The OpenXR™ 1.1.50 Specification (with all ratified extensions)

The Khronos® OpenXR Working Group

Version 1.1.50: from git ref release-1.1.50

Table of Contents

Preamble

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

Some parts of this Specification are purely informative and so are EXCLUDED 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.

Khronos® and Vulkan® are registered trademarks, and glTF™ is a trademark of The Khronos Group Inc. OpenXR™ is a trademark owned by The Khronos Group Inc. and is registered as a trademark in China, the European Union, Japan and the United Kingdom. OpenGL® is a registered trademark and the OpenGL ES™ and OpenGL SC™ logos are trademarks of Hewlett Packard Enterprise 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

https://registry.khronos.org/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:

https://tools.ietf.org/html/rfc2119

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.

> ℹ️ 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.

```
// Provided by XR_VERSION_1_0
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. Runtimes **may** use patch version number to determine whether to enable implementation changes, such as bug fixes, that impact functionality. Runtimes **should** document any changes that are tied to the patch version. Application developers **should** retest their application on all runtimes they support after compiling with a new version.

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 continues where the last minor version left off, making patch versions unique inside major 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 during an update to any of the major, minor, or patch version numbers:

Table 1. Scenarios Which May Cause a Version Change

Reason	*Major Version*	*Minor Version*	*Patch Version*
Extensions Added/Removed*	**may**	**may**	**may**
Spec-Optional Behavior Changed*	**may**	**may**	**may**
Spec Required Behavior Changed*	**may**	**may**	**must** not
Core Interfaces Added*	**may**	**may**	**must** not
Weak Deprecation*	**may**	**may**	**must** not
Strong Deprecation*	**may**	**must** not	**must** not
Core Interfaces Changed/Removed*	**may**	**must** not	**must** not

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 normative language **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, lower latency, 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.

Externally Synchronized Parameters

- The `instance` parameter, and any child handles, in xrDestroyInstance
- The `session` parameter, and any child handles, in xrDestroySession
- The `space` parameter, and any child handles, in xrDestroySpace
- The `swapchain` parameter, and any child handles, in xrDestroySwapchain
- The `actionSet` parameter, and any child handles, in xrDestroyActionSet
- The `action` parameter, and any child handles, in xrDestroyAction
- The `handTracker` parameter, and any child handles, in xrDestroyHandTrackerEXT
- The `future` member of the `cancelInfo` parameter in xrCancelFutureEXT
- The `spatialContext` parameter, and any child handles, in xrDestroySpatialContextEXT
- The `snapshot` parameter, and any child handles, in xrDestroySpatialSnapshotEXT
- The `spatialEntity` parameter, and any child handles, in xrDestroySpatialEntityEXT
- The `persistenceContext` parameter, and any child handles, in xrDestroySpatialPersistenceContextEXT

Implicit Externally Synchronized Parameters

- The `session` parameter by any other xrWaitFrame call in xrWaitFrame
- The `session` parameter by any other xrBeginFrame or xrEndFrame call in xrBeginFrame
- The `session` parameter by any other xrBeginFrame or xrEndFrame call in xrEndFrame

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 grows through the addition of new features. Similar to other Khronos APIs, extensions **may** 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 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 and configured for inclusion in this document 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 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 potentially expensive validation of correct API usage to be implemented in a "validation" layer. Such a layer allows the application developer to develop their application with a validation layer active to ensure that the application is using the API correctly. A 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 a 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 a validation layer to allow the application to run in a production environment without repeatedly incurring the validation expense. (Note that some validation of correct API usage is required to be implemented by the runtime.)

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 are not 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 by that API layer.

2.8. 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 assume that the actual type matches the declared type in some configurations, and **may** be so configured by default at common optimization levels. In such a compiler configured with that assumption, violating the assumption **may** produce undefined behavior. 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 code:* `const`. However, each element of 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.

```
// Provided by XR_VERSION_1_0
#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 is defined to a 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.

2.9. 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.

2.9.1. 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.

2.9.2. 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 all of the following conditions hold:

> ### 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.
>
> - 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.

2.9.3. 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.

2.9.4. 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.

2.9.5. Valid Usage for Flags

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

```
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 but will appear in the specification with all other valid flag values for that type.)
>
> - 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.

2.9.6. 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
>
> - OpenGLES ⇒ _OPENGL_ES
>
> - EGL ⇒ _EGL
>
> - D3D ⇒ _D3D

2.9.7. 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:

```
// Provided by XR_VERSION_1_0
typedef struct XrBaseInStructure {
    XrStructureType                    type;
    const struct XrBaseInStructure*    next;
} XrBaseInStructure;
```

<div align="center">

Member Descriptions

</div>

- 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.

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

The XrBaseOutStructure structure is defined as:

```
// Provided by XR_VERSION_1_0
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.

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.

2.9.8. 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.

2.10. Return Codes

The core API is designed to capture most, but not all, instances of incorrect usage. As such, most functions provide 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,
    XR_ERROR_LIMIT_REACHED = -10,
    XR_ERROR_SIZE_INSUFFICIENT = -11,
    XR_ERROR_HANDLE_INVALID = -12,
    XR_ERROR_INSTANCE_LOST = -13,
    XR_ERROR_SESSION_RUNNING = -14,
    XR_ERROR_SESSION_NOT_RUNNING = -16,
    XR_ERROR_SESSION_LOST = -17,
    XR_ERROR_SYSTEM_INVALID = -18,
    XR_ERROR_PATH_INVALID = -19,
    XR_ERROR_PATH_COUNT_EXCEEDED = -20,
    XR_ERROR_PATH_FORMAT_INVALID = -21,
    XR_ERROR_PATH_UNSUPPORTED = -22,
    XR_ERROR_LAYER_INVALID = -23,
    XR_ERROR_LAYER_LIMIT_EXCEEDED = -24,
    XR_ERROR_SWAPCHAIN_RECT_INVALID = -25,
    XR_ERROR_SWAPCHAIN_FORMAT_UNSUPPORTED = -26,
    XR_ERROR_ACTION_TYPE_MISMATCH = -27,
    XR_ERROR_SESSION_NOT_READY = -28,
    XR_ERROR_SESSION_NOT_STOPPING = -29,
    XR_ERROR_TIME_INVALID = -30,
```

```
      XR_ERROR_REFERENCE_SPACE_UNSUPPORTED = -31,
      XR_ERROR_FILE_ACCESS_ERROR = -32,
      XR_ERROR_FILE_CONTENTS_INVALID = -33,
      XR_ERROR_FORM_FACTOR_UNSUPPORTED = -34,
      XR_ERROR_FORM_FACTOR_UNAVAILABLE = -35,
      XR_ERROR_API_LAYER_NOT_PRESENT = -36,
      XR_ERROR_CALL_ORDER_INVALID = -37,
      XR_ERROR_GRAPHICS_DEVICE_INVALID = -38,
      XR_ERROR_POSE_INVALID = -39,
      XR_ERROR_INDEX_OUT_OF_RANGE = -40,
      XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED = -41,
      XR_ERROR_ENVIRONMENT_BLEND_MODE_UNSUPPORTED = -42,
      XR_ERROR_NAME_DUPLICATED = -44,
      XR_ERROR_NAME_INVALID = -45,
      XR_ERROR_ACTIONSET_NOT_ATTACHED = -46,
      XR_ERROR_ACTIONSETS_ALREADY_ATTACHED = -47,
      XR_ERROR_LOCALIZED_NAME_DUPLICATED = -48,
      XR_ERROR_LOCALIZED_NAME_INVALID = -49,
      XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING = -50,
      XR_ERROR_RUNTIME_UNAVAILABLE = -51,
    // Provided by XR_VERSION_1_1
      XR_ERROR_EXTENSION_DEPENDENCY_NOT_ENABLED = -1000710001,
    // Provided by XR_VERSION_1_1
      XR_ERROR_PERMISSION_INSUFFICIENT = -1000710000,
    // Provided by XR_KHR_android_thread_settings
      XR_ERROR_ANDROID_THREAD_SETTINGS_ID_INVALID_KHR = -1000003000,
    // Provided by XR_KHR_android_thread_settings
      XR_ERROR_ANDROID_THREAD_SETTINGS_FAILURE_KHR = -1000003001,
    // Provided by XR_EXT_future
      XR_ERROR_FUTURE_PENDING_EXT = -1000469001,
    // Provided by XR_EXT_future
      XR_ERROR_FUTURE_INVALID_EXT = -1000469002,
    // Provided by XR_EXT_spatial_entity
      XR_ERROR_SPATIAL_CAPABILITY_UNSUPPORTED_EXT = -1000740001,
    // Provided by XR_EXT_spatial_entity
      XR_ERROR_SPATIAL_ENTITY_ID_INVALID_EXT = -1000740002,
    // Provided by XR_EXT_spatial_entity
      XR_ERROR_SPATIAL_BUFFER_ID_INVALID_EXT = -1000740003,
    // Provided by XR_EXT_spatial_entity
      XR_ERROR_SPATIAL_COMPONENT_UNSUPPORTED_FOR_CAPABILITY_EXT = -1000740004,
    // Provided by XR_EXT_spatial_entity
      XR_ERROR_SPATIAL_CAPABILITY_CONFIGURATION_INVALID_EXT = -1000740005,
    // Provided by XR_EXT_spatial_entity
      XR_ERROR_SPATIAL_COMPONENT_NOT_ENABLED_EXT = -1000740006,
    // Provided by XR_EXT_spatial_persistence
      XR_ERROR_SPATIAL_PERSISTENCE_SCOPE_UNSUPPORTED_EXT = -1000763001,
    // Provided by XR_EXT_spatial_persistence_operations
      XR_ERROR_SPATIAL_PERSISTENCE_SCOPE_INCOMPATIBLE_EXT = -1000781001,
```

```
  // Provided by XR_KHR_maintenance1
    XR_ERROR_EXTENSION_DEPENDENCY_NOT_ENABLED_KHR =
 XR_ERROR_EXTENSION_DEPENDENCY_NOT_ENABLED,
  // Provided by XR_KHR_maintenance1
    XR_ERROR_PERMISSION_INSUFFICIENT_KHR = XR_ERROR_PERMISSION_INSUFFICIENT,
    XR_RESULT_MAX_ENUM = 0x7FFFFFFF
} XrResult;
```

All return codes in the API are reported via XrResult return values.

The following are common suffixes shared across many of the return codes:

- `_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 the handle taken by this function, but is not usable at this moment.

Success Codes

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

Error Codes

Enum	Description
XR_ERROR_VALIDATION_FAILURE	The function usage was invalid in some way.
XR_ERROR_RUNTIME_FAILURE	The runtime failed to handle the function in an unexpected way that is not covered by another error result.
XR_ERROR_OUT_OF_MEMORY	A memory allocation has failed.

Enum	Description
XR_ERROR_API_VERSION_UNSUPPORTED	The runtime does not support the requested API version.
XR_ERROR_INITIALIZATION_FAILED	Initialization of object could not be completed.
XR_ERROR_FUNCTION_UNSUPPORTED	The requested function was not found or is otherwise unsupported.
XR_ERROR_FEATURE_UNSUPPORTED	The requested feature is not supported.
XR_ERROR_EXTENSION_NOT_PRESENT	A requested extension is not supported.
XR_ERROR_LIMIT_REACHED	The runtime supports no more of the requested resource.
XR_ERROR_SIZE_INSUFFICIENT	The supplied size was smaller than required.
XR_ERROR_HANDLE_INVALID	A supplied object handle was invalid.
XR_ERROR_INSTANCE_LOST	The XrInstance was lost or could not be found. It will need to be destroyed and optionally recreated.
XR_ERROR_SESSION_RUNNING	The session is already running.
XR_ERROR_SESSION_NOT_RUNNING	The session is not yet running.
XR_ERROR_SESSION_LOST	The XrSession was lost. It will need to be destroyed and optionally recreated.
XR_ERROR_SYSTEM_INVALID	The provided XrSystemId was invalid.
XR_ERROR_PATH_INVALID	The provided XrPath was not valid.
XR_ERROR_PATH_COUNT_EXCEEDED	The maximum number of supported semantic paths has been reached.
XR_ERROR_PATH_FORMAT_INVALID	The semantic path character format is invalid.
XR_ERROR_PATH_UNSUPPORTED	The semantic path is unsupported.
XR_ERROR_LAYER_INVALID	The layer was NULL or otherwise invalid.
XR_ERROR_LAYER_LIMIT_EXCEEDED	The number of specified layers is greater than the supported number.
XR_ERROR_SWAPCHAIN_RECT_INVALID	The image rect was negatively sized or otherwise invalid.
XR_ERROR_SWAPCHAIN_FORMAT_UNSUPPORTED	The image format is not supported by the runtime or platform.
XR_ERROR_ACTION_TYPE_MISMATCH	The API used to retrieve an action's state does not match the action's type.
XR_ERROR_SESSION_NOT_READY	The session is not in the ready state.

Enum	Description
XR_ERROR_SESSION_NOT_STOPPING	The session is not in the stopping state.
XR_ERROR_TIME_INVALID	The provided XrTime was zero, negative, or out of range.
XR_ERROR_REFERENCE_SPACE_UNSUPPORTED	The specified reference space is not supported by the runtime or system.
XR_ERROR_FILE_ACCESS_ERROR	The file could not be accessed.
XR_ERROR_FILE_CONTENTS_INVALID	The file's contents were invalid.
XR_ERROR_FORM_FACTOR_UNSUPPORTED	The specified form factor is not supported by the current runtime or platform.
XR_ERROR_FORM_FACTOR_UNAVAILABLE	The specified form factor is supported, but the device is currently not available, e.g. not plugged in or powered off.
XR_ERROR_API_LAYER_NOT_PRESENT	A requested API layer is not present or could not be loaded.
XR_ERROR_CALL_ORDER_INVALID	The call was made without having made a previously required call.
XR_ERROR_GRAPHICS_DEVICE_INVALID	The given graphics device is not in a valid state. The graphics device could be lost or initialized without meeting graphics requirements.
XR_ERROR_POSE_INVALID	The supplied pose was invalid with respect to the requirements.
XR_ERROR_INDEX_OUT_OF_RANGE	The supplied index was outside the range of valid indices.
XR_ERROR_VIEW_CONFIGURATION_TYPE_UNSUPPORTED	The specified view configuration type is not supported by the runtime or platform.
XR_ERROR_ENVIRONMENT_BLEND_MODE_UNSUPPORTED	The specified environment blend mode is not supported by the runtime or platform.
XR_ERROR_NAME_DUPLICATED	The name provided was a duplicate of an already-existing resource.
XR_ERROR_NAME_INVALID	The name provided was invalid.
XR_ERROR_ACTIONSET_NOT_ATTACHED	A referenced action set is not attached to the session.
XR_ERROR_ACTIONSETS_ALREADY_ATTACHED	The session already has attached action sets.
XR_ERROR_LOCALIZED_NAME_DUPLICATED	The localized name provided was a duplicate of an already-existing resource.

Enum	Description
XR_ERROR_LOCALIZED_NAME_INVALID	The localized name provided was invalid.
XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING	The xrGetGraphicsRequirements* call was not made before calling xrCreateSession.
XR_ERROR_RUNTIME_UNAVAILABLE	The loader was unable to find or load a runtime.
XR_ERROR_EXTENSION_DEPENDENCY_NOT_ENABLED	One or more of the extensions being enabled has dependency on extensions that are not enabled.
XR_ERROR_PERMISSION_INSUFFICIENT	Insufficient permissions. This error is included for use by vendor extensions. The precise definition of XR_ERROR_PERMISSION_INSUFFICIENT and actions possible by the developer or user to resolve it can vary by platform, extension or function. The developer should refer to the documentation of the function that returned the error code and extension it was defined.
XR_ERROR_ANDROID_THREAD_SETTINGS_ID_INVALID_KHR	xrSetAndroidApplicationThreadKHR failed as thread id is invalid. (Added by the XR_KHR_android_thread_settings extension)
XR_ERROR_ANDROID_THREAD_SETTINGS_FAILURE_KHR	xrSetAndroidApplicationThreadKHR failed setting the thread attributes/priority. (Added by the XR_KHR_android_thread_settings extension)
XR_ERROR_FUTURE_PENDING_EXT	Returned by completion function to indicate future is not ready. (Added by the XR_EXT_future extension)
XR_ERROR_FUTURE_INVALID_EXT	Returned by completion function to indicate future is not valid. (Added by the XR_EXT_future extension)
XR_ERROR_EXTENSION_DEPENDENCY_NOT_ENABLED_KHR	Alias for XR_ERROR_EXTENSION_DEPENDENCY_NOT_ENABLED (Added by the XR_KHR_maintenance1 extension)
XR_ERROR_PERMISSION_INSUFFICIENT_KHR	Alias for XR_ERROR_PERMISSION_INSUFFICIENT (Added by the XR_KHR_maintenance1 extension)
XR_ERROR_SPATIAL_CAPABILITY_UNSUPPORTED_EXT	The specified spatial capability is not supported by the runtime or the system. (Added by the XR_EXT_spatial_entity extension)
XR_ERROR_SPATIAL_ENTITY_ID_INVALID_EXT	The specified spatial entity id is invalid or an entity with that id does not exist in the environment. (Added by the XR_EXT_spatial_entity extension)

Enum	Description
`XR_ERROR_SPATIAL_BUFFER_ID_INVALID_EXT`	The specified spatial buffer id is invalid or does not exist in the spatial snapshot being used to query for the buffer data. (Added by the `XR_EXT_spatial_entity` extension)
`XR_ERROR_SPATIAL_COMPONENT_UNSUPPORTED_FOR_CAPABILITY_EXT`	The specified spatial component is not supported by the runtime or the system for the given capability. (Added by the `XR_EXT_spatial_entity` extension)
`XR_ERROR_SPATIAL_CAPABILITY_CONFIGURATION_INVALID_EXT`	The specified spatial capability configuration is invalid. (Added by the `XR_EXT_spatial_entity` extension)
`XR_ERROR_SPATIAL_COMPONENT_NOT_ENABLED_EXT`	The specified spatial component is not enabled for the spatial context. (Added by the `XR_EXT_spatial_entity` extension)
`XR_ERROR_SPATIAL_PERSISTENCE_SCOPE_UNSUPPORTED_EXT`	The specified spatial persistence scope is not supported by the runtime or the system. (Added by the `XR_EXT_spatial_persistence` extension)
`XR_ERROR_SPATIAL_PERSISTENCE_SCOPE_INCOMPATIBLE_EXT`	The scope configured for the persistence context is incompatible for the current spatial entity. (Added by the `XR_EXT_spatial_persistence_operations` extension)

2.10.1. Convenience Macros

```
// Provided by XR_VERSION_1_0
#define XR_SUCCEEDED(result) ((result) >= 0)
```

A convenience macro that **can** be used to test if a function succeeded. Note that this evaluates to true for all success codes, including a qualified success such as `XR_FRAME_DISCARDED`.

```
// Provided by XR_VERSION_1_0
#define XR_FAILED(result) ((result) < 0)
```

A convenience macro that **can** be used to test if a function has failed in some way. It evaluates to true for all failure codes.

```
// Provided by XR_VERSION_1_0
#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 evaluates to true exclusively when the provided XrResult is equal to XR_SUCCESS (0).

2.10.2. 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.

2.11. 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 passed by the application if it does not pass 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 of XrSession.

2.12. 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.

```
// Provided by XR_VERSION_1_0
typedef enum XrObjectType {
    XR_OBJECT_TYPE_UNKNOWN = 0,
    XR_OBJECT_TYPE_INSTANCE = 1,
    XR_OBJECT_TYPE_SESSION = 2,
    XR_OBJECT_TYPE_SWAPCHAIN = 3,
    XR_OBJECT_TYPE_SPACE = 4,
    XR_OBJECT_TYPE_ACTION_SET = 5,
    XR_OBJECT_TYPE_ACTION = 6,
  // Provided by XR_EXT_hand_tracking
    XR_OBJECT_TYPE_HAND_TRACKER_EXT = 1000051000,
  // Provided by XR_EXT_spatial_entity
    XR_OBJECT_TYPE_SPATIAL_ENTITY_EXT = 1000740000,
  // Provided by XR_EXT_spatial_entity
    XR_OBJECT_TYPE_SPATIAL_CONTEXT_EXT = 1000740001,
  // Provided by XR_EXT_spatial_entity
    XR_OBJECT_TYPE_SPATIAL_SNAPSHOT_EXT = 1000740002,
  // Provided by XR_EXT_spatial_persistence
    XR_OBJECT_TYPE_SPATIAL_PERSISTENCE_CONTEXT_EXT = 1000763000,
    XR_OBJECT_TYPE_MAX_ENUM = 0x7FFFFFFF
} XrObjectType;
```

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 in the core specification:

XrObjectType	OpenXR Handle Type
XR_OBJECT_TYPE_UNKNOWN	Unknown/Undefined Handle
XR_OBJECT_TYPE_INSTANCE	XrInstance
XR_OBJECT_TYPE_SESSION	XrSession
XR_OBJECT_TYPE_SWAPCHAIN	XrSwapchain
XR_OBJECT_TYPE_SPACE	XrSpace
XR_OBJECT_TYPE_ACTION_SET	XrActionSet
XR_OBJECT_TYPE_ACTION	XrAction

2.13. Buffer Size Parameters

Functions with buffer or array parameters passed as pointers, rather than declared with a static array size, follow different conventions depending on whether the buffer size is known to the application or variable per call.

2.13.1. Variable size buffer parameters

Functions with variable size output buffer parameters take on either parameter form or structure form, as in one of the following examples, with the element type being `float` in this case:

Parameter form:

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

Structure form:

```
XrResult xrFunction(XrBuffer* buffer);

struct XrBuffer {
    uint32_t                elementCapacityInput;
    uint32_t                elementCountOutput;
    float*                  elements;
};
```

A "two-call idiom" **should** be employed by the application, 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 structure) 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. If the element type of `elements` is a structure with `type` and `next` fields, the application **must** set the `type` to the correct value, and **must** set `next` to a valid value. A valid value for `next` is generally either `NULL` or another structure with related data, in which `type` and `next` are also valid, recursively. (See Valid Usage for Structure Pointer Chains for details.)

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 following behavior with respect to the array/buffer and its size parameters:

> ### Buffer Size Parameter Behavior
>
> - The `elementCapacityInput` and `elementCountOutput` 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 application **must** pass a valid pointer for `elementCountOutput` if the function uses parameter form.
>
> - Independent of `elementCapacityInput` or `elements` parameters, the function sets `elementCountOutput`.
>
> - The application **may** pass `0` for the `elementCapacityInput` parameter, to indicate a request for the required array size. That is, passing a capacity of `0` does not return `XR_ERROR_SIZE_INSUFFICIENT`. In this case, the following two points apply.
>
>   - The function **must** set `elementCountOutput` to the required size in number of elements.
>
>   - The `elements` parameter is ignored (any value passed is considered valid usage).
>
> - If the `elementCapacityInput` is non-zero but less than required, the function **must** set `elementCountOutput` to the required capacity, and **must** return `XR_ERROR_SIZE_INSUFFICIENT`. After the function returns, the data in the array `elements` is undefined.
>
> - If the `elementCapacityInput` is non-zero and the function returns successfully, the function sets `elementCountOutput` to the count of the elements that have been written to `elements`.
>
> - If the function returns any `XR_ERROR_*` result other than `XR_ERROR_SIZE_INSUFFICIENT`, the contents of the values of (or pointed to by) `elementCountOutput` and `elements` are undefined.
>
> - For clarity, if the element array refers to a string (`element` is of type `char*`), `elementCapacityInput` and `elementCountOutput` refer to the string `strlen` plus `1` for a `NULL` terminator.

Some functions have a given `elementCapacityInput` and `elementCountOutput` associated with more than one element array (i.e. parallel arrays). In this case, the capacity/count and all its associated arrays will share a common name prefix. All of the preceding general requirements continue to apply.

