or 0 to indicate a request to retrieve the required capacity.
- **bufferCountOutput** is a pointer to the count of name characters written (including the terminating \0), or a pointer to the required capacity in the case that bufferCapacityInput is insufficient.
- **buffer** is a pointer to an application-allocated buffer that will be filled with the source name. It can be NULL if bufferCapacityInput is 0.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required buffer size.

xrGetInputSourceLocalizedName returns a string for the input source in the current system locale.

If `xrAttachSessionActionSets` has not yet been called for the session, the runtime must return XR_ERROR_ACTIONSET_NOT_ATTACHED.
Valid Usage (Implicit)

- `session` must be a valid `XrSession` handle
- `getInfo` must be a pointer to a valid `XrInputSourceLocalizedNameGetInfo` structure
- `bufferCountOutput` must be a pointer to a `uint32_t` value
- If `bufferCapacityInput` is not 0, `buffer` must be a pointer to an array of `bufferCapacityInput` char values

Return Codes

Success
- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

Failure
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_SIZE_INSUFFICIENT`
- `XR_ERROR_PATH_UNSUPPORTED`
- `XR_ERROR_PATH_INVALID`
- `XR_ERROR_ACTIONSET_NOT_ATTACHED`

The `XrInputSourceLocalizedNameGetInfo` structure is defined as:

```c
typedef struct XrInputSourceLocalizedNameGetInfo {
    XrStructureType                    type;
    const void*                        next;  
    XrPath                             sourcePath;  
    XrInputSourceLocalizedNameFlags    whichComponents; 
} XrInputSourceLocalizedNameGetInfo;
```
Member Descriptions

- **type** is the *XrStructureType* of this structure.
- **next** is **NULL** or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR.
- **source** is an *XrPath* representing the source. Typically this was returned by a call to *xrEnumerateBoundSourcesForAction*.
- **whichComponents** is any set of flags from *XrInputSourceLocalizedNameFlagBits*.

Valid Usage (Implicit)

- **type** **must** be *XR_TYPE_INPUT_SOURCE_LOCALIZED_NAME_GET_INFO*
- **next** **must** be **NULL** or a valid pointer to the next structure in a structure chain
- **whichComponents** **must** be a valid combination of *XrInputSourceLocalizedNameFlagBits* values
- **whichComponents** **must** not be 0

The *xrGetInputSourceLocalizedName::whichComponents* parameter takes bitwise-OR of any of the following values:

```c
// Flag bits for XrInputSourceLocalizedNameFlags
static const XrInputSourceLocalizedNameFlags XR_INPUT_SOURCE_LOCALIZED_NAME_USER_PATH_BIT = 0x00000001;
static const XrInputSourceLocalizedNameFlags XR_INPUT_SOURCE_LOCALIZED_NAME_INTERACTION_PROFILE_BIT = 0x00000002;
static const XrInputSourceLocalizedNameFlags XR_INPUT_SOURCE_LOCALIZED_NAME_COMPONENT_BIT = 0x00000004;
```

Flag Descriptions

- **XR_INPUT_SOURCE_LOCALIZED_NAME_USER_PATH_BIT** indicates that the runtime **must** include the user path portion of the string in the result, if available. E.g. *Left Hand*.
- **XR_INPUT_SOURCE_LOCALIZED_NAME_INTERACTION_PROFILE_BIT** indicates that the runtime **must** include the interaction profile portion of the string in the result, if available. E.g. *Vive Controller*.
- **XR_INPUT_SOURCE_LOCALIZED_NAME_COMPONENT_BIT** indicates that the runtime **must** include the input component portion of the string in the result, if available. E.g. *Trigger*. 
Chapter 12. List of Extensions

- XR_KHR_android_create_instance
- XR_KHR_android_surface_swapchain
- XR_KHR_android_thread_settings
- XR_KHR_binding_modification
- XR_KHR_composition_layer_color_scale_bias
- XR_KHR_composition_layer_cube
- XR_KHR_composition_layer_cylinder
- XR_KHR_composition_layer_depth
- XR_KHR_composition_layer_equirect
- XR_KHR_composition_layer_equirect2
- XR_KHR_convert_timespec_time
- XR_KHR_D3D11_enable
- XR_KHR_D3D12_enable
- XR_KHR_loader_init
- XR_KHR_loader_init_android
- XR_KHR_opengl_enable
- XR_KHR_opengl_es_enable
- XR_KHR_swapchain_usage_input_attachment_bit
- XR_KHR_visibility_mask
- XR_KHR_vulkan_enable
- XR_KHR_vulkan_enable2
- XR_KHR_vulkan_swapchain_format_list
- XR_KHR_win32_convert_performance_counter_time
12.1. XR_KHR_android_create_instance

Name String
XR_KHR_android_create_instance

Extension Type
Instance extension

Registered Extension Number
9

Revision
3

Extension and Version Dependencies
• Requires OpenXR 1.0

Last Modified Date
2019-07-17

IP Status
No known IP claims.

Contributors
Robert Menzel, NVIDIA
Martin Renschler, Qualcomm
Krzysztof Kosiński, Google

Overview
When the application creates an XrInstance object on Android systems, additional information from the application has to be provided to the XR runtime.

The Android XR runtime must return error XR_ERROR_VALIDATION_FAILURE if the additional information is not provided by the application or if the additional parameters are invalid.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

• XR_TYPE_INSTANCE_CREATE_INFO_ANDROID_KHR
New Enums

New Structures

The `XrInstanceCreateInfoAndroidKHR` structure is defined as:

```c
typedef struct XrInstanceCreateInfoAndroidKHR {
    XrStructureType type;
    const void* next;
    void* applicationVM;
    void* applicationActivity;
} XrInstanceCreateInfoAndroidKHR;
```

**Member Descriptions**

- **type** is the `XrStructureType` of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **applicationVM** is a pointer to the JNI’s opaque `JavaVM` structure, cast to a void pointer.
- **applicationActivity** is a JNI reference to an `android.app.Activity` that will drive the session lifecycle of this instance, cast to a void pointer.

`XrInstanceCreateInfoAndroidKHR` contains additional Android specific information needed when calling `xrCreateInstance`. The `applicationVM` field should be populated with the `JavaVM` structure received by the `JNI_OnLoad` function, while the `applicationActivity` field will typically contain a reference to a Java activity object received through an application-specific native method. The `XrInstanceCreateInfoAndroidKHR` structure must be provided in the `next` chain of the `XrInstanceCreateInfo` structure when calling `xrCreateInstance`.

**Valid Usage (Implicit)**

- The `XR_KHR_android_create_instance` extension must be enabled prior to using `XrInstanceCreateInfoAndroidKHR`
- **type** must be `XR_TYPE_INSTANCE_CREATE_INFO_ANDROID_KHR`
- **next** must be NULL or a valid pointer to the next structure in a structure chain
- **applicationVM** must be a pointer value
- **applicationActivity** must be a pointer value
New Functions

Issues

Version History

• Revision 1, 2017-05-26 (Robert Menzel)
  ◦ Initial draft
• Revision 2, 2019-01-24 (Martin Renschler)
  ◦ Added error code, reformatted
• Revision 3, 2019-07-17 (Krzysztof Kosiński)
  ◦ Non-substantive clarifications.

12.2. XR_KHR_android_surface_swapchain

Name String

XR_KHR_android_surface_swapchain

Extension Type

Instance extension

Registered Extension Number

5

Revision

4

Extension and Version Dependencies

• Requires OpenXR 1.0

Last Modified Date

2019-05-30

IP Status

No known IP claims.

Contributors

Krzysztof Kosiński, Google
Johannes van Waveren, Oculus
Martin Renschler, Qualcomm

Overview

A common activity in XR is to view an image stream. Image streams are often the result of camera
previews or decoded video streams. On Android, the basic primitive representing the producer end of an image queue is the class `android.view.Surface`. This extension provides a special swapchain that uses an `android.view.Surface` as its producer end.

**New Object Types**

**New Flag Types**

**New Enum Constants**

**New Enums**

**New Structures**

**New Functions**

To create an `XrSwapchain` object and an Android Surface object call:

```c
XrResult xrCreateSwapchainAndroidSurfaceKHR(
    XrSession                                   session,
    const XrSwapchainCreateInfo*                info,
    XrSwapchain*                                swapchain,
    jobject*                                    surface);
```

**Parameter Descriptions**

- `session` is an `XrSession` handle previously created with `xrCreateSession`.
- `info` is a pointer to an `XrSwapchainCreateInfo` structure.
- `swapchain` is a pointer to a handle in which the created `XrSwapchain` is returned.
- `surface` is a pointer to a `jobject` where the created Android Surface is returned.

`xrCreateSwapchainAndroidSurfaceKHR` creates an `XrSwapchain` object returned in `swapchain` and an Android Surface `jobject` returned in `surface`. The `jobject` must be valid to be passed back to Java code using JNI and must be valid to be used with ordinary Android APIs for submitting images to Surfaces. The returned `XrSwapchain` must be valid to be referenced in `XrSwapchainSubImage` structures to show content on the screen. The width and height passed in `XrSwapchainCreateInfo` may not be persistent throughout the life cycle of the created swapchain, since on Android, the size of the images is controlled by the producer and possibly changes at any time.

The only function that is allowed to be called on the `XrSwapchain` returned from this function is `xrDestroySwapchain`. For example, calling any of the functions `xrEnumerateSwapchainImages`, `xrAcquireSwapchainImage`, `xrWaitSwapchainImage` or `xrReleaseSwapchainImage` is invalid.
When the application receives the `XrEventDataSessionStateChanged` event with the `XR_SESSION_STATE_STOPPING` state, it **must** ensure that no threads are writing to any of the Android surfaces created with this extension before calling `xrEndSession`. The effect of writing frames to the Surface when the session is in states other than `XR_SESSION_STATE_VISIBLE` or `XR_SESSION_STATE_FOCUSED` is undefined.

`xrCreateSwapchainAndroidSurfaceKHR` **must** return the same set of error codes as `xrCreateSwapchain` under the same circumstances, plus `XR_ERROR_FUNCTION_UNSUPPORTED` in case the function is not supported.

### Valid Usage of `XrSwapchainCreateInfo` members

- The `XrSwapchainCreateInfo::format`, `XrSwapchainCreateInfo::sampleCount`, `XrSwapchainCreateInfo::faceCount`, `XrSwapchainCreateInfo::arraySize` and `XrSwapchainCreateInfo::mipCount` members of the structure passed as the `info` parameter **must** be zero.

### Valid Usage (Implicit)

- The `XR_KHR_android_surface_swapchain` extension **must** be enabled prior to calling `xrCreateSwapchainAndroidSurfaceKHR`
- `session` **must** be a valid `XrSession` handle
- `info` **must** be a pointer to a valid `XrSwapchainCreateInfo` structure
- `swapchain` **must** be a pointer to an `XrSwapchain` handle
- `surface` **must** be a pointer to a `jobject` value
Return Codes

Success
- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

Failure
- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_OUT_OF_MEMORY
- XR_ERROR_LIMIT_REACHED

Issues

Version History
- Revision 1, 2017-01-17 (Johannes van Waveren)
  ◦ Initial draft
- Revision 2, 2017-10-30 (Kaye Mason)
  ◦ Changed images to swapchains, used snippet includes. Added issue for Surfaces.
- Revision 3, 2018-05-16 (Krzysztof Kosiński)
  ◦ Refactored to use Surface instead of SurfaceTexture.
- Revision 4, 2019-01-24 (Martin Renschler)
  ◦ Refined the specification of the extension

12.3. XR_KHR_android_thread_settings

Name String
XR_KHR_android_thread_settings

Extension Type
Instance extension
Overview

For XR to be comfortable, it is important for applications to deliver frames quickly and consistently. In order to make sure the important application threads get their full share of time, these threads must be identified to the system, which will adjust their scheduling priority accordingly.

New Object Types

New Flag Types

New Enum Constants

$\text{XrResult}$ enumeration is extended with:

- $\text{XR_ERROR_ANDROID_THREAD_SETTINGS_ID_INVALID_KHR}$
- $\text{XR_ERROR_ANDROID_THREAD_SETTINGS_FAILURE_KHR}$

New Enums

The possible thread types are specified by the $\text{XrAndroidThreadTypeKHR}$ enumeration:
typedef enum XrAndroidThreadTypeKHR {
    XR_ANDROID_THREAD_TYPE_APPLICATION_MAIN_KHR = 1,
    XR_ANDROID_THREAD_TYPE_APPLICATION_WORKER_KHR = 2,
    XR_ANDROID_THREAD_TYPE_RENDERER_MAIN_KHR = 3,
    XR_ANDROID_THREAD_TYPE_RENDERER_WORKER_KHR = 4,
    XR_ANDROID_THREAD_TYPE_MAX_ENUM_KHR = 0x7FFFFFFF
} XrAndroidThreadTypeKHR;

**Enumerants**

- **XR_ANDROID_THREAD_TYPE_APPLICATION_MAIN_KHR**
  hints the XR runtime that the thread is doing background CPU tasks

- **XR_ANDROID_THREAD_TYPE_APPLICATION_WORKER_KHR**
  hints the XR runtime that the thread is doing time critical CPU tasks

- **XR_ANDROID_THREAD_TYPE_RENDERER_MAIN_KHR**
  hints the XR runtime that the thread is doing background graphics device tasks

- **XR_ANDROID_THREAD_TYPE_RENDERER_WORKER_KHR**
  hints the XR runtime that the thread is doing time critical graphics device tasks

**New Structures**

**New Functions**

To declare a thread to be of a certain `XrAndroidThreadTypeKHR` type call:

```c
XrResult xrSetAndroidApplicationThreadKHR(
    XrSession session,
    XrAndroidThreadTypeKHR threadType,
    uint32_t threadId);
```
Parameter Descriptions

- **session** is a valid XrSession handle.

- **threadType** is a classification of the declared thread allowing the XR runtime to apply the relevant priority and attributes. If such settings fail, the runtime **must** return XR_ERROR_ANDROID_THREAD_SETTINGS_FAILURE_KHR.

- **threadId** is the kernel thread ID of the declared thread, as returned by gettid() or android.os.process.myTid(). If the thread ID is invalid, the runtime **must** return XR_ERROR_ANDROID_THREAD_SETTINGS_ID_INVALID_KHR.

xrSetAndroidApplicationThreadKHR allows to declare an XR-critical thread and to classify it.

Valid Usage (Implicit)

- The XR_KHR_android_thread_settings extension **must** be enabled prior to calling xrSetAndroidApplicationThreadKHR
- **session** **must** be a valid XrSession handle
- **threadType** **must** be a valid XrAndroidThreadTypeKHR value

Return Codes

**Success**

- XR_SUCCESS
- XR_SESSION_LOSS_PENDING

**Failure**

- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SESSION_LOST
- XR_ERROR_ANDROID_THREAD_SETTINGS_ID_INVALID_KHR
- XR_ERROR_ANDROID_THREAD_SETTINGS_FAILURE_KHR

Version History
• Revision 1, 2017-01-17 (Johannes van Waveren)
  ◦ Initial draft.
• Revision 2, 2017-10-31 (Armelle Laine)
  ◦ Move the performance settings to EXT extension.
• Revision 3, 2018-12-20 (Paul Pedriana)
  ◦ Revised the error code naming to use KHR and renamed xrSetApplicationThreadKHR → xrSetAndroidApplicationThreadKHR.
• Revision 4, 2019-01-24 (Martin Renschler)
  ◦ Added enum specification, reformatting
• Revision 5, 2019-07-17 (Krzysztof Kosiński)
  ◦ Clarify the type of thread identifier used by the extension.

12.4. XR_KHR_binding_modification

Name String
  XR_KHR_binding_modification

Extension Type
  Instance extension

Registered Extension Number
  121

Revision
  1

Extension and Version Dependencies
  • Requires OpenXR 1.0

Last Modified Date
  2020-07-29

IP Status
  No known IP claims.

Contributors
  Joe Ludwig, Valve

Contacts
  Joe Ludwig, Valve
Overview

This extension adds an optional structure that can be included on the `XrInteractionProfileSuggestedBinding::next` chain passed to `xrSuggestInteractionProfileBindings` to specify additional information to modify default binding behavior.

This extension does not define any actual modification structs, but includes the list of modifications and the `XrBindingModificationBaseHeaderKHR` structure to allow other extensions to provide specific modifications.

New Object Types

New Flag Types

New Enum Constants

The `XrStructureType` enumeration is extended with:

- XR_TYPE_BINDING_MODIFICATIONS_KHR

New Enums

New Structures

The `XrBindingModificationsKHR` structure is defined as:

```c
typedef struct XrBindingModificationsKHR {
    XrStructureType                                     type;
    const void*                                         next;
    uint32_t                                            bindingModificationCount;
    const XrBindingModificationBaseHeaderKHR* const*    bindingModifications;
} XrBindingModificationsKHR;
```

Member Descriptions

- `type` is the `XrStructureType` of this structure.
- `next` is NULL or a pointer to the next structure in a structure chain.
- `bindingModificationCount` is the number of binding modifications in the array pointed to by `bindingModifications`.
- `bindingModifications` is a pointer to an array of pointers to binding modification structures based on `XrBindingModificationBaseHeaderKHR`, that define all of the application's suggested binding modifications for the specified interaction profile.
Valid Usage (Implicit)

• The XR_KHR_binding_modification extension must be enabled prior to using XrBindingModificationsKHR

• type must be XR_TYPE_BINDING_MODIFICATIONS_KHR

• next must be NULL or a valid pointer to the next structure in a structure chain

• If bindingModificationCount is not 0, bindingModifications must be a pointer to an array of bindingModificationCount valid XrBindingModificationBaseHeaderKHR-based structures

The XrBindingModificationBaseHeaderKHR structure is defined as:

```
typedef struct XrBindingModificationBaseHeaderKHR {
    XrStructureType type;
    const void* next;
} XrBindingModificationBaseHeaderKHR;
```

Member Descriptions

• type is the XrStructureType of this structure. This base structure itself has no associated XrStructureType value.

• next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or in this extension.

The XrBindingModificationBaseHeaderKHR is a base structure is overridden by XrBindingModification* child structures.

Valid Usage (Implicit)

• The XR_KHR_binding_modification extension must be enabled prior to using XrBindingModificationBaseHeaderKHR

• next must be NULL or a valid pointer to the next structure in a structure chain

New Functions

Issues

Version History
12.5. XR_KHR_composition_layer_color_scale_bias

Name String

XR_KHR_composition_layer_color_scale_bias

Extension Type

Instance extension

Registered Extension Number

35

Revision

5

Extension and Version Dependencies

• Requires OpenXR 1.0

Last Modified Date

2019-01-28

IP Status

No known IP claims.

Contributors

Paul Pedriana, Oculus
Cass Everitt, Oculus
Martin Renschler, Qualcomm

Overview

Color scale and bias are applied to a layer color during composition, after its conversion to premultiplied alpha representation.

If specified, colorScale and colorBias must be used to alter the LayerColor as follows:

• colorScale = max( vec4( 0, 0, 0, 0 ), colorScale )
• LayerColor.RGB = LayerColor.A > 0 ? LayerColor.RGB / LayerColor.A : vec3( 0, 0, 0 )
• LayerColor = LayerColor * colorScale + colorBias
• LayerColor.RGB *= LayerColor.A

This extension specifies the XrCompositionLayerColorScaleBiasKHR structure, which, if present in the
XrCompositionLayerBaseHeader::next chain, **must** be applied to the composition layer.

This extension does not define a new composition layer type, but rather it defines a transform that may be applied to the color derived from existing composition layer types.

**New Object Types**

**New Flag Types**

**New Enum Constants**

XrStructureType enumeration is extended with:

- XR_TYPE_COMPOSITION_LAYER_COLOR_SCALE_BIAS_KHR

**New Enums**

**New Structures**

The **XrCompositionLayerColorScaleBiasKHR** structure is defined as:

```c
typedef struct XrCompositionLayerColorScaleBiasKHR {
    XrStructureType    type;
    const void*        next;
    XrColor4f          colorScale;
    XrColor4f          colorBias;
} XrCompositionLayerColorScaleBiasKHR;
```

**Member Descriptions**

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **colorScale** is an XrColor4f which will modulate the color sourced from the images.
- **colorBias** is an XrColor4f which will offset the color sourced from the images.

XrCompositionLayerColorScaleBiasKHR contains the information needed to scale and bias the color of layer textures.

The XrCompositionLayerColorScaleBiasKHR structure **can** be applied by applications to composition layers by adding an instance of the struct to the XrCompositionLayerBaseHeader::next list.
Valid Usage (Implicit)

- The `XR_KHR_composition_layer_color_scale_bias` extension must be enabled prior to using `XrCompositionLayerColorScaleBiasKHR`.
- `type` must be `XR_TYPE_COMPOSITION_LAYER_COLOR_SCALE_BIAS_KHR`.
- `next` must be `NULL` or a valid pointer to the next structure in a structure chain.

New Functions

Issues

Version History

- Revision 1, 2017-09-13 (Paul Pedriana)
  - Initial implementation.
- Revision 2, 2019-01-24 (Martin Renschler)
  - Formatting, spec language changes
- Revision 3, 2019-01-28 (Paul Pedriana)
  - Revised math to remove premultiplied alpha before applying color scale and offset, then restoring.
- Revision 4, 2019-07-17 (Cass Everitt)
  - Non-substantive updates to the spec language and equations.
- Revision 5, 2020-05-20 (Cass Everitt)
  - Changed extension name, simplified language.

12.6. `XR_KHR_composition_layer_cube`

Name String

- `XR_KHR_composition_layer_cube`

Extension Type

- Instance extension

Registered Extension Number

- 7

Revision

- 8
Extension and Version Dependencies

- Requires OpenXR 1.0

Last Modified Date

2019-01-24

IP Status

No known IP claims.

Contributors

Johannes van Waveren, Oculus
Cass Everitt, Oculus
Paul Pedriana, Oculus
Gloria Kennickell, Oculus
Sam Martin, ARM
Kaye Mason, Google, Inc.
Martin Renschler, Qualcomm

Contacts

Cass Everitt, Oculus
Paul Pedriana, Oculus

Overview

This extension adds an additional layer type that enables direct sampling from cubemaps.

The cube layer is the natural layer type for hardware accelerated environment maps. Without updating the image source, the user can look all around, and the compositor can display what they are looking at without intervention from the application.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- XR_TYPE_COMPOSITION_LAYER_CUBE_KHR

New Enums

New Structures

The XrCompositionLayerCubeKHR structure is defined as:
```c
typedef struct XrCompositionLayerCubeKHR {
    XrStructureType            type;
    const void*                next;
    XrCompositionLayerFlags    layerFlags;
    XrSpace                    space;
    XrEyeVisibility            eyeVisibility;
    XrSwapchain                swapchain;
    uint32_t                   imageArrayIndex;
    XrQuaternionf              orientation;
} XrCompositionLayerCubeKHR;
```

### Member Descriptions

- **type** is the `XrStructureType` of this structure.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **layerFlags** is any flags to apply to this layer.
- **space** is the `XrSpace` in which the **orientation** of the cube layer is evaluated over time.
- **eye** is the eye represented by this layer.
- **swapchain** is the swapchain.
- **imageArrayIndex** is the image array index, with 0 meaning the first or only array element.
- **orientation** is the orientation of the environment map in the **space**.