Some functions fill multiple element arrays of varying sizes in one call. For these functions, the `elementCapacityInput`, `elementCountOutput`, and `elements` array parameters or fields are repeated with different prefixes. In this case, all of the preceding general requirements still apply, with these additional requirements:

- If the application sets **any** `elementCapacityInput` parameter or field to `0`, the runtime **must** treat **all** `elementCapacityInput` values as if they were set to `0`.

- If all `elementCapacityInput` values are non-zero but **any** is insufficient to fit all elements of its corresponding array, the runtime **must** return `XR_ERROR_SIZE_INSUFFICIENT`. As in the case of the single array, the data in all arrays is undefined when the function returns any `XR_ERROR_*` result.

2.13.2. Known size buffer parameters

Functions with known size input and/or output buffer parameters, or buffer parameters of an application-chosen size, take a slightly different approach than variable size buffer parameters. Such functions also take on either parameter form or structure form, as in the following examples:

Parameter form:

```
XrResult xrFunction(uint32_t elementCount, float* elements);
```

Structure form:

```
XrResult xrFunction(XrBuffer* buffer);

struct XrBuffer {
    uint32_t              elementCount;
    float*                elements;
};
```

Unlike for variable size buffer parameters, only a single "count" is specified per buffer/array. Functions following this convention have the following behavior with respect to the array/buffer and its count parameters:

Buffer Size Parameter Behavior

- The `elementCount` argument precedes the array to which it refers, in argument order.

- `elementCount` specifies the number of elements passed as input, or both the capacity in number of elements of the buffer to be written and the exact number of elements expected.

- `elements` **must** be at least `elementCount` elements in size.

- If the array is used as output or input/output and `elementCount` is not equal to the exact number of elements expected, the function **must** return `XR_ERROR_VALIDATION_FAILURE`.

- If the array is used as output or input/output and the function returns any `XR_ERROR_*`, including `XR_ERROR_VALIDATION_FAILURE`, the contents of the data in the array `elements` is undefined.

- If the array is used as input, all `elementCount` elements of `elements` **must** be valid unless otherwise specified.

Some functions have a given `elementCount` associated with more than one element array (i.e. parallel arrays). In this case, the count and all its associated arrays will share a common name prefix. All of the preceding general requirements continue to apply.

Some functions operate on multiple element arrays of known sizes in one call. For these functions, the `elementCount`, and `elements` array parameters or fields are repeated with different prefixes. As in the case of the single array, the data in all arrays is undefined when the function returns any `XR_ERROR_*` result. All of the preceding general requirements continue to apply.

2.14. 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.

2.14.1. 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.

2.15. Duration

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

2.15.1. 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. XrDuration is a base value type that represents duration as a signed 64-bit integer, representing the signed number of nanoseconds between two timepoints.

Functions that refer to durations use XrDuration as opposed to XrTime. When an XrDuration is used as a timeout parameter, the constants XR_NO_DURATION and XR_INFINITE_DURATION have special meaning. A timeout with a duration that refers to the past (that is, a negative duration) **must** be interpreted as a timeout of XR_NO_DURATION.

The interpretation of zero and negative durations in non-timeout uses is specified along with each such use.

```
// Provided by XR_VERSION_1_0
#define XR_NO_DURATION 0
```

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

```
// Provided by XR_VERSION_1_0
#define XR_INFINITE_DURATION 0x7fffffffffffffffLL
```

XR_INFINITE_DURATION is a special value that **can** be used to indicate that the timeout never occurs.

2.16. Prediction Time Limits

Some functions involve prediction or history retrieval for a supplied `XrTime` timepoint. For example, xrLocateViews accepts a display time for which to return the resulting data. Timepoints 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 the accuracy possible at varying past (historical or backwards prediction) and future timepoints. The runtime **must** treat as valid any future time requested by an application, except when disqualified by size limitations of the underlying types, 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. The runtime **must** retain and return at least 50 milliseconds of historical data, interpolating as required, preceding the most recently received pose. 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.

2.17. Colors

The XrColor3f structure is defined as:

```
// Provided by XR_VERSION_1_1
typedef struct XrColor3f {
    float    r;
    float    g;
    float    b;
} XrColor3f;
```

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.

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.

The XrColor4f structure is defined as:

```
// Provided by XR_VERSION_1_0
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.

2.18. Coordinate System

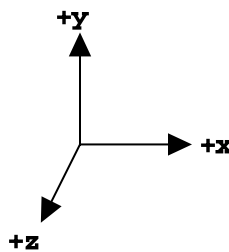This API uses a Cartesian right-handed coordinate system.

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:

```
// Provided by XR_VERSION_1_0
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:

```
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.

2.19. Common Data Types

Some OpenXR data types are used in multiple structures. Those include the `XrVector*f` family of types, the spatial types specified above, and the following categories of structures:

- offset

- extents

- rectangle

- field of view

Offsets are used to describe the direction and distance of an offset in two dimensions.

A floating-point offset is defined by the structure:

```
// Provided by XR_VERSION_1_0
typedef struct XrOffset2Df {
    float    x;
    float    y;
} XrOffset2Df;
```

This structure is used for component values that may be real numbers, represented with single-precision floating point. For representing offsets in discrete values, such as texels, the integer variant XrOffset2Di is used instead.

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;
```

This variant is for representing discrete values such as texels. For representing physical distances, the floating-point variant XrOffset2Df is used instead.

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

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

```
// Provided by XR_VERSION_1_0
typedef struct XrExtent2Df {
    float     width;
    float     height;
} XrExtent2Df;
```

This structure is used for component values that may be real numbers, represented with single-precision floating point. For representing extents in discrete values, such as texels, the integer variant XrExtent2Di is used instead.

If used to represent physical distances, values **must** be in meters.

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

The XrExtent3Df structure is defined as:

```
// Provided by XR_VERSION_1_1
typedef struct XrExtent3Df {
    float    width;
    float    height;
    float    depth;
} XrExtent3Df;
```

Member Descriptions

- `width` is the floating-point width of the extent (x).
- `height` is the floating-point height of the extent (y).
- `depth` is the floating-point depth of the extent (z).

This structure is used for component values that may be real numbers, represented with single-precision floating point.

If used to represent physical distances, values **must** be in meters. The width, height, and depth values **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` is the integer width of the extent.
- `height` is the integer height of the extent.

This variant is for representing discrete values such as texels. For representing physical distances, the floating-point variant XrExtent2Df is 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:

```
// Provided by XR_VERSION_1_0
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 real numbers, represented with single-precision floating point.

The `offset` is the position of the rectangle corner with minimum value coordinates. The other three corners are computed by adding the XrExtent2Df::`width` to the `x` offset, XrExtent2Df::`height` to the `y` offset, or both.

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 XrRect2Df is used instead.

The `offset` is the position of the rectangle corner with minimum value coordinates. The other three corners are computed by adding the XrExtent2Di::`width` to the `x` offset, XrExtent2Di::`height` to the `y` offset, or both.

An XrSpheref structure describes the center and radius of a sphere bounds.

```
// Provided by XR_VERSION_1_1
typedef struct XrSpheref {
    XrPosef    center;
    float      radius;
} XrSpheref;
```

Member Descriptions

- `center` is an XrPosef representing the pose of the center of the sphere within the reference frame of the corresponding XrSpace.
- `radius` is the finite non-negative radius of the sphere.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` if `radius` is not a finite positive value.

An XrBoxf structure describes the pose and extents of an oriented box.

```
// Provided by XR_VERSION_1_1
typedef struct XrBoxf {
    XrPosef       center;
    XrExtent3Df   extents;
} XrBoxf;
```

Member Descriptions

- center is an XrPosef defining the center position and orientation of the oriented bounding box bound within the reference frame of the corresponding XrSpace.

- extents is an XrExtent3Df defining the edge-to-edge length of the box along each dimension with center as the center.

The runtime **must** return XR_ERROR_VALIDATION_FAILURE if width, height or depth values are negative.

An XrFrustumf structure describes the pose, field of view, and far distance of a frustum.

```
// Provided by XR_VERSION_1_1
typedef struct XrFrustumf {
    XrPosef   pose;
    XrFovf    fov;
    float     nearZ;
    float     farZ;
} XrFrustumf;
```

Member Descriptions

- pose is an XrPosef defining the position and orientation of the tip of the frustum within the reference frame of the corresponding XrSpace.

- fov is an XrFovf for the four sides of the frustum where angleLeft and angleRight are along the X axis and angleUp and angleDown are along the Y axis of the frustum space.

- nearZ is the positive distance of the near plane of the frustum bound along the -Z direction of the frustum space.

- farZ is the positive distance of the far plane of the frustum bound along the -Z direction of the frustum space.

The runtime **must** return XR_ERROR_VALIDATION_FAILURE if farZ is less than or equal to zero.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` if `nearZ` is less than zero.

See XrFovf for validity requirements on `fov`.

The XrUuid structure is a 128-bit Universally Unique Identifier and is defined as:

```
// Provided by XR_VERSION_1_1
typedef struct XrUuid {
    uint8_t    data[XR_UUID_SIZE];
} XrUuid;
```

Member Descriptions

- `data` is a 128-bit Universally Unique Identifier.

The structure is composed of 16 octets, with the size and order of the fields defined in RFC 4122 section 4.1.2.

2.20. 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;
```

- `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 `angleRight` minus `angleLeft`, and the total vertical field of view is `angleUp` minus `angleDown`. For a symmetric FoV, `angleRight` and `angleUp` will have positive values, `angleLeft` will be `-angleRight`, and `angleDown` will be `-angleUp`.

The angles **must** be specified in radians, and **must** be between $-\pi/2$ and $\pi/2$ exclusively.

When `angleLeft` > `angleRight`, the content of the view **must** be flipped horizontally. When `angleDown` > `angleUp`, the content of the view **must** be flipped vertically.

2.21. Boolean Values

```
typedef uint32_t XrBool32;
```

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

Enumerant Descriptions

- `XR_TRUE` represents a true value.
- `XR_FALSE` represents a false value.

```
#define XR_TRUE                         1
```

```
#define XR_FALSE                        0
```

2.22. Events

Events are messages sent from the runtime to the application.

2.22.1. Event Polling

Events are placed in a queue within the runtime. The application **must** read from the queue with regularity. Events are read from the queue one at a time via xrPollEvent. Every type of event is identified by an individual structure type, with each such structure beginning with an XrEventDataBaseHeader.

Example 1. Proper Method for Receiving OpenXR Event Data

```
XrInstance instance; // previously initialized

// Initialize an event buffer to hold the output.
XrEventDataBuffer event = {XR_TYPE_EVENT_DATA_BUFFER};
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

The xrPollEvent function is defined as:

```
// Provided by XR_VERSION_1_0
XrResult xrPollEvent(
    XrInstance                                  instance,
    XrEventDataBuffer*                          eventData);
```

Parameter Descriptions

- `instance` is a valid XrInstance.

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

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`.

The runtime **must** discard queued events which contain destroyed or otherwise invalid handles. The runtime **must** not return events containing handles that have been destroyed or are otherwise invalid at the time of the call to xrPollEvent.

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`

Table 2. Event Descriptions

Event	Description
XrEventDataEventsLost	event queue has overflowed and some events were lost
XrEventDataInstanceLossPending	application is about to lose the instance
XrEventDataInteractionProfileChanged	current interaction profile for one or more top level user paths has changed
XrEventDataReferenceSpaceChangePending	runtime will begin operating with updated definitions or bounds for a reference space
XrEventDataSessionStateChanged	the application's session has changed lifecycle state

The XrEventDataBaseHeader structure is defined as:

```
// Provided by XR_VERSION_1_0
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 type 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.

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.

An application **can** set (or reset) only the `type` member and clear the `next` member of an XrEventDataBuffer before passing it as an input to xrPollEvent. The runtime **must** ignore the contents of the `varying` field and overwrite it without reading it.

A pointer to an XrEventDataBuffer **may** be type-cast to an XrEventDataBaseHeader pointer, or a pointer to any other appropriate event data based on the `type` parameter.

```
// Provided by XR_VERSION_1_0
#define XR_MAX_EVENT_DATA_SIZE sizeof(XrEventDataBuffer)
```

XR_MAX_EVENT_DATA_SIZE is the size of XrEventDataBuffer, including the size of the XrEventDataBuffer::type and XrEventDataBuffer::next members.

XrEventDataEventsLost

The XrEventDataEventsLost structure is defined as:

```
// Provided by XR_VERSION_1_0
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.

2.23. 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. 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.

```
// Provided by XR_VERSION_1_0
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 **must** return 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 by the application 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 3. xrGetInstanceProcAddr behavior

`instance` **parameter**	`name` **parameter**	**return value**
*	`NULL`	undefined
invalid instance	*	undefined

instance parameter	name parameter	return value
NULL	xrEnumerateInstanceExtensionProperties	fp
NULL	xrEnumerateApiLayerProperties	fp
NULL	xrCreateInstance	fp
NULL	* (any name not covered above)	NULL
instance	core OpenXR function	fp[1]
instance	enabled extension function for instance	fp[1]
instance	* (any name not covered above)	NULL

1

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);
```

<div align="center">

Parameter Descriptions

</div>

- no parameters.

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

```
typedef XrResult (XRAPI_PTR *PFN_xrGetInstanceProcAddr)(XrInstance instance, const char*
name, PFN_xrVoidFunction* function);
```

PFN_xrGetInstanceProcAddr is a function pointer type for xrGetInstanceProcAddr.

```
typedef struct XrApiLayerCreateInfo XrApiLayerCreateInfo;
typedef XrResult (XRAPI_PTR *PFN_xrCreateApiLayerInstance)(
            const XrInstanceCreateInfo* info,
            const XrApiLayerCreateInfo* apiLayerInfo,
            XrInstance* instance);
```

PFN_xrCreateApiLayerInstance is a function pointer type for xrCreateApiLayerInstance.

Note: This function pointer type is only used by an OpenXR loader library, and never by an application.

3.3. Runtime Interface Negotiation

In order to negotiate the runtime interface version with the loader, the runtime **must** implement the xrNegotiateLoaderRuntimeInterface function.

The xrNegotiateLoaderRuntimeInterface function is defined as:

```
// Provided by XR_LOADER_VERSION_1_0
XrResult xrNegotiateLoaderRuntimeInterface(
    const XrNegotiateLoaderInfo*                loaderInfo,
    XrNegotiateRuntimeRequest*                  runtimeRequest);
```

Parameter Descriptions

- `loaderInfo` **must** be a pointer to a valid XrNegotiateLoaderInfo structure.

- `runtimeRequest` **must** be a valid pointer to an XrNegotiateRuntimeRequest structure, with minimal initialization, as subsequently described, to be fully populated by the called runtime.

xrNegotiateLoaderRuntimeInterface **should** be directly exported by a runtime so that using e.g. `GetProcAddress` on Windows or `dlsym` on POSIX platforms returns a valid function pointer to it.

The runtime **must** return `XR_ERROR_INITIALIZATION_FAILED` if any of the following conditions on `loaderInfo` are true:

- XrNegotiateLoaderInfo::`structType` is not `XR_LOADER_INTERFACE_STRUCT_LOADER_INFO`
- XrNegotiateLoaderInfo::`structVersion` is not `XR_LOADER_INFO_STRUCT_VERSION`
- XrNegotiateLoaderInfo::`structSize` is not `sizeof(XrNegotiateLoaderInfo)`

The runtime **must** also return `XR_ERROR_INITIALIZATION_FAILED` if any of the following conditions on `runtimeRequest` are true:

- XrNegotiateRuntimeRequest::`structType` is not `XR_LOADER_INTERFACE_STRUCT_RUNTIME_REQUEST`
- XrNegotiateRuntimeRequest::`structVersion` is not `XR_RUNTIME_INFO_STRUCT_VERSION`
- XrNegotiateRuntimeRequest::`structSize` is not `sizeof(XrNegotiateRuntimeRequest)`

The runtime **must** determine if it supports the loader's request. The runtime does not support the loader's request if either of the following is true:

- The runtime does not support any of the interface versions supported by the loader, as specified by the range `XrNegotiateLoaderInfo::minInterfaceVersion` through `XrNegotiateLoaderInfo::maxInterfaceVersion` inclusive.
- The runtime does not support any of the API versions supported by the loader, ignoring "patch" version components, as specified by the range `XrNegotiateLoaderInfo::minApiVersion` through `XrNegotiateLoaderInfo::maxApiVersion` inclusive.

The runtime **must** return `XR_ERROR_INITIALIZATION_FAILED` if it does not support the loader's request.

If the function succeeds, the runtime **must** set the `XrNegotiateRuntimeRequest::runtimeInterfaceVersion` with the runtime interface version it desires to support. The `XrNegotiateRuntimeRequest::runtimeInterfaceVersion` set **must** be in the range `XrNegotiateLoaderInfo::minInterfaceVersion` through `XrNegotiateLoaderInfo::maxInterfaceVersion` inclusive.

If the function succeeds, the runtime **must** set the `XrNegotiateRuntimeRequest::runtimeApiVersion` with the API version of OpenXR it will execute under. The `XrNegotiateRuntimeRequest::runtimeApiVersion` set **must** be in the range `XrNegotiateLoaderInfo::minApiVersion` through `XrNegotiateLoaderInfo::maxApiVersion` inclusive.

If the function succeeds, the runtime **must** set the `XrNegotiateRuntimeRequest::getInstanceProcAddr` with a valid function pointer for the loader to use to query function pointers to the remaining OpenXR functions supported by the runtime.

If the function succeeds, the runtime **must** return `XR_SUCCESS`.

Valid Usage (Implicit)

- `loaderInfo` **must** be a pointer to a valid XrNegotiateLoaderInfo structure
- `runtimeRequest` **must** be a pointer to an XrNegotiateRuntimeRequest structure

---

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_INITIALIZATION_FAILED`

The XrNegotiateLoaderInfo structure is used to pass information about the loader to a runtime or an

API layer.

The XrNegotiateLoaderInfo structure is defined as:

```
typedef struct XrNegotiateLoaderInfo {
    XrLoaderInterfaceStructs    structType;
    uint32_t                    structVersion;
    size_t                      structSize;
    uint32_t                    minInterfaceVersion;
    uint32_t                    maxInterfaceVersion;
    XrVersion                   minApiVersion;
    XrVersion                   maxApiVersion;
} XrNegotiateLoaderInfo;
```

Member Descriptions

- structType **must** be XR_LOADER_INTERFACE_STRUCT_LOADER_INFO.

- structVersion **must** be a valid version of the structure. The value XR_LOADER_INFO_STRUCT_VERSION describes the current latest version of this structure.

- structSize **must** be the size in bytes of the current version of the structure (i.e. sizeof(XrNegotiateLoaderInfo)).

- minInterfaceVersion is the minimum runtime or API layer interface version supported by the loader.

- maxInterfaceVersion is the maximum valid version of the runtime or API layer interface version supported by the loader, currently defined using XR_CURRENT_LOADER_RUNTIME_VERSION or XR_CURRENT_LOADER_API_LAYER_VERSION.

- minApiVersion is the minimum supported version of the OpenXR API by the loader as formatted by XR_MAKE_VERSION. Patch is ignored.

- maxApiVersion is the maximum supported version of the OpenXR API by the loader as formatted by XR_MAKE_VERSION. Patch is ignored.

This structure is an input from the loader to the runtime in an xrNegotiateLoaderRuntimeInterface call, as well as from the loader to an API layer in an xrNegotiateLoaderApiLayerInterface call.

Valid Usage (Implicit)

- structType **must** be a valid XrLoaderInterfaceStructs value

The XrLoaderInterfaceStructs enumeration is defined as:

```
typedef enum XrLoaderInterfaceStructs {
    XR_LOADER_INTERFACE_STRUCT_UNINTIALIZED = 0,
    XR_LOADER_INTERFACE_STRUCT_LOADER_INFO = 1,
    XR_LOADER_INTERFACE_STRUCT_API_LAYER_REQUEST = 2,
    XR_LOADER_INTERFACE_STRUCT_RUNTIME_REQUEST = 3,
    XR_LOADER_INTERFACE_STRUCT_API_LAYER_CREATE_INFO = 4,
    XR_LOADER_INTERFACE_STRUCT_API_LAYER_NEXT_INFO = 5,
    XR_LOADER_INTERFACE_STRUCTS_MAX_ENUM = 0x7FFFFFFF
} XrLoaderInterfaceStructs;
```

This enumeration serves a similar purpose in the runtime and API layer interface negotiation (loader) API as XrStructureType serves in the application-facing API.

```
// Provided by XR_LOADER_VERSION_1_0
#define XR_LOADER_INFO_STRUCT_VERSION 1
```

XR_LOADER_INFO_STRUCT_VERSION is the current version of the XrNegotiateLoaderInfo structure. It is used to populate the XrNegotiateLoaderInfo::structVersion field.

```
// Provided by XR_LOADER_VERSION_1_0
#define XR_CURRENT_LOADER_RUNTIME_VERSION 1
```

XR_CURRENT_LOADER_RUNTIME_VERSION is the current version of the overall OpenXR Loader Runtime interface. It is used to populate maximum and minimum interface version fields in XrNegotiateLoaderInfo when loading a runtime.

```
// Provided by XR_LOADER_VERSION_1_0
#define XR_CURRENT_LOADER_API_LAYER_VERSION 1
```

XR_CURRENT_LOADER_API_LAYER_VERSION is the current version of the overall OpenXR Loader API Layer interface. It is used to populate maximum and minimum interface version fields in XrNegotiateLoaderInfo when loading an API layer.

The XrNegotiateRuntimeRequest structure is used to pass information about the runtime back to the loader.

The XrNegotiateRuntimeRequest structure is defined as:

```
typedef struct XrNegotiateRuntimeRequest {
    XrLoaderInterfaceStructs    structType;
    uint32_t                    structVersion;
    size_t                      structSize;
    uint32_t                    runtimeInterfaceVersion;
    XrVersion                   runtimeApiVersion;
    PFN_xrGetInstanceProcAddr   getInstanceProcAddr;
} XrNegotiateRuntimeRequest;
```

Member Descriptions

- structType **must** be XR_LOADER_INTERFACE_STRUCT_RUNTIME_REQUEST.

- structVersion **must** be a valid version of the structure. The value XR_RUNTIME_INFO_STRUCT_VERSION is used to describe the current version of this structure.

- structSize **must** be the size in bytes of the current version of the structure (i.e. sizeof(XrNegotiateRuntimeRequest))

- runtimeInterfaceVersion is the version of the runtime interface version being requested by the runtime. Must: not be outside of the bounds of the range XrNegotiateLoaderInfo ::minInterfaceVersion through XrNegotiateLoaderInfo::maxInterfaceVersion (inclusive).

- runtimeApiVersion is the version of the OpenXR API supported by this runtime as formatted by XR_MAKE_VERSION. Patch is ignored.

- getInstanceProcAddr is a pointer to the runtime's xrGetInstanceProcAddr implementation that will be used by the loader to populate a dispatch table of OpenXR functions supported by the runtime.

This is an output structure from runtime negotiation. The loader **must** populate structType, structVersion, and structSize to ensure correct interpretation by the runtime, while the runtime populates the rest of the fields in a successful call to xrNegotiateLoaderRuntimeInterface.