**XrCompositionLayerCubeKHR** contains the information needed to render a cube map when calling `xrEndFrame`. **XrCompositionLayerCubeKHR** is an alias type for the base struct `XrCompositionLayerBaseHeader` used in `XrFrameEndInfo`. 
Valid Usage (Implicit)

- The `XR_KHR_composition_layer_cube` extension must be enabled prior to using `XrCompositionLayerCubeKHR`.
- `type` must be `XR_TYPE_COMPOSITION_LAYER_CUBE_KHR`.
- `next` must be `NULL` or a valid pointer to the next structure in a structure chain.
- `layerFlags` must be `0` or a valid combination of `XrCompositionLayerFlagBits` values.
- `space` must be a valid `XrSpace` handle.
- `eyeVisibility` must be a valid `XrEyeVisibility` value.
- `swapchain` must be a valid `XrSwapchain` handle.
- Both of `space` and `swapchain` must have been created, allocated, or retrieved from the same `XrSession`.

New Functions

Issues

Version History

- Revision 0, 2017-02-01 (Johannes van Waveren)
  - Initial draft.
- Revision 1, 2017-05-19 (Sam Martin)
  - Initial draft, moving the 3 layer types to an extension.
- Revision 2, 2017-08-30 (Paul Pedriana)
  - Updated the specification.
- Revision 3, 2017-10-12 (Cass Everitt)
  - Updated to reflect per-eye structs and the change to swapchains.
- Revision 4, 2017-10-18 (Kaye Mason)
  - Update to flatten structs to remove per-eye arrays.
- Revision 5, 2017-12-05 (Paul Pedriana)
  - Updated to break out the cylinder and equirect features into separate extensions.
- Revision 6, 2017-12-07 (Paul Pedriana)
  - Updated to use transform components instead of transform matrices.
- Revision 7, 2017-12-07 (Paul Pedriana)
  - Updated to convert `XrPosef` to `XrQuaternionf` (there’s no position component).
12.7. XR_KHR_composition_layer_cylinder

Name String
XR_KHR_composition_layer_cylinder

Extension Type
Instance extension

Registered Extension Number
18

Revision
4

Extension and Version Dependencies
• Requires OpenXR 1.0

Last Modified Date
2019-01-24

IP Status
No known IP claims.

Contributors
James Hughes, Oculus
Paul Pedriana, Oculus
Martin Renschler, Qualcomm

Contacts
Paul Pedriana, Oculus
Cass Everitt, Oculus

Overview
This extension adds an additional layer type where the XR runtime must map a texture stemming from a swapchain onto the inside of a cylinder section. It can be imagined much the same way a curved television display looks to a viewer. This is not a projection type of layer but rather an object-in-world type of layer, similar to XrCompositionLayerQuad. Only the interior of the cylinder surface must be visible; the exterior of the cylinder is not visible and must not be drawn by the runtime.

The cylinder characteristics are specified by the following parameters:
These can be understood via the following diagram, which is a top-down view of a horizontally oriented cylinder. The aspect ratio drives how tall the cylinder will appear based on the other parameters. Typically the `aspectRatio` would be set to be the aspect ratio of the texture being used, so that it looks the same within the cylinder as it does in 2D.

### Figure 4. Cylinder Layer Parameters

- **r** — Radius
- **a** — Central angle in (0, 2\(\pi\))
- **p** — Origin of pose transform
- **U/V** — UV coordinates

### New Object Types

### New Flag Types

### New Enum Constants

**XrStructureType** enumeration is extended with:

- **XR_TYPE_COMPOSITION_LAYER_CYLINDER_KHR**

### New Enums
New Structures

The XrCompositionLayerCylinderKHR structure is defined as:

```c
typedef struct XrCompositionLayerCylinderKHR {
    XrStructureType            type;
    const void*                next;
    XrCompositionLayerFlags    layerFlags;
    XrSpace                    space;
    XrEyeVisibility            eyeVisibility;
    XrSwapchainSubImage        subImage;
    XrPosef                    pose;
    float                      radius;
    float                      centralAngle;
    float                      aspectRatio;
} XrCompositionLayerCylinderKHR;
```

Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **layerFlags** specifies options for the layer.
- **space** is the XrSpace in which the pose of the cylinder layer is evaluated over time.
- **eye** is the eye represented by this layer.
- **subImage** identifies the image XrSwapchainSubImage to use.
- **pose** is an XrPosef defining the position and orientation of the center point of the view of the cylinder within the reference frame of the space.
- **radius** is the non-negative radius of the cylinder. Values of zero or floating point positive infinity are treated as an infinite cylinder.
- **centralAngle** is the angle of the visible section of the cylinder, based at 0 radians, in the range of [0, 2\pi). It grows symmetrically around the 0 radian angle.
- **aspectRatio** is the ratio of the visible cylinder section width / height. The height of the cylinder is given by: (cylinder radius \times \text{cylinder angle}) / \text{aspectRatio}.

XrCompositionLayerCylinderKHR contains the information needed to render a texture onto a cylinder when calling xrEndFrame. XrCompositionLayerCylinderKHR is an alias type for the base struct XrCompositionLayerBaseHeader used in XrFrameEndInfo.
Valid Usage (Implicit)

- The `XR_KHR_composition_layer_cylinder` extension **must** be enabled prior to using `XrCompositionLayerCylinderKHR`.
- `type` **must** be `XR_TYPE_COMPOSITION_LAYER_CYLINDER_KHR`.
- `next` **must** be `NULL` or a valid pointer to the `next` structure in a structure chain.
- `layerFlags` **must** be `0` or a valid combination of `XrCompositionLayerFlagBits` values.
- `space` **must** be a valid `XrSpace` handle.
- `eyeVisibility` **must** be a valid `XrEyeVisibility` value.
- `subImage` **must** be a valid `XrSwapchainSubImage` structure.

**New Functions**

**Issues**

**Version History**

- Revision 1, 2017-05-19 (Paul Pedriana)
  - Initial version. This was originally part of a single extension which supported multiple such extension layer types.
- Revision 2, 2017-12-07 (Paul Pedriana)
  - Updated to use transform components instead of transform matrices.
- Revision 3, 2018-03-05 (Paul Pedriana)
  - Added improved documentation and brought the documentation in line with the existing core spec.
- Revision 4, 2019-01-24 (Martin Renschler)
  - Reformatted, spec language changes, eye parameter description update

### 12.8. XR_KHR_composition_layer_depth

**Name String**

`XR_KHR_composition_layer_depth`

**Extension Type**

- Instance extension

**Registered Extension Number**

11
Revision
6

Extension and Version Dependencies
- Requires OpenXR 1.0

Last Modified Date
2019-01-24

IP Status
No known IP claims.

Contributors
Paul Pedriana, Oculus
Bryce Hutchings, Microsoft
Andreas Loeve Selvik, Arm
Martin Renschler, Qualcomm

Overview
This extension defines an extra layer type which allows applications to submit depth images along with color images in projection layers, i.e. **XrCompositionLayerProjection**.

The XR runtime may use this information to perform more accurate reprojections taking depth into account. Use of this extension does not affect the order of layer composition as described in Compositing.

New Object Types

New Flag Types

New Enum Constants

**XrStructureType** enumeration is extended with:

- **XR_TYPE_COMPOSITION_LAYER_DEPTH_INFO_KHR**

New Enums

New Structures

When submitting depth images along with projection layers, add the **XrCompositionLayerDepthInfoKHR** to the next chain for all **XrCompositionLayerProjectionView** structures in the given layer.

The **XrCompositionLayerDepthInfoKHR** structure is defined as:
typedef struct XrCompositionLayerDepthInfoKHR {
    XrStructureType        type;
    const void*            next;
    XrSwapchainSubImage    subImage;
    float                  minDepth;
    float                  maxDepth;
    float                  nearZ;
    float                  farZ;
} XrCompositionLayerDepthInfoKHR;

**Member Descriptions**

- **type** is the XrStructureType of this structure.

- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.

- **subImage** identifies the depth image XrSwapchainSubImage to be associated with the color swapchain.

- **minDepth** and **maxDepth** are the window space depths that correspond to the near and far frustum planes, respectively. **minDepth** must be less than **maxDepth**. **minDepth** and **maxDepth** must be in the range [0, 1].

- **nearZ** and **farZ** are the positive distances in meters to the near and far frustum planes, respectively. **nearZ** and **farZ** must not be equal. **nearZ** and **farZ** must be in the range (0, +infinity].

**Note**
The window space depth values **minDepth** and **maxDepth** are akin to the parameters of glDepthRange that specify the mapping from normalized device coordinates into window space.

**Note**
A reversed mapping of depth, such that points closer to the view have a window space depth that is greater than points further away can be achieved by making nearZ > farZ.

XrCompositionLayerDepthInfoKHR contains the information needed to associate depth with the color information in a projection layer. When submitting depth images along with projection layers, add the XrCompositionLayerDepthInfoKHR to the next chain for all XrCompositionLayerProjectionView structures in the given layer.

The homogeneous transform from view space z to window space depth is given by the following matrix, where a = minDepth, b = maxDepth, n = nearZ, and f = farZ.
\[\mathbf{T} = \begin{bmatrix} b - a & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} -\frac{f}{f-n} & -\frac{fn}{f-n} \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} -\frac{bf-an}{f-n} & -\frac{fn(b-a)}{f-n} \\ -1 & 0 \end{bmatrix}\]

\[\mathbf{p}_w = \mathbf{T}\mathbf{p}_v\]

\[\mathbf{p}_w = [z_w \quad w_w]^t, \text{homogeneous window space depth}\]

\[\mathbf{p}_v = [z_v \quad w_v]^t, \text{homogeneous view space depth}\]

Figure 5. Homogeneous transform from view space to window space depth

Homogeneous values are constructed from real values by appending a \(w\) component with value 1.0.

General homogeneous values are projected back to real space by dividing by the \(w\) component.

**Valid Usage (Implicit)**

- The XR_KHR_composition_layer_depth extension must be enabled prior to using XrCompositionLayerDepthInfoKHR
- type must be XR_TYPE_COMPOSITION_LAYER_DEPTH_INFO_KHR
- next must be NULL or a valid pointer to the next structure in a structure chain
- subImage must be a valid XrSwapchainSubImage structure

**New Functions**

**Issues**

1. Should the range of minDepth and maxDepth be constrained to [0,1]?

   **RESOLVED:** Yes.

   There is no compelling mathematical reason for this constraint, however, it does not impose any hardship currently, and the constraint could be relaxed in a future version of the extension if needed.

2. Should we require minDepth be less than maxDepth?

   **RESOLVED:** Yes.

   There is no compelling mathematical reason for this constraint, however, it does not impose any
hardship currently, and the constraint could be relaxed in a future version of the extension if needed. Reverse z mappings can be achieved by making $\text{near}Z > \text{far}Z$.

3. Does this extension support view space depth images?

**RESOLVED:** No.

The formulation of the transform between view and window depths implies projected depth. A different extension would be needed to support a different interpretation of depth.

4. Is there any constraint on the resolution of the depth subimage?

**RESOLVED:** No.

The resolution of the depth image need not match that of the corresponding color image.

**Version History**

- Revision 1, 2017-08-18 (Paul Pedriana)
  - Initial proposal.
- Revision 2, 2017-10-30 (Kaye Mason)
  - Migration from Images to Swapchains.
- Revision 3, 2018-07-20 (Bryce Hutchings)
  - Support for swapchain texture arrays
- Revision 4, 2018-12-17 (Andreas Loeve Selvik)
  - depthImageRect in pixels instead of UVs
- Revision 5, 2019-01-24 (Martin Renschler)
  - changed depthSwapchain/depthImageRect/depthImageArrayIndex to XrSwapchainSubImage
  - reformat and spec language changes
  - removed vendor specific terminology
- Revision 6, 2022-02-16 (Cass Everitt)
  - Provide homogeneous transform as function of provided parameters

**12.9. XR_KHR_composition_layer_equirect**

**Name String**

XR_KHR_composition_layer_equirect

**Extension Type**

Instance extension
Registered Extension Number
19

Revision
3

Extension and Version Dependencies
• Requires OpenXR 1.0

Last Modified Date
2019-01-24

IP Status
No known IP claims.

Contributors
Johannes van Waveren, Oculus
Cass Everitt, Oculus
Paul Pedriana, Oculus
Gloria Kennickell, Oculus
Martin Renschler, Qualcomm

Contacts
Cass Everitt, Oculus
Paul Pedriana, Oculus

Overview
This extension adds an additional layer type where the XR runtime must map an equirectangular coded image stemming from a swapchain onto the inside of a sphere.

The equirect layer type provides most of the same benefits as a cubemap, but from an equirect 2D image source. This image source is appealing mostly because equirect environment maps are very common, and the highest quality you can get from them is by sampling them directly in the compositor.

This is not a projection type of layer but rather an object-in-world type of layer, similar to XrCompositionLayerQuad. Only the interior of the sphere surface must be visible; the exterior of the sphere is not visible and must not be drawn by the runtime.

New Object Types

New Flag Types

New Enum Constants
XrStructureType enumeration is extended with:

- XR_TYPE_COMPOSITION_LAYER_EQUIRECT_KHR

New Enums

New Structures

The XrCompositionLayerEquirectKHR structure is defined as:

typedef struct XrCompositionLayerEquirectKHR {
    XrStructureType            type;
    const void*                next;
    XrCompositionLayerFlags    layerFlags;
    XrSpace                    space;
    XrEyeVisibility            eyeVisibility;
    XrSwapchainSubImage        subImage;
    XrPosef                    pose;
    float                      radius;
    XrVector2f                 scale;
    XrVector2f                 bias;
} XrCompositionLayerEquirectKHR;

Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **layerFlags** specifies options for the layer.
- **space** is the XrSpace in which the pose of the equirect layer is evaluated over time.
- **eye** is the eye represented by this layer.
- **subImage** identifies the image XrSwapchainSubImage to use.
- **pose** is an XrPosef defining the position and orientation of the center point of the sphere onto which the equirect image data is mapped, relative to the reference frame of the space.
- **radius** is the non-negative radius of the sphere onto which the equirect image data is mapped. Values of zero or floating point positive infinity are treated as an infinite sphere.
- **scale** is an XrVector2f indicating a scale of the texture coordinates after the mapping to 2D.
- **bias** is an XrVector2f indicating a bias of the texture coordinates after the mapping to 2D.
XrCompositionLayerEquirectKHR contains the information needed to render an equirectangular image onto a sphere when calling xrEndFrame. XrCompositionLayerEquirectKHR is an alias type for the base struct XrCompositionLayerBaseHeader used in XrFrameEndInfo.

### Valid Usage (Implicit)

- The XR_KHR_composition_layer_equirect extension must be enabled prior to using XrCompositionLayerEquirectKHR
- type must be XR_TYPE_COMPOSITION_LAYER_EQUIRECT_KHR
- next must be NULL or a valid pointer to the next structure in a structure chain
- layerFlags must be 0 or a valid combination of XrCompositionLayerFlagBits values
- space must be a valid XrSpace handle
- eyeVisibility must be a valid XrEyeVisibility value
- subImage must be a valid XrSwapchainSubImage structure

### New Functions

### Issues

### Version History

- Revision 1, 2017-05-19 (Paul Pedriana)
  - Initial version. This was originally part of a single extension which supported multiple such extension layer types.
- Revision 2, 2017-12-07 (Paul Pedriana)
  - Updated to use transform components instead of transform matrices.
- Revision 3, 2019-01-24 (Martin Renschler)
  - Reformatted, spec language changes, eye parameter description update

### 12.10. XR_KHR_composition_layer_equirect2

#### Name String

XR_KHR_composition_layer_equirect2

#### Extension Type

- Instance extension

#### Registered Extension Number

92
Overview

This extension adds an additional layer type where the XR runtime must map an equirectangular coded image stemming from a swapchain onto the inside of a sphere.

The equirect layer type provides most of the same benefits as a cubemap, but from an equirect 2D image source. This image source is appealing mostly because equirect environment maps are very common, and the highest quality you can get from them is by sampling them directly in the compositor.

This is not a projection type of layer but rather an object-in-world type of layer, similar to XrCompositionLayerQuad. Only the interior of the sphere surface must be visible; the exterior of the sphere is not visible and must not be drawn by the runtime.

This extension uses a different parameterization more in keeping with the formulation of KHR_composition_layer_cylinder but is functionally equivalent to KHR_composition_layer_equirect.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:
New Enums

New Structures

The XrCompositionLayerEquirect2KHR structure is defined as:

typedef struct XrCompositionLayerEquirect2KHR {
    XrStructureType        type;
    const void*            next;
    XrCompositionLayerFlags layerFlags;
    XrSpace                space;
    XrEyeVisibility        eyeVisibility;
    XrSwapchainSubImage    subImage;
    XrPosef                pose;
    float                  radius;
    float                  centralHorizontalAngle;
    float                  upperVerticalAngle;
    float                  lowerVerticalAngle;
} XrCompositionLayerEquirect2KHR;
Member Descriptions

• **type** is the `XrStructureType` of this structure.

• **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.

• **layerFlags** specifies options for the layer.

• **space** is the `XrSpace` in which the **pose** of the equirect layer is evaluated over time.

• **eye** is the eye represented by this layer.

• **subImage** identifies the image `XrSwapchainSubImage` to use.

• **pose** is an `XrPosef` defining the position and orientation of the center point of the sphere onto which the equirect image data is mapped, relative to the reference frame of the **space**.

• **radius** is the non-negative radius of the sphere onto which the equirect image data is mapped. Values of zero or floating point positive infinity are treated as an infinite sphere.

• **centralHorizontalAngle** defines the visible horizontal angle of the sphere, based at 0 radians, in the range of [0, 2π]. It grows symmetrically around the 0 radian angle.

• **upperVerticalAngle** defines the upper vertical angle of the visible portion of the sphere, in the range of [-π/2, π/2].

• **lowerVerticalAngle** defines the lower vertical angle of the visible portion of the sphere, in the range of [-π/2, π/2].

`XrCompositionLayerEquirect2KHR` contains the information needed to render an equirectangular image onto a sphere when calling `xrEndFrame`. `XrCompositionLayerEquirect2KHR` is an alias type for the base struct `XrCompositionLayerBaseHeader` used in `XrFrameEndInfo`.

Valid Usage (Implicit)

• The `XR_KHR_composition_layer_equirect2` extension **must** be enabled prior to using `XrCompositionLayerEquirect2KHR`

• **type** **must** be `XR_TYPE_COMPOSITION_LAYER_EQUIRECT2_KHR`

• **next** **must** be `NULL` or a valid pointer to the **next** structure in a structure chain

• **layerFlags** **must** be 0 or a valid combination of `XrCompositionLayerFlagBits` values

• **space** **must** be a valid `XrSpace` handle

• **eyeVisibility** **must** be a valid `XrEyeVisibility` value

• **subImage** **must** be a valid `XrSwapchainSubImage` structure

New Functions
Overview

This extension provides two functions for converting between timespec monotonic time and \texttt{XrTime}. The \texttt{xrConvertTimespecTimeToTimeKHR} function converts from timespec time to \texttt{XrTime}, while the \texttt{xrConvertTimeToTimespecTimeKHR} function converts \texttt{XrTime} to timespec monotonic time. The primary use case for this functionality is to be able to synchronize events between the local system and the OpenXR system.
To convert from timespec monotonic time to \textit{XrTime}, call:

\begin{verbatim}
XrResult xrConvertTimespecTimeToTimeKHR(
    XrInstance instance,
    const struct timespec* timespecTime,
    XrTime* time);
\end{verbatim}

\textbf{Parameter Descriptions}

- \textit{instance} is an \textit{XrInstance} handle previously created with \textit{xrCreateInstance}.
- \textit{timespecTime} is a \textit{timespec} obtained from \textit{clock_gettime} with \textit{CLOCK_MONOTONIC}.
- \textit{time} is the resulting \textit{XrTime} that is equivalent to the \textit{timespecTime}.

The \textit{xrConvertTimespecTimeToTimeKHR} function converts a time obtained by the \textit{clock_gettime} function to the equivalent \textit{XrTime}.

If the output \textit{time} cannot represent the input \textit{timespecTime}, the runtime \textbf{must} return \textit{XR_ERROR_TIME_INVALID}.

\textbf{Valid Usage (Implicit)}

- The \textit{XR_KHR_convert_timespec_time} extension \textbf{must} be enabled prior to calling \textit{xrConvertTimespecTimeToTimeKHR}
- \textit{instance} \textbf{must} be a valid \textit{XrInstance} handle
- \textit{timespecTime} \textbf{must} be a pointer to a valid \textit{timespec} value
- \textit{time} \textbf{must} be a pointer to an \textit{XrTime} value
Return Codes

Success
- XR_SUCCESS

Failure
- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_TIME_INVALID

To convert from XrTime to timespec monotonic time, call:

```c
XrResult xrConvertTimeToTimespecTimeKHR(
    XrInstance instance,
    XrTime time,
    struct timespec* timespecTime);
```

Parameter Descriptions

- `instance` is an XrInstance handle previously created with xrCreateInstance.
- `time` is an XrTime.
- `timespecTime` is the resulting timespec time that is equivalent to a timespec obtained from clock_gettime with CLOCK_MONOTONIC.

The xrConvertTimeToTimespecTimeKHR function converts an XrTime to time as if generated by clock_gettime.

If the output timespecTime cannot represent the input time, the runtime must return XR_ERROR_TIME_INVALID.
Valid Usage (Implicit)

- The `XR_KHR_convert_timespec_time` extension must be enabled prior to calling `xrConvertTimeToTimespecTimeKHR`.
- `instance` must be a valid `XrInstance` handle.
- `timespecTime` must be a pointer to a `timespec` value.

Return Codes

**Success**
- `XR_SUCCESS`

**Failure**
- `XR_ERROR_FUNCTION_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_TIME_INVALID`

Issues

Version History

- Revision 1, 2019-01-24 (Paul Pedriana)
  - Initial draft

12.12. XR_KHR_D3D11_enable

**Name String**

`XR_KHR_D3D11_enable`

**Extension Type**

Instance extension

**Registered Extension Number**

28
Overview

This extension enables the use of the D3D11 graphics API in an OpenXR runtime. Without this extension, the OpenXR runtime may not be able to use any D3D11 swapchain images.

This extension provides the mechanisms necessary for an application to generate a valid XrGraphicsBindingD3D11KHR structure in order to create a D3D11-based XrSession. Note that during this process the application is responsible for creating all the required D3D11 objects, including a graphics device to be used for rendering.

This extension also provides mechanisms for the application to interact with images acquired by calling xrEnumerateSwapchainImages.

In order to expose the structures, types, and functions of this extension, you must define XR_USE_GRAPHICS_API_D3D11 before including the OpenXR platform header openxr_platform.h, in all portions of your library or application that include it.

Swapchain Flag Bits

All XrSwapchainUsageFlags values passed in a session created using XrGraphicsBindingD3D11KHR must be interpreted as follows by the runtime, so that the returned swapchain images used by the application may be used as if they were created with the corresponding D3D11_BIND_FLAG flags. The runtime may set additional bind flags but must not restrict usage.

<table>
<thead>
<tr>
<th>XrSwapchainUsageFlagBits</th>
<th>Corresponding D3D11 bind flag bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>XR_SWAPCHAIN_USAGE_COLOR_ATTACHMENT_BIT</td>
<td>D3D11_BIND_RENDER_TARGET</td>
</tr>
<tr>
<td>XR_SWAPCHAIN_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
<td>D3D11_BIND_DEPTH_STENCIL</td>
</tr>
<tr>
<td>XR_SWAPCHAIN_USAGE_UNORDERED_ACCESS_BIT</td>
<td>D3D11_BIND_UNORDERED_ACCESS</td>
</tr>
<tr>
<td>XrSwapchainUsageFlagBits</td>
<td>Corresponding D3D11 bind flag bits</td>
</tr>
<tr>
<td>--------------------------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>XR_SWAPCHAIN_USAGE_TRANSFER_SRC_BIT</td>
<td>ignored</td>
</tr>
<tr>
<td>XR_SWAPCHAIN_USAGE_TRANSFER_DST_BIT</td>
<td>ignored</td>
</tr>
<tr>
<td>XR_SWAPCHAIN_USAGE_SAMPLED_BIT</td>
<td>D3D11_BIND_SHADER_RESOURCE</td>
</tr>
<tr>
<td>XR_SWAPCHAIN_USAGE_MUTABLE_FORMAT_BIT</td>
<td>ignored</td>
</tr>
<tr>
<td>XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_KHR</td>
<td>ignored</td>
</tr>
<tr>
<td>(Added by XR_KHR_swapchain_usage_input_attachment_bit and only available when that extension is enabled)</td>
<td>ignored</td>
</tr>
</tbody>
</table>

All D3D11 swapchain textures are created with D3D11_USAGE_DEFAULT usage.