```
// Provided by XR_LOADER_VERSION_1_0
#define XR_RUNTIME_INFO_STRUCT_VERSION 1
```

XR_RUNTIME_INFO_STRUCT_VERSION is the current version of the XrNegotiateRuntimeRequest structure. It is used to populate the XrNegotiateRuntimeRequest::structVersion field.

3.4. API Layer Interface Negotiation

In order to negotiate the API layer interface version with the loader, an OpenXR API layer **must** implement the xrNegotiateLoaderApiLayerInterface function.

> *Note*
>
> The API described in this section is solely intended for use between an OpenXR loader and an API layer. Applications use the appropriate loader library for their platform to load the active runtime and configured API layers, rather than making these calls directly. This section is included in the specification to ensure consistency between runtimes in their interactions with the loader.
>
> Be advised that as this is not application-facing API, some of the typical OpenXR API conventions are not followed in this section.

The xrNegotiateLoaderApiLayerInterface function is defined as:

```
// Provided by XR_LOADER_VERSION_1_0
XrResult xrNegotiateLoaderApiLayerInterface(
    const XrNegotiateLoaderInfo*                loaderInfo,
    const char*                                 layerName,
    XrNegotiateApiLayerRequest*                 apiLayerRequest);
```

- `loaderInfo` **must** be a pointer to a valid XrNegotiateLoaderInfo structure.

- `layerName` **must** be NULL or a valid C-style NULL-terminated string listing the name of an API layer which the loader is attempting to negotiate with.

- `apiLayerRequest` **must** be a valid pointer to an XrNegotiateApiLayerRequest structure, with minimal initialization, as subsequently described, to be fully populated by the called API layer.

xrNegotiateLoaderApiLayerInterface **should** be directly exported by an API layer so that using e.g. `GetProcAddress` on Windows or `dlsym` on POSIX platforms returns a valid function pointer to it.

The API layer **must** return `XR_ERROR_INITIALIZATION_FAILED` if any of the following conditions on `loaderInfo` are true:

- XrNegotiateLoaderInfo::`structType` is not `XR_LOADER_INTERFACE_STRUCT_LOADER_INFO`

- XrNegotiateLoaderInfo::`structVersion` is not XR_LOADER_INFO_STRUCT_VERSION

- XrNegotiateLoaderInfo::`structSize` is not `sizeof(XrNegotiateLoaderInfo)`

The API layer **must** also return `XR_ERROR_INITIALIZATION_FAILED` if any of the following conditions on `apiLayerRequest` are true:

- XrNegotiateApiLayerRequest::`structType` is not `XR_LOADER_INTERFACE_STRUCT_API_LAYER_REQUEST`

- XrNegotiateApiLayerRequest::`structVersion` is not XR_API_LAYER_INFO_STRUCT_VERSION

- XrNegotiateApiLayerRequest::`structSize` is not `sizeof(XrNegotiateApiLayerRequest)`

The API layer **must** determine if it supports the loader's request. The API layer does not support the loader's request if either of the following is true:

- The API layer does not support the interface versions supported by the loader, as specified by the range XrNegotiateLoaderInfo::`minInterfaceVersion` through XrNegotiateLoaderInfo ::`maxInterfaceVersion` inclusive.

- The API layer does not support the API versions supported by the loader, ignoring "patch" version components, as specified by the range XrNegotiateLoaderInfo::`minApiVersion` through XrNegotiateLoaderInfo::`maxApiVersion` inclusive.

The API layer **must** return `XR_ERROR_INITIALIZATION_FAILED` if it does not support the loader's request.

If the function succeeds, the API layer **must** set the XrNegotiateApiLayerRequest ::`layerInterfaceVersion` with the API layer interface version it desires to support. The XrNegotiateApiLayerRequest::`layerInterfaceVersion` set **must** be in the range XrNegotiateLoaderInfo ::`minInterfaceVersion` through XrNegotiateLoaderInfo::`maxInterfaceVersion` inclusive.

If the function succeeds, the API layer **must** set the XrNegotiateApiLayerRequest::`layerApiVersion` with the API version of OpenXR it will execute under. The XrNegotiateApiLayerRequest::`layerApiVersion` set **must** be in the range XrNegotiateLoaderInfo::`minApiVersion` through XrNegotiateLoaderInfo::`maxApiVersion` inclusive.

If the function succeeds, the API layer **must** set the XrNegotiateApiLayerRequest::`getInstanceProcAddr` with a valid function pointer for the loader to use to query function pointers to the remaining OpenXR functions supported by the API layer.

If the function succeeds, the API layer **must** set the XrNegotiateApiLayerRequest::`createApiLayerInstance` with a valid function pointer to an implementation of xrCreateApiLayerInstance for the loader to use to create the instance through the API layer call chain.

If the function succeeds, the API layer **must** return `XR_SUCCESS`.

The API layer **must** not call into another API layer from its implementation of the xrNegotiateLoaderApiLayerInterface function. The loader **must** handle all API layer negotiations with each API layer individually.

Valid Usage (Implicit)

- `loaderInfo` **must** be a pointer to a valid XrNegotiateLoaderInfo structure
- `layerName` **must** be a null-terminated UTF-8 string
- `apiLayerRequest` **must** be a pointer to an XrNegotiateApiLayerRequest structure

---

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_INITIALIZATION_FAILED`

The XrNegotiateApiLayerRequest structure is used to pass information about the API layer back to the loader.

The XrNegotiateApiLayerRequest structure is defined as:

```
typedef struct XrNegotiateApiLayerRequest {
    XrLoaderInterfaceStructs        structType;
    uint32_t                        structVersion;
    size_t                          structSize;
    uint32_t                        layerInterfaceVersion;
    XrVersion                       layerApiVersion;
    PFN_xrGetInstanceProcAddr       getInstanceProcAddr;
    PFN_xrCreateApiLayerInstance    createApiLayerInstance;
} XrNegotiateApiLayerRequest;
```

Member Descriptions

- structType **must** be XR_LOADER_INTERFACE_STRUCT_API_LAYER_REQUEST.

- structVersion **must** be a valid version of the structure. The value XR_API_LAYER_INFO_STRUCT_VERSION is used to describe the current latest version of this structure.

- structSize **must** be the size in bytes of the current version of the structure (i.e. sizeof(XrNegotiateApiLayerRequest)).

- layerInterfaceVersion is the version of the API layer interface version being requested by the API layer. Should not be outside of the bounds of the range XrNegotiateLoaderInfo ::minInterfaceVersion through XrNegotiateLoaderInfo::maxInterfaceVersion (inclusive).

- layerApiVersion is the version of the OpenXR API supported by this API layer as formatted by XR_MAKE_VERSION. Patch is ignored.

- getInstanceProcAddr is a pointer to the API layer's xrGetInstanceProcAddr implementation that will be used by the loader to populate a dispatch table of OpenXR functions supported by the API layer.

- createApiLayerInstance is a pointer to the API layer's xrCreateApiLayerInstance implementation that will be used by the loader during a call to xrCreateInstance when an API layer is active. This is used because API layers need additional information at xrCreateInstance time.

This is an output structure from API layer negotiation. The loader **must** populate structType, structVersion, and structSize before calling to ensure correct interpretation by the API layer, while the API layer populates the rest of the fields in a successful call to xrNegotiateLoaderApiLayerInterface.

```
// Provided by XR_LOADER_VERSION_1_0
#define XR_API_LAYER_INFO_STRUCT_VERSION 1
```

XR_API_LAYER_INFO_STRUCT_VERSION is the current version of the XrNegotiateApiLayerRequest structure. It is used to populate the XrNegotiateApiLayerRequest::structVersion field.

The xrCreateApiLayerInstance function is defined as:

```
// Provided by XR_LOADER_VERSION_1_0
XrResult xrCreateApiLayerInstance(
    const XrInstanceCreateInfo*              info,
    const XrApiLayerCreateInfo*              layerInfo,
    XrInstance*                              instance);
```

Parameter Descriptions

- info is a pointer to the XrInstanceCreateInfo information passed by the application into the outer xrCreateInstance function.

- layerInfo is a pointer to an XrApiLayerCreateInfo structure that contains special information required by a API layer during its create instance process. This is generated by the loader.

- instance is a pointer to store the returned instance in, just as in the standard xrCreateInstance function.

An API layer's implementation of the xrCreateApiLayerInstance function is invoked during the loader's implementation of xrCreateInstance, if the layer in question is enabled.

An API layer needs additional information during xrCreateInstance calls, so each API layer **must** implement the xrCreateApiLayerInstance function, which is a special API layer function.

An API layer **must** not implement xrCreateInstance.

xrCreateApiLayerInstance **must** be called by the loader during its implementation of the xrCreateInstance function.

The loader **must** call the first API layer's xrCreateApiLayerInstance function passing in the pointer to the created XrApiLayerCreateInfo.

The XrApiLayerCreateInfo::`nextInfo` **must** be a linked-list of XrApiLayerNextInfo structures with information about each of the API layers that are to be enabled. Note that this does not operate like a `next` chain in the OpenXR application API, but instead describes the enabled API layers from outermost to innermost.

The API layer **may** validate that it is getting the correct next information by checking that the XrApiLayerNextInfo::`layerName` matches the expected value.

The API layer **must** use the information in its XrApiLayerNextInfo to call down the call chain to the next xrCreateApiLayerInstance:

- The API layer **must** copy the XrApiLayerCreateInfo structure into its own structure.
- The API layer **must** then update its copy of the XrApiLayerCreateInfo structure, setting XrApiLayerCreateInfo::`nextInfo` to point to the XrApiLayerNextInfo for the next API layer (e.g. `layerInfoCopy→nextInfo = layerInfo→nextInfo→next;`).
- The API layer **must** then use the pointer to its XrApiLayerCreateInfo structure (instead of the one that was passed in) when it makes a call to the xrCreateApiLayerInstance function.
- If the nested xrCreateApiLayerInstance call succeeds, the API layer **may** choose to setup its own dispatch table to the next API layer's functions using the returned XrInstance and the next API layer's xrGetInstanceProcAddr.
- The API layer **must** return the XrResult returned from the next API layer.

Valid Usage (Implicit)

- `info` **must** be a pointer to a valid XrInstanceCreateInfo structure
- `layerInfo` **must** be a pointer to a valid XrApiLayerCreateInfo structure
- `instance` **must** be a pointer to an XrInstance handle

Return Codes

Success

- `XR_SUCCESS`

Failure

- `XR_ERROR_INITIALIZATION_FAILED`

The XrApiLayerCreateInfo structure contains special information required by a API layer during its create instance process.

The XrApiLayerCreateInfo structure is defined as:

```
typedef struct XrApiLayerCreateInfo {
    XrLoaderInterfaceStructs    structType;
    uint32_t                    structVersion;
    size_t                      structSize;
    void*                       loaderInstance;
    char                        settings_file_location
[XR_API_LAYER_MAX_SETTINGS_PATH_SIZE];
    XrApiLayerNextInfo*         nextInfo;
} XrApiLayerCreateInfo;
```

Member Descriptions

- structType **must** be XR_LOADER_INTERFACE_STRUCT_API_LAYER_CREATE_INFO.

- structVersion is the version of the structure being supplied by the loader (i.e. XR_API_LAYER_CREATE_INFO_STRUCT_VERSION)

- structSize **must** be the size in bytes of the current version of the structure (i.e. sizeof(XrApiLayerCreateInfo))

- loaderInstance is deprecated and **must** be ignored.

- settings_file_location is the location of any usable API layer settings file. The size of settings_file_location is given by XR_API_LAYER_MAX_SETTINGS_PATH_SIZE. This is currently unused.

- nextInfo is a pointer to the XrApiLayerNextInfo structure which contains information to work with the next API layer in the chain.

Valid Usage (Implicit)

- structType **must** be a valid XrLoaderInterfaceStructs value

- loaderInstance **must** be a pointer value

- settings_file_location **must** be a null-terminated UTF-8 string whose length is less than or equal to XR_API_LAYER_MAX_SETTINGS_PATH_SIZE

- nextInfo **must** be a pointer to an XrApiLayerNextInfo structure

```
// Provided by XR_LOADER_VERSION_1_0
#define XR_API_LAYER_CREATE_INFO_STRUCT_VERSION 1
```

XR_API_LAYER_CREATE_INFO_STRUCT_VERSION is the current version of the XrApiLayerCreateInfo structure. It is used to populate the XrApiLayerCreateInfo::structVersion field.

```
// Provided by XR_LOADER_VERSION_1_0
#define XR_API_LAYER_MAX_SETTINGS_PATH_SIZE 512
```

XR_API_LAYER_MAX_SETTINGS_PATH_SIZE is the size of the XrApiLayerCreateInfo::settings_file_location field.

The XrApiLayerNextInfo structure:

The XrApiLayerNextInfo structure is defined as:

```
typedef struct XrApiLayerNextInfo {
    XrLoaderInterfaceStructs      structType;
    uint32_t                      structVersion;
    size_t                        structSize;
    char                          layerName[XR_MAX_API_LAYER_NAME_SIZE];
    PFN_xrGetInstanceProcAddr     nextGetInstanceProcAddr;
    PFN_xrCreateApiLayerInstance  nextCreateApiLayerInstance;
    struct XrApiLayerNextInfo*    next;
} XrApiLayerNextInfo;
```

Member Descriptions

- `structType` **must** be `XR_LOADER_INTERFACE_STRUCT_API_LAYER_NEXT_INFO`

- `structVersion` **must** be a valid version of the structure and the version being supplied by the loader (i.e. `XR_API_LAYER_NEXT_INFO_STRUCT_VERSION`).

- `structSize` **must** be the size in bytes of the current version of the structure (i.e. `sizeof(XrApiLayerNextInfo)`)

- `layerName` is the name of the intended next API layer, used to verify and debug the API layer chain.

- `nextGetInstanceProcAddr` is a pointer to the next API layer's `xrGetInstanceProcAddr`. This is intended for use in populating a dispatch table to the next implementations in the chain.

- `nextCreateApiLayerInstance` is a pointer to the `xrCreateApiLayerInstance` function implementation in the next API layer. This is to be called **after** the API layer has done any localized creation, but **before** the API layer records any function addresses from the next API layer using `xrGetInstanceProcAddr`.

- `next` is a pointer to the `XrApiLayerNextInfo` for the next API layer. If no API layer is after this, it will be `NULL`.

Valid Usage (Implicit)

- `structType` **must** be a valid `XrLoaderInterfaceStructs` value

- `layerName` **must** be a null-terminated UTF-8 string whose length is less than or equal to `XR_MAX_API_LAYER_NAME_SIZE`

- `nextGetInstanceProcAddr` **must** be a valid `PFN_xrGetInstanceProcAddr` value

- `nextCreateApiLayerInstance` **must** be a valid `PFN_xrCreateApiLayerInstance` value

- `next` **must** be a pointer to an `XrApiLayerNextInfo` structure

```
// Provided by XR_LOADER_VERSION_1_0
#define XR_API_LAYER_NEXT_INFO_STRUCT_VERSION 1
```

XR_API_LAYER_NEXT_INFO_STRUCT_VERSION is the current version of the XrApiLayerNextInfo structure. It is used to populate the XrApiLayerNextInfo::`structVersion` field.

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:

```
// Provided by XR_VERSION_1_0
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

The XrApiLayerProperties structure is defined as:

```c
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 XrInstanceCreateInfo ::`enabledApiLayerNames` 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:

```
// Provided by XR_VERSION_1_0
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.

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:

```
// Provided by XR_VERSION_1_0
XrResult xrCreateInstance(
    const XrInstanceCreateInfo*                 createInfo,
    XrInstance*                                 instance);
```

- `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_EXTENSION_DEPENDENCY_NOT_ENABLED

- XR_ERROR_API_VERSION_UNSUPPORTED

- XR_ERROR_API_LAYER_NOT_PRESENT

The XrInstanceCreateInfo structure is defined as:

```
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 XrInstanceCreateInfo::`createFlags` member is of the following type, and contains a bitwise-OR of zero or more of the bits defined in XrInstanceCreateFlagBits.

```
typedef XrFlags64 XrInstanceCreateFlags;
```

Valid bits for XrInstanceCreateFlags are defined by XrInstanceCreateFlagBits.

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

There are currently no instance creation flag bits defined. This is reserved for future use.

The XrApplicationInfo structure is defined as:

```
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;
```

<div align="center">

Member Descriptions

</div>

- 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.

Useful values for apiVersion include XR_API_VERSION_1_0 and XR_API_VERSION_1_1.

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:

```
// Provided by XR_VERSION_1_0
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.

4.3. Instance Information

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

```
// Provided by XR_VERSION_1_0
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.

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.

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

The XrEventDataInstanceLossPending structure is defined as:

```
// Provided by XR_VERSION_1_0
typedef struct XrEventDataInstanceLossPending {
    XrStructureType    type;
    const void*        next;
    XrTime             lossTime;
} XrEventDataInstanceLossPending;
```

- `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.

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.

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.

```
// Provided by XR_VERSION_1_0
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_LOST`

The `XR_MAX_RESULT_STRING_SIZE` enumerant defines the size of the buffer passed to xrResultToString.

```
#define XR_MAX_RESULT_STRING_SIZE          64
```

The xrStructureTypeToString function is defined as:

```
// Provided by XR_VERSION_1_0
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`

The XR_MAX_STRUCTURE_NAME_SIZE enumerant defines the size of the buffer passed to xrStructureTypeToString.

```
#define XR_MAX_STRUCTURE_NAME_SIZE        64
```

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 XrSystemId is returned by xrGetSystem representing the system of devices the runtime will use to support a given 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 session, which can then be used to accept input from the user and output rendered frames. The application also provides suggested bindings from its actions to any number of input sources. The runtime **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 **may** vary from runtime to runtime. For example, a runtime that is capable of mapping from one tracking system's space to another's **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 **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 **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.

```
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 **may** be supported by OpenXR runtimes are:

- `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.

```
// Provided by XR_VERSION_1_0
#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:

```
// Provided by XR_VERSION_1_0
XrResult xrGetSystem(
    XrInstance                                  instance,
    const XrSystemGetInfo*                      getInfo,
    XrSystemId*                                 systemId);
```

- `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 = {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:

```
// Provided by XR_VERSION_1_0
XrResult xrGetSystemProperties(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    XrSystemProperties*                         properties);
```

<div style="border:1px solid #ccc; padding:1em;">

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.

</div>

An application **can** call xrGetSystemProperties to retrieve information about the system such as vendor ID, system name, and graphics and tracking properties.

<div style="border:1px solid #ccc; padding:1em;">

Valid Usage (Implicit)

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

- `properties` **must** be a pointer to an XrSystemProperties structure

</div>

- XR_SUCCESS

- 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:

```
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.

- 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.

The runtime **must** report a valid vendor ID for the system. The vendor ID **must** be either the USB vendor ID defined for the physical device or a Khronos vendor ID.

The XrSystemGraphicsProperties structure is defined as:

```
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.

```
// Provided by XR_VERSION_1_0
#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:

```
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.

```
// Provided by XR_VERSION_1_0
#define XR_NULL_PATH 0
```

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:

```
// Provided by XR_VERSION_1_0
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

```
// Provided by XR_VERSION_1_0
XrResult xrPathToString(
    XrInstance                                  instance,
    XrPath                                      path,
    uint32_t                                    bufferCapacityInput,
    uint32_t*                                   bufferCountOutput,
    char*                                       buffer);
```

- `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 to `buffer` (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 the Buffer Size Parameters section 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 XrInstance, in the form of a `NULL` terminated string placed into a *caller-allocated* buffer. Since the mapping between a well-formed path name string and an `XrPath` is bijective, there will always be exactly one string for each valid `XrPath` value. This can be useful if the calling application receives an `XrPath` value that they had not previously retrieved via xrStringToPath. During the lifetime of the given XrInstance, the path name string retrieved by this function for a given valid `XrPath` will not change. For invalid paths, including XR_NULL_PATH, `XR_ERROR_PATH_INVALID` **must** be returned.

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. Top level /user paths

Some paths are used to refer to entities that are filling semantic roles in the system. These paths are all under the *per* subtree.

The reserved user paths are:

Reserved Semantic Paths

- *per/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.

- *per/hand/right* represents the user's right hand in analog to the left hand.

- *per/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.

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

- *per/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

Interaction profiles 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 subpaths that use identifiers and component names that do not appear in this specification or otherwise do not follow the pattern specified below. Input subpaths further qualify top-level */user* paths to form binding paths. For this reason, they are often shown starting with … or omitting path components before `/input` or `/output` entirely. The input subpaths considered valid when combined with any given top-level */user* path vary by interaction profile.

Each input subpath **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.

- thumb_resting_surfaces - Any surfaces that a thumb may naturally rest on. This may include, but is not limited to, face buttons, thumbstick, and thumbrest (Provided by `XR_VERSION_1_1`)

- stylus - Tip that can be used for writing or drawing. May be able to detect various pressure levels (Provided by `XR_VERSION_1_1`)

- trigger_curl - This sensor detects how pointed or curled the user's finger is on the trigger: 0 = fully pointed, 1 = finger flat on surface (Provided by `XR_VERSION_1_1`)

- trigger_slide - This sensor represents how far the user is sliding their index finger along the surface of the trigger: 0 = finger flat on the surface, 1 = finger fully drawn back (Provided by `XR_VERSION_1_1`)

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 *ID/user/hand/left* and *ID/user/hand/right* are:
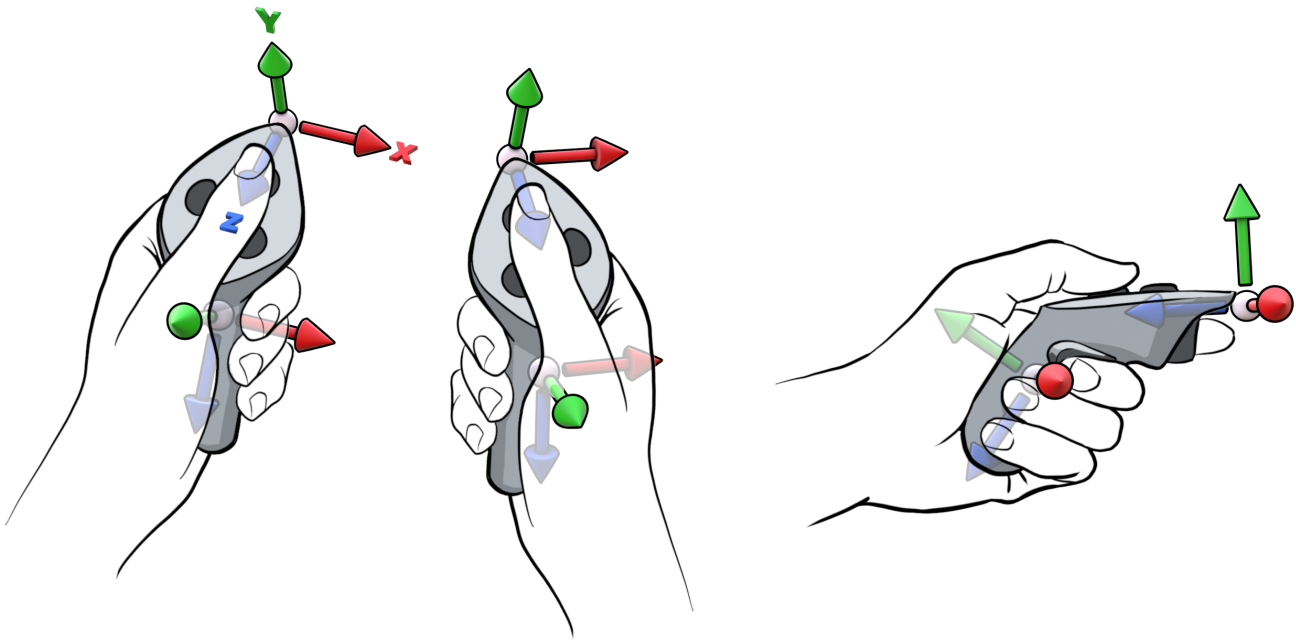
Figure 2. Example grip and aim poses for generic motion controllers

- 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, often a ray emerging from the hand at a target pointed by moving 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.

• grip_surface - (Provided by `XR_VERSION_1_1`) A pose that allows applications to reliably anchor visual content relative to the user's physical hand, whether the user's hand is tracked directly or its position and orientation is inferred by a physical controller. The grip_surface pose is defined as follows:

  ◦ The grip_surface position: The user's physical palm centroid, at the surface of the palm. For the avoidance of doubt, the palm does not include fingers.

  ◦ The grip_surface orientation's +X axis: When a user is holding the controller and straightens their index fingers pointing forward, the ray that is normal (perpendicular) to the user's palm (away from the palm in the left hand, into the palm in the right hand).

  ◦ The grip_surface orientation's -Z axis: When a user is holding the controller and straightens their index finger, the ray that is parallel to their finger's pointing direction.

  ◦ The grip_surface orientation's +Y axis: orthogonal to +Z and +X using the right-hand rule.
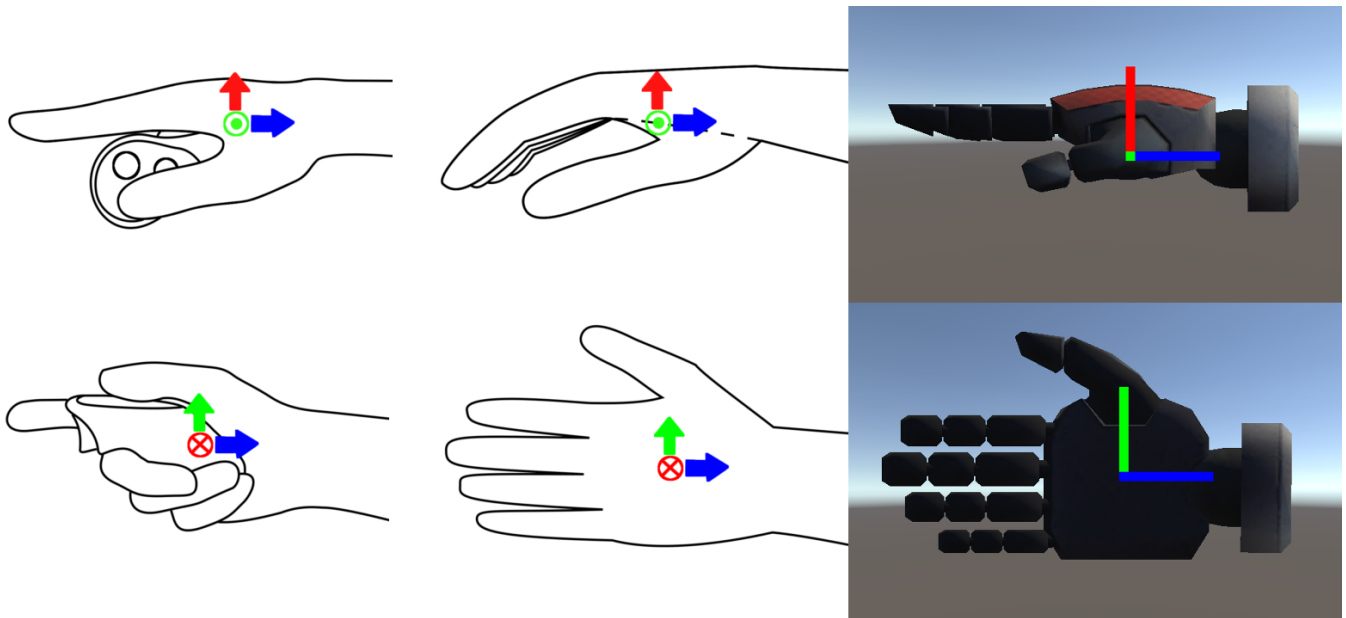
Figure 3. Example grip_surface pose for (from left to right) a generic motion controller, tracked hand, and a digital hand avatar. The X axis is depicted in red. The Y axis is depicted in green. The Z axis is depicted in blue.

> **Note**
>
> When the `XR_EXT_hand_interaction` extension is available and enabled, additional "pinch_ext" and "poke_ext" standard pose identifiers are available, and a path is added to all interaction profiles valid for */user/hand/left* or */user/hand/right*. This includes interaction profiles defined in the core spec and in extensions.

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.

- proximity - The user is in physical proximity of input source. This **may** be present for any kind of input source representing a physical component, such as a button, if the device includes the

necessary sensor. The state of a "proximity" component **must** be `XR_TRUE` if the same input source is returning `XR_TRUE` for either a "touch" or any other component that implies physical contact. The runtime **may** return `XR_TRUE` for "proximity" when "touch" returns `XR_FALSE` which would indicate that the user is hovering just above, but not touching the input source in question. "proximity" components are always boolean. (Provided by `XR_VERSION_1_1`)

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
- haptic_trigger - A haptic element located in the trigger (Provided by `XR_VERSION_1_1`)
- haptic_thumb - A haptic element located in the resting place of the thumb, like under the touchpad (Provided by `XR_VERSION_1_1`)

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 defacto 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*

> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> *Note*
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> *Note*
>
> When the `XR_EXT_hand_interaction` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/pinch_ext/pose*
> - *…/input/poke_ext/pose*

6.4.2. ByteDance PICO Neo 3 controller Profile

(Provided by `XR_VERSION_1_1`)

Path: */interaction_profiles/bytedance/pico_neo3_controller*

Valid for user paths:

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

This interaction profile represents the input sources and haptics on the ByteDance PICO Neo3 Controller.

- On /user/hand/left only:
  - *…/input/x/click*
  - *…/input/x/touch*
  - *…/input/y/click*
  - *…/input/y/touch*
- On /user/hand/right only:
  - *…/input/a/click*
  - *…/input/a/touch*
  - *…/input/b/click*
  - *…/input/b/touch*
- *…/input/menu/click*
- *…/input/system/click* (**may** not be available for application use)
- *…/input/trigger/click*
- *…/input/trigger/value*
- *…/input/trigger/touch*
- *…/input/thumbstick/y*
- *…/input/thumbstick/x*
- *…/input/thumbstick/click*
- *…/input/thumbstick/touch*
- *…/input/squeeze/click*
- *…/input/squeeze/value*
- *…/input/grip/pose*
- *…/input/aim/pose*

- *…/output/haptic*

> **Note**
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> **Note**
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> **Note**
>
> When the `XR_EXT_hand_interaction` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/pinch_ext/pose*
> - *…/input/poke_ext/pose*

6.4.3. ByteDance PICO 4 controller Profile

(Provided by `XR_VERSION_1_1`)

Path: */interaction_profiles/bytedance/pico4_controller*

Valid for user paths:

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

This interaction profile represents the input sources and haptics on the ByteDance PICO 4 Controller.

- 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/trigger/click*
- *…/input/trigger/value*
- *…/input/trigger/touch*
- *…/input/thumbstick/y*
- *…/input/thumbstick/x*
- *…/input/thumbstick/click*
- *…/input/thumbstick/touch*
- *…/input/squeeze/click*
- *…/input/squeeze/value*
- *…/input/grip/pose*
- *…/input/aim/pose*
- *…/output/haptic*

> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> *Note*
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> *Note*
>
> When the `XR_EXT_hand_interaction` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/pinch_ext/pose*
> - *…/input/poke_ext/pose*

6.4.4. ByteDance PICO G3 controller Profile

(Provided by `XR_VERSION_1_1`)

Path: */interaction_profiles/bytedance/pico_g3_controller*

Valid for user paths:

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

This interaction profile represents the input sources and haptics on the ByteDance PICO G3 Controller.

- *…/input/trigger/click*
- *…/input/trigger/value*
- *…/input/menu/click*
- *…/input/grip/pose*
- *…/input/aim/pose*
- *…/input/thumbstick*
- *…/input/thumbstick/click*

> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> *Note*
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> *Note*
>
> When the `XR_EXT_hand_interaction` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/pinch_ext/pose*
> - *…/input/poke_ext/pose*

6.4.5. 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*

> **ℹ**
> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> **ℹ**
> *Note*
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

6.4.6. HP Mixed Reality Motion Controller Profile

(Provided by `XR_VERSION_1_1`)

Path: */interaction_profiles/hp/mixed_reality_controller*

Valid for user paths:

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

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

- On */user/hand/left* only:
  - *…/input/x/click*
  - *…/input/y/click*
- On */user/hand/right* only:
  - *…/input/a/click*
  - *…/input/b/click*
- *…/input/menu/click*
- *…/input/squeeze/value*
- *…/input/trigger/value*
- *…/input/thumbstick/x*
- *…/input/thumbstick/y*
- *…/input/thumbstick/click*
- *…/input/grip/pose*
- *…/input/aim/pose*
- *…/output/haptic*

> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *.../input/grip_surface/pose*

> *Note*
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *.../input/grip_surface/pose*

> *Note*
>
> When the `XR_EXT_hand_interaction` extension is available and enabled, this interaction profile **must** also support
>
> - *.../input/pinch_ext/pose*
> - *.../input/poke_ext/pose*

6.4.7. 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*

> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *.../input/grip_surface/pose*

> *Note*
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *.../input/grip_surface/pose*

> *Note*
>
> When the `XR_EXT_hand_interaction` extension is available and enabled, this interaction profile **must** also support
>
> - *.../input/pinch_ext/pose*
> - *.../input/poke_ext/pose*

6.4.8. HTC Vive Cosmos Controller Profile

(Provided by `XR_VERSION_1_1`)

Path: */interaction_profiles/htc/vive_cosmos_controller*

Valid for user paths:

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

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

Supported component paths:

- On */user/hand/left* only:
  - *.../input/x/click*
  - *.../input/y/click*
  - *.../input/menu/click*

- On */user/hand/right* only:
  - *…/input/a/click*
  - *…/input/b/click*
  - *…/input/system/click* (**may** not be available for application use)
- *…/input/shoulder/click*
- *…/input/squeeze/click*
- *…/input/trigger/click*
- *…/input/trigger/value*
- *…/input/thumbstick/x*
- *…/input/thumbstick/y*
- *…/input/thumbstick/click*
- *…/input/thumbstick/touch*
- *…/input/grip/pose*
- *…/input/aim/pose*
- *…/output/haptic*

> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> *Note*
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> *Note*
>
> When the `XR_EXT_hand_interaction` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/pinch_ext/pose*
> - *…/input/poke_ext/pose*

6.4.9. HTC Vive Focus 3 Controller Profile

(Provided by `XR_VERSION_1_1`)

Path: */interaction_profiles/htc/vive_focus3_controller*

Valid for user paths:

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

This interaction profile represents the input sources and haptics on the Vive Focus 3 Controller.

- On */user/hand/left* only:
  - *…/input/x/click*
  - *…/input/y/click*
  - *…/input/menu/click*
- On */user/hand/right* only:
  - *…/input/a/click*
  - *…/input/b/click*
  - *…/input/system/click* (**may** not be available for application use)
- *…/input/squeeze/click*
- *…/input/squeeze/touch*
- *…/input/squeeze/value*
- *…/input/trigger/click*
- *…/input/trigger/touch*
- *…/input/trigger/value*
- *…/input/thumbstick/x*
- *…/input/thumbstick/y*
- *…/input/thumbstick/click*
- *…/input/thumbstick/touch*
- *…/input/thumbrest/touch*
- *…/input/grip/pose*
- *…/input/aim/pose*
- *…/output/haptic*

> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> **Note**
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *.../input/grip_surface/pose*

> **Note**
>
> When the `XR_EXT_hand_interaction` extension is available and enabled, this interaction profile **must** also support
>
> - *.../input/pinch_ext/pose*
> - *.../input/poke_ext/pose*

6.4.10. 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*
- *.../input/mute_mic/click*

6.4.11. Magic Leap 2 Controller Profile

(Provided by `XR_VERSION_1_1`)

Path: */interaction_profiles/ml/ml2_controller*

Valid for user paths:

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

This interaction profile represents the input sources and haptics on the Magic Leap 2 controller.

Supported component paths:

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

> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *.../input/grip_surface/pose*

> *Note*
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *.../input/grip_surface/pose*

> *Note*
>
> When the `XR_EXT_hand_interaction` extension is available and enabled, this interaction profile **must** also support
>
> - *.../input/pinch_ext/pose*
> - *.../input/poke_ext/pose*

6.4.12. 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/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*

> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> *Note*
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> *Note*
>
> When the `XR_EXT_hand_interaction` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/pinch_ext/pose*
> - *…/input/poke_ext/pose*

6.4.13. 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:

- *…/input/menu/click*
- *…/input/view/click*
- *…/input/a/click*
- *…/input/b/click*
- *…/input/x/click*
- *…/input/y/click*
- *…/input/dpad_down/click*
- *…/input/dpad_right/click*
- *…/input/dpad_up/click*
- *…/input/dpad_left/click*
- *…/input/shoulder_left/click*
- *…/input/shoulder_right/click*
- *…/input/thumbstick_left/click*
- *…/input/thumbstick_right/click*
- *…/input/trigger_left/value*
- *…/input/trigger_right/value*
- *…/input/thumbstick_left/x*
- *…/input/thumbstick_left/y*
- *…/input/thumbstick_right/x*

- *…/input/thumbstick_right/y*
- *…/output/haptic_left*
- *…/output/haptic_right*
- *…/output/haptic_left_trigger*
- *…/output/haptic_right_trigger*

6.4.14. 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*

> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> *Note*
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> *Note*
>
> When the `XR_EXT_hand_interaction` extension is available and enabled, this interaction profile **must** also support
>
> - *.../input/pinch_ext/pose*
> - *.../input/poke_ext/pose*

6.4.15. 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*
- *.../input/trigger/value*
- *.../input/trigger/touch*
- *.../input/trigger/proximity* (Provided by `XR_VERSION_1_1`)
- *.../input/thumb_resting_surfaces/proximity* (Provided by `XR_VERSION_1_1`)
- *.../input/thumbstick/x*

- *…/input/thumbstick/y*
- *…/input/thumbstick/click*
- *…/input/thumbstick/touch*
- *…/input/thumbrest/touch*
- *…/input/grip/pose*
- *…/input/aim/pose*
- *…/output/haptic*

> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> *Note*
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> *Note*
>
> When the `XR_EXT_hand_interaction` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/pinch_ext/pose*
> - *…/input/poke_ext/pose*

6.4.16. Meta Touch Pro Controller Profile

(Provided by `XR_VERSION_1_1`)

Path: */interaction_profiles/meta/touch_pro_controller*

Valid for user paths:

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

This interaction profile represents the input sources and haptics on the Meta Touch Pro 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*
- *…/input/trigger/value*
- *…/input/trigger/touch*
- *…/input/trigger/proximity*
- *…/input/trigger_curl/value*
- *…/input/trigger_slide/value*
- *…/input/thumb_resting_surfaces/proximity*
- *…/input/thumbstick/x*
- *…/input/thumbstick/y*
- *…/input/thumbstick/click*
- *…/input/thumbstick/touch*
- *…/input/thumbrest/touch*
- *…/input/thumbrest/force*
- *…/input/stylus/force*
- *…/input/grip/pose*
- *…/input/aim/pose*
- *…/output/haptic*
- *…/output/haptic_trigger*
- *…/output/haptic_thumb*

Note

When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support

- *…/input/grip_surface/pose*

Note

When the `XR_EXT_hand_interaction` extension is available and enabled, this interaction profile **must** also support

- *…/input/pinch_ext/pose*
- *…/input/poke_ext/pose*

6.4.17. Meta Touch Plus Controller Profile

(Provided by `XR_VERSION_1_1`)

Path: */interaction_profiles/meta/touch_plus_controller*

Valid for user paths:

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

This interaction profile represents the input sources and haptics on the Meta Touch Plus 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*

- *…/input/trigger/value*

- *…/input/trigger/touch*

- *…/input/trigger/force*

- *…/input/trigger/proximity*

- *…/input/trigger_curl/value*

- *…/input/trigger_slide/value*

- *…/input/thumb_resting_surfaces/proximity*

- *…/input/thumbstick/x*

- *…/input/thumbstick/y*

- *…/input/thumbstick/click*

- *…/input/thumbstick/touch*

- *…/input/thumbrest/touch*

- *…/input/grip/pose*

- *…/input/aim/pose*

- *…/output/haptic*

> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> *Note*
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> *Note*
>
> When the `XR_EXT_hand_interaction` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/pinch_ext/pose*
> - *…/input/poke_ext/pose*

6.4.18. Meta Touch Controller (Rift CV1) Profile

(Provided by `XR_VERSION_1_1`)

Path: *interaction_profiles/meta/touch_controller_rift_cv1*

Valid for user paths:

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

This interaction profile represents the input sources and haptics on the Oculus Touch controller and is a legacy profile added to specifically represent the controller shipped with the Rift CV1.

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*
- *…/input/trigger/value*
- *…/input/trigger/touch*
- *…/input/trigger/proximity*

- *.../input/thumb_resting_surfaces/proximity*

- *.../input/thumbstick/x*

- *.../input/thumbstick/y*

- *.../input/thumbstick/click*

- *.../input/thumbstick/touch*

- *.../input/thumbrest/touch*

- *.../input/grip/pose*

- *.../input/aim/pose*

- *.../output/haptic*

> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *.../input/grip_surface/pose*

> *Note*
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *.../input/grip_surface/pose*

> *Note*
>
> When the `XR_EXT_hand_interaction` extension is available and enabled, this interaction profile **must** also support
>
> - *.../input/pinch_ext/pose*
> - *.../input/poke_ext/pose*

6.4.19. Meta Touch Controller (Rift S / Quest 1) Profile

(Provided by `XR_VERSION_1_1`)

Path: */interaction_profiles/meta/touch_controller_quest_1_rift_s*

Valid for user paths:

- */user/hand/left*

- */user/hand/right*

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

a legacy profile added to specifically represent the controller shipped with the Rift S and Quest 1.

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*
- *…/input/trigger/value*
- *…/input/trigger/touch*
- *…/input/trigger/proximity*
- *…/input/thumb_resting_surfaces/proximity*
- *…/input/thumbstick/x*
- *…/input/thumbstick/y*
- *…/input/thumbstick/click*
- *…/input/thumbstick/touch*
- *…/input/grip/pose*
- *…/input/aim/pose*
- *…/output/haptic*

> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

6.4.20. Meta Touch Controller (Quest 2) Profile

(Provided by `XR_VERSION_1_1`)

Path: */interaction_profiles/meta/touch_controller_quest_2*

Valid for user paths:

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

This interaction profile represents the input sources and haptics on the Oculus Touch controller and is a legacy profile added to specifically represent the controller shipped with the Quest 2.

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*
- *.../input/trigger/value*
- *.../input/trigger/touch*
- *.../input/trigger/proximity*
- *.../input/thumb_resting_surfaces/proximity*
- *.../input/thumbstick/x*
- *.../input/thumbstick/y*
- *.../input/thumbstick/click*
- *.../input/thumbstick/touch*
- *.../input/thumbrest/touch*
- *.../input/grip/pose*
- *.../input/aim/pose*
- *.../output/haptic*

> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *.../input/grip_surface/pose*

> *Note*
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *.../input/grip_surface/pose*

> *Note*
>
> When the `XR_EXT_hand_interaction` extension is available and enabled, this interaction profile **must** also support
>
> - *.../input/pinch_ext/pose*
> - *.../input/poke_ext/pose*

6.4.21. Samsung Odyssey Controller Profile

(Provided by `XR_VERSION_1_1`)

Path: *interaction_profiles/samsung/odyssey_controller*

Valid for user paths:

- *\/user\/hand\/left*
- *\/user\/hand\/right*

This interaction profile represents the input sources and haptics on the Samsung Odyssey Controller. It is exactly the same, with the exception of the name of the interaction profile, as the Microsoft Mixed Reality Controller interaction profile. It enables the application to differentiate the newer form factor of motion controller released with the Samsung Odyssey headset. It enables the application to customize the appearance and experience of the controller differently from the original mixed reality motion controller.

Supported component paths:

- *…\/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*

> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *…\/input\/grip_surface\/pose*

> *Note*
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *…\/input\/grip_surface\/pose*

> *Note*
>
> When the `XR_EXT_hand_interaction` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/pinch_ext/pose*
> - *…/input/poke_ext/pose*

6.4.22. 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*

> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> *Note*
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> *Note*
>
> When the `XR_EXT_hand_interaction` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/pinch_ext/pose*
> - *…/input/poke_ext/pose*

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.

```
XR_DEFINE_HANDLE(XrSpace)
```

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, LOCAL_FLOOR, 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

The XrReferenceSpaceType enumeration is defined as:

```
typedef enum XrReferenceSpaceType {
    XR_REFERENCE_SPACE_TYPE_VIEW = 1,
    XR_REFERENCE_SPACE_TYPE_LOCAL = 2,
    XR_REFERENCE_SPACE_TYPE_STAGE = 3,
  // Provided by XR_VERSION_1_1
    XR_REFERENCE_SPACE_TYPE_LOCAL_FLOOR = 1000426000,
    XR_REFERENCE_SPACE_TYPE_MAX_ENUM = 0x7FFFFFFF
} XrReferenceSpaceType;
```

Brief introductions to core reference space types follow. Each has full requirements in a subsequent section, linked from these descriptions.

<div style="border: 1px solid #ccc; padding: 1em;">

<p align="center">**Enumerant Descriptions**</p>

- `XR_REFERENCE_SPACE_TYPE_VIEW`. The `VIEW` reference 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.

  Runtimes **must** support `VIEW` 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.

  Runtimes **must** support `LOCAL` 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 xrGetReferenceSpaceBoundsRect to determine the extents of the `STAGE` reference space's XZ bounds rectangle, if defined.

  Support for the `STAGE` reference space is **optional**.

- `XR_REFERENCE_SPACE_TYPE_LOCAL_FLOOR` (provided by `XR_VERSION_1_1`) Similar to `LOCAL` space, the `LOCAL_FLOOR` reference space establishes a world-locked origin, gravity-aligned to exclude pitch and roll, with +Y up, +X to the right, and -Z forward. However, the origin of this space is defined to be on an estimate of the floor level.

  Runtimes **must** support `LOCAL_FLOOR` reference space.

</div>

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. 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.

7.1.1. View Reference Space

The `XR_REFERENCE_SPACE_TYPE_VIEW` or `VIEW` reference 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.

The `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 the `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.

7.1.2. Local Reference Space

The `XR_REFERENCE_SPACE_TYPE_LOCAL` or `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.

When a user needs to recenter the `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 XrEventDataReferenceSpaceChangePending::`changeTime` 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 original origin of `LOCAL` space.

7.1.3. Stage Reference Space

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 xrGetReferenceSpaceBoundsRect to determine the extents of the `STAGE` reference space's XZ bounds rectangle, if defined.

The `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 space definition, the runtime **must** queue the XrEventDataReferenceSpaceChangePending event, with the new STAGE space origin only taking effect for xrLocateSpace or xrLocateViews calls whose XrTime parameter is greater than or equal to the XrEventDataReferenceSpaceChangePending::changeTime 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 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 original origin of the STAGE space.

7.1.4. Local Floor Reference Space

Local floor reference space, indicated by XR_REFERENCE_SPACE_TYPE_LOCAL_FLOOR, is closely related to the LOCAL reference space. It always aligns with the LOCAL space, and matches it in X and Z position. However, unlike the LOCAL space, the LOCAL_FLOOR space has its Y axis origin on the runtime's best estimate of the floor level under the origin of the LOCAL space.

The location of the origin of the LOCAL_FLOOR space **must** match the LOCAL space in the X and Z coordinates but not in the Y coordinate.

The orientation of the LOCAL_FLOOR space **must** match the LOCAL space.

The runtime **must** establish the Y axis origin at its best estimate of the floor level under the origin of the LOCAL space space, subject to requirements under the following conditions to match the floor level of the STAGE space.

If all of the following conditions are true, the Y axis origin of the LOCAL_FLOOR space **must** match the Y axis origin of the STAGE space:

- the STAGE space is supported
- the location of the LOCAL space relative to the STAGE space has valid position (XR_SPACE_LOCATION_POSITION_VALID_BIT is set)
- bounds are available from xrGetReferenceSpaceBoundsRect for the STAGE space
- the position of the LOCAL space relative to the STAGE space is within the STAGE space XZ bounds

That is, if there is a stage with bounds, and if the local space and thus the local floor is logically within the stage, the local floor and the stage share the same floor level.

When the origin of the LOCAL space is changed in orientation or XZ position, the origin of the

`LOCAL_FLOOR` space **must** also change accordingly.

When a change in origin of the `LOCAL_FLOOR` space occurs, the runtime **must** queue the XrEventDataReferenceSpaceChangePending event, with the changed `LOCAL_FLOOR` space origin only taking effect for xrLocateSpace or xrLocateViews calls whose `XrTime` parameter is greater than or equal to the XrEventDataReferenceSpaceChangePending::`changeTime` in that event.