**New Object Types**

**New Flag Types**

**New Enum Constants**

XrStructureType enumeration is extended with:

- XR_TYPE_GRAPHICS_REQUIREMENTS_D3D11_KHR
- XR_TYPE_GRAPHICS_BINDING_D3D11_KHR
- XR_TYPE_SWAPCHAIN_IMAGE_D3D11_KHR

**New Enums**

**New Structures**

The following structures are provided to supply supporting runtimes the necessary information required to work with the D3D11 API executing on certain operating systems.

The XrGraphicsBindingD3D11KHR structure is defined as:

```c
typedef struct XrGraphicsBindingD3D11KHR {
    XrStructureType   type;
    const void*       next;
    ID3D11Device*     device;
} XrGraphicsBindingD3D11KHR;
```
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **device** is a pointer to a valid ID3D11Device to use.

When creating a D3D11-backed XrSession, the application will provide a pointer to an XrGraphicsBindingD3D11KHR in the XrSessionCreateInfo::next field of structure passed to xrCreateSession. The D3D11 device specified in XrGraphicsBindingD3D11KHR::device must be created in accordance with the requirements retrieved through xrGetD3D11GraphicsRequirementsKHR, otherwise xrCreateSession must return XR_ERROR_GRAPHICS_DEVICE_INVALID.

Valid Usage (Implicit)

- The XR_KHR_D3D11_enable extension must be enabled prior to using XrGraphicsBindingD3D11KHR
- **type** must be XR_TYPE_GRAPHICS_BINDING_D3D11_KHR
- **next** must be NULL or a valid pointer to the next structure in a structure chain
- **device** must be a pointer to an ID3D11Device value

The XrSwapchainImageD3D11KHR structure is defined as:

```c
typedef struct XrSwapchainImageD3D11KHR {
    XrStructureType type;
    void* next;
    ID3D11Texture2D* texture;
} XrSwapchainImageD3D11KHR;
```

Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **texture** is a pointer to a valid ID3D11Texture2D to use.
If a given session was created with `XrGraphicsBindingD3D11KHR`, the following conditions **must** apply.

- Calls to `xrEnumerateSwapchainImages` on an `XrSwapchain` in that session **must** return an array of `XrSwapchainImageD3D11KHR` structures.
- Whenever an OpenXR function accepts an `XrSwapchainImageBaseHeader` pointer as a parameter in that session, the runtime **must** also accept a pointer to an `XrSwapchainImageD3D11KHR`.

The OpenXR runtime **must** interpret the top-left corner of the swapchain image as the coordinate origin unless specified otherwise by extension functionality.

The OpenXR runtime **must** interpret the swapchain images in a clip space of positive Y pointing up, near Z plane at 0, and far Z plane at 1.

---

### Valid Usage (Implicit)

- The `XR_KHR_D3D11_enable` extension **must** be enabled prior to using `XrSwapchainImageD3D11KHR`
- `type` **must** be `XR_TYPE_SWAPCHAIN_IMAGE_D3D11_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `texture` **must** be a pointer to an `ID3D11Texture2D` value

---

The `XrGraphicsRequirementsD3D11KHR` structure is defined as:

```c
typedef struct XrGraphicsRequirementsD3D11KHR {
    XrStructureType      type;
    void*                next;
    LUID                 adapterLuid;
    D3D_FEATURE_LEVEL    minFeatureLevel;
} XrGraphicsRequirementsD3D11KHR;
```

### Member Descriptions

- `type` is the `XrStructureType` of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `adapterLuid` identifies what graphics device needs to be used.
- `minFeatureLevel` is the minimum feature level that the D3D11 device must be initialized with.
XrGraphicsRequirementsD3D11KHR is populated by xrGetD3D11GraphicsRequirementsKHR.

### Valid Usage (Implicit)

- The `XR_KHR_D3D11_enable` extension must be enabled prior to using `XrGraphicsRequirementsD3D11KHR`.
- `type` must be `XR_TYPE_GRAPHICS_REQUIREMENTS_D3D11_KHR`.
- `next` must be `NULL` or a valid pointer to the next structure in a structure chain.
- `adapterLuid` must be a valid `LUID` value.
- `minFeatureLevel` must be a valid `D3D_FEATURE_LEVEL` value.

### New Functions

Some computer systems may have multiple graphics devices, each of which may have independent external display outputs. XR systems that connect to such graphics devices are typically connected to a single device. Applications need to know what graphics device the XR system is connected to so that they can use that graphics device to generate XR images.

To retrieve the D3D11 feature level and graphics device for an instance and system, call:

```c
XrResult xrGetD3D11GraphicsRequirementsKHR(
    XrInstance instance,
    XrSystemId systemId,
    XrGraphicsRequirementsD3D11KHR* graphicsRequirements);
```

### Parameter Descriptions

- `instance` is an `XrInstance` handle previously created with `xrCreateInstance`.
- `systemId` is an `XrSystemId` handle for the system which will be used to create a session.
- `graphicsRequirements` is the `XrGraphicsRequirementsD3D11KHR` output structure.

The `xrGetD3D11GraphicsRequirementsKHR` function identifies to the application what graphics device (Windows LUID) needs to be used and the minimum feature level to use. The runtime must return `XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING` (`XR_ERROR_VALIDATION_FAILURE` may be returned due to legacy behavior) on calls to `xrCreateSession` if `xrGetD3D11GraphicsRequirementsKHR` has not been called for the same `instance` and `systemId`. The LUID and feature level that `xrGetD3D11GraphicsRequirementsKHR` returns must be used to create the `ID3D11Device` that the application passes to `xrCreateSession` in the `XrGraphicsBindingD3D11KHR`. 
Valid Usage (Implicit)

- The `XR_KHR_D3D11_enable` extension must be enabled prior to calling `xrGetD3D11GraphicsRequirementsKHR`
- `instance` must be a valid `XrInstance` handle
- `graphicsRequirements` must be a pointer to an `XrGraphicsRequirementsD3D11KHR` structure

Return Codes

Success
- `XR_SUCCESS`

Failure
- `XR_ERROR_FUNCTION_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SYSTEM_INVALID`

Issues

Version History

- Revision 1, 2018-05-07 (Mark Young)
  - Initial draft
- Revision 2, 2018-06-21 (Bryce Hutchings)
  - Split `XR_KHR_D3D_enable` into `XR_KHR_D3D11_enable`
  - Rename and expand `xrGetD3DGraphicsDeviceKHR` functionality to `xrGetD3D11GraphicsRequirementsKHR`
- Revision 3, 2018-11-15 (Paul Pedriana)
  - Specified the swapchain texture coordinate origin.
- Revision 4, 2018-11-16 (Minmin Gong)
  - Specified Y direction and Z range in clip space
- Revision 5, 2020-08-06 (Bryce Hutchings)
  - Added new `XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING` error code
12.13. XR_KHR_D3D12_enable

Name String
XR_KHR_D3D12_enable

Extension Type
Instance extension

Registered Extension Number
29

Revision
9

Extension and Version Dependencies
• Requires OpenXR 1.0

Last Modified Date
2020-03-18

IP Status
No known IP claims.

Contributors
Bryce Hutchings, Microsoft
Paul Pedriana, Oculus
Mark Young, LunarG
Minmin Gong, Microsoft
Dan Ginsburg, Valve
Matthieu Bucchianeri, Microsoft

Overview

This extension enables the use of the D3D12 graphics API in an OpenXR runtime. Without this extension, the OpenXR runtime may not be able to use any D3D12 swapchain images.

This extension provides the mechanisms necessary for an application to generate a valid XrGraphicsBindingD3D12KHR structure in order to create a D3D12-based XrSession. Note that during this process the application is responsible for creating all the required D3D12 objects, including a
graphics device and queue to be used for rendering.

This extension also provides mechanisms for the application to interact with images acquired by calling `xrEnumerateSwapchainImages`.

In order to expose the structures, types, and functions of this extension, you **must** define `XR_USE_GRAPHICS_API_D3D12` before including the OpenXR platform header `openxr_platform.h`, in all portions of your library or application that include it.

**Swapchain Image Resource State**

When an application acquires a swapchain image by calling `xrAcquireSwapchainImage` in a session created using `XrGraphicsBindingD3D12KHR`, the OpenXR runtime **must** guarantee that:

- The color rendering target image has a resource state match with `D3D12_RESOURCE_STATE_RENDER_TARGET`
- The depth rendering target image has a resource state match with `D3D12_RESOURCE_STATE_DEPTH_WRITE`
- The `ID3D12CommandQueue` specified in `XrGraphicsBindingD3D12KHR` can write to the image.

When an application releases a swapchain image by calling `xrReleaseSwapchainImage`, in a session created using `XrGraphicsBindingD3D12KHR`, the OpenXR runtime **must** interpret the image as:

- Having a resource state match with `D3D12_RESOURCE_STATE_RENDER_TARGET` if the image is a color rendering target
- Having a resource state match with `D3D12_RESOURCE_STATE_DEPTH_WRITE` if the image is a depth rendering target
- Being available for read/write on the `ID3D12CommandQueue` specified in `XrGraphicsBindingD3D12KHR`.

The application is responsible for transitioning the swapchain image back to the resource state and queue availability that the OpenXR runtime requires. If the image is not in a resource state match with the above specifications the runtime **may** exhibit undefined behavior.

All `XrSwapchainUsageFlags` values passed in a session created using `XrGraphicsBindingD3D12KHR` **must** be interpreted as follows by the runtime, so that the returned swapchain images used by the application may be used as if they were created with the corresponding D3D12_BIND_FLAG flags and heap type. The runtime **may** set additional resource flags but **must** not restrict usage.

<table>
<thead>
<tr>
<th><code>XrSwapchainUsageFlagBits</code></th>
<th>Corresponding D3D12 resource flag bits</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>XR_SWAPCHAIN_USAGE_COLOR_ATTACHMENT_BIT</code></td>
<td><code>D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET</code></td>
</tr>
<tr>
<td><code>XR_SWAPCHAIN_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT</code></td>
<td><code>D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL</code></td>
</tr>
<tr>
<td><code>XR_SWAPCHAIN_USAGE_UNORDERED_ACCESS_BIT</code></td>
<td><code>D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS</code></td>
</tr>
<tr>
<td><code>XR_SWAPCHAIN_USAGE_TRANSFER_SRC_BIT</code></td>
<td><code>ignored</code></td>
</tr>
<tr>
<td><code>XR_SWAPCHAIN_USAGE_TRANSFER_DST_BIT</code></td>
<td><code>ignored</code></td>
</tr>
<tr>
<td>XrSwapchainUsageFlagBits</td>
<td>Corresponding D3D12 resource flag bits</td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>XR_SWAPCHAIN_USAGE_SAMPLED_BIT omitted</td>
<td>D3D12_RESOURCE_FLAG_DENY_SHADER_RESOURCE</td>
</tr>
<tr>
<td>XR_SWAPCHAIN_USAGE_MUTABLE_FORMAT_BIT</td>
<td>ignored</td>
</tr>
<tr>
<td>XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_KHR (Added by XR_KHR_swapchain_usage_input_attachment_bit and only available when that extension is enabled)</td>
<td>ignored</td>
</tr>
</tbody>
</table>

All D3D12 swapchain textures are created with D3D12_HEAP_TYPE_DEFAULT usage.

**New Object Types**

**New Flag Types**

**New Enum Constants**

*XrStructureType* enumeration is extended with:

- XR_TYPE_GRAPHICS_REQUIREMENTS_D3D12_KHR
- XR_TYPE_GRAPHICS_BINDING_D3D12_KHR
- XR_TYPE_SWAPCHAIN_IMAGE_D3D12_KHR

**New Enums**

**New Structures**

The following structures are provided to supply supporting runtimes the necessary information required to work with the D3D12 API executing on certain operating systems.

The *XrGraphicsBindingD3D12KHR* structure is defined as:

```c
typedef struct XrGraphicsBindingD3D12KHR {
    XrStructureType        type;
    const void*            next;
    ID3D12Device*          device;
    ID3D12CommandQueue*    queue;
} XrGraphicsBindingD3D12KHR;
```
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **device** is a pointer to a valid ID3D12Device to use.
- **queue** is a pointer to a valid ID3D12CommandQueue to use.

When creating a D3D12-backed XrSession, the application will provide a pointer to an XrGraphicsBindingD3D12KHR in the XrSessionCreateInfo::next field of structure passed to xrCreateSession. The D3D12 device specified in XrGraphicsBindingD3D12KHR::device must be created in accordance with the requirements retrieved through xrGetD3D12GraphicsRequirementsKHR, otherwise xrCreateSession must return XR_ERROR_GRAPHICS_DEVICE_INVALID.

Valid Usage ( Implicit)

- The XR_KHR_D3D12_enable extension must be enabled prior to using XrGraphicsBindingD3D12KHR
- **type** must be XR_TYPE_GRAPHICS_BINDING_D3D12_KHR
- **next** must be NULL or a valid pointer to the next structure in a structure chain
- **device** must be a pointer to an ID3D12Device value
- **queue** must be a pointer to an ID3D12CommandQueue value

The XrSwapchainImageD3D12KHR structure is defined as:

```c
typedef struct XrSwapchainImageD3D12KHR {
    XrStructureType    type;
    void*               next;
    ID3D12Resource*     texture;
} XrSwapchainImageD3D12KHR;
```
Member Descriptions

- **type** is the \texttt{XrStructureType} of this structure.
- **next** is \texttt{NULL} or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **texture** is a pointer to a valid \texttt{ID3D12Texture2D} to use.

If a given session was created with \texttt{XrGraphicsBindingD3D12KHR}, the following conditions \textbf{must} apply.

- Calls to \texttt{xrEnumerateSwapchainImages} on an \texttt{XrSwapchain} in that session \textbf{must} return an array of \texttt{XrSwapchainImageD3D12KHR} structures.
- Whenever an OpenXR function accepts an \texttt{XrSwapchainImageBaseHeader} pointer as a parameter in that session, the runtime \textbf{must} also accept a pointer to an \texttt{XrSwapchainImageD3D12KHR}.

The OpenXR runtime \textbf{must} interpret the top-left corner of the swapchain image as the coordinate origin unless specified otherwise by extension functionality.

The OpenXR runtime \textbf{must} interpret the swapchain images in a clip space of positive Y pointing up, near Z plane at 0, and far Z plane at 1.

Valid Usage (Implicit)

- The \texttt{XR_KHR_D3D12_enable} extension \textbf{must} be enabled prior to using \texttt{XrSwapchainImageD3D12KHR}
- **type** \textbf{must} be \texttt{XR_TYPE_SWAPCHAIN_IMAGE_D3D12_KHR}
- **next** \textbf{must} be \texttt{NULL} or a valid pointer to the next structure in a structure chain
- **texture** \textbf{must} be a pointer to an \texttt{ID3D12Resource} value

The \texttt{XrGraphicsRequirementsD3D12KHR} structure is defined as:

```c
typedef struct XrGraphicsRequirementsD3D12KHR {
    XrStructureType      type;
    void*                next;
    LUID                 adapterLuid;
    D3D_FEATURE_LEVEL    minFeatureLevel;
} XrGraphicsRequirementsD3D12KHR;
```
Member Descriptions

- **type** is the `XrStructureType` of this structure.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **adapterLuid** identifies what graphics device needs to be used.
- **minFeatureLevel** is the minimum feature level that the D3D12 device must be initialized with.

`XrGraphicsRequirementsD3D12KHR` is populated by `xrGetD3D12GraphicsRequirementsKHR`.

Valid Usage (Implicit)

- The **XR_KHR_D3D12_enable** extension must be enabled prior to using `XrGraphicsRequirementsD3D12KHR`
- **type** must be `XR_TYPE_GRAPHICS_REQUIREMENTS_D3D12_KHR`
- **next** must be `NULL` or a valid pointer to the next structure in a structure chain
- **adapterLuid** must be a valid `LUID` value
- **minFeatureLevel** must be a valid `D3D_FEATURE_LEVEL` value

New Functions

Some computer systems may have multiple graphics devices, each of which may have independent external display outputs. XR systems that connect to such graphics devices are typically connected to a single device. Applications need to know what graphics device the XR system is connected to so that they can use that graphics device to generate XR images.

To retrieve the D3D12 feature level and graphics device for an instance and system, call:

```c
XrResult xrGetD3D12GraphicsRequirementsKHR(
    XrInstance instance,
    XrSystemId systemId,
    XrGraphicsRequirementsD3D12KHR* graphicsRequirements);
```
Parameter Descriptions

- **instance** is an `XrInstance` handle previously created with `xrCreateInstance`.
- **systemId** is an `XrSystemId` handle for the system which will be used to create a session.
- **graphicsRequirements** is the `XrGraphicsRequirementsD3D12KHR` output structure.

The `xrGetD3D12GraphicsRequirementsKHR` function identifies to the application what graphics device (Windows LUID) needs to be used and the minimum feature level to use. The runtime **must** return `XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING` (`XR_ERROR_VALIDATION_FAILURE` may be returned due to legacy behavior) on calls to `xrCreateSession` if `xrGetD3D12GraphicsRequirementsKHR` has not been called for the same `instance` and `systemId`. The LUID and feature level that `xrGetD3D12GraphicsRequirementsKHR` returns **must** be used to create the `ID3D12Device` that the application passes to `xrCreateSession` in the `XrGraphicsBindingD3D12KHR`.

Valid Usage (Implicit)

- The `XR_KHR_D3D12_enable` extension **must** be enabled prior to calling `xrGetD3D12GraphicsRequirementsKHR`
- **instance** **must** be a valid `XrInstance` handle
- **graphicsRequirements** **must** be a pointer to an `XrGraphicsRequirementsD3D12KHR` structure

Return Codes

**Success**

- `XR_SUCCESS`

**Failure**

- `XR_ERROR_FUNCTION_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SYSTEM_INVALID`

Issues

Version History
• Revision 1, 2018-05-07 (Mark Young)
  ◦ Initial draft
• Revision 2, 2018-06-21 (Bryce Hutchings)
  ◦ Split XR_KHR_D3D_enable into XR_KHR_D3D12_enable
  ◦ Rename and expand xrGetD3DGraphicsDeviceKHR functionality to xrGetD3D12GraphicsRequirementsKHR
• Revision 3, 2018-11-15 (Paul Pedriana)
  ◦ Specified the swapchain texture coordinate origin.
• Revision 4, 2018-11-16 (Minmin Gong)
  ◦ Specified Y direction and Z range in clip space
• Revision 5, 2019-01-29 (Dan Ginsburg)
  ◦ Added swapchain image resource state details.
• Revision 6, 2020-03-18 (Minmin Gong)
  ◦ Specified depth swapchain image resource state.
• Revision 7, 2020-08-06 (Bryce Hutchings)
  ◦ Added new XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING error code
• Revision 8, 2021-09-09 (Bryce Hutchings)
  ◦ Document mapping for XrSwapchainUsageFlags
• Revision 9, 2021-12-28 (Matthieu Bucchianeri)
  ◦ Added missing XR_ERROR_GRAPHICS_DEVICE_INVALID error condition

12.14. XR_KHR_loader_init

Name String
  XR_KHR_loader_init

Extension Type
  Instance extension

Registered Extension Number
  89

Revision
  1

Extension and Version Dependencies
  • Requires OpenXR 1.0
Overview

On some platforms, before loading can occur the loader must be initialized with platform-specific parameters. Unlike other extensions, the presence of this extension is signaled by a successful call to `xrGetInstanceProcAddr` to retrieve the function pointer for `xrInitializeLoaderKHR` using a null instance handle. If this extension is supported, its use may be required on some platforms and the use of the `xrInitializeLoaderKHR` function must precede other OpenXR calls except `xrGetInstanceProcAddr`. This function exists as part of the loader library that the application is using.

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

The `XrLoaderInitInfoBaseHeaderKHR` structure is defined as:

```c
typedef struct XrLoaderInitInfoBaseHeaderKHR {
    XrStructureType    type;
    const void*        next;
} XrLoaderInitInfoBaseHeaderKHR;
```

Member Descriptions

- **type** is the `XrStructureType` of this structure. This base structure itself has no associated `XrStructureType` value.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
Valid Usage (Implicit)

- The XR_KHR_loader_init extension must be enabled prior to using XrLoaderInitInfoBaseHeaderKHR
- type must be XR_TYPE_LOADER_INIT_INFO_ANDROID_KHR
- next must be NULL or a valid pointer to the next structure in a structure chain

New Functions

To initialize an OpenXR loader with platform or implementation-specific parameters, call:

```c
XrResult xrInitializeLoaderKHR(
    const XrLoaderInitInfoBaseHeaderKHR* loaderInitInfo);
```

Parameter Descriptions

- loaderInitInfo is a pointer to an XrLoaderInitInfoBaseHeaderKHR structure, which is a polymorphic type defined by other platform- or implementation-specific extensions.

Issues

Version History

- Revision 1, 2020-05-07 (Cass Everitt)
  - Initial draft

12.15. XR_KHR_loader_init_android

Name String

XR_KHR_loader_init_android

Extension Type

Instance extension

Registered Extension Number

90

Revision

1
Extension and Version Dependencies

- Requires OpenXR 1.0
- Requires XR_KHR_loader_init

Last Modified Date

2020-05-07

IP Status

No known IP claims.

Contributors

Cass Everitt, Facebook

Overview

On Android, some loader implementations need the application to provide additional information on initialization. This extension defines the parameters needed by such implementations. If this is available on a given implementation, an application must make use of it.

On implementations where use of this is required, the following condition must apply:

- Whenever an OpenXR function accepts an XrLoaderInitInfoBaseHeaderKHR pointer, the runtime (and loader) must also accept a pointer to an XrLoaderInitInfoAndroidKHR.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- XR_TYPE_LOADER_INIT_INFO_ANDROID_KHR

New Enums

New Structures

The XrLoaderInitInfoAndroidKHR structure is defined as:
typedef struct XrLoaderInitInfoAndroidKHR {
    XrStructureType    type;
    const void*        next;
    void*              applicationVM;
    void*              applicationContext;
} XrLoaderInitInfoAndroidKHR;

Member Descriptions

• type is the XrStructureType of this structure.
• next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
• applicationVM is a pointer to the JNI's opaque JavaVM structure, cast to a void pointer.
• applicationContext is a JNI reference to an android.content.Context associated with the application, cast to a void pointer.

Valid Usage (Implicit)

• The XR_KHR_loader_init_android extension must be enabled prior to using XrLoaderInitInfoAndroidKHR
• type must be XR_TYPE_LOADER_INIT_INFO_ANDROID_KHR
• next must be NULL or a valid pointer to the next structure in a structure chain
• applicationVM must be a pointer value
• applicationContext must be a pointer value

New Functions

Issues

Version History

• Revision 1, 2020-05-07 (Cass Everitt)
  ◦ Initial draft

12.16. XR_KHR_opengl_enable

Name String

XR_KHR_opengl_enable
Extension Type

Instance extension

Registered Extension Number

24

Revision

10

Extension and Version Dependencies

• Requires OpenXR 1.0

Last Modified Date

2019-07-02

IP Status

No known IP claims.