The xrGetReferenceSpaceBoundsRect function is defined as:

```
// Provided by XR_VERSION_1_0
XrResult xrGetReferenceSpaceBoundsRect(
    XrSession                               session,
    XrReferenceSpaceType                    referenceSpaceType,
    XrExtent2Df*                            bounds);
```

Parameter Descriptions

- `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.

XR systems **may** have limited real world spatial ranges in which users can freely move around while remaining tracked. Applications sometimes 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 xrGetReferenceSpaceBoundsRect.

When called, 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 XrExtent2Df::`width` and 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 support boundaries. If a runtime is unable to provide bounds for a given space, `XR_SPACE_BOUNDS_UNAVAILABLE` **must** be returned and all fields of `bounds` **must** be set to 0.

The returned extents are expressed relative to the natural origin of the provided XrReferenceSpaceType and **must** not incorporate any origin offsets specified by the application during calls to xrCreateReferenceSpace.

The runtime **must** return `XR_ERROR_REFERENCE_SPACE_UNSUPPORTED` if the XrReferenceSpaceType passed in `referenceSpaceType` is not supported by this `session`.

When a runtime will begin operating with updated space bounds, the runtime **must** queue a

corresponding XrEventDataReferenceSpaceChangePending event.

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_PENDING

- 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 structure is defined as:

```
// Provided by XR_VERSION_1_0
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 `poseInPreviousSpace` 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.

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 XrFrameState::`predictedDisplayTime` of future frames to avoid threshold issues with applications that calculate future frame times using XrFrameState::`predictedDisplayPeriod`.

The `poseInPreviousSpace` 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.

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

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:

```
// Provided by XR_VERSION_1_0
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 the Buffer Size Parameters section 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:

```
// Provided by XR_VERSION_1_0
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`.

<div>

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

</div>

<div>

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`

</div>

The XrReferenceSpaceCreateInfo structure is defined as:

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

7.3.3. xrCreateActionSpace

The xrCreateActionSpace function is defined as:

```
// Provided by XR_VERSION_1_0
XrResult xrCreateActionSpace(
    XrSession                           session,
    const XrActionSpaceCreateInfo*      createInfo,
    XrSpace*                            space);
```

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 XrSession handle. See Action Spaces Lifetime for details.

The runtime **must** return `XR_ERROR_ACTION_TYPE_MISMATCH` if the action provided in XrActionSpaceCreateInfo::`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:

```
// Provided by XR_VERSION_1_0
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.

```
// Provided by XR_VERSION_1_0
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 XrPosef::`position` and XrPosef::`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.

During tracking loss, the runtime **should** return a location with `XR_SPACE_LOCATION_POSITION_VALID_BIT` and `XR_SPACE_LOCATION_ORIENTATION_VALID_BIT` set and `XR_SPACE_LOCATION_POSITION_TRACKED_BIT` and `XR_SPACE_LOCATION_ORIENTATION_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 continue to make use of the runtime's latest knowledge of the world.

If an XrSpaceVelocity structure is chained to the XrSpaceLocation::`next` pointer, 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 XrSpaceLocation::`next` pointer, 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.

```
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.

- `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: XrEyeGazeSampleTimeEXT, XrSpaceVelocity

- `locationFlags` **must** be `0` or a valid combination of XrSpaceLocationFlagBits values

The XrSpaceLocation::`locationFlags` member is of the following type, and contains a bitwise-OR of zero or more of the bits defined in XrSpaceLocationFlagBits.

```
typedef XrFlags64 XrSpaceLocationFlags;
```

Valid bits for XrSpaceLocationFlags are defined by XrSpaceLocationFlagBits, which is specified as:

```
// 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;
```

The flag bits have the following meanings:

- `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:

```
// Provided by XR_VERSION_1_0
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 the origin 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 XrSpaceVelocity::`velocityFlags` member is of the following type, and contains a bitwise-OR of zero or more of the bits defined in XrSpaceVelocityFlagBits.

```
typedef XrFlags64 XrSpaceVelocityFlags;
```

Valid bits for XrSpaceVelocityFlags are defined by XrSpaceVelocityFlagBits, which is specified as:

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

The flag bits have the following meanings:

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.

7.4.2. Locate spaces

Applications **can** use xrLocateSpaces function to locate an array of spaces.

The xrLocateSpaces function is defined as:

```
// Provided by XR_VERSION_1_1
XrResult xrLocateSpaces(
    XrSession                              session,
    const XrSpacesLocateInfo*              locateInfo,
    XrSpaceLocations*                      spaceLocations);
```

Parameter Descriptions

- `session` is an XrSession handle previously created with xrCreateSession.

- `locateInfo` is a pointer to an XrSpacesLocateInfo that provides the input information to locate spaces.

- `spaceLocations` is a pointer to an XrSpaceLocations for the runtime to return the locations of the specified spaces in the base space.

xrLocateSpaces provides the physical location of one or more spaces in a base space at a specified time, if currently known by the runtime.

The XrSpacesLocateInfo::`time`, the XrSpacesLocateInfo::`baseSpace`, and each space in XrSpacesLocateInfo::`spaces`, in the `locateInfo` parameter, all follow the same specifics as the corresponding inputs to the xrLocateSpace function.

The XrSpacesLocateInfo structure is defined as:

```
// Provided by XR_VERSION_1_1
typedef struct XrSpacesLocateInfo {
    XrStructureType    type;
    const void*        next;
    XrSpace            baseSpace;
    XrTime             time;
    uint32_t           spaceCount;
    const XrSpace*     spaces;
} XrSpacesLocateInfo;
```

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.

- `baseSpace` identifies the underlying space in which to locate `spaces`.

- `time` is the time for which the location is requested.

- `spaceCount` is a `uint32_t` specifying the count of elements in the `spaces` array.

- `spaces` is an array of valid XrSpace handles to be located.

The `time`, the `baseSpace`, and each space in `spaces` all follow the same specifics as the corresponding inputs to the xrLocateSpace function.

The `baseSpace` and all of the XrSpace handles in the `spaces` array **must** be valid and share the same parent XrSession.

If the `time` is invalid, the xrLocateSpaces **must** return `XR_ERROR_TIME_INVALID`.

The `spaceCount` **must** be a positive number, i.e. the array `spaces` **must** not be empty. Otherwise, the runtime **must** return `XR_ERROR_VALIDATION_FAILURE`.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_SPACES_LOCATE_INFO`

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

- `baseSpace` **must** be a valid XrSpace handle

- `spaces` **must** be a pointer to an array of `spaceCount` valid XrSpace handles

- The `spaceCount` parameter **must** be greater than `0`

- Both of `baseSpace` and the elements of `spaces` **must** have been created, allocated, or retrieved from the same XrSession

The XrSpaceLocations structure is defined as:

```
// Provided by XR_VERSION_1_1
typedef struct XrSpaceLocations {
    XrStructureType         type;
    void*                   next;
    uint32_t                locationCount;
    XrSpaceLocationData*    locations;
} XrSpaceLocations;
```

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 XrSpaceVelocities.
- `locationCount` is a `uint32_t` specifying the count of elements in the `locations` array.
- `locations` is an array of XrSpaceLocationData structures for the runtime to populate with the locations of the specified spaces in the XrSpacesLocateInfo::`baseSpace` at the specified XrSpacesLocateInfo::`time`.

The XrSpaceLocations structure contains an array of space locations in the member `locations`, to be used as output for xrLocateSpaces. The application **must** allocate this array to be populated with the function output. The `locationCount` value **must** be the same as XrSpacesLocateInfo::`spaceCount`, otherwise, the xrLocateSpaces function **must** return `XR_ERROR_VALIDATION_FAILURE`.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_SPACE_LOCATIONS`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrSpaceVelocities
- `locations` **must** be a pointer to an array of `locationCount` XrSpaceLocationData structures
- The `locationCount` parameter **must** be greater than `0`

The XrSpaceLocationData structure is defined as:

```
// Provided by XR_VERSION_1_1
typedef struct XrSpaceLocationData {
    XrSpaceLocationFlags    locationFlags;
    XrPosef                 pose;
} XrSpaceLocationData;
```

Member Descriptions

- `locationFlags` is a bitfield, with bit masks defined in XrSpaceLocationFlagBits. It behaves the same as XrSpaceLocation::`locationFlags`.

- `pose` is an XrPosef that behaves the same as XrSpaceLocation::`pose`.

This is a single element of the array in XrSpaceLocations::`locations`, and is used to return the pose and location flags for a single space with respect to the specified base space from a call to xrLocateSpaces. It does not accept chained structures to allow for easier use in dynamically allocated container datatypes. Chained structures are possible with the XrSpaceLocations that describes an array of these elements.

7.4.3. Locate space velocities

Applications **can** request the velocities of spaces by chaining the XrSpaceVelocities structure to the next pointer of XrSpaceLocations when calling xrLocateSpaces.

The XrSpaceVelocities structure is defined as:

```
// Provided by XR_VERSION_1_1
typedef struct XrSpaceVelocities {
    XrStructureType        type;
    void*                  next;
    uint32_t               velocityCount;
    XrSpaceVelocityData*   velocities;
} XrSpaceVelocities;
```

Member Descriptions

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `velocityCount` is a `uint32_t` specifying the count of elements in the `velocities` array.

- `velocities` is an array of XrSpaceVelocityData for the runtime to populate with the velocities of the specified spaces in the XrSpacesLocateInfo::`baseSpace` at the specified XrSpacesLocateInfo::`time`.

The `velocities` member contains an array of space velocities in the member `velocities`, to be used as output for xrLocateSpaces. The application **must** allocate this array to be populated with the function output. The `velocityCount` value **must** be the same as XrSpacesLocateInfo::`spaceCount`, otherwise, the

xrLocateSpaces function **must** return `XR_ERROR_VALIDATION_FAILURE`.

Valid Usage (Implicit)

- `type` **must** be `XR_TYPE_SPACE_VELOCITIES`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `velocities` **must** be a pointer to an array of `velocityCount` XrSpaceVelocityData structures
- The `velocityCount` parameter **must** be greater than `0`

The XrSpaceVelocityData structure is defined as:

```
// Provided by XR_VERSION_1_1
typedef struct XrSpaceVelocityData {
    XrSpaceVelocityFlags    velocityFlags;
    XrVector3f              linearVelocity;
    XrVector3f              angularVelocity;
} XrSpaceVelocityData;
```

Member Descriptions

- `velocityFlags` is a bitfield, with bit values defined in XrSpaceVelocityFlagBits. It behaves the same as XrSpaceVelocity::`velocityFlags`.
- `linearVelocity` is an XrVector3f. It behaves the same as XrSpaceVelocity::`linearVelocity`.
- `angularVelocity` is an XrVector3f. It behaves the same as XrSpaceVelocity::`angularVelocity`.

This is a single element of the array in XrSpaceVelocities::`velocities`, and is used to return the linear and angular velocity and velocity flags for a single space with respect to the specified base space from a call to xrLocateSpaces. It does not accept chained structures to allow for easier use in dynamically allocated container datatypes.

7.4.4. Example code for xrLocateSpaces

The following example code shows how an application retrieves both the location and velocity of one or more spaces in a base space at a given time using the xrLocateSpaces function.

```
XrInstance instance; // previously initialized
XrSession session; // previously initialized
XrSpace baseSpace; // previously initialized
```

```
std::vector<XrSpace> spacesToLocate;  // previously initialized

// Prepare output buffers to receive data and get reused in frame loop.
std::vector<XrSpaceLocationData> locationBuffer(spacesToLocate.size());
std::vector<XrSpaceVelocityData> velocityBuffer(spacesToLocate.size());

// Get function pointer for xrLocateSpaces.
PFN_xrLocateSpaces xrLocateSpaces;
CHK_XR(xrGetInstanceProcAddr(instance, "xrLocateSpaces",
                             reinterpret_cast<PFN_xrVoidFunction*>(
                             &xrLocateSpaces)));

// application frame loop
while (1) {
    // Typically the time is the predicted display time returned from xrWaitFrame.
    XrTime displayTime; // previously initialized.

    XrSpacesLocateInfo locateInfo{XR_TYPE_SPACES_LOCATE_INFO};
    locateInfo.baseSpace = baseSpace;
    locateInfo.time = displayTime;
    locateInfo.spaceCount = (uint32_t)spacesToLocate.size();
    locateInfo.spaces = spacesToLocate.data();

    XrSpaceLocations locations{XR_TYPE_SPACE_LOCATIONS};
    locations.locationCount = (uint32_t)locationBuffer.size();
    locations.locations = locationBuffer.data();

    XrSpaceVelocities velocities{XR_TYPE_SPACE_VELOCITIES};
    velocities.velocityCount = (uint32_t)velocityBuffer.size();
    velocities.velocities = velocityBuffer.data();

    locations.next = &velocities;
    CHK_XR(xrLocateSpaces(session, &locateInfo, &locations));

    for (uint32_t i = 0; i < spacesToLocate.size(); i++) {
        const auto positionAndOrientationTracked =
          XR_SPACE_LOCATION_POSITION_TRACKED_BIT |
XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT;
        const auto orientationOnlyTracked = XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT;

        if ((locationBuffer[i].locationFlags & positionAndOrientationTracked) ==
positionAndOrientationTracked) {
            // if the location is 6dof tracked
            do_something(locationBuffer[i].pose.position);
            do_something(locationBuffer[i].pose.orientation);

            const auto velocityValidBits =
              XR_SPACE_VELOCITY_LINEAR_VALID_BIT | XR_SPACE_VELOCITY_ANGULAR_VALID_BIT;
```

```
            if ((velocityBuffer[i].velocityFlags & velocityValidBits) ==
velocityValidBits) {
                do_something(velocityBuffer[i].linearVelocity);
                do_something(velocityBuffer[i].angularVelocity);
            }
        }
        else if ((locationBuffer[i].locationFlags & orientationOnlyTracked) ==
orientationOnlyTracked) {
            // if the location is 3dof tracked
            do_something(locationBuffer[i].pose.orientation);

            if ((velocityBuffer[i].velocityFlags & XR_SPACE_VELOCITY_ANGULAR_VALID_BIT)
== XR_SPACE_VELOCITY_ANGULAR_VALID_BIT) {
                do_something(velocityBuffer[i].angularVelocity);
            }
        }
    }
}
```

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

```
typedef enum XrViewConfigurationType {
    XR_VIEW_CONFIGURATION_TYPE_PRIMARY_MONO = 1,
    XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO = 2,
  // Provided by XR_VERSION_1_1
    XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO_WITH_FOVEATED_INSET = 1000037000,
    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 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 each XrCompositionLayerProjection layer. View index 0 **must** represent the left eye and view index 1 **must** represent the right eye.

- `XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO_WITH_FOVEATED_INSET`. Four views representing the form factor's primary stereo displays. This view configuration type represents a hardware independent way of providing foveated rendering. The view configuration adds two foveated inset views for the left and right eye separately to the already defined two views specified in the `XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO` view configuration. View index 0 **must** represent the left eye and view index 1 **must** represent the right eye as specified in `XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO` view configuration, and view index 2 **must** represent the left eye inset view and view index 3 **must** represent the right eye inset view. The new inset view 2 and view 3 **must**, after applying the pose and FoV projection to same plane, be contained within view 0 and 1 respectively. The inset views **may** have a higher resolution with respect to the same field of view as the corresponding wide FoV view for each eye. The runtime **may** blend between the views at the edges, so the application **must** not omit the inner field of view from being rendered in the outer view. The `fov` returned by xrLocateViews for each inset view relative to the corresponding outer stereo view **may** change at run-time, the `pose` for inset view and stereo view for each eye respectively **must** have the same values.

  > ℹ️ The benefits of the `XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO_WITH_FOVEATED_INSET` view configuration type can be demonstrated by looking at the rendered pixel count. For example, a Varjo Aero requires a pair of stereo views rendered at 4148 x 3556 (14.7 million pixels) to achieve a pixel density of 35 pixels per degree. By using four views, with an eye-tracked foveated inset covering about 1/9th of the full FoV and rendered with the same 35 pixels per degree and while the remaining views are dropped to 14 pixels per degree, the resolution of the inset is 1076 x 1076 (1.1 million pixels) and the resolution of the stereo views is 1660 x 1420 (2.3 million pixels). The total pixel count is 75% less with `XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO_WITH_FOVEATED_INSET` over the `XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO` view configuration type.

```
                    0 - mono


XR_VIEW_CONFIGURATION_TYPE_PRIMARY_MONO



      0 - left              1 - right


XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO



      2 - left    ↕       3 - right   ↕
        inset                inset

          0 - left      1 - right

XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO_INSET
```
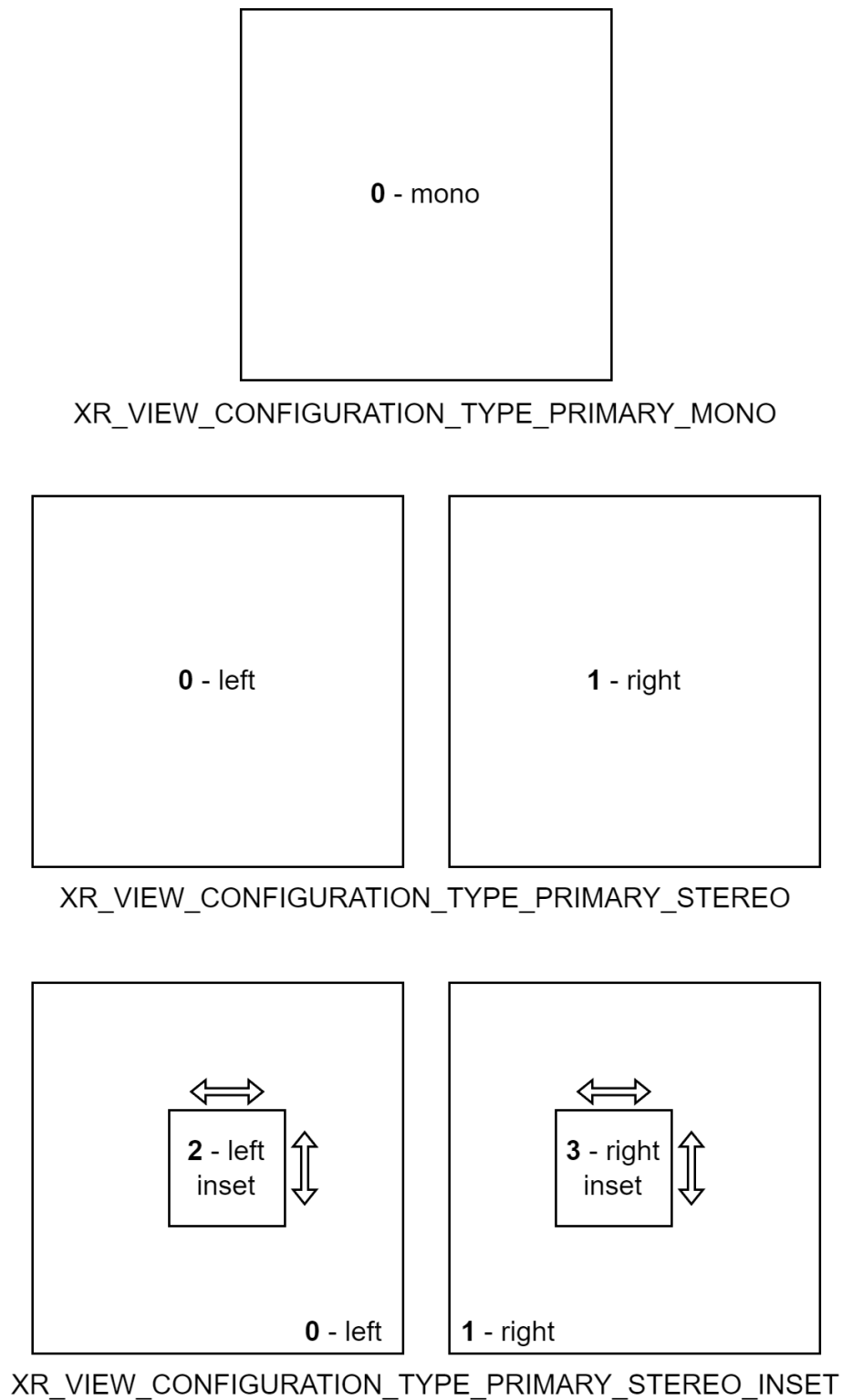
Figure 4. View configurations. The numbers in the figure is the view indices of the specific view.