Contributors

Mark Young, LunarG
Bryce Hutchings, Microsoft
Paul Pedriana, Oculus
Minmin Gong, Microsoft
Robert Menzel, NVIDIA
Jakob Bornecrantz, Collabora
Paulo Gomes, Samsung Electronics

Overview

This extension enables the use of the OpenGL graphics API in an OpenXR runtime. Without this extension, the OpenXR runtime may not be able to provide any OpenGL swapchain images.

This extension provides the mechanisms necessary for an application to generate a valid XrGraphicsBindingOpenGL*KHR structure in order to create an OpenGL-based XrSession. Note that during this process the application is responsible for creating an OpenGL context to be used for rendering. The runtime however will provide the OpenGL textures to render into in the form of a swapchain.

This extension provides mechanisms for the application to interact with images acquired by calling xrEnumerateSwapchainImages.

In order to expose the structures, types, and functions of this extension, the application must define XR_USE_GRAPHICS_API_OPENGL, as well as an appropriate window system define supported by this extension, before including the OpenXR platform header openxr_platform.h, in all portions of the library or application that include it. The window system defines currently supported by this extension are:
• XR_USE_PLATFORM_WIN32
• XR_USE_PLATFORM_XLIB
• XR_USE_PLATFORM_XCB
• XR_USE_PLATFORM_WAYLAND

Note that a runtime implementation of this extension is only required to support the structs introduced by this extension which belong to the platform it is running on.

Note that the OpenGL context given to the call `xrCreateSession` must not be bound in another thread when calling the functions: `xrCreateSession`, `xrDestroySession`, `xrBeginFrame`, `xrEndFrame`, `xrCreateSwapchain`, `xrDestroySwapchain`, `xrEnumerateSwapchainImages`, `xrAcquireSwapchainImage`, `xrWaitSwapchainImage` and `xrReleaseSwapchainImage`. It may be bound in the thread calling those functions. The runtime must not access the context from any other function. In particular the application must be able to call `xrWaitFrame` from a different thread than the rendering thread.

**Swapchain Flag Bits**

All `XrSwapchainUsageFlags` valid values passed in a session created using `XrGraphicsBindingOpenGLWin32KHR`, `XrGraphicsBindingOpenGLXlibKHR`, `XrGraphicsBindingOpenGLXcbKHR` or `XrGraphicsBindingOpenGLWaylandKHR` should be ignored as there is no mapping to OpenGL texture settings.

```
Note
In such a session, a runtime may use a supporting graphics API, such as Vulkan, to allocate images that are intended to alias with OpenGL textures, and be part of an XrSwapchain. A runtime which allocates the texture with a different graphics API may need to enable several usage flags on the underlying native texture resource to ensure compatibility with OpenGL.
```

**New Object Types**

**New Flag Types**

**New Enum Constants**

`XrStructureType` enumeration is extended with:

- `XR_TYPE_GRAPHICS_REQUIREMENTS_OPENGL_KHR`
- `XR_TYPE_GRAPHICS_BINDING_OPENGL_WIN32_KHR`
- `XR_TYPE_GRAPHICS_BINDING_OPENGL_XLIB_KHR`
- `XR_TYPE_GRAPHICS_BINDING_OPENGL_XCB_KHR`
- `XR_TYPE_GRAPHICS_BINDING_OPENGL_WAYLAND_KHR`
- `XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_KHR`
New Enums

New Structures

The following structures are provided to supply supporting runtimes the necessary information required to work with the OpenGL API executing on certain operating systems.

These structures are only available when the corresponding `XR_USE_PLATFORM_` macro is defined before including `openxr_platform.h`.

The `XrGraphicsBindingOpenGLWin32KHR` structure is defined as:

```c
typedef struct XrGraphicsBindingOpenGLWin32KHR {
    XrStructureType    type;
    const void*        next;
    HDC                hDC;
    HGLRC              hGLRC;
} XrGraphicsBindingOpenGLWin32KHR;
```

**Member Descriptions**

- **type** is the `XrStructureType` of this structure.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **hDC** is a valid Windows HW device context handle.
- **hGLRC** is a valid Windows OpenGL rendering context handle.

When creating an OpenGL-backed `XrSession` on Microsoft Windows, the application will provide a pointer to an `XrGraphicsBindingOpenGLWin32KHR` in the `next` chain of the `XrSessionCreateInfo`. As no standardized way exists for OpenGL to create the graphics context on a specific GPU, the runtime **must** assume that the application uses the operating systems default GPU. If the GPU used by the runtime does not match the GPU on which the OpenGL context of the application got created, `xrCreateSession` **must** return `XR_ERROR_GRAPHICS_DEVICE_INVALID`.

The required window system configuration define to expose this structure type is `XR_USE_PLATFORM_WIN32`.
Valid Usage (Implicit)

- The `XR_KHR_opengl_enable` extension must be enabled prior to using `XrGraphicsBindingOpenGLWin32KHR`
- `type` must be `XR_TYPE_GRAPHICS_BINDING_OPENGL_WIN32_KHR`
- `next` must be `NULL` or a valid pointer to the next structure in a structure chain
- `hDC` must be a valid `HDC` value
- `hGLRC` must be a valid `HGLRC` value

The `XrGraphicsBindingOpenGLXlibKHR` structure is defined as:

```c
typedef struct XrGraphicsBindingOpenGLXlibKHR {
    XrStructureType    type;
    const void*        next;
    Display*           xDisplay;
    uint32_t           visualid;
    GLXFBConfig        glxFBConfig;
    GLXDrawable        glxDrawable;
    GLXContext         glxContext;
} XrGraphicsBindingOpenGLXlibKHR;
```

Member Descriptions

- `type` is the `XrStructureType` of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `xDisplay` is a valid X11 `Display`.
- `visualid` is a valid X11 visual identifier.
- `glxFBConfig` is a valid X11 OpenGL GLX `GLXFBConfig`.
- `glxDrawable` is a valid X11 OpenGL GLX `GLXDrawable`.
- `glxContext` is a valid X11 OpenGL GLX `GLXContext`.

When creating an OpenGL-backed `XrSession` on any Linux/Unix platform that utilizes X11 and GLX, via the Xlib library, the application will provide a pointer to an `XrGraphicsBindingOpenGLXlibKHR` in the `next` chain of the `XrSessionCreateInfo`.

The required window system configuration define to expose this structure type is
Valid Usage (Implicit)

- The `XR_KHR_opengl_enable` extension must be enabled prior to using `XrGraphicsBindingOpenGLXlibKHR`.
- `type` must be `XR_TYPE_GRAPHICS_BINDING_OPENGL_XLIB_KHR`.
- `next` must be `NULL` or a valid pointer to the next structure in a structure chain.
- `xDisplay` must be a pointer to a `Display` value.
- `glxFBConfig` must be a valid `GLXFBConfig` value.
- `glxDrawable` must be a valid `GLXDrawable` value.
- `glxContext` must be a valid `GLXContext` value.

The `XrGraphicsBindingOpenGLXcbKHR` structure is defined as:

```c
typedef struct XrGraphicsBindingOpenGLXcbKHR {
    XrStructureType       type;
    const void*           next;
    xcb_connection_t*     connection;
    uint32_t              screenNumber;
    xcb_glx_fbconfig_t    fbconfigid;
    xcb_visualid_t        visualid;
    xcb_glx_drawable_t    glxDrawable;
    xcb_glx_context_t     glxContext;
} XrGraphicsBindingOpenGLXcbKHR;
```
Member Descriptions

• **type** is the XrStructureType of this structure.
• **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
• **connection** is a valid xcb_connection_t.
• **screenNumber** is an index indicating which screen should be used for rendering.
• **fbconfigid** is a valid XCB OpenGL GLX xcb_glx_fbconfig_t.
• **visualid** is a valid XCB OpenGL GLX xcb_visualid_t.
• **glxDrawable** is a valid XCB OpenGL GLX xcb_glx_drawable_t.
• **glxContext** is a valid XCB OpenGL GLX xcb_glx_context_t.

When creating an OpenGL-backed XrSession on any Linux/Unix platform that utilizes X11 and GLX, via the Xlib library, the application will provide a pointer to an XrGraphicsBindingOpenGLXcbKHR in the next chain of the XrSessionCreateInfo.

The required window system configuration define to expose this structure type is XR_USE_PLATFORM_XCB.

Valid Usage (Implicit)

• The XR_KHR_opengl_enable extension must be enabled prior to using XrGraphicsBindingOpenGLXcbKHR
• **type** must be XR_TYPE_GRAPHICS_BINDING_OPENGL_XCB_KHR
• **next** must be NULL or a valid pointer to the next structure in a structure chain
• **connection** must be a pointer to an xcb_connection_t value
• **fbconfigid** must be a valid xcb_glx_fbconfig_t value
• **visualid** must be a valid xcb_visualid_t value
• **glxDrawable** must be a valid xcb_glx_drawable_t value
• **glxContext** must be a valid xcb_glx_context_t value

The XrGraphicsBindingOpenGLWaylandKHR structure is defined as:
typedef struct XrGraphicsBindingOpenGLWaylandKHR {
    XrStructureType type;
    const void* next;
    struct wl_display* display;
} XrGraphicsBindingOpenGLWaylandKHR;

### Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **display** is a valid Wayland `wl_display`.

When creating an OpenGL-backed XrSession on any Linux/Unix platform that utilizes the Wayland protocol with its compositor, the application will provide a pointer to an `XrGraphicsBindingOpenGLWaylandKHR` in the `next` chain of the XrSessionCreateInfo.

The required window system configuration define to expose this structure type is XR_USE_PLATFORM_WAYLAND.

### Valid Usage (Implicit)

- The XR_KHR_opengl_enable extension **must** be enabled prior to using `XrGraphicsBindingOpenGLWaylandKHR`
- **type** **must** be XR_TYPE_GRAPHICS_BINDING_OPENGL_WAYLAND_KHR
- **next** **must** be NULL or a valid pointer to the next structure in a structure chain
- **display** **must** be a pointer to a `wl_display` value

The XrSwapchainImageOpenGLKHR structure is defined as:

typedef struct XrSwapchainImageOpenGLKHR {
    XrStructureType type;
    void* next;
    uint32_t image;
} XrSwapchainImageOpenGLKHR;
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **image** is the OpenGL texture handle associated with this swapchain image.

If a given session was created with a XrGraphicsBindingOpenGLKHR, the following conditions **must** apply.

- Calls to xrEnumerateSwapchainImages on an XrSwapchain in that session **must** return an array of XrSwapchainImageOpenGLKHR structures.
- Whenever an OpenXR function accepts an XrSwapchainImageBaseHeader pointer as a parameter in that session, the runtime **must** also accept a pointer to an XrSwapchainImageOpenGLKHR.

The OpenXR runtime **must** interpret the bottom-left corner of the swapchain image as the coordinate origin unless specified otherwise by extension functionality.

The OpenXR runtime **must** interpret the swapchain images in a clip space of positive Y pointing up, near Z plane at -1, and far Z plane at 1.

### Valid Usage (Implicit)

- The XR_KHR_opengl_enable extension **must** be enabled prior to using XrSwapchainImageOpenGLKHR
- **type** **must** be XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_KHR
- **next** **must** be NULL or a valid pointer to the next structure in a structure chain

The XrGraphicsRequirementsOpenGLKHR structure is defined as:

```c
typedef struct XrGraphicsRequirementsOpenGLKHR {
  XrStructureType    type;
  void*              next;
  XrVersion          minApiVersionSupported;
  XrVersion          maxApiVersionSupported;
} XrGraphicsRequirementsOpenGLKHR;
```
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **minApiVersionSupported** is the minimum version of OpenGL that the runtime supports. Uses XR_MAKE_VERSION on major and minor API version, ignoring any patch version component.
- **maxApiVersionSupported** is the maximum version of OpenGL that the runtime has been tested on and is known to support. Newer OpenGL versions might work if they are compatible. Uses XR_MAKE_VERSION on major and minor API version, ignoring any patch version component.

XrGraphicsRequirementsOpenGLKHR is populated by xrGetOpenGLGraphicsRequirementsKHR with the runtime’s OpenGL API version requirements.

Valid Usage (Implicit)

- The XR_KHR_opengl_enable extension must be enabled prior to using XrGraphicsRequirementsOpenGLKHR
- **type** must be XR_TYPE_GRAPHICS_REQUIREMENTS_OPENGL_KHR
- **next** must be NULL or a valid pointer to the next structure in a structure chain

New Functions

To query OpenGL API version requirements for an instance and system, call:

```c
XrResult xrGetOpenGLGraphicsRequirementsKHR(
    XrInstance instance,
    XrSystemId systemId,
    XrGraphicsRequirementsOpenGLKHR* graphicsRequirements);
```

Parameter Descriptions

- **instance** is an XrInstance handle previously created with xrCreateInstance.
- **systemId** is an XrSystemId handle for the system which will be used to create a session.
- **graphicsRequirements** is the XrGraphicsRequirementsOpenGLKHR output structure.

The xrGetOpenGLGraphicsRequirementsKHR function identifies to the application the minimum
OpenGL version requirement and the highest known tested OpenGL version. The runtime must return 
XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING (XR_ERROR_VALIDATION_FAILURE may be returned due to 
legacy behavior) on calls to xrCreateSession if xrGetOpenGLGraphicsRequirementsKHR has not been 
called for the same instance and systemId.

Valid Usage (Implicit)

- The XR_KHR_opengl_enable extension must be enabled prior to calling 
xrGetOpenGLGraphicsRequirementsKHR
- instance must be a valid XrInstance handle
- graphicsRequirements must be a pointer to an XrGraphicsRequirementsOpenGLKHR structure

Return Codes

Success
- XR_SUCCESS

Failure
- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SYSTEM_INVALID

Issues

Version History

- Revision 1, 2018-05-07 (Mark Young)
  - Initial draft
- Revision 2, 2018-06-21 (Bryce Hutchings)
  - Add new xrGetOpenGLGraphicsRequirementsKHR
- Revision 3, 2018-11-15 (Paul Pedriana)
  - Specified the swapchain texture coordinate origin.
- Revision 4, 2018-11-16 (Minmin Gong)
  - Specified Y direction and Z range in clip space
12.17. XR_KHR_opengl_es_enable

**Name String**

*XR_KHR_opengl_es_enable*

**Extension Type**

Instance extension

**Registered Extension Number**

25

**Revision**

8

**Extension and Version Dependencies**

- Requires OpenXR 1.0

**Last Modified Date**

2019-07-12

**IP Status**

No known IP claims.

**Contributors**

Mark Young, LunarG
Bryce Hutchings, Microsoft
Paul Pedriana, Oculus
This extension must be provided by runtimes supporting applications using OpenGL ES APIs for rendering. OpenGL ES applications need this extension to obtain compatible swapchain images which the runtime is required to supply. The runtime needs the following OpenGL ES objects from the application in order to interact properly with the OpenGL ES driver: EGLDisplay, EGLConfig and EGLContext.

These are passed from the application to the runtime in a XrGraphicsBindingOpenGLESAndroidKHR structure when creating the XrSession. Although not restricted to Android, the OpenGL ES extension is currently tailored for Android.

Note that the application is responsible for creating the required OpenGL ES objects, including an OpenGL ES context to be used for rendering.

This extension also provides mechanisms for the application to interact with images acquired by calling xrEnumerateSwapchainImages.

In order to expose the structures, types, and functions of this extension, the application source code must define XR_USE_GRAPHICS_API_OPENGL_ES, as well as an appropriate window system define, before including the OpenXR platform header openxr_platform.h, in all portions of your library or application that include it. The only window system define currently supported by this extension is:

- XR_USE_PLATFORM_ANDROID

### Swapchain Flag Bits

All XrSwapchainUsageFlags valid values passed in a session created using XrGraphicsBindingOpenGLESAndroidKHR should be ignored as there is no mapping to OpenGL ES texture settings.

Note

In such a session, a runtime may use a supporting graphics API, such as Vulkan, to allocate images that are intended to alias with OpenGLES textures, and be part of an XrSwapchain. A runtime which allocates the texture with a different graphics API may need to enable several usage flags on the underlying native texture resource to ensure compatibility with OpenGL ES.

### New Object Types

### New Flag Types
New Enum Constants

**XrStructureType** enumeration is extended with:

- **XR_TYPE_GRAPHICS_REQUIREMENTS_OPENGL_ES_KHR**
- **XR_TYPE_GRAPHICS_BINDING_OPENGL_ES_ANDROID_KHR**
- **XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_ES_KHR**

New Enums

New Structures

The following structures are provided to supply supporting runtimes the necessary information required to work with the OpenGL ES API executing on certain operating systems.

These structures are only available when the corresponding **XR_USE_PLATFORM_** macro is defined before including `openxr_platform.h`.

The **XrGraphicsBindingOpenGLESAndroidKHR** structure is defined as:

```c
typedef struct XrGraphicsBindingOpenGLESAndroidKHR {
    XrStructureType    type;
    const void*        next;
    EGLDisplay         display;
    EGLConfig          config;
    EGLContext         context;
} XrGraphicsBindingOpenGLESAndroidKHR;
```

**Member Descriptions**

- **type** is the **XrStructureType** of this structure.
- **next** is **NULL** or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **display** is a valid Android OpenGL ES **EGLDisplay**.
- **config** is a valid Android OpenGL ES **EGLConfig**.
- **context** is a valid Android OpenGL ES **EGLContext**.

When creating an OpenGL ES-backed **XrSession** on Android, the application will provide a pointer to an **XrGraphicsBindingOpenGLESAndroidKHR** structure in the **next** chain of the **XrSessionCreateInfo**.

The required window system configuration define to expose this structure type is...
Valid Usage (Implicit)

- The `XR_KHR_opengl_es_enable` extension must be enabled prior to using `XrGraphicsBindingOpenGLESAndroidKHR`
- `type` must be `XR_TYPE_GRAPHICS_BINDING_OPENGL_ES_ANDROID_KHR`
- `next` must be `NULL` or a valid pointer to the next structure in a structure chain
- `display` must be a valid `EGLDisplay` value
- `config` must be a valid `EGLConfig` value
- `context` must be a valid `EGLContext` value

The `XrSwapchainImageOpenGLESKHR` structure is defined as:

```c
typedef struct XrSwapchainImageOpenGLESKHR {
    XrStructureType    type;
    void*              next;
    uint32_t           image;
} XrSwapchainImageOpenGLESKHR;
```

Member Descriptions

- `type` is the `XrStructureType` of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `image` is an index indicating the current OpenGL ES swapchain image to use.

If a given session was created with a `XrGraphicsBindingOpenGLESKHR`, the following conditions must apply.

- Calls to `xrEnumerateSwapchainImages` on an `XrSwapchain` in that session must return an array of `XrSwapchainImageOpenGLESKHR` structures.
- Whenever an OpenXR function accepts an `XrSwapchainImageBaseHeader` pointer as a parameter in that session, the runtime must also accept a pointer to an `XrSwapchainImageOpenGLESKHR` structure.

The OpenXR runtime must interpret the bottom-left corner of the swapchain image as the coordinate origin unless specified otherwise by extension functionality.
The OpenXR runtime **must** interpret the swapchain images in a clip space of positive Y pointing up, near Z plane at -1, and far Z plane at 1.

**Valid Usage (Implicit)**

- The `XR_KHR_opengl_es_enable` extension **must** be enabled prior to using `XrSwapchainImageOpenGLESKHR`
- `type` **must** be `XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_ES_KHR`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

The `XrGraphicsRequirementsOpenGLESKHR` structure is defined as:

```c
typedef struct XrGraphicsRequirementsOpenGLESKHR {
    XrStructureType    type;
    void*              next;
    XrVersion          minApiVersionSupported;
    XrVersion          maxApiVersionSupported;
} XrGraphicsRequirementsOpenGLESKHR;
```

**Member Descriptions**

- `type` is the `XrStructureType` of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `minApiVersionSupported` is the minimum version of OpenGL ES that the runtime supports. Uses `XR_MAKE_VERSION` on major and minor API version, ignoring any patch version component.
- `maxApiVersionSupported` is the maximum version of OpenGL ES that the runtime has been tested on and is known to support. Newer OpenGL ES versions might work if they are compatible. Uses `XR_MAKE_VERSION` on major and minor API version, ignoring any patch version component.

`XrGraphicsRequirementsOpenGLESKHR` is populated by `xrGetOpenGLESGraphicsRequirementsKHR` with the runtime’s OpenGL ES API version requirements.
Valid Usage (Implicit)

- The `XR_KHR_opengl_es_enable` extension must be enabled prior to using `XrGraphicsRequirementsOpenGLESKHR`
- `type` must be `XR_TYPE_GRAPHICS_REQUIREMENTS_OPENGL_ES_KHR`
- `next` must be `NULL` or a valid pointer to the next structure in a structure chain

New Functions

To query OpenGL ES API version requirements for an instance and system, call:

```c
XrResult xrGetOpenGLESGraphicsRequirementsKHR(
  XrInstance instance,
  XrSystemId systemId,
  XrGraphicsRequirementsOpenGLESKHR* graphicsRequirements);
```

Parameter Descriptions

- `instance` is an `XrInstance` handle previously created with `xrCreateInstance`.
- `systemId` is an `XrSystemId` handle for the system which will be used to create a session.
- `graphicsRequirements` is the `XrGraphicsRequirementsOpenGLESKHR` output structure.

The `xrGetOpenGLESGraphicsRequirementsKHR` function identifies to the application the minimum OpenGL ES version requirement and the highest known tested OpenGL ES version. The runtime must return `XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING` (`XR_ERROR_VALIDATION_FAILURE` may be returned due to legacy behavior) on calls to `xrCreateSession` if `xrGetOpenGLESGraphicsRequirementsKHR` has not been called for the same `instance` and `systemId`.

Valid Usage (Implicit)

- The `XR_KHR_opengl_es_enable` extension must be enabled prior to calling `xrGetOpenGLESGraphicsRequirementsKHR`
- `instance` must be a valid `XrInstance` handle
- `graphicsRequirements` must be a pointer to an `XrGraphicsRequirementsOpenGLESKHR` structure
Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SYSTEM_INVALID

Issues

Version History

- Revision 1, 2018-05-07 (Mark Young)
  - Initial draft
- Revision 2, 2018-06-21 (Bryce Hutchings)
  - Add new xrGetOpenGLESGraphicsRequirementsKHR
- Revision 3, 2018-11-15 (Paul Pedriana)
  - Specified the swapchain texture coordinate origin.
- Revision 4, 2018-11-16 (Minmin Gong)
  - Specified Y direction and Z range in clip space
- Revision 5, 2019-01-25 (Robert Menzel)
  - Description updated
- Revision 6, 2019-07-12 (Martin Renschler)
  - Description updated
- Revision 7, 2020-08-06 (Bryce Hutchings)
  - Added new XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING error code
- Revision 8, 2021-08-27 (Paulo F. Gomes)
  - Document handling of XrSwapchainUsageFlags
12.18. XR_KHR_swapchain_usage_input_attachment_bit

Name String
   XR_KHR_swapchain_usage_input_attachment_bit

Extension Type
   Instance extension

Registered Extension Number
   166

Revision
   3

Extension and Version Dependencies
   • Requires OpenXR 1.0

Last Modified Date
   2021-05-11

IP Status
   No known IP claims.

Contributors
   Jakob Bornecrantz, Collabora
   Ryan Pavlik, Collabora

Overview
   This extension enables an application to specify that swapchain images should be created in a way so that they can be used as input attachments. At the time of writing this bit only affects Vulkan swapchains.

New Object Types

New Flag Types

New Enum Constants

XrSwapchainUsageFlagBits enumeration is extended with:
   • XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_KHR - indicates that the image format may be used as an input attachment.

New Enums
New Structures

New Functions

Issues

Version History

• Revision 1, 2020-07-23 (Jakob Bornecrantz)
  ◦ Initial draft

• Revision 2, 2020-07-24 (Jakob Bornecrantz)
  ◦ Added note about only affecting Vulkan
  ◦ Changed from MNDX to MND

• Revision 3, 2021-05-11 (Ryan Pavlik)
  ◦ Updated for promotion from MND to KHR

12.19. XR_KHR_visibility_mask

Name String

XR_KHR_visibility_mask

Extension Type

Instance extension

Registered Extension Number

32

Revision

2

Extension and Version Dependencies

• Requires OpenXR 1.0

Last Modified Date

2018-07-05

IP Status

No known IP claims.

Contributors

Paul Pedriana, Oculus
Alex Turner, Microsoft
Overview

This extension supports the providing of a per-view drawing mask for applications. The primary purpose of this is to enable performance improvements that result from avoiding drawing on areas that aren't visible to the user. A common occurrence in head-mounted VR hardware is that the optical system's frustum doesn't intersect precisely with the rectangular display it is viewing. As a result, it may be that there are parts of the display that aren't visible to the user, such as the corners of the display. In such cases it would be unnecessary for the application to draw into those parts.

New Object Types

New Flag Types

New Enum Constants

New Enums

XrVisibilityMaskTypeKHR identifies the different types of mask specification that is supported. The application can request a view mask in any of the formats identified by these types.

```c
typedef enum XrVisibilityMaskTypeKHR {
    XR_VISIBILITY_MASK_TYPE_HIDDEN_TRIANGLE_MESH_KHR = 1,
    XR_VISIBILITY_MASK_TYPE_VISIBLE_TRIANGLE_MESH_KHR = 2,
    XR_VISIBILITY_MASK_TYPE_LINE_LOOP_KHR = 3,
    XR_VISIBILITY_MASK_TYPE_MAX_ENUM_KHR = 0x7FFFFFFF
} XrVisibilityMaskTypeKHR;
```
Enumerant Descriptions

- **XR_VISIBILITY_MASK_TYPE_HIDDEN_TRIANGLE_MESH_KHR** refers to a two dimensional triangle mesh on the view surface which should not be drawn to by the application. *XrVisibilityMaskKHR* refers to a set of triangles identified by vertices and vertex indices. The index count will thus be a multiple of three. The triangle vertices will be returned in counter-clockwise order as viewed from the user perspective.

- **XR_VISIBILITY_MASK_TYPE_VISIBLE_TRIANGLE_MESH_KHR** refers to a two dimensional triangle mesh on the view surface which should be drawn to by the application. *XrVisibilityMaskKHR* refers to a set of triangles identified by vertices and vertex indices. The index count will thus be a multiple of three. The triangle vertices will be returned in counter-clockwise order as viewed from the user perspective.

- **XR_VISIBILITY_MASK_TYPE_LINE_LOOP_KHR** refers to a single multi-segmented line loop on the view surface which encompasses the view area which should be drawn by the application. It is the border that exists between the visible and hidden meshes identified by **XR_VISIBILITY_MASK_TYPE_HIDDEN_TRIANGLE_MESH_KHR** and **XR_VISIBILITY_MASK_TYPE_VISIBLE_TRIANGLE_MESH_KHR**. The line is counter-clockwise, contiguous, and non-self crossing, with the last point implicitly connecting to the first point. There is one vertex per point, the index count will equal the vertex count, and the indices will refer to the vertices.

New Structures

The *XrVisibilityMaskKHR* structure is an input/output struct which specifies the view mask.

```c
typedef struct XrVisibilityMaskKHR {
    XrStructureType    type;
    void*              next;
    uint32_t           vertexCapacityInput;
    uint32_t           vertexCountOutput;
    XrVector2f*        vertices;
    uint32_t           indexCapacityInput;
    uint32_t           indexCountOutput;
    uint32_t*          indices;
} XrVisibilityMaskKHR;
```
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **vertexCapacityInput** is the capacity of the vertices array, or 0 to indicate a request to retrieve the required capacity.
- **vertexCountOutput** is filled in by the runtime with the count of vertices written or the required capacity in the case that vertexCapacityInput or indexCapacityInput is insufficient.
- **vertices** is an array of vertices filled in by the runtime that specifies mask coordinates in the z=-1 plane of the rendered view—i.e. one meter in front of the view. When rendering the mask for use in a projection layer, these vertices must be transformed by the application's projection matrix used for the respective XrCompositionLayerProjectionView.
- **indexCapacityInput** is the capacity of the indices array, or 0 to indicate a request to retrieve the required capacity.
- **indexCountOutput** is filled in by the runtime with the count of indices written or the required capacity in the case that vertexCapacityInput or indexCapacityInput is insufficient.
- **indices** is an array of indices filled in by the runtime, specifying the indices of the mask geometry in the vertices array.

Valid Usage (Implicit)

- The XR_KHR_visibility_mask extension must be enabled prior to using XrVisibilityMaskKHR
- **type** must be XR_TYPE_VISIBILITY_MASK_KHR
- **next** must be NULL or a valid pointer to the next structure in a structure chain
- If vertexCapacityInput is not 0, vertices must be a pointer to an array of vertexCapacityInput XrVector2f structures
- If indexCapacityInput is not 0, indices must be a pointer to an array of indexCapacityInput uint32_t values

The XrEventDataVisibilityMaskChangedKHR structure specifies an event which indicates that a given view mask has changed. The application should respond to the event by calling xrGetVisibilityMaskKHR to retrieve the updated mask. This event is per-view, so if the masks for multiple views in a configuration change then multiple instances of this event will be sent to the application, one per view.
typedef struct XrEventDataVisibilityMaskChangedKHR {
    XrStructureType            type;
    const void*                next;
    XrSession                  session;
    XrViewConfigurationType    viewConfigurationType;
    uint32_t                   viewIndex;
} XrEventDataVisibilityMaskChangedKHR;

Member Descriptions

• **type** is the **XrStructureType** of this structure.
• **next** is **NULL** or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
• **session** is the **XrSession** for which the view mask has changed.
• **viewConfigurationType** is the view configuration whose mask has changed.
• **viewIndex** is the individual view within the view configuration to which the change refers.

Valid Usage (Implicit)

• The **XR_KHR_visibility_mask** extension **must** be enabled prior to using **XrEventDataVisibilityMaskChangedKHR**
• **type** **must** be **XR_TYPE_EVENT_DATA_VISIBILITY_MASK_CHANGED_KHR**
• **next** **must** be **NULL** or a valid pointer to the next structure in a structure chain
• **session** **must** be a valid **XrSession** handle
• **viewConfigurationType** **must** be a valid **XrViewConfigurationType** value

New Functions

The **xrGetVisibilityMaskKHR** function is defined as:

XrResult xrGetVisibilityMaskKHR(
    XrSession session,
    XrViewConfigurationType viewConfigurationType,
    uint32_t viewIndex,
    XrVisibilityMaskTypeKHR visibilityMaskType,
    XrVisibilityMaskKHR* visibilityMask);
Parameter Descriptions

- **session** is an XrSession handle previously created with `xrCreateSession`.
- **viewConfigurationType** is the view configuration from which to retrieve mask information.
- **viewIndex** is the individual view within the view configuration from which to retrieve mask information.
- **visibilityMaskType** is the type of visibility mask requested.
- **visibilityMask** is an input/output struct which specifies the view mask.

`xrGetVisibilityMaskKHR` retrieves the view mask for a given view. This function follows the two-call idiom for filling multiple buffers in a struct. Specifically, if either `vertexCapacityInput` or `indexCapacityInput` is 0, the runtime must respond as if both fields were set to 0, returning the vertex count and index count through `vertexCountOutput` or `indexCountOutput` respectively. If a view mask for the specified view isn't available, the returned vertex and index counts must be 0.

Valid Usage (Implicit)

- The `XR_KHR_visibility_mask` extension must be enabled prior to calling `xrGetVisibilityMaskKHR`
- **session** must be a valid XrSession handle
- **viewConfigurationType** must be a valid XrViewConfigurationType value
- **visibilityMaskType** must be a valid XrVisibilityMaskTypeKHR value
- **visibilityMask** must be a pointer to an XrVisibilityMaskKHR structure
Return Codes

Success

• XR_SUCCESS
• XR_SESSION_LOSS_PENDING

Failure

• XR_ERROR_FUNCTION_UNSUPPORTED
• XR_ERROR_VALIDATION_FAILURE
• XR_ERROR_RUNTIME_FAILURE
• XR_ERROR_HANDLE_INVALID
• XR_ERROR_INSTANCE_LOST
• XR_ERROR_SESSION_LOST
• XR_ERROR_SIZE_INSUFFICIENT
• XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED

Issues

Version History

• Revision 1, 2018-07-05 (Paul Pedriana)
  ◦ Initial version.
• Revision 2, 2019-07-15 (Alex Turner)
  ◦ Adjust two-call idiom usage.

12.20. XR_KHR_vulkan_enable

Name String

XR_KHR_vulkan_enable

Extension Type

Instance extension

Registered Extension Number

26

Revision

8
Overview

This extension enables the use of the Vulkan graphics API in an OpenXR runtime. Without this extension, the OpenXR runtime may not be able to use any Vulkan swapchain images.

This extension provides the mechanisms necessary for an application to generate a valid XrGraphicsBindingVulkanKHR structure in order to create a Vulkan-based XrSession. Note that during this process the application is responsible for creating all the required Vulkan objects.

This extension also provides mechanisms for the application to interact with images acquired by calling xrEnumerateSwapchainImages.

In order to expose the structures, types, and functions of this extension, you must define XR_USE_GRAPHICS_API_VULKAN before including the OpenXR platform header openxr_platform.h, in all portions of your library or application that include it.

Initialization

Some of the requirements for creating a valid XrGraphicsBindingVulkanKHR include correct initialization of a VkInstance, VkPhysicalDevice, and VkDevice.

A runtime may require that the VkInstance be initialized to a specific Vulkan API version. Additionally, the runtime may require a set of instance extensions to be enabled in the VkInstance. These requirements can be queried by the application using xrGetVulkanGraphicsRequirementsKHR and xrGetVulkanInstanceExtensionsKHR, respectively.

Similarly, the runtime may require the VkDevice to have a set of device extensions enabled, which can
be queried using \texttt{xrGetVulkanDeviceExtensionsKHR}.

In order to satisfy the \texttt{VkPhysicalDevice} requirements, the application can query \texttt{xrGetVulkanGraphicsDeviceKHR} to identify the correct \texttt{VkPhysicalDevice}.

Populating an \texttt{XrGraphicsBindingVulkanKHR} with a \texttt{VkInstance}, \texttt{VkDevice}, or \texttt{VkPhysicalDevice} that does not meet the requirements outlined by this extension \textbf{may} result in undefined behavior by the OpenXR runtime.

The API version, instance extension, device extension and physical device requirements only apply to the \texttt{VkInstance}, \texttt{VkDevice}, and \texttt{VkPhysicalDevice} objects which the application wishes to associate with an \texttt{XrGraphicsBindingVulkanKHR}.

\textbf{Concurrency}

Vulkan requires that concurrent access to a \texttt{VkQueue} from multiple threads be externally synchronized. Therefore, OpenXR functions that may access the \texttt{VkQueue} specified in the \texttt{XrGraphicsBindingVulkanKHR} must also be externally synchronized.

The list of OpenXR functions where the OpenXR runtime \textbf{may} access the \texttt{VkQueue} are:

- \texttt{xrBeginFrame}
- \texttt{xrEndFrame}
- \texttt{xrAcquireSwapchainImage}
- \texttt{xrReleaseSwapchainImage}

The runtime \textbf{must} not access the \texttt{VkQueue} in any OpenXR function that is not listed above or in an extension definition.

\textbf{Swapchain Image Layout}

When an application acquires a swapchain image by calling \texttt{xrAcquireSwapchainImage} in a session created using \texttt{XrGraphicsBindingVulkanKHR}, the OpenXR runtime \textbf{must} guarantee that:

- The image has a memory layout compatible with \texttt{VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL} for color images, or \texttt{VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL} for depth images.
- The \texttt{VkQueue} specified in \texttt{XrGraphicsBindingVulkanKHR} has ownership of the image.

When an application releases a swapchain image by calling \texttt{xrReleaseSwapchainImage}, in a session created using \texttt{XrGraphicsBindingVulkanKHR}, the OpenXR runtime \textbf{must} interpret the image as:

- Having a memory layout compatible with \texttt{VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL} for color images, or \texttt{VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL} for depth images.
- Being owned by the \texttt{VkQueue} specified in \texttt{XrGraphicsBindingVulkanKHR}.

The application is responsible for transitioning the swapchain image back to the image layout and...
queue ownership that the OpenXR runtime requires. If the image is not in a layout compatible with the above specifications the runtime may exhibit undefined behavior.

**Swapchain Flag Bits**

All `XrSwapchainUsageFlags` values passed in a session created using `XrGraphicsBindingVulkanKHR` must be interpreted as follows by the runtime, so that the returned swapchain images used by the application may be used as if they were created with at least the specified `VkImageUsageFlagBits` or `VkImageCreateFlagBits` set.

<table>
<thead>
<tr>
<th><code>XrSwapchainUsageFlagBits</code></th>
<th>Corresponding Vulkan flag bit</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>XR_SWAPCHAIN_USAGE_COLOR_ATTACHMENT_BIT</code></td>
<td><code>VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT</code></td>
</tr>
<tr>
<td><code>XR_SWAPCHAIN_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT</code></td>
<td><code>VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT</code></td>
</tr>
<tr>
<td><code>XR_SWAPCHAIN_USAGE_UNORDERED_ACCESS_BIT</code></td>
<td><code>VK_IMAGE_USAGE_STORAGE_BIT</code></td>
</tr>
<tr>
<td><code>XR_SWAPCHAIN_USAGE_TRANSFER_SRC_BIT</code></td>
<td><code>VK_IMAGE_USAGE_TRANSFER_SRC_BIT</code></td>
</tr>
<tr>
<td><code>XR_SWAPCHAIN_USAGE_TRANSFER_DST_BIT</code></td>
<td><code>VK_IMAGE_USAGE_TRANSFER_DST_BIT</code></td>
</tr>
<tr>
<td><code>XR_SWAPCHAIN_USAGE_SAMPLED_BIT</code></td>
<td><code>VK_IMAGE_USAGE_SAMPLED_BIT</code></td>
</tr>
<tr>
<td><code>XR_SWAPCHAIN_USAGE_MUTABLE_FORMAT_BIT</code></td>
<td><code>VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT</code></td>
</tr>
<tr>
<td><code>XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_KHR</code> (Added by <code>XR_KHR_swapchain_usage_input_attachment_bit</code> and only available when that extension is enabled)</td>
<td><code>VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT</code></td>
</tr>
</tbody>
</table>

**New Object Types**

**New Flag Types**

**New Enum Constants**

`XrStructureType` enumeration is extended with:

- `XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN_KHR`
- `XR_TYPE_GRAPHICS_BINDING_VULKAN_KHR`
- `XR_TYPE_SWAPCHAIN_IMAGE_VULKAN_KHR`

**New Enums**

**New Structures**

The following structures are provided to supply supporting runtimes the necessary information required to work with the Vulkan API executing on certain operating systems.

The `XrGraphicsBindingVulkanKHR` structure is defined as:
typedef struct XrGraphicsBindingVulkanKHR {
    XrStructureType     type;
    const void*         next;
    VkInstance          instance;
    VkPhysicalDevice    physicalDevice;
    VkDevice            device;
    uint32_t            queueFamilyIndex;
    uint32_t            queueIndex;
} XrGraphicsBindingVulkanKHR;

**Member Descriptions**

- **type** is the `XrStructureType` of this structure.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **instance** is a valid Vulkan `VkInstance`.
- **physicalDevice** is a valid Vulkan `VkPhysicalDevice`.
- **device** is a valid Vulkan `VkDevice`.
- **queueFamilyIndex** is a valid queue family index on `device`.
- **queueIndex** is a valid queue index on `device` to be used for synchronization.

When creating a Vulkan-backed `XrSession`, the application will provide a pointer to an `XrGraphicsBindingVulkanKHR` in the `next` chain of the `XrSessionCreateInfo`.

**Valid Usage**

- **instance** **must** have enabled a Vulkan API version in the range specified by `XrGraphicsBindingVulkanKHR`
- **instance** **must** have enabled all the instance extensions specified by `xrGetVulkanInstanceExtensionsKHR`
- **physicalDevice** `VkPhysicalDevice` **must** match the device specified by `xrGetVulkanGraphicsDeviceKHR`
- **device** **must** have enabled all the device extensions specified by `xrGetVulkanDeviceExtensionsKHR`
Valid Usage (Implicit)

- The `XR_KHR_vulkan_enable` extension must be enabled prior to using `XrGraphicsBindingVulkanKHR`
- `type` must be `XR_TYPE_GRAPHICS_BINDING_VULKAN_KHR`
- `next` must be `NULL` or a valid pointer to the next structure in a structure chain
- `instance` must be a valid `VkInstance` value
- `physicalDevice` must be a valid `VkPhysicalDevice` value
- `device` must be a valid `VkDevice` value

The `XrSwapchainImageVulkanKHR` structure is defined as:

```c
typedef struct XrSwapchainImageVulkanKHR {
    XrStructureType    type;
    void*              next;
    VkImage            image;
} XrSwapchainImageVulkanKHR;
```

Member Descriptions

- `type` is the `XrStructureType` of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- `image` is a valid Vulkan `VkImage` to use.

If a given session was created with `XrGraphicsBindingVulkanKHR`, the following conditions must apply.

- Calls to `xrEnumerateSwapchainImages` on an `XrSwapchain` in that session must return an array of `XrSwapchainImageVulkanKHR` structures.
- Whenever an OpenXR function accepts an `XrSwapchainImageBaseHeader` pointer as a parameter in that session, the runtime must also accept a pointer to an `XrSwapchainImageVulkanKHR`.

The OpenXR runtime must interpret the top-left corner of the swapchain image as the coordinate origin unless specified otherwise by extension functionality.

The OpenXR runtime must interpret the swapchain images in a clip space of positive Y pointing down, near Z plane at 0, and far Z plane at 1.
Valid Usage (Implicit)

- The XR_KHR_vulkan_enable extension must be enabled prior to using XrSwapchainImageVulkanKHR
- type must be XR_TYPE_SWAPCHAIN_IMAGE_VULKAN_KHR
- next must be NULL or a valid pointer to the next structure in a structure chain
- image must be a valid VkImage value

The XrGraphicsRequirementsVulkanKHR structure is defined as:

```c
typedef struct XrGraphicsRequirementsVulkanKHR {
  XrStructureType type;
  void*              next;
  XrVersion          minApiVersionSupported;
  XrVersion          maxApiVersionSupported;
} XrGraphicsRequirementsVulkanKHR;
```

**Member Descriptions**

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- minApiVersionSupported is the minimum Vulkan Instance API version that the runtime supports. Uses XR_MAKE_VERSION on major and minor API version, ignoring any patch version component.
- maxApiVersionSupported is the maximum Vulkan Instance API version that the runtime has been tested on and is known to support. Newer Vulkan Instance API versions might work if they are compatible. Uses XR_MAKE_VERSION on major and minor API version, ignoring any patch version component.

XrGraphicsRequirementsVulkanKHR is populated by xrGetVulkanGraphicsRequirementsKHR with the runtime’s Vulkan API version requirements.
Valid Usage (Implicit)

• The `XR_KHR_vulkan_enable` extension must be enabled prior to using `XrGraphicsRequirementsVulkanKHR`
• `type` must be `XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN_KHR`
• `next` must be `NULL` or a valid pointer to the next structure in a structure chain

New Functions

To query Vulkan API version requirements, call:

```c
XrResult xrGetVulkanGraphicsRequirementsKHR(
    XrInstance instance,
    XrSystemId systemId,
    XrGraphicsRequirementsVulkanKHR* graphicsRequirements);
```

Parameter Descriptions

• `instance` is an `XrInstance` handle previously created with `xrCreateInstance`.
• `systemId` is an `XrSystemId` handle for the system which will be used to create a session.
• `graphicsRequirements` is the `XrGraphicsRequirementsVulkanKHR` output structure.

The `xrGetVulkanGraphicsRequirementsKHR` function identifies to the application the minimum Vulkan version requirement and the highest known tested Vulkan version. The runtime must return `XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING` (`XR_ERROR_VALIDATION_FAILURE` may be returned due to legacy behavior) on calls to `xrCreateSession` if `xrGetVulkanGraphicsRequirementsKHR` has not been called for the same `instance` and `systemId`.

Valid Usage (Implicit)

• The `XR_KHR_vulkan_enable` extension must be enabled prior to calling `xrGetVulkanGraphicsRequirementsKHR`
• `instance` must be a valid `XrInstance` handle
• `graphicsRequirements` must be a pointer to an `XrGraphicsRequirementsVulkanKHR` structure
### Return Codes

**Success**
- XR_SUCCESS

**Failure**
- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_VALIDATION_FAILURE
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID
- XR_ERROR_INSTANCE_LOST
- XR_ERROR_SYSTEM_INVALID

Some computer systems may have multiple graphics devices, each of which may have independent external display outputs. XR systems that connect to such graphics devices are typically connected to a single device. Applications need to know what graphics device the XR system is connected to so that they can use that graphics device to generate XR images.

To identify what graphics device needs to be used for an instance and system, call:

```c
XrResult xrGetVulkanGraphicsDeviceKHR(
    XrInstance instance,         // instance is an XrInstance handle previously created with xrCreateInstance.
    XrSystemId systemId,         // systemId is an XrSystemId handle for the system which will be used to create a session.
    VkInstance vkInstance,       // vkInstance is a valid Vulkan VkInstance.
    VkPhysicalDevice* vkPhysicalDevice); // vkPhysicalDevice is a pointer to a VkPhysicalDevice value to populate.
```

xrGetVulkanGraphicsDeviceKHR function identifies to the application what graphics device (Vulkan VkPhysicalDevice) needs to be used. xrGetVulkanGraphicsDeviceKHR must be called prior to calling xrCreateSession, and the VkPhysicalDevice that xrGetVulkanGraphicsDeviceKHR returns should be passed to xrCreateSession in the XrGraphicsBindingVulkanKHR.
Valid Usage (Implicit)

- The `XR_KHR_vulkan_enable` extension must be enabled prior to calling `xrGetVulkanGraphicsDeviceKHR`
- `instance` must be a valid `XrInstance` handle
- `vkInstance` must be a valid `VkInstance` value
- `vkPhysicalDevice` must be a pointer to a `VkPhysicalDevice` value

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_FUNCTION_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SYSTEM_INVALID`

```c
XrResult xrGetVulkanInstanceExtensionsKHR(
    XrInstance instance,
    XrSystemId systemId,
    uint32_t bufferCapacityInput,
    uint32_t* bufferCountOutput,
    char* buffer);
```
### Parameter Descriptions

- **instance** is an `XrInstance` handle previously created with `xrCreateInstance`.
- **systemId** is an `XrSystemId` handle for the system which will be used to create a session.
- **bufferCapacityInput** is the capacity of the `buffer`, or 0 to indicate a request to retrieve the required capacity.
- **bufferCountOutput** is a pointer to the count of characters written (including terminating `\0`), or a pointer to the required capacity in the case that `bufferCapacityInput` is insufficient.
- **buffer** is a pointer to an array of characters, but can be `NULL` if `bufferCapacityInput` is 0. The format of the output is a single space (ASCII `\0x20`) delimited string of extension names.
- See **Buffer Size Parameters** chapter for a detailed description of retrieving the required `buffer` size.