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:

```
// Provided by XR_VERSION_1_0
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.

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

- `viewConfigurationTypeCountOutput` is a pointer to the count of `viewConfigurationTypes` written, or a pointer to the required capacity in the case that `viewConfigurationTypeCapacityInput` is insufficient.

- `viewConfigurationTypes` is a pointer to an array of XrViewConfigurationType values, but **can** be `NULL` if `viewConfigurationTypeCapacityInput` is 0.

- See the Buffer Size Parameters section for a detailed description of retrieving the required `viewConfigurationTypes` 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.

8.2.2. xrGetViewConfigurationProperties

The xrGetViewConfigurationProperties function is defined as:

```
// Provided by XR_VERSION_1_0
XrResult xrGetViewConfigurationProperties(
    XrInstance                          instance,
    XrSystemId                          systemId,
    XrViewConfigurationType             viewConfigurationType,
    XrViewConfigurationProperties*      configurationProperties);
```

<div style="border:1px solid #ccc; padding:1em; background:#f2f2f2">

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.

</div>

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`.

<div style="border:1px solid #ccc; padding:1em; background:#f2f2f2">

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

</div>

<div style="border:1px solid #ccc; padding:1em; background:#f2f2f2">

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`

</div>

8.2.3. XrViewConfigurationProperties

The XrViewConfigurationProperties structure is defined as:

```
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:

```
// Provided by XR_VERSION_1_0
XrResult xrEnumerateViewConfigurationViews(
    XrInstance                            instance,
    XrSystemId                            systemId,
    XrViewConfigurationType               viewConfigurationType,
    uint32_t                              viewCapacityInput,
    uint32_t*                             viewCountOutput,
    XrViewConfigurationView*              views);
```

- `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

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;
```

- `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 XrSwapchainSubImage::`imageRect` to use when rendering this view into a swapchain.

- `maxImageRectWidth` is the maximum width of XrSwapchainSubImage::`imageRect` supported when rendering this view into a swapchain.

- `recommendedImageRectHeight` is the optimal height of XrSwapchainSubImage::`imageRect` to use when rendering this view into a swapchain.

- `maxImageRectHeight` is the maximum height of XrSwapchainSubImage::`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 XrSwapchainSubImage::`imageRect` values, and XrSwapchainCreateInfo for more information about creating swapchains appropriately sized to support those XrSwapchainSubImage::`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.

- `type` **must** be `XR_TYPE_VIEW_CONFIGURATION_VIEW`

- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrViewConfigurationDepthRangeEXT

8.3. Example View Configuration Code

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


// Enumerate the view configurations paths.
```

```cpp
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;
XrViewConfigurationType viewConfig = XR_VIEW_CONFIGURATION_TYPE_MAX_ENUM;
for(uint32_t i = 0; i < configurationCount; ++i)
{
    if (configurationTypes[i] == XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO)
    {
        configFound = true;
        viewConfig = configurationTypes[i];
        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,
    viewConfig,
    0,
    &viewCount,
    nullptr));

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

// Set the primary view configuration for the session.
XrSessionBeginInfo beginInfo = {XR_TYPE_SESSION_BEGIN_INFO};
beginInfo.primaryViewConfigurationType = viewConfig;
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)
```

```cpp
{
    // 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.viewConfigurationType = viewConfig;
        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

```
XR_DEFINE_HANDLE(XrSession)
```

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

9.1. Session Lifecycle
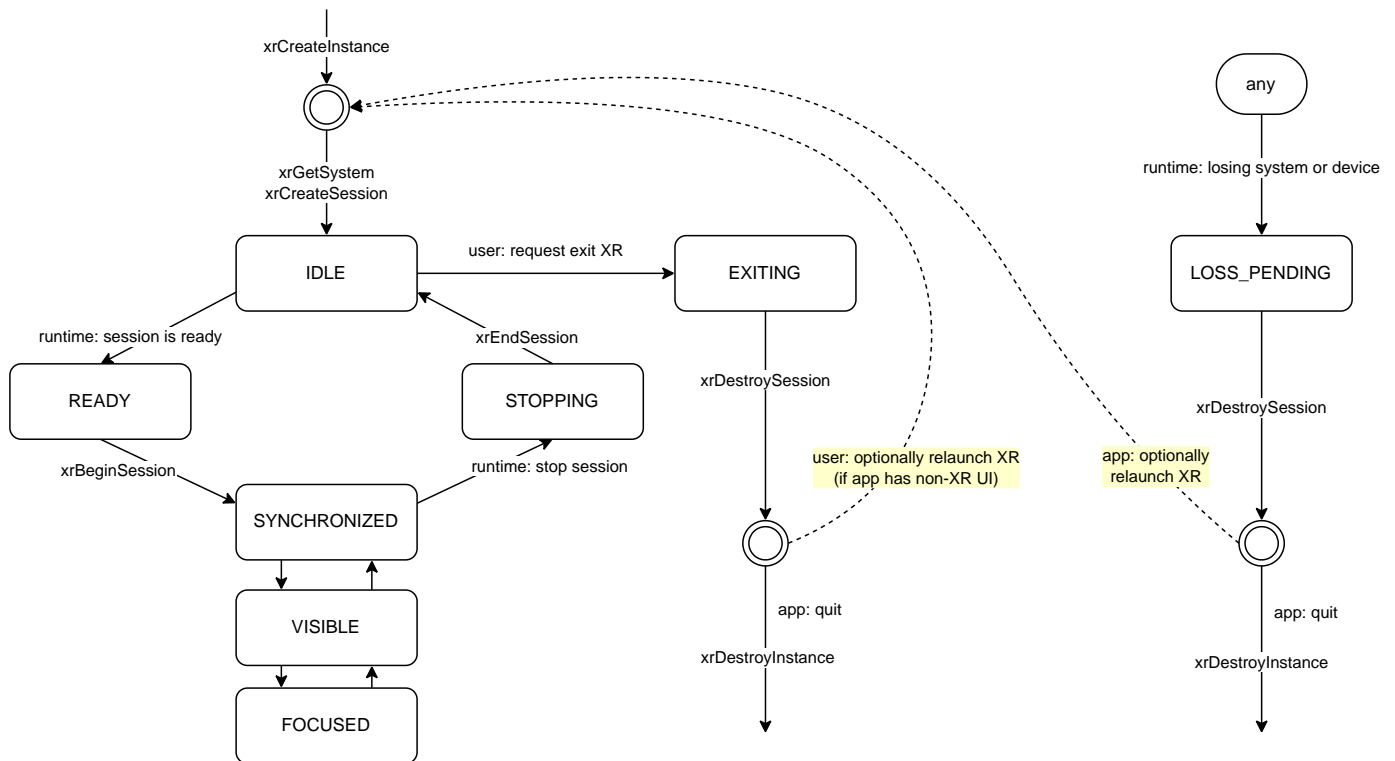
Figure 5. Session Lifecycle

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 can regularly call xrPollEvent to monitor 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 `XR_SESSION_STATE_READY`. Once the application is also ready to proceed and display its XR content, it calls 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 xrWaitFrame, xrBeginFrame and 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 `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 `XR_SESSION_STATE_VISIBLE` state, and sets XrFrameState::`shouldRender` to `true` in xrWaitFrame. The application should render XR content and submit the composition layers to 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 `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 `XR_SESSION_STATE_FOCUSED` to `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 `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 `XR_SESSION_STATE_STOPPING`. When the application receives the `XR_SESSION_STATE_STOPPING` event, it should stop its frame loop and then call xrEndSession to tell the runtime to stop the running session.

9. After xrEndSession, the runtime transitions the session state to `XR_SESSION_STATE_IDLE`. If the XR session is temporarily paused in the background, the runtime will keep the session state at `XR_SESSION_STATE_IDLE` and later transition the session state back to `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 `XR_SESSION_STATE_IDLE` to `XR_SESSION_STATE_EXITING`.

10. When the application receives the `XR_SESSION_STATE_EXITING` event, it releases the resources related to the session and calls xrDestroySession.

A session is considered **running** after a successful call to xrBeginSession and remains running until any call is made to xrEndSession. Certain functions are only valid to call when a session is running, such as xrWaitFrame, or else the `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:

```
// Provided by XR_VERSION_1_0
XrResult xrCreateSession(
    XrInstance                                  instance,
    const XrSessionCreateInfo*                  createInfo,
    XrSession*                                  session);
```

Parameter Descriptions

- `instance` is the instance from which XrSessionCreateInfo::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

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:

```
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, XrGraphicsBindingMetalKHR, XrGraphicsBindingOpenGLESAndroidKHR, XrGraphicsBindingOpenGLWaylandKHR, XrGraphicsBindingOpenGLWin32KHR, XrGraphicsBindingOpenGLXcbKHR, XrGraphicsBindingOpenGLXlibKHR, XrGraphicsBindingVulkanKHR

- createFlags **must** be 0

The XrSessionCreateInfo::createFlags member is of the following type, and contains a bitwise-OR of zero or more of the bits defined in XrSessionCreateFlagBits.

```
typedef XrFlags64 XrSessionCreateFlags;
```

Valid bits for XrSessionCreateFlags are defined by XrSessionCreateFlagBits.

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

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

The xrDestroySession function is defined as.

```
// Provided by XR_VERSION_1_0
XrResult xrDestroySession(
    XrSession                                   session);
```

- `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:

```
// Provided by XR_VERSION_1_0
XrResult xrBeginSession(
    XrSession                                   session,
    const XrSessionBeginInfo*                   beginInfo);
```

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 XrFrameState::`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. Frame state in this context includes xrWaitFrame, xrBeginFrame, and xrEndFrame call order enforcement.

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

The XrSessionBeginInfo structure is defined as:

```
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.

The xrEndSession function is defined as:

```
// Provided by XR_VERSION_1_0
XrResult xrEndSession(
    XrSession                                   session);
```

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`.

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:

```
// Provided by XR_VERSION_1_0
XrResult xrRequestExitSession(
    XrSession                                   session);
```

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

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:

```
// Provided by XR_VERSION_1_0
typedef struct XrEventDataSessionStateChanged {
    XrStructureType    type;
    const void*        next;
    XrSession          session;
    XrSessionState     state;
    XrTime             time;
} XrEventDataSessionStateChanged;
```

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

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;
```

<div style="border: 1px solid #ccc; padding: 1em;">

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.

</div>

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, xrBeginFrame and xrEndFrame 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 **may** wait until the user starts engaging with the device before transitioning to the `XR_SESSION_STATE_READY` state.

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, xrBeginFrame and xrEndFrame, 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 xrEndSession 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 **may** transition to the XR_SESSION_STATE_STOPPING state when the user stops engaging with the device.

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 are assumed to be internally performed by the runtime in linear color space. Images intended to be interpreted as being non-linear-encoded ("sRGB") **must** be created using an API-specific "sRGB" format (e.g. DXGI_FORMAT_R8G8B8A8_UNORM_SRGB, GL_SRGB8_ALPHA8, VK_FORMAT_R8G8B8A8_SRGB) to signal the need for sRGB-to-linear conversion (whether automatic or manual) when sampled 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.
>
> Gritz, L. and d'Eon, E. 2007. The Importance of Being Linear. In: H. Nguyen, ed., *GPU Gems 3*. Addison-Wesley Professional. https://developer.nvidia.com/gpugems/gpugems3/part-iv-image-effects/chapter-24-importance-being-linear

> *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:

```
// Provided by XR_VERSION_1_0
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 the Buffer Size Parameters section 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

The xrCreateSwapchain function is defined as:

```
// Provided by XR_VERSION_1_0
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 runtime. 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 or usage flags specified in the XrSwapchainCreateInfo is unsupported.

- `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 **must** 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 XrSwapchainCreateInfo::`createFlags` member is of the following type, and contains a bitwise-OR of zero or more of the bits defined in XrSwapchainCreateFlagBits.

```
typedef XrFlags64 XrSwapchainCreateFlags;
```

Valid bits for XrSwapchainCreateFlags are defined by XrSwapchainCreateFlagBits, which is specified as:

```
// Flag bits for XrSwapchainCreateFlags
static const XrSwapchainCreateFlags XR_SWAPCHAIN_CREATE_PROTECTED_CONTENT_BIT =
0x00000001;
static const XrSwapchainCreateFlags XR_SWAPCHAIN_CREATE_STATIC_IMAGE_BIT = 0x00000002;
```

The flag bits have the following meanings:

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.

XrSwapchainUsageFlags specify the intended usage of the swapchain images. The XrSwapchainCreateInfo::usageFlags member is of this type, and contains a bitwise-OR of one or more of the bits defined in XrSwapchainUsageFlagBits.

```
typedef XrFlags64 XrSwapchainUsageFlags;
```

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 **must** match the intended usage.

Flags include:

```
// 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 =
0x00000080;  // alias of XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_MND
```

The flag bits have the following meanings:

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 a 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 a input attachment. (Added by the `XR_KHR_swapchain_usage_input_attachment_bit` extension)

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.

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:

```
// Provided by XR_VERSION_1_0
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

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:

```
// Provided by XR_VERSION_1_0
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 the Buffer Size Parameters section 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`.

> **ℹ** *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 XrSwapchainImageBaseHeader::`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, XrSwapchainImageOpenGLESKHR, XrSwapchainImageOpenGLKHR, 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 is extended by graphics API-specific XrSwapchainImage* child structures.

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 builds 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 to 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 (non-static) swapchain at a time, for example if the application implements a multiple frame deep rendering pipeline.

The xrAcquireSwapchainImage function is defined as:

```
// Provided by XR_VERSION_1_0
XrResult xrAcquireSwapchainImage(
    XrSwapchain                                 swapchain,
    const XrSwapchainImageAcquireInfo*          acquireInfo,
    uint32_t*                                   index);
```

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.

This function only provides the index of the swapchain image, for example for use in recording command buffers. It does not wait for the image to be usable by the application. The application **must** call xrWaitSwapchainImage for each "acquire" call before submitting graphics commands that write to the image.

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

The XrSwapchainImageAcquireInfo structure is defined as:

```
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 xrAcquireSwapchainImage::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:

```
// Provided by XR_VERSION_1_0
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 begins 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.

Valid Usage (Implicit)

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

- waitInfo **must** be a pointer to a valid XrSwapchainImageWaitInfo structure

The XrSwapchainImageWaitInfo structure describes a swapchain image wait operation. It is defined as:

```
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 **may** block waiting for the image to become available for writing.

- 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 **may** be used in one frame.

The xrReleaseSwapchainImage function is defined as:

```
// Provided by XR_VERSION_1_0
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

The XrSwapchainImageReleaseInfo structure is defined as:

```
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 xrReleaseSwapchainImage::releaseInfo for applications that are not using any relevant extensions.

Valid Usage (Implicit)

- type **must** be XR_TYPE_SWAPCHAIN_IMAGE_RELEASE_INFO

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

10.2. View and Projection State

An application uses xrLocateViews to 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:

```
// Provided by XR_VERSION_1_0
XrResult xrLocateViews(
    XrSession                               session,
    const XrViewLocateInfo*                 viewLocateInfo,
    XrViewState*                            viewState,
    uint32_t                                viewCapacityInput,
    uint32_t*                               viewCountOutput,
    XrView*                                 views);
```

Parameter Descriptions

- `session` is a handle to the provided XrSession.

- `viewLocateInfo` is a pointer to a valid XrViewLocateInfo structure.

- `viewState` is the output structure with the viewer state information.

- `viewCapacityInput` is an input parameter which specifies the capacity of the `views` array. The required capacity **must** be same as defined by the corresponding XrViewConfigurationType.

- `viewCountOutput` is an output parameter which identifies the valid count of `views`.

- `views` is an array of XrView.

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

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.

xrLocateViews returns an array of XrView elements, one for each view of the specified view configuration type, along with an XrViewState containing additional state data shared across all views. The eye each view corresponds to is statically defined in XrViewConfigurationType in case the application wants to apply eye-specific rendering traits. The XrViewState and XrView member data

may change on subsequent calls to xrLocateViews, and so applications **must** not assume it to be constant.

If an application gives a `viewLocateInfo` with a XrViewLocateInfo::`viewConfigurationType` that was not passed in the session's call to xrBeginSession via the XrSessionBeginInfo::`primaryViewConfigurationType`, or enabled though an extension, then the runtime **must** return `XR_ERROR_VALIDATION_FAILURE`.

Valid Usage (Implicit)

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

- `viewLocateInfo` **must** be a pointer to a valid XrViewLocateInfo structure

- `viewState` **must** be a pointer to an XrViewState structure

- `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` XrView structures

---

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_VIEW_CONFIGURATION_TYPE_UNSUPPORTED`

- `XR_ERROR_TIME_INVALID`

The XrViewLocateInfo structure is defined as:

```
typedef struct XrViewLocateInfo {
    XrStructureType           type;
    const void*               next;
    XrViewConfigurationType   viewConfigurationType;
    XrTime                    displayTime;
    XrSpace                   space;
} XrViewLocateInfo;
```

Member Descriptions

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `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;
```

- `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:

```
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.

The XrViewStateFlags specifies the validity and quality of the corresponding XrView array returned by xrLocateViews. The XrViewState::`viewStateFlags` member is of this type, and contains a bitwise-OR of zero or more of the bits defined in XrViewStateFlagBits.

```
typedef XrFlags64 XrViewStateFlags;
```

Valid bits for XrViewStateFlags are defined by XrViewStateFlagBits, which is specified as:

```
// 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;
```

The flag bits have the following meanings:

<div style="text-align: center;">**Flag Descriptions**</div>

- `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:

```
// Provided by XR_VERSION_1_0
XrResult xrWaitFrame(
    XrSession                                   session,
    const XrFrameWaitInfo*                      frameWaitInfo,
    XrFrameState*                               frameState);
```

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. 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. Refer to xrBeginSession for details on how a transition to session running resets the frame function call order.

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 xrWaitFrame::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;
```

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.

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:

```
// Provided by XR_VERSION_1_0
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. Refer to xrBeginSession for details on how a transition to session running resets the frame function call order.

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

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, [xrBeginFrame](#) will accept a

NULL argument for xrBeginFrame::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 xrEndFrame function is defined as:

```
// Provided by XR_VERSION_1_0
XrResult xrEndFrame(
    XrSession                                   session,
    const XrFrameEndInfo*                       frameEndInfo);
```

Parameter Descriptions

- session is a valid XrSession handle.

- frameEndInfo is a pointer to a valid XrFrameEndInfo.

---

xrEndFrame **may** return immediately to the application. XrFrameEndInfo::displayTime **should** be computed using values returned by xrWaitFrame. The runtime **should** be robust against variations in the timing of calls to xrWaitFrame, since a pipelined system may call xrWaitFrame on a separate thread from xrBeginFrame and xrEndFrame 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. Refer to xrBeginSession for details on how a transition to session running resets the frame function call order. 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 and only if the XrFrameEndInfo::`environmentBlendMode` was not enumerated by xrEnumerateEnvironmentBlendModes for the XrInstance and `XrSystemId` used to create `session`.

`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

The XrFrameEndInfo structure is defined as:

```
typedef struct XrFrameEndInfo {
    XrStructureType                            type;
    const void*                                next;
    XrTime                                     displayTime;
    XrEnvironmentBlendMode                     environmentBlendMode;
    uint32_t                                   layerCount;
    const XrCompositionLayerBaseHeader* const* layers;
} XrFrameEndInfo;
```

- `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.

10.5. 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.

10.6. 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.

10.6.1. Composition Layer Flags

XrCompositionLayerFlags specifies options for individual composition layers, and contains a bitwise-OR of zero or more of the bits defined in XrCompositionLayerFlagBits.

```
typedef XrFlags64 XrCompositionLayerFlags;
```

Valid bits for XrCompositionLayerFlags are defined by XrCompositionLayerFlagBits, which is specified as:

```
// Flag bits for XrCompositionLayerFlags
// XR_COMPOSITION_LAYER_CORRECT_CHROMATIC_ABERRATION_BIT is deprecated and should not be
used
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;
static const XrCompositionLayerFlags XR_COMPOSITION_LAYER_INVERTED_ALPHA_BIT_EXT =
0x00000008;
```

The flag bits have the following meanings:

<div style="border: 1px solid #ccc; padding: 1em;">

Flag Descriptions

- `XR_COMPOSITION_LAYER_CORRECT_CHROMATIC_ABERRATION_BIT` *(deprecated — ignored)* — Enables chromatic aberration correction when not done by default. This flag has no effect on any known conformant runtime, and is officially deprecated in OpenXR 1.1.

- `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.

</div>

10.6.2. 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:

```
LayerColor.RGB *= LayerColor.A
```

LayerColor is then clamped to a range of [0.0, 1.0].

The layer blending operation is defined as:

```
CompositeColor  = LayerColor + CompositeColor * (1 - LayerColor.A)
```

Before the first layer is composited, all components of CompositeColor are initialized to zero.

10.6.3. 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:

```
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.

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:

```
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.

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:

```
// Provided by XR_VERSION_1_0
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;
```

- `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 swapchain **must** have been created with a XrSwapchainCreateInfo::`faceCount` of 1.

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:

```
// Provided by XR_VERSION_1_0
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. The swapchain **must** have been created with a XrSwapchainCreateInfo::`faceCount` of 1.

- `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.

The XrEyeVisibility enum selects which of the viewer's eyes to display a layer to:

```
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.

10.6.4. 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 environment 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.

The environment **may** be perceived in two ways. It could be the user's view of the physical world that exists beyond the displays, or it could be a synthetic environment including virtual components generated externally from the application. Alternatively, it could be a combination of both these elements.

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 support video/optical passthrough or synthetic environments, they **may** support the XR_ENVIRONMENT_BLEND_MODE_ADDITIVE or XR_ENVIRONMENT_BLEND_MODE_ALPHA_BLEND modes. Selecting one of these modes would display the environment in the background, contingent upon the capability and status of the headsets.

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:

```
// Provided by XR_VERSION_1_0
XrResult xrEnumerateEnvironmentBlendModes(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    XrViewConfigurationType                     viewConfigurationType,
    uint32_t                                    environmentBlendModeCapacityInput,
    uint32_t*                                   environmentBlendModeCountOutput,
    XrEnvironmentBlendMode*                     environmentBlendModes);
```

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.

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:

```
XrInstance instance; // previously initialized
XrSession session; // previously initialized

// Create an action set
XrActionSetCreateInfo actionSetInfo{XR_TYPE_ACTION_SET_CREATE_INFO};
strcpy(actionSetInfo.actionSetName, "gameplay");
strcpy(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};
strcpy(actioninfo.actionName, "teleport");
actioninfo.actionType = XR_ACTION_TYPE_BOOLEAN_INPUT;
strcpy(actioninfo.localizedActionName, "Teleport");
XrAction teleportAction;
CHK_XR(xrCreateAction(inGameActionSet, &actioninfo, &teleportAction));

// create a "player_hit" output action
XrActionCreateInfo hapticsactioninfo{XR_TYPE_ACTION_CREATE_INFO};
strcpy(hapticsactioninfo.actionName, "player_hit");
hapticsactioninfo.actionType = XR_ACTION_TYPE_VIBRATION_OUTPUT;
strcpy(hapticsactioninfo.localizedActionName, "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

```
XR_DEFINE_HANDLE(XrActionSet)
```

Action sets are application-defined collections of actions. They are attached to a given XrSession with a xrAttachSessionActionSets call. Enabled action sets are indicated by the application via xrSyncActions depending on the current application context.

For example, consider using one collection of actions that apply to controlling a character and another collection for navigating a menu system. When these actions are structured as two XrActionSet handles, the applicable action set is easy to specify according to application logic using a single function call.

Further, suppose some actions only apply when operating a vehicle as a character. This is intended to be modeled as another separate action set. While the user is operating a vehicle, the application enables both the character-control and vehicle action sets simultaneously in each xrSyncActions 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:

```
// Provided by XR_VERSION_1_0
XrResult xrCreateActionSet(
    XrInstance                                  instance,
    const XrActionSetCreateInfo*                createInfo,
    XrActionSet*                                actionSet);
```

The xrCreateActionSet function creates an action set and returns a handle to the created action set.

The XrActionSetCreateInfo structure is defined as:

```
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:

```
// Provided by XR_VERSION_1_0
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

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:

```
// Provided by XR_VERSION_1_0
XrResult xrCreateAction(
    XrActionSet                                 actionSet,
    const XrActionCreateInfo*                   createInfo,
    XrAction*                                   action);
```

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.

If `actionSet` has been included in a call to xrAttachSessionActionSets, the implementation **must** return `XR_ERROR_ACTIONSETS_ALREADY_ATTACHED`.

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 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:

```
// Provided by XR_VERSION_1_0
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 suggest bindings for their actions to runtimes so that raw input data is mapped appropriately to the application's actions. Suggested bindings also serve as a signal indicating the hardware that has been tested by the application developer. Applications **can** suggest bindings by calling xrSuggestInteractionProfileBindings for each interaction profile that the application is developed and tested with. 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 xrSuggestInteractionProfileBindings function is defined as:

```
// Provided by XR_VERSION_1_0
XrResult xrSuggestInteractionProfileBindings(
    XrInstance                                  instance,
    const XrInteractionProfileSuggestedBinding* suggestedBindings);
```

The xrSuggestInteractionProfileBindings function provides action bindings for a single interaction profile. 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 Input subpaths (further described in XrActionSuggestedBinding), the runtime **must** return XR_ERROR_PATH_UNSUPPORTED. If the interaction profile path or binding path (top level */user* path plus input subpath) 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.

The XrInteractionProfileSuggestedBinding structure is defined as:

```
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.

- 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:

```
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 "binding path" is any top level *user* path plus an applicable input subpath, 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:

```
// Provided by XR_VERSION_1_0
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 XrSessionActionSetsAttachInfo ::`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`

11.5. Current Interaction Profile

The xrGetCurrentInteractionProfile function is defined as:

```
// Provided by XR_VERSION_1_0
XrResult xrGetCurrentInteractionProfile(
    XrSession                              session,
    XrPath                                 topLevelUserPath,
    XrInteractionProfileState*             interactionProfile);
```

xrGetCurrentInteractionProfile retrieves the current interaction profile for a top level user path.

The runtime **must** return only interaction profiles for which the application has provided suggested 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 an anticipated interaction profile, from the list of interaction profiles with suggested bindings (as supplied by the application through xrSuggestInteractionProfileBindings) for this top level /user path, in the event that no controllers are active. Whether the runtime reports an interaction profile path or XR_NULL_PATH does **not** provide any signal to the application regarding presence or absence of a controller or other interaction method.

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 top level user paths described in Top level /user paths, the runtime **must** return `XR_ERROR_PATH_UNSUPPORTED`.

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 xrGetCurrentInteractionProfile::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.

<div style="background:#eee;padding:1em">

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

</div>

The XrEventDataInteractionProfileChanged structure is defined as:

```
// Provided by XR_VERSION_1_0
typedef struct XrEventDataInteractionProfileChanged {
    XrStructureType    type;
    const void*        next;
    XrSession          session;
} XrEventDataInteractionProfileChanged;
```

<div style="background:#eee;padding:1em">

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.

</div>

The XrEventDataInteractionProfileChanged event is queued to notify the application that the current interaction profile 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 queued for running sessions.

Upon receiving this event, an application **can** call xrGetCurrentInteractionProfile for each top level user path in use, if its behavior depends on the current interaction profile.