### Valid Usage (Implicit)

- The `XR_KHR_vulkan_enable` extension must be enabled prior to calling `xrGetVulkanInstanceExtensionsKHR`.
- **instance** must be a valid `XrInstance` handle.
- **bufferCountOutput** must be a pointer to a `uint32_t` value.
- If `bufferCapacityInput` is not 0, **buffer** must be a pointer to an array of `bufferCapacityInput` char values.

### Return Codes

**Success**

- `XR_SUCCESS`

**Failure**

- `XR_ERROR_FUNCTION_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SIZE_INSUFFICIENT`
- `XR_ERROR_SYSTEM_INVALID`
XrResult xrGetVulkanDeviceExtensionsKHR(
    XrInstance instance,
    XrSystemId systemId,
    uint32_t bufferCapacityInput,
    uint32_t* bufferCountOutput,
    char* buffer);

Parameter Descriptions

- **instance** is an XrInstance handle previously created with xrCreateInstance.
- **systemId** is an XrSystemId handle for the system which will be used to create a session.
- **bufferCapacityInput** is the capacity of the buffer, or 0 to indicate a request to retrieve the required capacity.
- **bufferCountOutput** is a pointer to the count of characters written (including terminating \0), or a pointer to the required capacity in the case that bufferCapacityInput is insufficient.
- **buffer** is a pointer to an array of characters, but can be NULL if bufferCapacityInput is 0. The format of the output is a single space (ASCII 0x20) delimited string of extension names.
- See Buffer Size Parameters chapter for a detailed description of retrieving the required buffer size.

Valid Usage (Implicit)

- The XR_KHR_vulkan_enable extension must be enabled prior to calling xrGetVulkanDeviceExtensionsKHR.
- **instance** must be a valid XrInstance handle
- **bufferCountOutput** must be a pointer to a uint32_t value
- If bufferCapacityInput is not 0, **buffer** must be a pointer to an array of bufferCapacityInput char values
Return Codes

Success

• XR_SUCCESS

Failure

• XR_ERROR_FUNCTION_UNSUPPORTED
• XR_ERROR_VALIDATION_FAILURE
• XR_ERROR_RUNTIME_FAILURE
• XR_ERROR_HANDLE_INVALID
• XR_ERROR_INSTANCE_LOST
• XR_ERROR_SIZE_INSUFFICIENT
• XR_ERROR_SYSTEM_INVALID

Issues

Version History

• Revision 1, 2018-05-07 (Mark Young)
  ◦ Initial draft

• Revision 2, 2018-06-21 (Bryce Hutchings)
  ◦ Replace session parameter with instance and systemId parameters.
  ◦ Move xrGetVulkanDeviceExtensionsKHR, xrGetVulkanInstanceExtensionsKHR, and xrGetVulkanGraphicsDeviceKHR functions into this extension
  ◦ Add new XrGraphicsRequirementsVulkanKHR function.

• Revision 3, 2018-11-15 (Paul Pedriana)
  ◦ Specified the swapchain texture coordinate origin.

• Revision 4, 2018-11-16 (Minmin Gong)
  ◦ Specified Y direction and Z range in clip space

• Revision 5, 2019-01-24 (Robert Menzel)
  ◦ Description updated

• Revision 6, 2019-01-25 (Andres Rodriguez)
  ◦ Reword sections of the spec to shift requirements on to the runtime instead of the app

• Revision 7, 2020-08-06 (Bryce Hutchings)
  ◦ Added new XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING error code
12.21. XR_KHR_vulkan_enable2

Name String
XR_KHR_vulkan_enable2

Extension Type
Instance extension

Registered Extension Number
91

Revision
2

Extension and Version Dependencies
• Requires OpenXR 1.0

Last Modified Date
2020-05-04

IP Status
No known IP claims.

Contributors
Mark Young, LunarG
Paul Pedriana, Oculus
Ed Hutchins, Oculus
Andres Rodriguez, Valve
Dan Ginsburg, Valve
Bryce Hutchings, Microsoft
Minmin Gong, Microsoft
Robert Menzel, NVIDIA
Paulo Gomes, Samsung Electronics

12.21.1. Overview

This extension enables the use of the Vulkan graphics API in an OpenXR runtime. Without this extension, the OpenXR runtime may not be able to use any Vulkan swapchain images.

This extension provides the mechanisms necessary for an application to generate a valid XrGraphicsBindingVulkan2KHR structure in order to create a Vulkan-based XrSession.
This extension also provides mechanisms for the application to interact with images acquired by calling `xrEnumerateSwapchainImages`.

In order to expose the structures, types, and functions of this extension, you must define `XR_USE_GRAPHICS_API_VULKAN` before including the OpenXR platform header `openxr_platform.h`, in all portions of your library or application that include it.

Note
This extension is intended as an alternative to `XR_KHR_vulkan_enable`, and does not depend on it.

12.21.2. Initialization

When operating in Vulkan mode, the OpenXR runtime and the application will share the Vulkan queue described in the `XrGraphicsBindingVulkan2KHR` structure. This section of the document describes the mechanisms this extension exposes to ensure the shared Vulkan queue is compatible with the runtime and the application’s requirements.

Vulkan Version Requirements

First, a compatible Vulkan version must be agreed upon. To query the runtime’s Vulkan API version requirements an application will call:

```c
XrResult xrGetVulkanGraphicsRequirements2KHR(  
    XrInstance instance,  
    XrSystemId systemId,  
    XrGraphicsRequirementsVulkanKHR* graphicsRequirements);
```

The `xrGetVulkanGraphicsRequirements2KHR` function identifies to the application the runtime’s minimum Vulkan version requirement and the highest known tested Vulkan version. `xrGetVulkanGraphicsRequirements2KHR` must be called prior to calling `xrCreateSession`. The runtime must return `XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING` on calls to `xrCreateSession` if `xrGetVulkanGraphicsRequirements2KHR` has not been called for the same `instance` and `systemId`.

Parameter Descriptions

- `instance` is an `XrInstance` handle previously created with `xrCreateInstance`.
- `systemId` is an `XrSystemId` handle for the system which will be used to create a session.
- `graphicsRequirements` is the `XrGraphicsRequirementsVulkan2KHR` output structure.
Valid Usage (Implicit)

- The `XR_KHR_vulkan_enable2` extension **must** be enabled prior to calling `xrGetVulkanGraphicsRequirements2KHR`
- `instance` **must** be a valid `XrInstance` handle
- `graphicsRequirements` **must** be a pointer to an `XrGraphicsRequirementsVulkanKHR` structure

Return Codes

Success
- `XR_SUCCESS`

Failure
- `XR_ERROR_FUNCTION_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SYSTEM_INVALID`

The `XrGraphicsRequirementsVulkan2KHR` structure populated by `xrGetVulkanGraphicsRequirements2KHR` is defined as:

```c
// XrGraphicsRequirementsVulkan2KHR is an alias for XrGraphicsRequirementsVulkanKHR
typedef struct XrGraphicsRequirementsVulkanKHR {
    XrStructureType    type;
    void*              next;
    XrVersion          minApiVersionSupported;
    XrVersion          maxApiVersionSupported;
} XrGraphicsRequirementsVulkanKHR;

typedef XrGraphicsRequirementsVulkanKHR XrGraphicsRequirementsVulkan2KHR;
```
Member Descriptions

- **type** is the `XrStructureType` of this structure.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **minApiVersionSupported** is the minimum version of Vulkan that the runtime supports. Uses `XR_MAKE_VERSION` on major and minor API version, ignoring any patch version component.
- **maxApiVersionSupported** is the maximum version of Vulkan that the runtime has been tested on and is known to support. Newer Vulkan versions might work if they are compatible. Uses `XR_MAKE_VERSION` on major and minor API version, ignoring any patch version component.

Valid Usage (Implicit)

- The `XR_KHR_vulkan_enable2` extension must be enabled prior to using `XrGraphicsRequirementsVulkan2KHR`
- **Note:** `XrGraphicsRequirementsVulkan2KHR` is an alias for `XrGraphicsRequirementsVulkanKHR`, so implicit valid usage for `XrGraphicsRequirementsVulkanKHR` has been replicated below.
- **type** must be `XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN_KHR`
- **next** must be `NULL` or a valid pointer to the next structure in a structure chain

Vulkan Instance Creation

Second, a compatible `VkInstance` must be created. The `xrCreateVulkanInstanceKHR` entry point is a wrapper around `vkCreateInstance` intended for this purpose. When called, the runtime must aggregate the requirements specified by the application with its own requirements and forward the `VkInstance` creation request to the `vkCreateInstance` function pointer returned by `pfnGetInstanceProcAddr`.

```c
XrResult xrCreateVulkanInstanceKHR(
    XrInstance                                 instance,
    const XrVulkanInstanceCreateInfoKHR*       createInfo,
    VkInstance*                                vulkanInstance,
    VkResult*                                  vulkanResult);
```
Parameter Descriptions

- `instance` is an `XrInstance` handle previously created with `xrCreateInstance`.
- `createInfo` is an extensible input struct of type `XrVulkanInstanceCreateInfoKHR`.
- `vulkanInstance` points to a `VkInstance` handle to populate with the new Vulkan instance.
- `vulkanResult` points to a `VkResult` to populate with the result of the `vkCreateInstance` operation as returned by `pfnGetInstanceProcAddr`.

Valid Usage (Implicit)

- The `XR_KHR_vulkan_enable2` extension must be enabled prior to calling `xrCreateVulkanInstanceKHR`.
- `instance` must be a valid `XrInstance` handle.
- `createInfo` must be a pointer to a valid `XrVulkanInstanceCreateInfoKHR` structure.
- `vulkanInstance` must be a pointer to a `VkInstance` value.
- `vulkanResult` must be a pointer to a `VkResult` value.

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_FUNCTION_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_OUT_OF_MEMORY`
- `XR_ERROR_LIMIT_REACHED`
- `XR_ERROR_SYSTEM_INVALID`

The `XrVulkanInstanceCreateInfoKHR` structure contains the input parameters to `xrCreateVulkanInstanceKHR`. 
typedef struct XrVulkanInstanceCreateInfoKHR {
    XrStructureType type;
    const void* next;
    XrSystemId systemId;
    XrVulkanInstanceCreateFlagsKHR createFlags;
    PFN_vkGetInstanceProcAddr pfnGetInstanceProcAddr;
    const VkInstanceCreateInfo* vulkanCreateInfo;
    const VkAllocationCallbacks* vulkanAllocator;
} XrVulkanInstanceCreateInfoKHR;

Member Descriptions

- `systemId` is an `XrSystemId` handle for the system which will be used to create a session.
- `pfnGetInstanceProcAddr` is a function pointer to `vkGetInstanceProcAddr` or a compatible entry point.
- `vulkanCreateInfo` is the `VkInstanceCreateInfo` as specified by Vulkan.
- `vulkanAllocator` is the `VkAllocationCallbacks` as specified by Vulkan.

Valid Usage (Implicit)

- The `XR_KHR_vulkan_enable2` extension must be enabled prior to using `XrVulkanInstanceCreateInfoKHR`
- `type` must be `XR_TYPE_VULKAN_INSTANCE_CREATE_INFO_KHR`
- `next` must be `NULL` or a valid pointer to the next structure in a structure chain
- `createFlags` must be `0`
- `pfnGetInstanceProcAddr` must be a valid `PFN_vkGetInstanceProcAddr` value
- `vulkanCreateInfo` must be a pointer to a valid `VkInstanceCreateInfo` value
- If `vulkanAllocator` is not `NULL`, `vulkanAllocator` must be a pointer to a valid `VkAllocationCallbacks` value

typedef XrFlags64 XrVulkanInstanceCreateFlagsKHR;

Physical Device Selection

Third, a `VkPhysicalDevice` must be chosen. Some computer systems may have multiple graphics devices, each of which may have independent external display outputs. The runtime must report a
A `VkPhysicalDevice` that is compatible with the OpenXR implementation when `xrGetVulkanGraphicsDevice2KHR` is invoked. The application will use this `VkPhysicalDevice` to interact with the OpenXR runtime.

```c
XrResult xrGetVulkanGraphicsDevice2KHR(
    XrInstance                                  instance,
    const XrVulkanGraphicsDeviceGetInfoKHR*     getInfo,
    VkPhysicalDevice*                           vulkanPhysicalDevice);
```

**Parameter Descriptions**

- **instance** is an `XrInstance` handle previously created with `xrCreateInstance`.
- **getInfo** extensible input struct of type `XrVulkanGraphicsDeviceGetInfoKHR`
- **vulkanPhysicalDevice** is a pointer to a `VkPhysicalDevice` handle to populate.

**Valid Usage (Implicit)**

- The `XR_KHR_vulkan_enable2` extension **must** be enabled prior to calling `xrGetVulkanGraphicsDevice2KHR`
- **instance** **must** be a valid `XrInstance` handle
- **getInfo** **must** be a pointer to a valid `XrVulkanGraphicsDeviceGetInfoKHR` structure
- **vulkanPhysicalDevice** **must** be a pointer to a `VkPhysicalDevice` value

**Return Codes**

**Success**

- `XR_SUCCESS`

**Failure**

- `XR_ERROR_FUNCTION_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SYSTEM_INVALID`
The `XrVulkanGraphicsDeviceGetInfoKHR` structure contains the input parameters to `xrCreateVulkanInstanceKHR`.

```c
typedef struct XrVulkanGraphicsDeviceGetInfoKHR {
    XrStructureType    type;
    const void*        next;
    XrSystemId         systemId;
    VkInstance         vulkanInstance;
} XrVulkanGraphicsDeviceGetInfoKHR;
```

**Member Descriptions**

- `systemId` is an `XrSystemId` handle for the system which will be used to create a session.
- `vulkanInstance` is a valid Vulkan `VkInstance`.

**Valid Usage (Implicit)**

- The `XR_KHR_vulkan_enable2` extension must be enabled prior to using `XrVulkanGraphicsDeviceGetInfoKHR`
- `type` must be `XR_TYPE_VULKAN_GRAPHICS_DEVICE_GET_INFO_KHR`
- `next` must be `NULL` or a valid pointer to the next structure in a structure chain
- `vulkanInstance` must be a valid `VkInstance` value

**Vulkan Device Creation**

Fourth, a compatible `VkDevice` must be created. The `xrCreateVulkanDeviceKHR` entry point is a wrapper around `vkCreateDevice` intended for this purpose. When called, the runtime must aggregate the requirements specified by the application with its own requirements and forward the `VkDevice` creation request to the `vkCreateDevice` function pointer returned by `pfnGetInstanceProcAddr`.

```c
XrResult xrCreateVulkanDeviceKHR(
    XrInstance                                  instance,
    const XrVulkanDeviceCreateInfoKHR*          createInfo,
    VkDevice*                                   vulkanDevice,
    VkResult*                                   vulkanResult);
```
Parameter Descriptions

- **instance** is an `XrInstance` handle previously created with `xrCreateInstance`.
- **createInfo** extensible input struct of type `XrCreateVulkanDeviceCreateInfoKHR`.
- **vulkanDevice** points to a `VkDevice` handle to populate with the new Vulkan device.
- **vulkanResult** points to a `VkResult` to populate with the result of the `vkCreateDevice` operation as returned by `PFN.GetInstanceProcAddr`.

Valid Usage (Implicit)

- The `XR_KHR_vulkan_enable2` extension **must** be enabled prior to calling `xrCreateVulkanDeviceKHR`.
- **instance** **must** be a valid `XrInstance` handle.
- **createInfo** **must** be a pointer to a valid `XrVulkanDeviceCreateInfoKHR` structure.
- **vulkanDevice** **must** be a pointer to a `VkDevice` value.
- **vulkanResult** **must** be a pointer to a `VkResult` value.

Return Codes

Success

- **XR_SUCCESS**

Failure

- **XR_ERROR_FUNCTION_UNSUPPORTED**
- **XR_ERROR_VALIDATION_FAILURE**
- **XR_ERROR_RUNTIME_FAILURE**
- **XR_ERROR_HANDLE_INVALID**
- **XR_ERROR_INSTANCE_LOST**
- **XR_ERROR_OUT_OF_MEMORY**
- **XR_ERROR_LIMIT_REACHED**
- **XR_ERROR_SYSTEM_INVALID**

The `XrVulkanDeviceCreateInfoKHR` structure contains the input parameters to `xrCreateVulkanDeviceKHR`. 
typedef struct XrVulkanDeviceCreateInfoKHR {
    XrStructureType type;
    const void* next;
    XrSystemId systemId;
    XrVulkanDeviceCreateFlagsKHR createFlags;
    PFN_vkGetInstanceProcAddr pfnGetInstanceProcAddr;
    VkPhysicalDevice vulkanPhysicalDevice;
    const VkDeviceCreateInfo* vulkanCreateInfo;
    const VkAllocationCallbacks* vulkanAllocator;
} XrVulkanDeviceCreateInfoKHR;

Member Descriptions

- `systemId` is an `XrSystemId` handle for the system which will be used to create a session.
- `pfnGetInstanceProcAddr` is a function pointer to `vkGetInstanceProcAddr` or a compatible entry point.
- `vulkanPhysicalDevice` must match `xrGetVulkanGraphicsDeviceKHR`.
- `vulkanCreateInfo` is the `VkDeviceCreateInfo` as specified by Vulkan.
- `vulkanAllocator` is the `VkAllocationCallbacks` as specified by Vulkan.

Valid Usage (Implicit)

- The `XR_KHR_vulkan_enable2` extension must be enabled prior to using `XrVulkanDeviceCreateInfoKHR`.
- `type` must be `XR_TYPE_VULKAN_DEVICE_CREATE_INFO_KHR`.
- `next` must be `NULL` or a valid pointer to the next structure in a structure chain.
- `createFlags` must be `0`.
- `pfnGetInstanceProcAddr` must be a valid `PFN_vkGetInstanceProcAddr` value.
- `vulkanPhysicalDevice` must be a valid `VkPhysicalDevice` value.
- `vulkanCreateInfo` must be a pointer to a valid `VkDeviceCreateInfo` value.
- If `vulkanAllocator` is not `NULL`, `vulkanAllocator` must be a pointer to a valid `VkAllocationCallbacks` value.

typedef XrFlags64 XrVulkanDeviceCreateFlagsKHR;
If the `vulkanPhysicalDevice` parameter does not match the output of `xrGetVulkanGraphicsDeviceKHR`, then the runtime **must** return `XR_ERROR_HANDLE_INVALID`.

Queue Selection

Last, the application selects a `VkQueue` from the `VkDevice` that has the `VK_QUEUE_GRAPHICS_BIT` set.

**Note**

The runtime may schedule work on the `VkQueue` specified in the binding, or it may schedule work on any hardware queue in a foreign logical device.

Vulkan Graphics Binding

When creating a Vulkan-backed `XrSession`, the application will chain a pointer to an `XrGraphicsBindingVulkan2KHR` to the `XrSessionCreateInfo` parameter of `xrCreateSession`. With the data collected in the previous sections, the application now has all the necessary information to populate an `XrGraphicsBindingVulkan2KHR` structure for session creation.

```
// XrGraphicsBindingVulkan2KHR is an alias for XrGraphicsBindingVulkanKHR
typedef struct XrGraphicsBindingVulkanKHR {
    XrStructureType     type;
    const void*         next;
    VkInstance          instance;
    VkPhysicalDevice    physicalDevice;
    VkDevice            device;
    uint32_t            queueFamilyIndex;
    uint32_t            queueIndex;
} XrGraphicsBindingVulkanKHR;

typedef XrGraphicsBindingVulkanKHR XrGraphicsBindingVulkan2KHR;
```
Member Descriptions

- **type** is the `XrStructureType` of this structure.
- **next** is `NULL` or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **instance** is a valid Vulkan `VkInstance`.
- **physicalDevice** is a valid Vulkan `VkPhysicalDevice`.
- **device** is a valid Vulkan `VkDevice`.
- **queueFamilyIndex** is a valid queue family index on `device`.
- **queueIndex** is a valid queue index on `device` to be used for synchronization.

Valid Usage

- **instance** **must** have enabled a Vulkan API version in the range specified by `xrGetVulkanGraphicsRequirements2KHR`
- **instance** **must** have been created using `xrCreateVulkanInstanceKHR`
- **physicalDevice** `VkPhysicalDevice` **must** match the device specified by `xrGetVulkanGraphicsDevice2KHR`
- **device** **must** have been created using `xrCreateVulkanDeviceKHR`

Valid Usage (Implicit)

- The `XR_KHR_vulkan_enable2` extension **must** be enabled prior to using `XrGraphicsBindingVulkan2KHR`
- Note: `XrGraphicsBindingVulkan2KHR` is an alias for `XrGraphicsBindingVulkanKHR`, so implicit valid usage for `XrGraphicsBindingVulkanKHR` has been replicated below.

- **type** **must** be `XR_TYPE_GRAPHICS_BINDING_VULKAN_KHR`
- **next** **must** be `NULL` or a valid pointer to the next structure in a structure chain
- **instance** **must** be a valid `VkInstance` value
- **physicalDevice** **must** be a valid `VkPhysicalDevice` value
- **device** **must** be a valid `VkDevice` value

Populating an `XrGraphicsBindingVulkan2KHR` structure with a member that does not meet the requirements outlined by this extension **may** result in undefined behavior by the OpenXR runtime.

The requirements outlined in this extension only apply to the `VkInstance`, `VkDevice`, `VkPhysicalDevice`
and `VkQueue` objects which the application wishes to associate with an `XrGraphicsBindingVulkan2KHR`.

### 12.21.3. Concurrency

Vulkan requires that concurrent access to a `VkQueue` from multiple threads be externally synchronized. Therefore, OpenXR functions that may access the `VkQueue` specified in the `XrGraphicsBindingVulkan2KHR` **must** also be externally synchronized by the OpenXR application.

The list of OpenXR functions where the OpenXR runtime **may** access the `VkQueue` are:

- `xrBeginFrame`
- `xrEndFrame`
- `xrAcquireSwapchainImage`
- `xrReleaseSwapchainImage`

The runtime **must** not access the `VkQueue` in any OpenXR function that is not listed above or in an extension definition.

Failure by the application to synchronize access to `VkQueue` **may** result in undefined behavior in the OpenXR runtime.

### 12.21.4. Swapchain Interactions

#### Swapchain Images

When an application interacts with `XrSwapchainImageBaseHeader` structures in a Vulkan-backed `XrSession`, the application can interpret these to be `XrSwapchainImageVulkan2KHR` structures. These are defined as:

```c
// XrSwapchainImageVulkan2KHR is an alias for XrSwapchainImageVulkanKHR
typedef struct XrSwapchainImageVulkanKHR {
    XrStructureType    type;
    void*              next;
    VkImage            image;
} XrSwapchainImageVulkanKHR;

typedef XrSwapchainImageVulkanKHR XrSwapchainImageVulkan2KHR;
```
Member Descriptions

- **type** is the XrStructureType of this structure.
- **next** is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
- **image** is a valid Vulkan VkImage to use.

If a given session was created with XrGraphicsBindingVulkan2KHR, the following conditions must apply.

- Calls to xrEnumerateSwapchainImages on an XrSwapchain in that session must return an array of XrSwapchainImageVulkan2KHR structures.
- Whenever an OpenXR function accepts an XrSwapchainImageBaseHeader pointer as a parameter in that session, the runtime must also accept a pointer to an XrSwapchainImageVulkan2KHR.

The OpenXR runtime must interpret the top-left corner of the swapchain image as the coordinate origin unless specified otherwise by extension functionality.

The OpenXR runtime must interpret the swapchain images in a clip space of positive Y pointing down, near Z plane at 0, and far Z plane at 1.

Valid Usage (Implicit)

- The XR_KHR_vulkan_enable2 extension must be enabled prior to using XrSwapchainImageVulkan2KHR
- **Note:** XrSwapchainImageVulkan2KHR is an alias for XrSwapchainImageVulkanKHR, so implicit valid usage for XrSwapchainImageVulkanKHR has been replicated below.

  - **type** must be XR_TYPE_SWAPCHAIN_IMAGE_VULKAN_KHR
  - **next** must be NULL or a valid pointer to the next structure in a structure chain
  - **image** must be a valid VkImage value

Swapchain Image Layout

When an application acquires a swapchain image by calling xrAcquireSwapchainImage in a session created using XrGraphicsBindingVulkan2KHR, the OpenXR runtime must guarantee that:

- The image has a memory layout compatible with VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL for color images, or VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL for depth images.
- The VkQueue specified in XrGraphicsBindingVulkan2KHR has ownership of the image.

When an application releases a swapchain image by calling xrReleaseSwapchainImage, in a session
created using XrGraphicsBindingVulkan2KHR, the OpenXR runtime must interpret the image as:

- Having a memory layout compatible with VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL for color images, or VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL for depth images.
- Being owned by the VkQueue specified in XrGraphicsBindingVulkan2KHR.
- Being referenced by command buffers submitted to the VkQueue specified in XrGraphicsBindingVulkan2KHR which have not yet completed execution.

The application is responsible for transitioning the swapchain image back to the image layout and queue ownership that the OpenXR runtime requires. If the image is not in a layout compatible with the above specifications the runtime may exhibit undefined behavior.

**Swapchain Flag Bits**

All XrSwapchainUsageFlags values passed in a session created using XrGraphicsBindingVulkan2KHR must be interpreted as follows by the runtime, so that the returned swapchain images used by the application may be used as if they were created with at least the specified VkImageUsageFlagBits or VkImageCreateFlagBits set.

<table>
<thead>
<tr>
<th>XrSwapchainUsageFlagBits</th>
<th>Corresponding Vulkan flag bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>XR_SWAPCHAIN_USAGE_COLOR_ATTACHMENT_BIT</td>
<td>VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT</td>
</tr>
<tr>
<td>XR_SWAPCHAIN_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
<td>VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
</tr>
<tr>
<td>XR_SWAPCHAIN_USAGE_UNORDERED_ACCESS_BIT</td>
<td>VK_IMAGE_USAGE_STORAGE_BIT</td>
</tr>
<tr>
<td>XR_SWAPCHAIN_USAGE_TRANSFER_SRC_BIT</td>
<td>VK_IMAGE_USAGE_TRANSFER_SRC_BIT</td>
</tr>
<tr>
<td>XR_SWAPCHAIN_USAGE_TRANSFER_DST_BIT</td>
<td>VK_IMAGE_USAGE_TRANSFER_DST_BIT</td>
</tr>
<tr>
<td>XR_SWAPCHAIN_USAGE_SAMPLED_BIT</td>
<td>VK_IMAGE_USAGE_SAMPLED_BIT</td>
</tr>
<tr>
<td>XR_SWAPCHAIN_USAGE_MUTABLE_FORMAT_BIT</td>
<td>VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT</td>
</tr>
<tr>
<td>XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_KHR</td>
<td>VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT_KHR</td>
</tr>
</tbody>
</table>

**12.21.5. Appendix**

**Questions**

1. Should the xrCreateVulkanDeviceKHR and xrCreateVulkanInstanceKHR functions have an output parameter that returns the combined list of parameters used to create the Vulkan device/instance?
   - No. If the application is interested in capturing this data it can set the pfnGetInstanceProcAddr parameter to a local callback that captures the relevant information.
Quick Reference

New Enum Constants

**XrStructureType** enumeration is extended with:

- **XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN2_KHR** (alias of **XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN_KHR**)
- **XR_TYPE_GRAPHICS_BINDING_VULKAN2_KHR** (alias of **XR_TYPE_GRAPHICS_BINDING_VULKAN_KHR**)
- **XR_TYPE_SWAPCHAIN_IMAGE_VULKAN2_KHR** (alias of **XR_TYPE_SWAPCHAIN_IMAGE_VULKAN_KHR**)

New Structures

- **XrVulkanInstanceCreateInfoKHR**
- **XrVulkanDeviceCreateInfoKHR**
- **XrVulkanGraphicsDeviceGetInfoKHR**
- **XrGraphicsBindingVulkan2KHR** (alias of **XrGraphicsBindingVulkanKHR**)
- **XrSwapchainImageVulkan2KHR** (alias of **XrSwapchainImageVulkanKHR**)
- **XrGraphicsRequirementsVulkan2KHR** (alias of **XrGraphicsRequirementsVulkanKHR**)

New Functions

- **xrCreateVulkanInstanceKHR**
- **xrCreateVulkanDeviceKHR**
- **xrGetVulkanGraphicsDevice2KHR**
- **xrGetVulkanGraphicsRequirements2KHR**

Version History

- Revision 1, 2020-05-04 (Andres Rodriguez)
  - Initial draft
- Revision 2, 2021-01-21 (Ryan Pavlik)
  - Document mapping for **XrSwapchainUsageFlags**

12.22. **XR_KHR_vulkan_swapchain_format_list**

Name String

**XR_KHR_vulkan_swapchain_format_list**

Extension Type

Instance extension
Overview

Vulkan has the `VK_KHR_image_format_list` extension which allows applications to tell the `vkCreateImage` function which formats the application intends to use when `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` is specified. This OpenXR extension exposes that Vulkan extension to OpenXR applications. In the same way that a Vulkan-based application can pass a `VkImageFormatListCreateInfo` struct to the `vkCreateImage` function, an OpenXR application can pass an identically configured `XrVulkanSwapchainFormatListCreateInfoKHR` structure to `xrCreateSwapchain`.

Applications using this extension to specify more than one swapchain format must create OpenXR swapchains with the `XR_SWAPCHAIN_USAGE_MUTABLE_FORMAT_BIT` bit set.

Runtimes implementing this extension must support the `XR_KHR_vulkan_enable` or the `XR_KHR_vulkan_enable2` extension. When `XR_KHR_vulkan_enable` is used, the runtime must add `VK_KHR_image_format_list` to the list of extensions enabled in `xrCreateVulkanDeviceKHR`.

New Object Types

New Flag Types

New Enum Constants

`XrStructureType` enumeration is extended with:

```plaintext
XR_TYPE_VULKAN_SWAPCHAIN_FORMAT_LIST_CREATE_INFO_KHR
```
New Enums

New Structures

typedef struct XrVulkanSwapchainFormatListCreateInfoKHR {
    XrStructureType    type;
    const void*        next;
    uint32_t           viewFormatCount;
    const VkFormat*    viewFormats;
} XrVulkanSwapchainFormatListCreateInfoKHR;

Member Descriptions

• type is the XrStructureType of this structure.
• next is NULL or a pointer to the next structure in a structure chain. No such structures are defined in core OpenXR or this extension.
• viewFormatCount is the number of view formats passed in viewFormats.
• viewFormats is an array of VkFormat.

Valid Usage (Implicit)

• The XR_KHR_vulkan_swapchain_format_list extension must be enabled prior to using XrVulkanSwapchainFormatListCreateInfoKHR
• type must be XR_TYPE_VULKAN_SWAPCHAIN_FORMAT_LIST_CREATE_INFO_KHR
• next must be NULL or a valid pointer to the next structure in a structure chain
• If viewFormatCount is not 0, viewFormats must be a pointer to an array of viewFormatCount valid VkFormat values

New Functions

Issues

Version History

• Revision 1, 2017-09-13 (Paul Pedriana)
  ◦ Initial proposal.
• Revision 2, 2018-06-21 (Bryce Hutchings)
Update reference of `XR_KHR_vulkan_extension_requirements` to `XR_KHR_vulkan_enable`.

Revision 3, 2020-01-01 (Andres Rodriguez)
  - Update for `XR_KHR_vulkan_enable2`

Revision 4, 2021-01-21 (Ryan Pavlik)
  - Fix reference to the mutable-format bit in Vulkan.

### 12.23. `XR_KHR_win32_convert_performance_counter_time`

**Name String**

`XR_KHR_win32_convert_performance_counter_time`

**Extension Type**

Instance extension

**Registered Extension Number**

36

**Revision**

1

**Extension and Version Dependencies**

- Requires OpenXR 1.0

**Last Modified Date**

2019-01-24

**IP Status**

No known IP claims.

**Contributors**

- Paul Pedriana, Oculus
- Bryce Hutchings, Microsoft

**Overview**

This extension provides two functions for converting between the Windows performance counter (QPC) time stamps and `XrTime`. The `xrConvertWin32PerformanceCounterToTimeKHR` function converts from Windows performance counter time stamps to `XrTime`, while the `xrConvertTimeToWin32PerformanceCounterKHR` function converts `XrTime` to Windows performance counter time stamps. The primary use case for this functionality is to be able to synchronize events between the local system and the OpenXR system.
To convert from a Windows performance counter time stamp to XrTime, call:

```c
XrResult xrConvertWin32PerformanceCounterToTimeKHR(
    XrInstance instance,
    const LARGE_INTEGER* performanceCounter,
    XrTime* time);
```

### Parameter Descriptions

- **instance** is an XrInstance handle previously created with xrCreateInstance.
- **performanceCounter** is a time returned by QueryPerformanceCounter.
- **time** is the resulting XrTime that is equivalent to the performanceCounter.

The `xrConvertWin32PerformanceCounterToTimeKHR` function converts a time stamp obtained by the QueryPerformanceCounter Windows function to the equivalent XrTime.

If the output time cannot represent the input performanceCounter, the runtime must return XR_ERROR_TIME_INVALID.

### Valid Usage (Implicit)

- The XR_KHR_win32_convert_performance_counter_time extension must be enabled prior to calling `xrConvertWin32PerformanceCounterToTimeKHR`.
- **instance** must be a valid XrInstance handle
- **performanceCounter** must be a pointer to a valid LARGE_INTEGER value
- **time** must be a pointer to an XrTime value
Return Codes

Success
• XR_SUCCESS

Failure
• XR_ERROR_FUNCTION_UNSUPPORTED
• XR_ERROR_VALIDATION_FAILURE
• XR_ERROR_RUNTIME_FAILURE
• XR_ERROR_HANDLE_INVALID
• XR_ERROR_INSTANCE_LOST
• XR_ERROR_TIME_INVALID

To convert from XrTime to a Windows performance counter time stamp, call:

```c
XrResult xrConvertTimeToWin32PerformanceCounterKHR(
    XrInstance                                  instance,
    XrTime                                      time,
    LARGE_INTEGER*                              performanceCounter);
```

Parameter Descriptions

• instance is an XrInstance handle previously created with xrCreateInstance.
• time is an XrTime.
• performanceCounter is the resulting Windows performance counter time stamp that is equivalent to the time.

The xrConvertTimeToWin32PerformanceCounterKHR function converts an XrTime to time as if generated by the QueryPerformanceCounter Windows function.

If the output performanceCounter cannot represent the input time, the runtime must return XR_ERROR_TIME_INVALID.
Valid Usage (Implicit)

- The `XR_KHR_win32_convert_performance_counter_time` extension must be enabled prior to calling `xrConvertTimeToWin32PerformanceCounterKHR`
- `instance` must be a valid `XrInstance` handle
- `performanceCounter` must be a pointer to a `LARGE_INTEGER` value

Return Codes

**Success**
- `XR_SUCCESS`

**Failure**
- `XR_ERROR_FUNCTION_UNSUPPORTED`
- `XR_ERROR_VALIDATION_FAILURE`
- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_HANDLE_INVALID`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_TIME_INVALID`

**Issues**

**Version History**

- Revision 1, 2019-01-24 (Paul Pedriana)
  - Initial draft
Appendix

Code Style Conventions

These are the code style conventions used in this specification to define the API.

<table>
<thead>
<tr>
<th>Conventions</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Enumerants and defines are all upper case with words separated by an underscore.</td>
</tr>
<tr>
<td>• Neither type, function or member names contain underscores.</td>
</tr>
<tr>
<td>• Structure members start with a lower case character and each consecutive word starts with a capital.</td>
</tr>
<tr>
<td>• A structure that has a pointer to an array includes a structure member named <code>fooCount</code> of type <code>uint32_t</code> to denote the number of elements in the array of <code>foo</code>.</td>
</tr>
<tr>
<td>• A structure that has a pointer to an array lists the <code>fooCount</code> member first and then the array pointer.</td>
</tr>
<tr>
<td>• Unless a negative value has a clearly defined meaning all <code>fooCount</code> variables are unsigned.</td>
</tr>
<tr>
<td>• Function parameters that are modified are always listed last.</td>
</tr>
</tbody>
</table>

Prefixes are used in the API to denote specific semantic meaning of names, or as a label to avoid name clashes, and are explained here:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>XR_</code></td>
<td>Enumerants and defines are prefixed with these characters.</td>
</tr>
<tr>
<td><code>Xr</code></td>
<td>Non-function-pointer types are prefixed with these characters.</td>
</tr>
<tr>
<td><code>xr</code></td>
<td>Functions are prefixed with these characters.</td>
</tr>
<tr>
<td><code>PFN_xr</code></td>
<td>Function pointer types are prefixed with these characters.</td>
</tr>
</tbody>
</table>

Application Binary Interface

This section describes additional definitions and conventions that define the application binary interface.
typedef enum XrStructureType {
    XR_TYPE_UNKNOWN = 0,
    XR_TYPE_API_LAYER_PROPERTIES = 1,
    XR_TYPE_EXTENSION_PROPERTIES = 2,
    XR_TYPE_INSTANCE_CREATE_INFO = 3,
    XR_TYPE_SYSTEM_GET_INFO = 4,
    XR_TYPE_SYSTEM_PROPERTIES = 5,
    XR_TYPE_VIEW_LOCATE_INFO = 6,
    XR_TYPE_VIEW = 7,
    XR_TYPE_SESSION_CREATE_INFO = 8,
    XR_TYPE_SWAPCHAIN_CREATE_INFO = 9,
    XR_TYPE_SESSION_BEGIN_INFO = 10,
    XR_TYPE_VIEW_STATE = 11,
    XR_TYPE_FRAME_END_INFO = 12,
    XR_TYPE_HAPTIC_VIBRATION = 13,
    XR_TYPE_EVENT_DATA_BUFFER = 16,
    XR_TYPE_EVENT_DATA_INSTANCE_LOSS_PENDING = 17,
    XR_TYPE_EVENT_DATA_SESSION_STATE_CHANGED = 18,
    XR_TYPE_ACTION_STATE_BOOLEAN = 23,
    XR_TYPE_ACTION_STATE_FLOAT = 24,
    XR_TYPE_ACTION_STATE_VECTOR2F = 25,
    XR_TYPE_ACTION_STATE_POSE = 27,
    XR_TYPE_ACTION_SET_CREATE_INFO = 28,
    XR_TYPE_ACTION_CREATE_INFO = 29,
    XR_TYPE_INSTANCE_PROPERTIES = 32,
    XR_TYPE_FRAME_WAIT_INFO = 33,
    XR_TYPE_COMPOSITION_LAYER_PROJECTION = 35,
    XR_TYPE_COMPOSITION_LAYER_QUAD = 36,
    XR_TYPE_REFERENCE_SPACE_CREATE_INFO = 37,
    XR_TYPE_ACTION_SPACE_CREATE_INFO = 38,
    XR_TYPE_EVENT_DATA_REFERENCE_SPACE_CHANGE_PENDING = 40,
    XR_TYPE_VIEW_CONFIGURATION_VIEW = 41,
    XR_TYPE_SPACE_LOCATION = 42,
    XR_TYPE_SPACE_VELOCITY = 43,
    XR_TYPE_FRAME_STATE = 44,
    XR_TYPE_VIEW_CONFIGURATION_PROPERTIES = 45,
    XR_TYPE_FRAME_BEGIN_INFO = 46,
    XR_TYPE_COMPOSITION_LAYER_PROJECTION_VIEW = 48,
    XR_TYPE_EVENT_DATA_EVENTS_LOST = 49,
    XR_TYPE_INTERACTION_PROFILE_SUGGESTED_BINDING = 51,
    XR_TYPE_EVENT_DATA_INTERACTION_PROFILE_CHANGED = 52,
    XR_TYPE_INTERACTION_PROFILE_STATE = 53,
    XR_TYPE_SWAPCHAIN_IMAGE_ACQUIRE_INFO = 55,
}
XR_TYPE_SWAPCHAIN_IMAGE_WAIT_INFO = 56,
XR_TYPE_SWAPCHAIN_IMAGE_RELEASE_INFO = 57,
XR_TYPE_ACTION_STATE_GET_INFO = 58,
XR_TYPE_HAPTIC_ACTION_INFO = 59,
XR_TYPE_SESSION_ACTION_SETS_ATTACH_INFO = 60,
XR_TYPE_ACTIONS_SYNC_INFO = 61,
XR_TYPE_BOUND_SOURCES_FOR_ACTION_ENUMERATE_INFO = 62,
XR_TYPE_INPUT_SOURCE_LOCALIZED_NAME_GET_INFO = 63,
XR_TYPE_COMPOSITION_LAYER_CUBE_KHR = 1000006000,
XR_TYPE_INSTANCE_CREATE_INFO_ANDROID_KHR = 1000008000,
XR_TYPE_COMPOSITION_LAYER_DEPTH_INFO_KHR = 1000010000,
XR_TYPE_VULKAN_SWAPCHAIN_FORMAT_LIST_CREATE_INFO_KHR = 1000014000,
XR_TYPE_COMPOSITION_LAYER_CYLINDER_KHR = 1000017000,
XR_TYPE_COMPOSITION_LAYER_EQUIRECT_KHR = 1000018000,
XR_TYPE_GRAPHICS_BINDING_OPENGL_WAYLAND_KHR = 1000023003,
XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_KHR = 1000023004,
XR_TYPE_GRAPHICS_BINDING_OPENGL_XLIB_KHR = 1000023002,
XR_TYPE_GRAPHICS_BINDING_OPENGL_XCB_KHR = 1000023001,
XR_TYPE_GRAPHICS_BINDING_OPENGL_ES_ANDROID_KHR = 1000024001,
XR_TYPE_GRAPHICS_BINDING_OPENGL_ES_KHR = 1000024002,
XR_TYPE_GRAPHICS_REQUIREMENTS_OPENGL_KHR = 1000023005,
XR_TYPE_GRAPHICS_BINDING_VULKAN_KHR = 1000025000,
XR_TYPE_SWAPCHAIN_IMAGE_VULKAN_KHR = 1000025001,
XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN_KHR = 1000025002,
XR_TYPE_GRAPHICS_BINDING_D3D11_KHR = 1000027000,
XR_TYPE_SWAPCHAIN_IMAGE_D3D11_KHR = 1000027001,
XR_TYPE_GRAPHICS_REQUIREMENTS_D3D11_KHR = 1000027002,
XR_TYPE_GRAPHICS_BINDING_D3D12_KHR = 1000028000,
XR_TYPE_SWAPCHAIN_IMAGE_D3D12_KHR = 1000028001,
XR_TYPE_GRAPHICS_REQUIREMENTS_D3D12_KHR = 1000028002,
XR_TYPE_VISIBILITY_MASK_KHR = 1000031000,
XR_TYPE_EVENT_DATA_VISIBILITY_MASK_CHANGED_KHR = 1000031001,
XR_TYPE_COMPOSITION_LAYER_COLOR_SCALE_BIAS_KHR = 1000034000,
XR_TYPE_LOADER_INIT_INFO_ANDROID_KHR = 1000089000,
XR_TYPE_VULKAN_INSTANCE_CREATE_INFO_KHR = 1000090000,
XR_TYPE_VULKAN_DEVICE_CREATE_INFO_KHR = 1000090001,
XR_TYPE_VULKAN_GRAPHICS_DEVICE_GET_INFO_KHR = 1000090003,
XR_TYPE_COMPOSITION_LAYER_EQUIRECT2_KHR = 1000091000,
XR_TYPE_BINDING_MODIFICATIONS_KHR = 1000120000,
XR_TYPE_GRAPHICS_BINDING_VULKAN2_KHR = XR_TYPE_GRAPHICS_BINDING_VULKAN_KHR,
XR_TYPE_SWAPCHAIN_IMAGE_VULKAN2_KHR = XR_TYPE_SWAPCHAIN_IMAGE_VULKAN_KHR,
XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN2_KHR = XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN_KHR,
XR_STRUCTURE_TYPE_MAX_ENUM = 0x7FFFFFFF

} XrStructureType;
Most structures containing type members have a value of type matching the type of the structure, as described more fully in Valid Usage for Structure Types.

Note that all extension enums begin at the extension enum base of $10$ (base 10). Each extension is assigned a block of 1000 enums, starting at the enum base and arranged by the extension’s index.

For example, if an extension with index 5 wants to use an enum value of 3, the final enum is computed by:

$$\text{enum} = \text{enum\_base} + (\text{enum\_index} - 1) \times 1000 + \text{enum\_value} = 1000000000 + 4 \times 1000 + 3$$

**Flag Types**

Flag types are all bitmasks aliasing the base type XrFlags64 and with corresponding bit flag types defining the valid bits for that flag, as described in Valid Usage for Flags. Flag types supported by the API include:

```c
typedef XrFlags64 XrCompositionLayerFlags;
```

```c
typedef XrFlags64 XrInputSourceLocalizedNameFlags;
```

```c
typedef XrFlags64 XrInstanceCreateFlags;
```

```c
typedef XrFlags64 XrSessionCreateFlags;
```

```c
typedef XrFlags64 XrSpaceLocationFlags;
```

```c
typedef XrFlags64 XrSpaceVelocityFlags;
```
typedef XrFlags64 XrSwapchainCreateFlags;

typedef XrFlags64 XrSwapchainUsageFlags;

typedef XrFlags64 XrViewStateFlags;

General Macro Definitions

This API is defined in C and uses "C" linkage. The openxr.h header file is opened with:

```c
#ifdef __cplusplus
extern "C" {
#endif
```

and closed with:

```c
#ifdef __cplusplus
}
#endif
```

The supplied openxr.h header defines a small number of C preprocessor macros that are described below.

Version Number Macros

Two version numbers are defined in openxr.h. Each is packed into a 32-bit integer as described in API Version Number Function-like Macros.

```c
// OpenXR current version number.
#define XR_CURRENT_API_VERSION XR_MAKE_VERSION(1, 0, 24)
```

XR_CURRENT_API_VERSION is the current version of the OpenXR API.
API Version Number Function-like Macros

API Version Numbers are three components, packed into a single 64-bit integer. The following macros manipulate version components and packed version numbers.

```
#define XR_MAKE_VERSION(major, minor, patch)  
    (((major) & 0xffffULL) << 48) | (((minor) & 0xffffULL) << 32) | ((patch) & 0xffffffffULL)
```

Parameter Descriptions

- `major` is the major version number, packed into the most-significant 16 bits.
- `minor` is the minor version number, packed into the second-most-significant group of 16 bits.
- `patch` is the patch version number, in the least-significant 32 bits.

`XR_MAKE_VERSION` constructs a packed 64-bit integer API version number from three components. The format used is described in API Version Numbers and Semantics.

This macro can be used when constructing the `XrApplicationInfo::apiVersion` parameter passed to `xrCreateInstance`.

```
#define XR_VERSION_MAJOR(version) (uint16_t)(((uint64_t)(version) >> 48)& 0xffffULL)
```

Parameter Descriptions

- `version` is a packed version number, such as those produced with `XR_MAKE_VERSION`.

`XR_VERSION_MAJOR` extracts the API major version number from a packed version number.

```
#define XR_VERSION_MINOR(version) (uint16_t)(((uint64_t)(version) >> 32) & 0xffffULL)
```
Parameter Descriptions

- `version` is a packed version number, such as those produced with `XR_MAKE_VERSION`.