<div style="background:#eee;padding:1em">

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

</div>

11.6. 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`.

11.6.1. 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 current state **must** be the state of a single pose source. The source of the pose **must** only be changed during a call to xrSyncAction. The runtime **should** only change the source in response to user actions, such as picking up a new controller, or external events, such as a controller running out of battery.

- Haptic actions - The runtime **must** send output events to all bound haptic devices

11.6.2. 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:

```
typedef struct XrActionStateGetInfo {
    XrStructureType     type;
    const void*         next;
    XrAction            action;
    XrPath              subactionPath;
} XrActionStateGetInfo;
```

<div align="center">

Member Descriptions

</div>

- `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.

The XrHapticActionInfo structure is used to provide action and subaction paths when calling xr*HapticFeedback function. It is defined as:

```
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.

11.6.3. Boolean Actions

xrGetActionStateBoolean retrieves the current state of a boolean action. It is defined as:

```
// Provided by XR_VERSION_1_0
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

The XrActionStateBoolean structure is defined as:

```
typedef struct XrActionStateBoolean {
    XrStructureType    type;
    void*              next;
    XrBool32           currentState;
    XrBool32           changedSinceLastSync;
    XrTime             lastChangeTime;
    XrBool32           isActive;
} XrActionStateBoolean;
```

- `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 associated with the most recent change to this action's state.

- `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

11.6.4. Scalar and Vector Actions

xrGetActionStateFloat retrieves the current state of a floating-point action. It is defined as:

```
// Provided by XR_VERSION_1_0
XrResult xrGetActionStateFloat(
    XrSession                                   session,
    const XrActionStateGetInfo*                 getInfo,
    XrActionStateFloat*                         state);
```

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 associated with the most recent change to this action's state.

- `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:

```
// Provided by XR_VERSION_1_0
XrResult xrGetActionStateVector2f(
    XrSession                                   session,
    const XrActionStateGetInfo*                 getInfo,
    XrActionStateVector2f*                      state);
```

- `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_PATH_INVALID`

- `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 associated with the most recent change to this action's state.

- `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

11.6.5. Pose Actions

The xrGetActionStatePose function is defined as:

```
// Provided by XR_VERSION_1_0
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

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.

11.7. 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:

```
// Provided by XR_VERSION_1_0
XrResult xrApplyHapticFeedback(
    XrSession                                   session,
    const XrHapticActionInfo*                   hapticActionInfo,
    const XrHapticBaseHeader*                   hapticFeedback);
```

- `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

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.

The XrHapticVibration structure is defined as:

```
// Provided by XR_VERSION_1_0
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.

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.

```
// Provided by XR_VERSION_1_0
#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.

```
// Provided by XR_VERSION_1_0
#define XR_FREQUENCY_UNSPECIFIED 0
```

The xrStopHapticFeedback function is defined as:

```
// Provided by XR_VERSION_1_0
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

11.8. Input Action State Synchronization

The xrSyncActions function is defined as:

```
// Provided by XR_VERSION_1_0
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 XrActionsSyncInfo::`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:

```
typedef struct XrActiveActionSet {
    XrActionSet    actionSet;
    XrPath         subactionPath;
} XrActiveActionSet;
```

- `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.

- `actionSet` **must** be a valid XrActionSet handle

11.9. Bound 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. The bound `sources` are opaque `XrPath` values representing the physical controls that an action is bound to. 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 bound sources for an action are obtained, the application **can** gather additional information about it. xrGetInputSourceLocalizedName returns a localized human-readable string describing the bound physical control, e.g. 'A Button'.

The xrEnumerateBoundSourcesForAction function is defined as:

```
// Provided by XR_VERSION_1_0
XrResult xrEnumerateBoundSourcesForAction(
    XrSession                                   session,
    const XrBoundSourcesForActionEnumerateInfo* enumerateInfo,
    uint32_t                                    sourceCapacityInput,
    uint32_t*                                   sourceCountOutput,
    XrPath*                                     sources);
```

- `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 bound sources. It **can** be `NULL` if `sourceCapacityInput` is 0.

- See the Buffer Size Parameters section 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.

> *Note*
>
> The XrPath bound sources returned by the runtime are opaque values and **should** not be inspected or persisted. They are only intended for use in conjunction with xrGetInputSourceLocalizedName.

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.

The xrGetInputSourceLocalizedName function is defined as:

```
// Provided by XR_VERSION_1_0
XrResult xrGetInputSourceLocalizedName(
    XrSession                               session,
    const XrInputSourceLocalizedNameGetInfo*  getInfo,
    uint32_t                                bufferCapacityInput,
    uint32_t*                               bufferCountOutput,
    char*                                   buffer);
```

xrGetInputSourceLocalizedName returns a string for the bound 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`.

The XrInputSourceLocalizedNameGetInfo structure is defined as:

```
typedef struct XrInputSourceLocalizedNameGetInfo {
    XrStructureType                    type;
    const void*                        next;
    XrPath                             sourcePath;
    XrInputSourceLocalizedNameFlags    whichComponents;
} XrInputSourceLocalizedNameGetInfo;
```

The result of passing an XrPath `sourcePath` **not** retrieved from xrEnumerateBoundSourcesForAction is not specified.

The XrInputSourceLocalizedNameGetInfo::`whichComponents` parameter is of the following type, and contains a bitwise-OR of one or more of the bits defined in XrInputSourceLocalizedNameFlagBits.

```
typedef XrFlags64 XrInputSourceLocalizedNameFlags;
```

```
// 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;
```

The flag bits have the following meanings:

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 Current 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_extended_struct_name_lengths

- XR_KHR_loader_init

- XR_KHR_loader_init_android

- XR_KHR_metal_enable

- 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

- XR_EXT_conformance_automation

- XR_EXT_dpad_binding

- XR_EXT_eye_gaze_interaction

- XR_EXT_future

- XR_EXT_hand_interaction

- XR_EXT_hand_joints_motion_range

- XR_EXT_hand_tracking

- XR_EXT_hand_tracking_data_source

- XR_EXT_performance_settings

- XR_EXT_spatial_anchor

- XR_EXT_spatial_entity

- XR_EXT_spatial_marker_tracking

- XR_EXT_spatial_persistence

- XR_EXT_spatial_persistence_operations

- XR_EXT_spatial_plane_tracking

- XR_EXT_view_configuration_depth_range

12.1. XR_KHR_android_create_instance

Name String

    `XR_KHR_android_create_instance`

Extension Type

    Instance extension

Registered Extension Number

    9

Revision

    3

Ratification Status

    Ratified

Extension and Version Dependencies

    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:

```
// Provided by XR_KHR_android_create_instance
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.

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

Ratification Status

Ratified

Extension and Version Dependencies

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:

```
// Provided by XR_KHR_android_surface_swapchain
XrResult xrCreateSwapchainAndroidSurfaceKHR(
    XrSession                                   session,
    const XrSwapchainCreateInfo*                info,
    XrSwapchain*                                swapchain,
    jobject*                                    surface);
```

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 (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

Registered Extension Number

4

Revision

6

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0

Last Modified Date

2023-12-04

IP Status

No known IP claims.

Contributors

Cass Everitt, Oculus
Johannes van Waveren, Oculus
Martin Renschler, Qualcomm
Krzysztof Kosiński, Google
Xiang Wei, Meta

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

[XrResult](#) enumeration is extended with:

- `XR_ERROR_ANDROID_THREAD_SETTINGS_ID_INVALID_KHR`

- `XR_ERROR_ANDROID_THREAD_SETTINGS_FAILURE_KHR`

New Enums

The possible thread types are specified by the [XrAndroidThreadTypeKHR](#) enumeration:

```
// Provided by XR_KHR_android_thread_settings
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;
```

<div align="center">

Enumerants

</div>

- `XR_ANDROID_THREAD_TYPE_APPLICATION_MAIN_KHR`
  hints the XR runtime that the thread is doing time critical CPU tasks

- `XR_ANDROID_THREAD_TYPE_APPLICATION_WORKER_KHR`
  hints the XR runtime that the thread is doing background CPU tasks

- `XR_ANDROID_THREAD_TYPE_RENDERER_MAIN_KHR`
  hints the XR runtime that the thread is doing time critical graphics device tasks

- `XR_ANDROID_THREAD_TYPE_RENDERER_WORKER_KHR`
  hints the XR runtime that the thread is doing background graphics device tasks

New Structures

New Functions

To declare a thread to be of a certain [XrAndroidThreadTypeKHR](#) type call:

```
// Provided by XR_KHR_android_thread_settings
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

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.

- Revision 6, 2023-12-04 (Xiang Wei)

  - Revise/fix the hints of enum specification

12.4. XR_KHR_binding_modification

Name String

`XR_KHR_binding_modification`

Extension Type

Instance extension

Registered Extension Number

121

Revision

1

Ratification Status

Ratified

Extension and Version Dependencies

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

XrStructureType enumeration is extended with:

- `XR_TYPE_BINDING_MODIFICATIONS_KHR`

New Enums

New Structures

The XrBindingModificationsKHR structure is defined as:

```
// Provided by XR_KHR_binding_modification
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. See also: XrInteractionProfileDpadBindingEXT

The XrBindingModificationBaseHeaderKHR structure is defined as:

```
// Provided by XR_KHR_binding_modification
typedef struct XrBindingModificationBaseHeaderKHR {
    XrStructureType    type;
    const void*        next;
} XrBindingModificationBaseHeaderKHR;
```

<div>

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.

</div>

The XrBindingModificationBaseHeaderKHR is a base structure is overridden by XrBindingModification* child structures.

<div>

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

</div>

New Functions

Issues

Version History

- Revision 1, 2020-08-06 (Joe Ludwig)
  - Initial draft.

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

Ratification Status

    Ratified

Extension and Version Dependencies

    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:

```
// Provided by XR_KHR_composition_layer_color_scale_bias
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

Ratification Status

    Ratified

Extension and Version Dependencies

    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:

```
// Provided by XR_KHR_composition_layer_cube
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.

- `eyeVisibility` is the eye represented by this layer.

- `swapchain` is the swapchain, which **must** have been created with a XrSwapchainCreateInfo ::`faceCount` of 6.

- `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.

<div style="border: 1px solid #ccc; padding: 1em;">

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

</div>

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).

- Revision 8, 2019-01-24 (Martin Renschler)
  - Updated struct to use XrSwapchainSubImage, reformat and spec language changes, eye parameter description update

12.7. XR_KHR_composition_layer_cylinder

Name String

XR_KHR_composition_layer_cylinder

Extension Type

Instance extension

Registered Extension Number

18

Revision

4

Ratification Status

Ratified

Extension and Version Dependencies

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:

```
XrPosef          pose;
float            radius;
float            centralAngle;
float            aspectRatio;
```

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.
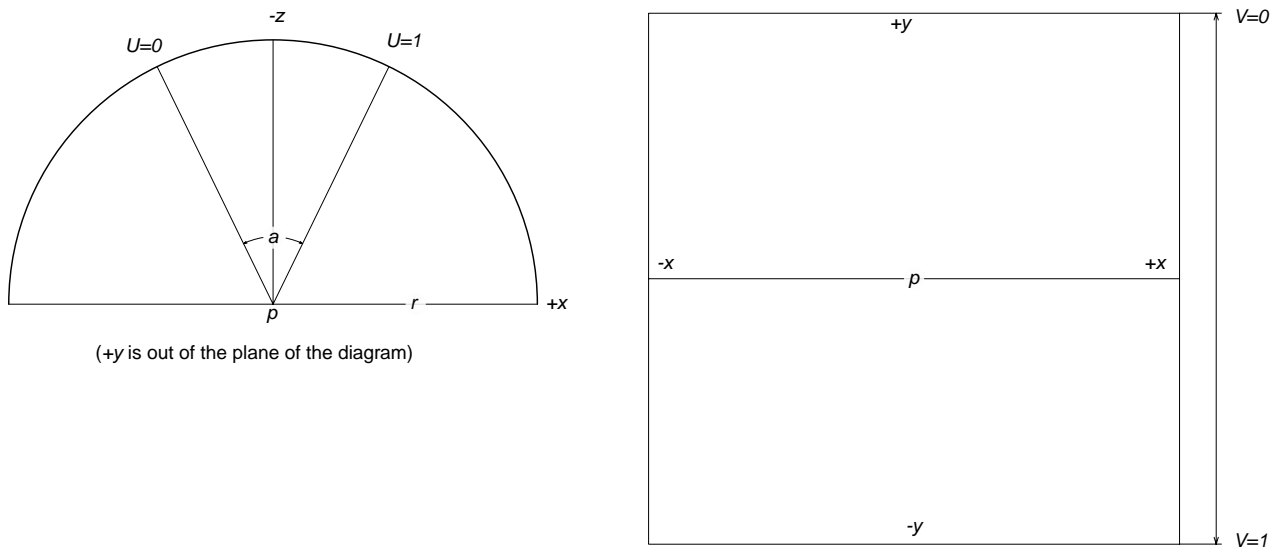
Figure 6. Cylinder Layer Parameters

- 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:

```
// Provided by XR_KHR_composition_layer_cylinder
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.

- `eyeVisibility` is the eye represented by this layer.

- `subImage` identifies the image XrSwapchainSubImage to use. The swapchain **must** have been created with a XrSwapchainCreateInfo::`faceCount` of 1.

- `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 × cylinder angle) / 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

Ratification Status

Ratified

Extension and Version Dependencies

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:

```
// Provided by XR_KHR_composition_layer_depth
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. The swapchain **must** have been created with a XrSwapchainCreateInfo ::`faceCount` of 1.

- `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.

$$\mathbf{T} = \begin{bmatrix} b - a & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} -\frac{f}{f-n} & -\frac{fn}{f-n} \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} -\frac{bf-an}{f-n} & -\frac{fn(b-a)}{f-n} \\ -1 & 0 \end{bmatrix}$$

$$\mathbf{p}_w = \mathbf{T}\mathbf{p}_v$$

$$\mathbf{p}_w = \begin{bmatrix} z_w & w_w \end{bmatrix}^t, \text{ homogeneous window space depth}$$

$$\mathbf{p}_v = \begin{bmatrix} z_v & w_v \end{bmatrix}^t, \text{ homogeneous view space depth}$$

Figure 7. 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 `nearZ` > `farZ`.

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

Ratification Status

> Ratified

Extension and Version Dependencies

> 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:

```
// Provided by XR_KHR_composition_layer_equirect
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;
```

- `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.

- `eyeVisibility` is the eye represented by this layer.

- `subImage` identifies the image XrSwapchainSubImage to use. The swapchain **must** have been created with a XrSwapchainCreateInfo::`faceCount` of 1.

- `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

Revision

1

Ratification Status

Ratified

Extension and Version Dependencies

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

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:

- `XR_TYPE_COMPOSITION_LAYER_EQUIRECT2_KHR`

New Enums

New Structures

The XrCompositionLayerEquirect2KHR structure is defined as:

```
// Provided by XR_KHR_composition_layer_equirect2
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.

- eyeVisibility is the eye represented by this layer.

- subImage identifies the image XrSwapchainSubImage to use. The swapchain **must** have been created with a XrSwapchainCreateInfo::faceCount of 1.

- 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 $[0, 2\pi]$. 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 $[-\pi/2, \pi/2]$.

- lowerVerticalAngle defines the lower vertical angle of the visible portion of the sphere, in the range of $[-\pi/2, \pi/2]$.

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.

- 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

Issues

Version History

- Revision 1, 2020-05-08 (Cass Everitt)

  ◦ Initial version.

  ◦ Kept contributors from the original equirect extension.

12.11. XR_KHR_convert_timespec_time

Name String

  `XR_KHR_convert_timespec_time`

Extension Type

  Instance extension

Registered Extension Number

  37

Revision

  1

Ratification Status

  Ratified

Extension and Version Dependencies

  OpenXR 1.0

Last Modified Date

2019-01-24

IP Status

No known IP claims.

Contributors

Paul Pedriana, Oculus

Overview

This extension provides two functions for converting between timespec monotonic time and XrTime. The xrConvertTimespecTimeToTimeKHR function converts from timespec time to XrTime, while the xrConvertTimeToTimespecTimeKHR function converts 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.

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

New Functions

To convert from timespec monotonic time to XrTime, call:

```
// Provided by XR_KHR_convert_timespec_time
XrResult xrConvertTimespecTimeToTimeKHR(
    XrInstance                                  instance,
    const struct timespec*                      timespecTime,
    XrTime*                                     time);
```

Parameter Descriptions

- instance is an XrInstance handle previously created with xrCreateInstance.

- timespecTime is a timespec obtained from clock_gettime with CLOCK_MONOTONIC.

- time is the resulting XrTime that is equivalent to the timespecTime.

The `xrConvertTimespecTimeToTimeKHR` function converts a time obtained by the `clock_gettime` function to the equivalent `XrTime`.

If the output `time` cannot represent the input `timespecTime`, the runtime **must** return `XR_ERROR_TIME_INVALID`.

Valid Usage (Implicit)

- The `XR_KHR_convert_timespec_time` extension **must** be enabled prior to calling xrConvertTimespecTimeToTimeKHR

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

- `timespecTime` **must** be a pointer to a valid `timespec` 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 timespec monotonic time, call:

```
// Provided by XR_KHR_convert_timespec_time
XrResult xrConvertTimeToTimespecTimeKHR(
    XrInstance                                  instance,
    XrTime                                      time,
    struct timespec*                            timespecTime);
```

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`.

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

Revision

10

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0

Last Modified Date

2025-03-07

IP Status

No known IP claims.

Contributors

Bryce Hutchings, Microsoft
Paul Pedriana, Oculus
Mark Young, LunarG
Minmin Gong, Microsoft

12.12.1. Overview

This extension enables the use of the Direct3D 11 (D3D11) graphics API in an OpenXR application. Without this extension, an OpenXR application **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. However, the runtime provides the D3D11 textures to render into. This extension provides mechanisms for the application to interact with those textures by calling xrEnumerateSwapchainImages and providing XrSwapchainImageD3D11KHR structures to populate.

In order to expose the structures, types, and functions of this extension, the application source code **must** define XR_USE_GRAPHICS_API_D3D11 before including the OpenXR platform header openxr_platform.h, in all portions of the library or application that interact with the types, values, and functions it defines.

12.12.2. Get Graphics Requirements

Some computer systems have multiple graphics devices, each of which **may** have independent external display outputs. XR systems that connect to such computer systems are typically connected to a single graphics device. Applications need to know the graphics device associated with the XR system, so that rendering takes place on the correct graphics device.

The xrGetD3D11GraphicsRequirementsKHR function is defined as:

```
// Provided by XR_KHR_D3D11_enable
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.

This call retrieves the D3D11 feature level and graphics device for an instance and system. The xrGetD3D11GraphicsRequirementsKHR function identifies to the application the graphics device (Windows LUID) 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.

The XrGraphicsRequirementsD3D11KHR structure is defined as:

```
// Provided by XR_KHR_D3D11_enable
typedef struct XrGraphicsRequirementsD3D11KHR {
    XrStructureType      type;
    void*                next;
    LUID                 adapterLuid;
    D3D_FEATURE_LEVEL    minFeatureLevel;
} XrGraphicsRequirementsD3D11KHR;
```

- `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 for which to initialize the D3D11 device.

XrGraphicsRequirementsD3D11KHR is populated by xrGetD3D11GraphicsRequirementsKHR with the runtime's D3D11 API feature level and adapter requirements.

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

12.12.3. Graphics Binding Structure

The XrGraphicsBindingD3D11KHR structure is defined as:

```
// Provided by XR_KHR_D3D11_enable
typedef struct XrGraphicsBindingD3D11KHR {
    XrStructureType    type;
    const void*        next;
    ID3D11Device*      device;
} XrGraphicsBindingD3D11KHR;
```

- `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.

To create a D3D11-backed XrSession, the application provides a pointer to an XrGraphicsBindingD3D11KHR structure in the XrSessionCreateInfo::`next` chain when calling 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`.

Creating a session using this structure triggers several requirements on the runtime regarding swapchain images. See the specification of XrSwapchainImageD3D11KHR for details.

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

12.12.4. Swapchain Images

The XrSwapchainImageD3D11KHR structure is defined as:

```
// Provided by XR_KHR_D3D11_enable
typedef struct XrSwapchainImageD3D11KHR {
    XrStructureType    type;
    void*              next;
    ID3D11Texture2D*   texture;
} XrSwapchainImageD3D11KHR;
```

If a given session was created with XrGraphicsBindingD3D11KHR, the following conditions 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.

The OpenXR runtime **must** return a texture created in accordance with D3D11 Swapchain Flag Bits.

12.12.5. D3D11 Swapchain Flag Bits

All valid 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.

XrSwapchainUsageFlagBits	Corresponding D3D11 bind flag bits
`XR_SWAPCHAIN_USAGE_COLOR_ATTACHMENT_BIT`	`D3D11_BIND_RENDER_TARGET`
`XR_SWAPCHAIN_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`	`D3D11_BIND_DEPTH_STENCIL`
`XR_SWAPCHAIN_USAGE_UNORDERED_ACCESS_BIT`	`D3D11_BIND_UNORDERED_ACCESS`

XrSwapchainUsageFlagBits	Corresponding D3D11 bind flag bits
XR_SWAPCHAIN_USAGE_TRANSFER_SRC_BIT	*ignored*
XR_SWAPCHAIN_USAGE_TRANSFER_DST_BIT	*ignored*
XR_SWAPCHAIN_USAGE_SAMPLED_BIT	D3D11_BIND_SHADER_RESOURCE
XR_SWAPCHAIN_USAGE_MUTABLE_FORMAT_BIT	*ignored*
XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_KHR (Added by XR_KHR_swapchain_usage_input_attachment_bit and only available when that extension is enabled)	*ignored*

All D3D11 swapchain textures are created with D3D11_USAGE_DEFAULT usage.

12.12.6. New Commands

- xrGetD3D11GraphicsRequirementsKHR

12.12.7. New Structures

- XrGraphicsRequirementsD3D11KHR

- XrSwapchainImageD3D11KHR

- Extending XrSessionCreateInfo:

    ◦ XrGraphicsBindingD3D11KHR

12.12.8. New Enum Constants

- XR_KHR_D3D11_ENABLE_EXTENSION_NAME

- XR_KHR_D3D11_enable_SPEC_VERSION

- Extending XrStructureType:

    ◦ XR_TYPE_GRAPHICS_BINDING_D3D11_KHR

    ◦ XR_TYPE_GRAPHICS_REQUIREMENTS_D3D11_KHR

    ◦ XR_TYPE_SWAPCHAIN_IMAGE_D3D11_KHR

12.12.9. 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
- Revision 8, 2021-09-09 (Bryce Hutchings)
  - Document mapping for `XrSwapchainUsageFlags`
- Revision 9, 2021-12-28 (Microsoft)
  - Added missing `XR_ERROR_GRAPHICS_DEVICE_INVALID` error condition
- Revision 10, 2025-03-07 (Rylie Pavlik, Collabora, Ltd.)
  - Re-organize, clarify, and make more uniform with other graphics binding extensions.

12.13. XR_KHR_D3D12_enable

Name String

`XR_KHR_D3D12_enable`

Extension Type

Instance extension

Registered Extension Number

29

Revision

10

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0

Last Modified Date

2025-03-07

IP Status

No known IP claims.

Contributors

> Bryce Hutchings, Microsoft
>
> Paul Pedriana, Oculus
>
> Mark Young, LunarG
>
> Minmin Gong, Microsoft
>
> Dan Ginsburg, Valve

12.13.1. Overview

This extension enables the use of the Direct3D 12 (D3D12) graphics API in an OpenXR application. Without this extension, an OpenXR application **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. However, the runtime provides the D3D12 images to render into. This extension provides mechanisms for the application to interact with those images by calling xrEnumerateSwapchainImages and providing XrSwapchainImageD3D12KHR structures to populate.

In order to expose the structures, types, and functions of this extension, the application source code **must** define XR_USE_GRAPHICS_API_D3D12 before including the OpenXR platform header `openxr_platform.h`, in all portions of the library or application that interact with the types, values, and functions it defines.