**XR_VERSION_MINOR** extracts the API minor version number from a packed version number.

```
#define XR_VERSION_MINOR(version) (uint32_t)((uint64_t)(version) & 0xffffffffULL)
```

Parameter Descriptions

- `version` is a packed version number, such as those produced with `XR_MAKE_VERSION`.

**XR_VERSION_PATCH** extracts the API patch version number from a packed version number.

Handle and Atom Macros

```
#if !defined(XR_DEFINE_HANDLE)
#if (XR_PTR_SIZE == 8)
   #define XR_DEFINE_HANDLE(object) typedef struct object##_T* object;
#else
   #define XR_DEFINE_HANDLE(object) typedef uint64_t object;
#endif
#else
   #endif
#endif
```

Parameter Descriptions

- `object` is the name of the resulting C type.

**XR_DEFINE_HANDLE** defines a handle type, which is an opaque 64 bit value, which may be implemented as an opaque, distinct pointer type on platforms with 64 bit pointers.

For further details, see Handles.
XR_NULL_HANDLE is a reserved value representing a non-valid object handle. It may be passed to and returned from API functions only when specifically allowed.

XR_DEFINE_ATOM(object) typedef uint64_t object;

Parameter Descriptions

- object is the name of the resulting C type.

XR_DEFINE_ATOM defines an atom type, which is an opaque 64 bit integer.

Platform-Specific Macro Definitions

Additional platform-specific macros and interfaces are defined using the included openxr_platform.h file. These macros are used to control platform-dependent behavior, and their exact definitions are under the control of specific platform implementations of the API.

Platform-Specific Calling Conventions

On many platforms the following macros are empty strings, causing platform- and compiler-specific default calling conventions to be used.

XRAPI_ATTR is a macro placed before the return type of an API function declaration. This macro controls calling conventions for C++11 and GCC/Clang-style compilers.

XRAPI_CALL is a macro placed after the return type of an API function declaration. This macro controls calling conventions for MSVC-style compilers.

XRAPI_PTR is a macro placed between the ( and * in API function pointer declarations. This macro also controls calling conventions, and typically has the same definition as XRAPI_ATTR or XRAPI_CALL, depending on the compiler.
Examples:

Function declaration:

```
XRAPI_ATTR <return_type> XRAPI_CALL <function_name>(<function_parameters>);
```

Function pointer type declaration:

```
typedef <return_type> (XRAPI_PTR *PFN_<function_name>(<function_parameters>);
```

**Platform-Specific Header Control**

If the `XR_NO_STDINT_H` macro is defined by the application at compile time, before including any OpenXR header, extended integer types normally found in `<stdint.h>` and used by the OpenXR headers, such as `uint8_t`, must also be defined (as `typedef` or with the preprocessor) before including any OpenXR header. Otherwise, `openxr.h` and related headers will not compile. If `XR_NO_STDINT_H` is not defined, the system-provided `<stdint.h>` is used to define these types. There is a fallback path for Microsoft Visual Studio version 2008 and earlier versions (which lack this header) that is automatically activated as needed.

**Graphics API Header Control**

<table>
<thead>
<tr>
<th>Compile Time Symbol</th>
<th>Graphics API Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>XR_USE_GRAPHICS_API_OPENGL</td>
<td>OpenGL</td>
</tr>
<tr>
<td>XR_USE_GRAPHICS_API_OPENGL_ES</td>
<td>OpenGL ES</td>
</tr>
<tr>
<td>XR_USE_GRAPHICS_API_VULKAN</td>
<td>Vulkan</td>
</tr>
<tr>
<td>XR_USE_GRAPHICS_API_D3D11</td>
<td>Direct3D 11</td>
</tr>
<tr>
<td>XR_USE_GRAPHICS_API_D3D12</td>
<td>Direct3D 12</td>
</tr>
</tbody>
</table>

**Window System Header Control**

<table>
<thead>
<tr>
<th>Compile Time Symbol</th>
<th>Window System Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>XR_USE_PLATFORM_WIN32</td>
<td>Microsoft Windows</td>
</tr>
<tr>
<td>XR_USE_PLATFORM_XLIB</td>
<td>X Window System Xlib</td>
</tr>
<tr>
<td>XR_USE_PLATFORM_XCB</td>
<td>X Window System Xcb</td>
</tr>
<tr>
<td>XR_USE_PLATFORM_WAYLAND</td>
<td>Wayland</td>
</tr>
<tr>
<td>XR_USE_PLATFORM_ANDROID</td>
<td>Android Native</td>
</tr>
</tbody>
</table>
## Android Notes

Android specific notes for using the OpenXR specification.

### Android Runtime category tag for immersive mode selection

Android applications should add the `<category android:name="org.khronos.openxr.intent.category.IMMERSIVE_HMD" />` tag inside the intent-filter to indicate that the activity starts in an immersive OpenXR mode and will not touch the native Android 2D surface.

The HMD suffix indicates the preferred form-factor used by the application and can be used by launchers to filter applications listed.

For example:

```xml
<intent-filter>
  <action android:name="android.intent.action.MAIN" />
  <category android:name="android.intent.category.LAUNCHER" />
  <category android:name="org.khronos.openxr.intent.category.IMMERSIVE_HMD" />
</intent-filter>
```

## Glossary

The terms defined in this section are used throughout this Specification. Capitalization is not significant for these definitions.

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>The XR application which calls the OpenXR API to communicate with an OpenXR runtime.</td>
</tr>
<tr>
<td>Deprecated</td>
<td>A feature/extension is deprecated if it is no longer recommended as the correct or best way to achieve its intended purpose. Generally a newer feature/extension will have been created that solves the same problem - in cases where no newer alternative feature exists, justification should be provided.</td>
</tr>
<tr>
<td>Handle</td>
<td>An opaque integer or pointer value used to refer to an object. Each object type has a unique handle type.</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Haptic</td>
<td>Haptic or kinesthetic communication recreates the sense of touch by applying forces, vibrations, or motions to the user.</td>
</tr>
<tr>
<td>In-Process</td>
<td>Something that executes in the application's process.</td>
</tr>
<tr>
<td>Instance</td>
<td>The top-level object, which represents the application's connection to the runtime. Represented by an XrInstance object.</td>
</tr>
<tr>
<td>Normalized</td>
<td>A value that is interpreted as being in the range [0,1], or a vector whose norm is in that range, as a result of being implicitly divided or scaled by some other value.</td>
</tr>
<tr>
<td>Out-Of-Process</td>
<td>Something that executes outside the application's process.</td>
</tr>
<tr>
<td>Promoted</td>
<td>A feature is promoted if it is taken from an older extension and made available as part of a new core version of the API, or a newer extension that is considered to be either as widely supported or more so. A promoted feature may have minor differences from the original such as:</td>
</tr>
<tr>
<td></td>
<td>• It may be renamed</td>
</tr>
<tr>
<td></td>
<td>• A small number of non-intrusive parameters may have been added</td>
</tr>
<tr>
<td></td>
<td>• The feature may be advertised differently by device features</td>
</tr>
<tr>
<td></td>
<td>• The author ID suffixes will be changed or removed as appropriate</td>
</tr>
<tr>
<td>Provisional</td>
<td>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.</td>
</tr>
<tr>
<td>Required Extensions</td>
<td>Extensions that must be enabled alongside extensions dependent on them, or that must be enabled to use given hardware.</td>
</tr>
</tbody>
</table>
### Term

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime</td>
<td>The software which implements the OpenXR API and allows applications to interact with XR hardware.</td>
</tr>
<tr>
<td>Swapchain</td>
<td>A resource that represents a chain of images in device memory. Represented by an XrSwapchain object.</td>
</tr>
<tr>
<td>Swapchain Image</td>
<td>Each element in a swapchain. Commonly these are simple formatted 2D images, but in other cases they may be array images. Represented by a structure related to XrSwapchainImageBaseHeader.</td>
</tr>
</tbody>
</table>

### Abbreviations

Abbreviations and acronyms are sometimes used in the API where they are considered clear and commonplace, and are defined here:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AR</td>
<td>Augmented Reality</td>
</tr>
<tr>
<td>ER</td>
<td>Eye Relief</td>
</tr>
<tr>
<td>IAD</td>
<td>Inter Axial Distance</td>
</tr>
<tr>
<td>IPD</td>
<td>Inter Pupillary Distance</td>
</tr>
<tr>
<td>MR</td>
<td>Mixed Reality</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>TSG</td>
<td>Technical Sub-Group. A specialized sub-group within a Khronos Working Group (WG).</td>
</tr>
<tr>
<td>VR</td>
<td>Virtual Reality</td>
</tr>
<tr>
<td>WG</td>
<td>Working Group. An organized group of people working to define/augment an API.</td>
</tr>
<tr>
<td>XR</td>
<td>VR + AR + MR</td>
</tr>
</tbody>
</table>
Dedication (Informative)

In memory of Johannes van Waveren: a loving father, husband, son, brother, colleague, and dear friend.

Johannes, known to his friends as "JP", had a great sense of humor, fierce loyalty, intense drive, a love of rainbow unicorns, and deep disdain for processed American cheese. Perhaps most distinguishing of all, though, was his love of technology and his extraordinary technical ability.

JP’s love of technology started at an early age --- instead of working on his homework, he built train sets, hovercrafts, and complex erector sets from scratch; fashioned a tool for grabbing loose change out of street grates; and played computer games. The passion for computer games continued at Delft University of Technology, where, armed with a T1 internet connection and sheer talent, he regularly destroyed his foes in arena matches without being seen, earning him the moniker "MrElusive". During this time, he wrote the Gladiator-bot AI, which earned him acclaim in the community and led directly to a job at the iconic American computer game company, id Software. From there, he quickly became an expert in every system he touched, contributing significantly to every facet of the technology: AI, path navigation, networking, skeletal animation, virtual texturing, advanced rendering, and physics. He became a master of all. He famously owned more lines of code than anyone else, but he was also a generous mentor, helping junior developers hone their skills and make their own contributions.

When the chance to work in the VR industry arose, he saw it as an opportunity to help shape the future. Having never worked on VR hardware did not phase him; he quickly became a top expert in the field. Many of his contributions directly moved the industry forward, most recently his work on asynchronous timewarp and open-standards development.

Time was not on his side. Even in his final days, JP worked tirelessly on the initial proposal for this specification. The treatments he had undergone took a tremendous physical toll, but he continued to work because of his love of technology, his dedication to the craft, and his desire to get OpenXR started on a solid footing. His focus was unwavering.

His proposal was unofficially adopted several days before his passing - and upon hearing, he mustered the energy for a smile. While it was his great dream to see this process through, he would be proud of the spirit of cooperation, passion, and dedication of the industry peers who took up the torch to drive this specification to completion.

JP lived a life full of accomplishment, as evidenced by many publications, credits, awards, and nominations where you will find his name. A less obvious accomplishment — but of equal importance — is the influence he had on people through his passionate leadership. He strove for excellence in everything that he did. He was always excited to talk about technology and share the discoveries made while working through complex problems. He created excitement and interest around engineering and technical excellence. He was a mentor and teacher who inspired those who knew him and many continue to benefit from his hard work and generosity.
JP was a rare gem; fantastically brilliant intellectually, but also warm, compassionate, generous, humble, and funny. Those of us lucky enough to have crossed paths with him knew what a privilege and great honor it was to know him. He is certainly missed.
Contributors (Informative)

OpenXR is the result of contributions from many people and companies participating in the Khronos OpenXR Working Group. Members of the Working Group, including the company that they represented at the time of their most recent contribution, are listed below.

Contributors to OpenXR 1.0

- Adam Gousetis, Google
- Alex Turner, Microsoft
- Andreas Loeve Selvik, Arm
- Andres Rodriguez, Valve Software
- Armelle Laine, Qualcomm Technologies, Inc
- Attila Maczak, CTRL-labs
- Blake Taylor, Magic Leap
- Brad Grantham, Google
- Brandon Jones, Google
- Brent E. Insko, Intel
- Brent Wilson, Microsoft
- Bryce Hutchings, Microsoft
- Cass Everitt, Facebook
- Charles Egenbacher, Epic Games
- Chris Osborn, CTRL-labs
- Christine Perey, Perey Research & Consulting
- Christoph Haag, Collabora
- Craig Donner, Google
- Dan Ginsburg, Valve Software
- Dave Houlton, LunarG
- Dave Shreiner, Unity Technologies
- Denny Rönngren, Tobii
- Dmitriy Vasilev, Samsung Electronics
- Doug Twileager, ZSpace
- Ed Hutchins, Facebook
- Gloria Kennickell, Facebook
• Gregory Greeby, AMD
• Guodong Chen, Huawei
• Jack Pritz, Unity Technologies
• Jakob Bornecrantz, Collabora
• Jared Cheshier, PlutoVR
• Javier Martinez, Intel
• Jeff Bellinghausen, Valve Software
• Jiehua Guo, Huawei
• Joe Ludwig, Valve Software
• Johannes van Waveren, Facebook
• Jon Leech, Khronos
• Jonathan Wright, Facebook
• Juan Wee, Samsung Electronics
• Jules Blok, Epic Games
• Karl Schultz, LunarG
• Kaye Mason, Google
• Krzysztof Kosiński, Google
• Lachlan Ford, Microsoft
• Lubosz Sarnecki, Collabora
• Mark Young, LunarG
• Martin Renschler, Qualcomm Technologies, Inc.
• Matias Koskela, Tampere University of Technology
• Matt Wash, Arm
• Mattias Brand, Tobii
• Mattias O. Karlsson, Tobii
• Michael Gatson, Dell
• Minmin Gong, Microsoft
• Mitch Singer, AMD
• Nell Waliczek, Microsoft
• Nick Whiting, Epic Games
• Nigel Williams, Sony
• Paul Pedriana, Facebook
• Paulo Gomes, Samsung Electronics
• Peter Kuhn, Unity Technologies
• Peter Peterson, HP Inc.
• Philippe Harscoet, Samsung Electronics
• Pierre-Loup Griffais, Valve Software
• Rajeev Gupta, Sony
• Remi Arnaud, Starbreeze
• Remy Zimmerman, Logitech
• River Gillis, Google
• Robert Memmott, Facebook
• Robert Menzel, NVIDIA
• Robert Simpson, Qualcomm Technologies, Inc.
• Robin Bourianes, Starbreeze
• Ryan A. Pavlik, Collabora
• Ryan Vance, Epic Games
• Sam Martin, Arm
• Satish Salian, NVIDIA
• Scott Flynn, Unity Technologies
• Sean Payne, CTRL-labs
• Sophia Baldonado, PlutoVR
• Steve Smith, Epic Games
• Sungye Kim, Intel
• Tom Flynn, Samsung Electronics
• Trevor F. Smith, Mozilla
• Vivek Viswanathan, Dell
• Yin Li, Microsoft
• Yuval Boger, Sensics
• Zheng Qin, Microsoft
Index

A
xrAcquireSwapchainImage (function), 147
XrAction (type), 186
XrActionCreateInfo (type), 187
XrActionSet (type), 182
XrActionSetCreateInfo (type), 183
XrActionSpaceCreateInfo (type), 100
XrActionsSyncInfo (type), 218
XrActionStateBoolean (type), 205
XrActionStateFloat (type), 207
XrActionStateGetInfo (type), 202
XrActionStatePose (type), 212
XrActionStateVector2f (type), 209
XrActionSuggestedBinding (type), 196
XrActionType (type), 190
XrActiveActionSet (type), 219
XrActiveActionSet (type), 219
XrAndroidThreadTypeKHR (type), 233
XrApiLayerProperties (type), 47
XrApplicationInfo (type), 54
xrApplyHapticFeedback (function), 213
xrAttachSessionActionSets (function), 196

B
XrBaseInStructure (type), 14
XrBaseOutStructure (type), 15
xrBeginFrame (function), 162
xrBeginSession (function), 127
XrBindingModificationBaseHeaderKHR (type), 238
XrBindingModificationsKHR (type), 237
XrBool32 (type), 35
XrBoundSourcesForActionEnumerateInfo (type), 222

C
XR_CURRENT_API_VERSION (define), 344
XrColor4f (type), 27
XrCompositionLayerBaseHeader (type), 169
XrCompositionLayerColorScaleBiasKHR (type), 240
XrCompositionLayerCubeKHR (type), 242
XrCompositionLayerCylinderKHR (type), 247
XrCompositionLayerDepthInfoKHR (type), 249
XrCompositionLayerEquirect2KHR (type), 257
XrCompositionLayerEquirectKHR (type), 254
XrCompositionLayerFlagBits (type), 168
XrCompositionLayerFlags (type), 343
XrCompositionLayerProjection (type), 171
XrCompositionLayerProjectionView (type), 172
XrCompositionLayerQuad (type), 173
xrConvertTimespecTimeToTimeKHR (function), 260
xrConvertTimeToTimespecTimeKHR (function), 261
xrConvertTimeToWin32PerformanceCounterKHR (function), 338
xrConvertWin32PerformanceCounterToTimeKHR (function), 337
xrCreateAction (function), 186
xrCreateActionSet (function), 182
xrCreateActionSpace (function), 99
xrCreateInstance (function), 51
xrCreateReferenceSpace (function), 97
xrCreateSession (function), 123
xrCreateSwapchain (function), 139
xrCreateSwapchainAndroidSurfaceKHR (function), 230
xrCreateVulkanDeviceKHR (function), 325
xrCreateVulkanInstanceKHR (function), 321

D
XR_DEFINE_ATOM (define), 347
XR_DEFINE_HANDLE (define), 346
xrDestroyAction (function), 190
xrDestroyActionSet (function), 185
xrDestroyInstance (function), 55
xrDestroySession (function), 126
xrDestroySpace (function), 101
xrDestroySwapchain (function), 143
XrDuration (type), 26

E
xrEndFrame (function), 164
xrEndSession (function), 129
xrEnumerateApiLayerProperties (function), 46
xrEnumerateBoundSourcesForAction (function), 220
xrEnumerateEnvironmentBlendModes (function), 176
xrEnumerateInstanceExtensionProperties (function), 48
xrEnumerateReferenceSpaces (function), 95
xrEnumerateSwapchainFormats (function), 136
xrEnumerateSwapchainImages (function), 144
xrEnumerateViewConfigurations (function), 111
xrEnumerateViewConfigurationViews (function), 115
XrEnvironmentBlendMode (type), 178
XrEventDataBaseHeader (type), 38
XrEventDataBuffer (type), 38
XrEventDataEventsLost (type), 39
XrEventDataInstanceLossPending (type), 58
XrEventDataInteractionProfileChanged (type), 192
XrEventDataReferenceSpaceChangePending (type), 93
XrEventDataSessionStateChanged (type), 132
XrEventDataVisibilityMaskChangedKHR (type), 302
XrExtensionProperties (type), 50
XrExtent2Df (type), 32
XrExtent2Di (type), 32
XrEyeVisibility (type), 175

F
XR_FAILED (define), 21
XR_FREQUENCY_UNSPECIFIED (define), 216
XrFlags64 (type), 13
XrFormFactor (type), 62
XrFovf (type), 34
XrFrameBeginInfo (type), 163
XrFrameEndInfo (type), 166
XrFrameState (type), 161
XrFrameWaitInfo (type), 160

G
xrGetActionStateBoolean (function), 204
xrGetActionStateFloat (function), 206
xrGetActionStatePose (function), 210
xrGetActionStateVector2f (function), 208
xrGetCurrentInteractionProfile (function), 198
xrGetD3D11GraphicsRequirementsKHR (function), 267
xrGetD3D12GraphicsRequirementsKHR (function), 274
xrGetInputSourceLocalized_Name (function), 223
xrGetInstanceProcAddr (function), 41
xrGetInstanceProperties (function), 56
xrGetOpenGLESGraphicsRequirementsKHR (function), 296
xrGetOpenGLGraphicsRequirementsKHR (function), 289
xrGetReferenceSpaceBoundsRect (function), 92
xrGetSystem (function), 63
xrGetSystemProperties (function), 66
xrGetViewConfigurationProperties (function), 113
xrGetVisibilityMaskKHR (function), 303
xrGetVulkanDeviceExtensionsKHR (function), 316
xrGetVulkanGraphicsDevice2KHR (function), 324
xrGetVulkanGraphicsDeviceKHR (function), 313
xrGetVulkanGraphicsRequirements2KHR (function), 319
xrGetVulkanGraphicsRequirementsKHR (function), 312
xrGetVulkanInstanceExtensionsKHR (function), 314
XrGraphicsBindingD3D11KHR (type), 264
XrGraphicsBindingD3D12KHR (type), 271
XrGraphicsBindingOpenGLESAndroidKHR (type), 293
XrGraphicsBindingOpenGLWaylandKHR (type), 286
XrGraphicsBindingOpenGLWin32KHR (type), 283
XrGraphicsBindingOpenGLXcbKHR (type), 285
XrGraphicsBindingOpenGLXlibKHR (type), 284
XrGraphicsBindingVulkan2KHR (type), 328
XrGraphicsBindingVulkanKHR (type), 308
XrGraphicsRequirementsD3D11KHR (type), 266
XrGraphicsRequirementsD3D12KHR (type), 273
XrGraphicsRequirementsOpenGLESKHR (type), 295
XrGraphicsRequirementsOpenGLKHR (type), 288
XrGraphicsRequirementsVulkanKHR (type), 311
XrGraphicsRequirementsVulkan2KHR (type), 320
XrSwapchain (type), 136
XrSwapchainCreateFlagBits (type), 142
XrSwapchainCreateFlags (type), 344
XrSwapchainCreateInfo (type), 140
XrSwapchainImageAcquireInfo (type), 148
XrSwapchainImageBaseHeader (type), 146
XrSwapchainImageD3D11KHR (type), 265
XrSwapchainImageD3D12KHR (type), 272
XrSwapchainImageOpenGLESKHR (type), 294
XrSwapchainImageOpenGLKHR (type), 287
XrSwapchainImageReleaseInfo (type), 152
XrSwapchainImageVulkan2KHR (type), 330
XrSwapchainImageVulkanKHR (type), 310
XrSwapchainImageWaitInfo (type), 150
XrSwapchainSubImage (type), 170
XrSwapchainUsageFlagBits (type), 138
XrSwapchainUsageFlags (type), 344
xrSyncActions (function), 217
XrSystemGetInfo (type), 64
XrSystemGraphicsProperties (type), 68
XrSystemId (type), 63
XrSystemProperties (type), 67
XrSystemTrackingProperties (type), 69

T
XrTime (type), 25

U
XR_UNQUALIFIED_SUCCESS (define), 21

V
XR_VERSION_MAJOR (define), 345
XR_VERSION_MINOR (define), 345
XR_VERSION_PATCH (define), 346
XrVector2f (type), 28
XrVector3f (type), 29
XrVector4f (type), 29
XrVersion (type), 5
XrView (type), 155
XrViewConfigurationProperties (type), 114
XrViewConfigurationType (type), 110
XrViewConfigurationView (type), 117
XrViewLocateInfo (type), 155
XrViewState (type), 156
XrViewStateFlagBits (type), 157
XrViewStateFlags (type), 344
XRVisibilityMaskKHR (type), 301
XRVisibilityMaskTypeKHR (type), 300
XrVulkanDeviceCreateFlagsKHR (type), 327
XrVulkanDeviceCreateInfoKHR (type), 326
XrVulkanGraphicsDeviceGetInfoKHR (type), 325
XrVulkanInstanceCreateInfoKHR (type), 323
XrVulkanInstanceCreateInfoKHR (type), 322
XrVulkanSwapchainFormatListCreateInfoKHR (type), 335

W
xrWaitFrame (function), 158
xrWaitSwapchainImage (function), 149