12.13.2. Get Graphics Requirements

Some computer systems have multiple graphics devices, each of which **may** have independent external display outputs. XR systems that connect to such computer systems are typically connected to a single graphics device. Applications need to know the graphics device associated with the XR system, so that rendering takes place on the correct graphics device.

The xrGetD3D12GraphicsRequirementsKHR function is defined as:

```
// Provided by XR_KHR_D3D12_enable
XrResult xrGetD3D12GraphicsRequirementsKHR(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    XrGraphicsRequirementsD3D12KHR*             graphicsRequirements);
```

This call retrieves the D3D12 feature level and graphics device for an instance and system. The xrGetD3D12GraphicsRequirementsKHR function identifies to the application the graphics device (Windows LUID) 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.

The XrGraphicsRequirementsD3D12KHR structure is defined as:

```
// Provided by XR_KHR_D3D12_enable
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 for which to initialize the D3D12 device.

XrGraphicsRequirementsD3D12KHR is populated by xrGetD3D12GraphicsRequirementsKHR with the runtime's D3D12 API feature level and adapter requirements.

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

12.13.3. Graphics Binding Structure

The XrGraphicsBindingD3D12KHR structure is defined as:

```
// Provided by XR_KHR_D3D12_enable
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.

To create a D3D12-backed XrSession, the application provides a pointer to an XrGraphicsBindingD3D12KHR structure in the XrSessionCreateInfo::`next` chain when calling 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`.

Creating a session using this structure triggers several requirements on the runtime regarding swapchain images. See the specification of XrSwapchainImageD3D12KHR for details.

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

12.13.4. Swapchain Images

The XrSwapchainImageD3D12KHR structure is defined as:

```
// Provided by XR_KHR_D3D12_enable
typedef struct XrSwapchainImageD3D12KHR {
    XrStructureType    type;
    void*              next;
    ID3D12Resource*    texture;
} XrSwapchainImageD3D12KHR;
```

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 `ID3D12Texture2D` to use.

If a given session was created with XrGraphicsBindingD3D12KHR, the following conditions apply.

- Calls to xrEnumerateSwapchainImages on an XrSwapchain in that session **must** return an array of XrSwapchainImageD3D12KHR structures.
- Whenever an OpenXR function accepts an XrSwapchainImageBaseHeader pointer as a parameter in that session, the runtime **must** also accept a pointer to an XrSwapchainImageD3D12KHR.

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.

The OpenXR runtime **must** return a texture created in accordance with D3D12 Swapchain Flag Bits.

The OpenXR runtime **must** manage image resource state in accordance with D3D12 Swapchain Image Resource State.

Valid Usage (Implicit)

- The `XR_KHR_D3D12_enable` extension **must** be enabled prior to using XrSwapchainImageD3D12KHR

- `type` **must** be `XR_TYPE_SWAPCHAIN_IMAGE_D3D12_KHR`

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

12.13.5. D3D12 Swapchain Flag Bits

All valid 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_RESOURCE_FLAGS flags and heap type. The runtime **may** set additional resource flags but **must** not restrict usage.

XrSwapchainUsageFlagBits	Corresponding D3D12 resource flag bits
XR_SWAPCHAIN_USAGE_COLOR_ATTACHMENT_BIT	D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET
XR_SWAPCHAIN_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT	D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL
XR_SWAPCHAIN_USAGE_UNORDERED_ACCESS_BIT	D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS
XR_SWAPCHAIN_USAGE_TRANSFER_SRC_BIT	*ignored*
XR_SWAPCHAIN_USAGE_TRANSFER_DST_BIT	*ignored*
XR_SWAPCHAIN_USAGE_SAMPLED_BIT **omitted**	D3D12_RESOURCE_FLAG_DENY_SHADER_RESOURCE
XR_SWAPCHAIN_USAGE_MUTABLE_FORMAT_BIT	*ignored*
XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_KHR (Added by XR_KHR_swapchain_usage_input_attachment_bit and only available when that extension is enabled)	*ignored*

All D3D12 swapchain textures are created with D3D12_HEAP_TYPE_DEFAULT heap type.

12.13.6. D3D12 Swapchain Image Resource State

If an application waits on a swapchain image by calling xrWaitSwapchainImage in a session created using XrGraphicsBindingD3D12KHR, and that call returns XR_SUCCESS or XR_SESSION_LOSS_PENDING, then the OpenXR runtime **must** guarantee that the following conditions are true:

- 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 is able to 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.

12.13.7. New Commands

- xrGetD3D12GraphicsRequirementsKHR

12.13.8. New Structures

- XrGraphicsRequirementsD3D12KHR
- XrSwapchainImageD3D12KHR
- Extending XrSessionCreateInfo:
  - XrGraphicsBindingD3D12KHR

12.13.9. New Enum Constants

- `XR_KHR_D3D12_ENABLE_EXTENSION_NAME`
- `XR_KHR_D3D12_enable_SPEC_VERSION`
- Extending XrStructureType:
  - `XR_TYPE_GRAPHICS_BINDING_D3D12_KHR`
  - `XR_TYPE_GRAPHICS_REQUIREMENTS_D3D12_KHR`
  - `XR_TYPE_SWAPCHAIN_IMAGE_D3D12_KHR`

12.13.10. 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 (Microsoft)

  ◦ Added missing `XR_ERROR_GRAPHICS_DEVICE_INVALID` error condition

- Revision 10, 2025-03-07 (Rylie Pavlik, Collabora, Ltd.)

  ◦ Re-organize, clarify, and make more uniform with other graphics binding extensions.

12.14. XR_KHR_extended_struct_name_lengths

Name String

`XR_KHR_extended_struct_name_lengths`

Extension Type

Instance extension

Registered Extension Number

149

Revision

1

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0

Last Modified Date

2024-08-12

IP Status

No known IP claims.

Contributors

Matthew Langille, Meta Platforms
Andreas Selvik, Meta Platforms

Rylie Pavlik, Collabora, Ltd.

12.14.1. Overview

This extension extends the maximum struct name sizes and provides a new function to access these new extended names.

xrStructureTypeToString2KHR is provided to allow retrieving the string names of structure type enumerants with lengths that exceed the original limit of 63 bytes (64 bytes including the null terminator). xrStructureTypeToString2KHR returns name strings for structure type enumerants up to 127 bytes in length (128 bytes including the null terminator). An application **can** use xrStructureTypeToString2KHR as a drop-in replacement for xrStructureTypeToString, as it works with all structure type enumerants, regardless of string name length.

12.14.2. Retrieving Structure Type Enumerant Strings

If the original xrStructureTypeToString is used to retrieve string names for structure type enumerants with name lengths in excess of 63 bytes, its behavior is clarified as follows. xrStructureTypeToString **must** populate the buffer with the correct name, except that the string **must** be truncated at a codepoint boundary to fit within the available buffer. That is, the returned string **must** always be valid UTF-8.

The xrStructureTypeToString2KHR function is defined as:

```
// Provided by XR_KHR_extended_struct_name_lengths
XrResult xrStructureTypeToString2KHR(
    XrInstance                                  instance,
    XrStructureType                             value,
    char                                        buffer
[XR_MAX_STRUCTURE_NAME_SIZE_EXTENDED_KHR]);
```

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 name of the provided XrStructureType value by copying a valid null-terminated UTF-8 string into `buffer`.

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

For structure type enumerants whose names fit within the original size limit of 63 bytes, xrStructureTypeToString2KHR **must** return the same resultant string as xrStructureTypeToString, up to the null terminator.

The `XR_MAX_STRUCTURE_NAME_SIZE_EXTENDED_KHR` enumerant defines the size of the buffer passed to xrStructureTypeToString2KHR.

```
#define XR_MAX_STRUCTURE_NAME_SIZE_EXTENDED_KHR 256
```

12.14.3. New Commands

- xrStructureTypeToString2KHR

12.14.4. New Enum Constants

- `XR_KHR_EXTENDED_STRUCT_NAME_LENGTHS_EXTENSION_NAME`

- `XR_KHR_extended_struct_name_lengths_SPEC_VERSION`

- `XR_MAX_STRUCTURE_NAME_SIZE_EXTENDED_KHR`

Version History

- Revision 1, 2024-02-29 (Matthew Langille)

  - Initial extension description

12.15. XR_KHR_loader_init

Name String

    `XR_KHR_loader_init`

Extension Type

    Instance extension

Registered Extension Number

    89

Revision

    2

Ratification Status

    Ratified

Extension and Version Dependencies

    OpenXR 1.0

Last Modified Date

    2023-05-08

IP Status

    No known IP claims.

Contributors

> Cass Everitt, Facebook
>
> Robert Blenkinsopp, Ultraleap

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 XR_NULL_HANDLE as the `instance` parameter.

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 and the loader **must** pass calls to xrInitializeLoaderKHR to the active runtime, and all enabled API layers that expose a xrInitializeLoaderKHR function exposed either through their manifest, or through their implementation of xrGetInstanceProcAddr.

If the xrInitializeLoaderKHR function is discovered through the manifest, xrInitializeLoaderKHR will be called before `xrNegotiateLoaderRuntimeInterface` or `xrNegotiateLoaderApiLayerInterface` has been called on the runtime or layer respectively.

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

The XrLoaderInitInfoBaseHeaderKHR structure is defined as:

```
// Provided by XR_KHR_loader_init
typedef struct XrLoaderInitInfoBaseHeaderKHR {
    XrStructureType    type;
    const void*        next;
} XrLoaderInitInfoBaseHeaderKHR;
```

- `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:

```
// Provided by XR_KHR_loader_init
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 2, 2023-05-08 (Robert Blenkinsoppp)

  - Explicitly state that the call to xrInitializeLoaderKHR should be passed to the runtime and enabled API layers.

- Revision 1, 2020-05-07 (Cass Everitt)

  - Initial draft

12.16. XR_KHR_loader_init_android

Name String

XR_KHR_loader_init_android

Extension Type

Instance extension

Registered Extension Number

90

Revision

1

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0
and
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:

```
// Provided by XR_KHR_loader_init_android
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

12.17. XR_KHR_metal_enable

Name String

XR_KHR_metal_enable

Extension Type

Instance extension

Registered Extension Number

30

Revision

2

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0

Last Modified Date

2025-03-07

IP Status

No known IP claims.

Contributors

Xiang Wei, Meta
Peter Kuhn, Unity
John Kearney, Meta
Andreas Selvik, Meta
Jakob Bornecrantz, Collabora
Rylie Pavlik, Collabora

12.17.1. Overview

This extension enables the use of the Metal® graphics API in an OpenXR application. Without this extension, an OpenXR application **may** not be able to use any Metal swapchain images.

This extension provides the mechanisms necessary for an application to generate a valid XrGraphicsBindingMetalKHR structure in order to create a Metal-based XrSession. Note that during this process, the runtime is responsible for creating the Metal device for the application's drawing operations, and the application is responsible for creating all the required Metal objects from that, including a Metal command queue to be used for rendering. The runtime however will provide the Metal textures to render into in the form of a swapchain.

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_METAL before including the OpenXR platform header openxr_platform.h, in all portions of the library or application that interact with the types, values, and functions it defines.

12.17.2. Get Graphics Requirements

Some computer systems **may** have multiple graphics devices, each of which **may** have independent external display outputs. XR systems that connect to such computer systems are typically connected to a single graphics device. Applications need to know the graphics device associated with the XR system, so that rendering takes place on the correct graphics device.

To retrieve the Metal device that **can** be used in drawing operations, call:

```
// Provided by XR_KHR_metal_enable
XrResult xrGetMetalGraphicsRequirementsKHR(
    XrInstance                              instance,
    XrSystemId                              systemId,
    XrGraphicsRequirementsMetalKHR*         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 XrGraphicsRequirementsMetalKHR output structure.

The xrGetMetalGraphicsRequirementsKHR function identifies to the application the Metal device to be used in drawing operations. The runtime **must** return XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING on calls to xrCreateSession if xrGetMetalGraphicsRequirementsKHR has not been called for the same instance and systemId.

The Metal device that xrGetMetalGraphicsRequirementsKHR returns **must** be used to create the Metal command queue that the application passes to xrCreateSession in the XrGraphicsBindingMetalKHR.

Valid Usage (Implicit)

- The `XR_KHR_metal_enable` extension **must** be enabled prior to calling xrGetMetalGraphicsRequirementsKHR
- `instance` **must** be a valid XrInstance handle
- `graphicsRequirements` **must** be a pointer to an XrGraphicsRequirementsMetalKHR 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 XrGraphicsRequirementsMetalKHR structure is defined as:

```
// Provided by XR_KHR_metal_enable
typedef struct XrGraphicsRequirementsMetalKHR {
    XrStructureType    type;
    void*              next;
    void*              metalDevice;
} XrGraphicsRequirementsMetalKHR;
```

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.

- `metalDevice` is the Metal device that the application **must** use for drawing operations, which **may** be bridged casted to an Objective-C object that conforms to the `MTLDevice` protocol.

[XrGraphicsRequirementsMetalKHR](#) is populated by [xrGetMetalGraphicsRequirementsKHR](#).

Valid Usage (Implicit)

- The `XR_KHR_metal_enable` extension **must** be enabled prior to using [XrGraphicsRequirementsMetalKHR](#)

- `type` **must** be `XR_TYPE_GRAPHICS_REQUIREMENTS_METAL_KHR`

- `next` **must** be `NULL` or a valid pointer to the [next structure in a structure chain](#)

- `metalDevice` **must** be a pointer value

12.17.3. Graphics Binding Structure

The [XrGraphicsBindingMetalKHR](#) structure is defined as:

```
// Provided by XR_KHR_metal_enable
typedef struct XrGraphicsBindingMetalKHR {
    XrStructureType    type;
    const void*        next;
    void*              commandQueue;
} XrGraphicsBindingMetalKHR;
```

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.

- `commandQueue` is a valid Metal command queue, which **must** be bridged casted from an Objective-C object that conforms to the `MTLCommandQueue` protocol.

To create a Metal-backed XrSession, the application provides a pointer to an XrGraphicsBindingMetalKHR in the XrSessionCreateInfo::next field of structure passed to xrCreateSession. The Metal command queue specified in XrGraphicsBindingMetalKHR::commandQueue **must** be created on the Metal device retrieved through XrGraphicsRequirementsMetalKHR ::metalDevice, otherwise xrCreateSession **must** return XR_ERROR_GRAPHICS_DEVICE_INVALID.

Creating a session using this structure triggers several requirements on the runtime regarding swapchain images. See the specification of XrSwapchainImageMetalKHR for details.

Valid Usage (Implicit)

- The `XR_KHR_metal_enable` extension **must** be enabled prior to using XrGraphicsBindingMetalKHR

- `type` **must** be `XR_TYPE_GRAPHICS_BINDING_METAL_KHR`

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

- `commandQueue` **must** be a pointer value

12.17.4. Swapchain Images

The XrSwapchainImageMetalKHR structure is defined as:

```
// Provided by XR_KHR_metal_enable
typedef struct XrSwapchainImageMetalKHR {
    XrStructureType    type;
    const void*        next;
    void*              texture;
} XrSwapchainImageMetalKHR;
```

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 populated with a valid Metal texture to use, which **must** be able to be bridged casted to an Objective-C object that conforms to the `MTLTexture` protocol.

If a given session was created with XrGraphicsBindingMetalKHR, the following conditions apply.

- Calls to xrEnumerateSwapchainImages on an XrSwapchain in that session **must** return an array of

XrSwapchainImageMetalKHR structures.

- Whenever an OpenXR function accepts an XrSwapchainImageBaseHeader pointer as a parameter in that session, the runtime **must** also accept a pointer to an XrSwapchainImageMetalKHR.

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_metal_enable` extension **must** be enabled prior to using XrSwapchainImageMetalKHR

- `type` **must** be `XR_TYPE_SWAPCHAIN_IMAGE_METAL_KHR`

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

- `texture` **must** be a pointer value

12.17.5. Metal Swapchain Flag Bits

All valid XrSwapchainUsageFlags values passed in a session created using XrGraphicsBindingMetalKHR **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 `MTLTextureUsage` flags. The runtime **may** set additional bind flags but **must** not restrict usage.

XrSwapchainUsageFlagBits	**Corresponding** `MTLTextureUsage` **bits**
`XR_SWAPCHAIN_USAGE_COLOR_ATTACHMENT_BIT`	`MTLTextureUsageRenderTarget`
`XR_SWAPCHAIN_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`	`MTLTextureUsageRenderTarget`
`XR_SWAPCHAIN_USAGE_UNORDERED_ACCESS_BIT`	`MTLTextureUsageShaderRead | MTLTextureUsageShaderWrite`
`XR_SWAPCHAIN_USAGE_TRANSFER_SRC_BIT`	*ignored*
`XR_SWAPCHAIN_USAGE_TRANSFER_DST_BIT`	*ignored*
`XR_SWAPCHAIN_USAGE_SAMPLED_BIT`	`MTLTextureUsageShaderRead`
`XR_SWAPCHAIN_USAGE_MUTABLE_FORMAT_BIT`	`MTLTextureUsagePixelFormatView`
`XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_KHR` (Added by `XR_KHR_swapchain_usage_input_attachment_bit` and only available when that extension is enabled)	*ignored*

All Metal swapchain textures are created with `MTLResourceStorageModePrivate` resource option, and are accessible only by the GPU.

12.17.6. Issues

- How to manage the resource state of the Swapchain textures, etc?

  ◦ The application uses the Metal device that is created by the runtime for the drawing operations. The runtime uses the same Metal device to create the swapchain images, and also create the synchronization events when necessary. On top of that, Metal tracks the write hazards and synchronizes the resources which are created from the same Metal device and directly bind to a pipeline. Please check this Apple documentation for more details: https://developer.apple.com/documentation/metal/resource_synchronization?language=objc

12.17.7. New Commands

- xrGetMetalGraphicsRequirementsKHR

12.17.8. New Structures

- XrGraphicsRequirementsMetalKHR

- XrSwapchainImageMetalKHR

- Extending XrSessionCreateInfo:

  ◦ XrGraphicsBindingMetalKHR

12.17.9. New Enum Constants

- `XR_KHR_METAL_ENABLE_EXTENSION_NAME`

- `XR_KHR_metal_enable_SPEC_VERSION`

- Extending XrStructureType:

  ◦ `XR_TYPE_GRAPHICS_BINDING_METAL_KHR`

  ◦ `XR_TYPE_GRAPHICS_REQUIREMENTS_METAL_KHR`

  ◦ `XR_TYPE_SWAPCHAIN_IMAGE_METAL_KHR`

12.17.10. Version History

- Revision 1, 2023-05-08 (Xiang Wei)

  ◦ Initial draft

- Revision 2, 2025-03-07 (Rylie Pavlik, Collabora, Ltd.)

  ◦ Re-organize, clarify, and make more uniform with other graphics binding extensions.

12.18. XR_KHR_opengl_enable

Name String

XR_KHR_opengl_enable

Extension Type

Instance extension

Registered Extension Number

24

Revision

11

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0

Last Modified Date

2025-03-07

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

12.18.1. Overview

This extension enables the use of the OpenGL graphics API in an OpenXR application. Without this extension, an OpenXR application **may** not be able to use 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 the application is responsible for creating an OpenGL context to be used for rendering. However, the runtime provides the OpenGL textures to render into. This extension provides mechanisms for the application to interact with those textures by calling xrEnumerateSwapchainImages and providing XrSwapchainImageOpenGLKHR structures to populate.

In order to expose the structures, types, and functions of this extension, the application source code **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 interact with the types, values, and functions it defines. 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 structures introduced by this extension which correspond to the platform it is running on.

12.18.2. OpenGL Context and Threading

Note that the OpenGL context given to the call to xrCreateSession **must** not be bound in *another* thread by the application when calling the functions:

- xrCreateSession

- xrDestroySession

- xrBeginFrame

- xrEndFrame

- xrCreateSwapchain

- xrDestroySwapchain

- xrEnumerateSwapchainImages

- xrAcquireSwapchainImage

- xrWaitSwapchainImage

- xrReleaseSwapchainImage

However, it **may** be bound in the thread calling one of 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.

12.18.3. Get Graphics Requirements

The xrGetOpenGLGraphicsRequirementsKHR function is defined as:

```
// Provided by XR_KHR_opengl_enable
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.

This call queries OpenGL API version requirements for an instance and system. 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

The XrGraphicsRequirementsOpenGLKHR structure is defined as:

```
// Provided by XR_KHR_opengl_enable
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 **may** 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.

12.18.4. Graphics Binding Structure

These structures are only available when the corresponding `XR_USE_PLATFORM_` window system/platform macro is defined before including `openxr_platform.h`.

The XrGraphicsBindingOpenGLWin32KHR structure is defined as:

```
// Provided by XR_KHR_opengl_enable
typedef struct XrGraphicsBindingOpenGLWin32KHR {
    XrStructureType     type;
    const void*         next;
    HDC                 hDC;
    HGLRC               hGLRC;
} XrGraphicsBindingOpenGLWin32KHR;
```

To create an OpenGL-backed XrSession on Microsoft Windows, the application provides a pointer to an XrGraphicsBindingOpenGLWin32KHR structure in the XrSessionCreateInfo::next chain when calling xrCreateSession. 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 system's default GPU when this structure is supplied. If the GPU used by the runtime does not match the GPU on which the OpenGL context of the application was created, xrCreateSession **must** return `XR_ERROR_GRAPHICS_DEVICE_INVALID`.

Creating a session using this structure triggers several requirements on the runtime regarding

swapchain images. See the specification of XrSwapchainImageOpenGLKHR for details.

By providing a context as `hGLRC`, the application becomes subject to restrictions on use of that context which effectively introduces additional external synchronization requirements on some OpenXR calls. See OpenGL Context and Threading for details.

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:

```
// Provided by XR_KHR_opengl_enable
typedef struct XrGraphicsBindingOpenGLXlibKHR {
    XrStructureType    type;
    const void*        next;
    Display*           xDisplay;
    uint32_t           visualid;
    GLXFBConfig        glxFBConfig;
    GLXDrawable        glxDrawable;
    GLXContext         glxContext;
} XrGraphicsBindingOpenGLXlibKHR;
```

- `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`.

To create an OpenGL-backed XrSession on any Linux/Unix platform that utilizes X11 and GLX, via the Xlib library, the application provides a pointer to an XrGraphicsBindingOpenGLXlibKHR in the XrSessionCreateInfo::`next` chain when calling xrCreateSession.

Creating a session using this structure triggers several requirements on the runtime regarding swapchain images. See the specification of XrSwapchainImageOpenGLKHR for details.

By providing a GLX context as `glxContext`, the application becomes subject to restrictions on use of that context which effectively introduces additional external synchronization requirements on some OpenXR calls. See OpenGL Context and Threading for details.

The required window system configuration define to expose this structure type is XR_USE_PLATFORM_XLIB.

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:

```
// Provided by XR_KHR_opengl_enable
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 the screen to 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`.

To create an OpenGL-backed XrSession on any Linux/Unix platform that utilizes X11 and GLX, via the Xlib library, the application provides a pointer to an XrGraphicsBindingOpenGLXcbKHR in the XrSessionCreateInfo::next chain when calling xrCreateSession.

Creating a session using this structure triggers several requirements on the runtime regarding swapchain images. See the specification of XrSwapchainImageOpenGLKHR for details.

By providing a GLX context as `glxContext`, the application becomes subject to restrictions on use of that context which effectively introduces additional external synchronization requirements on some OpenXR calls. See OpenGL Context and Threading for details.

The required window system configuration define to expose this structure type is XR_USE_PLATFORM_XCB.

- 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:

```
// Provided by XR_KHR_opengl_enable
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`.

To create an OpenGL-backed XrSession on any Linux/Unix platform that utilizes the Wayland protocol with its compositor, the application provides a pointer to an XrGraphicsBindingOpenGLWaylandKHR in the XrSessionCreateInfo::`next` chain when calling xrCreateSession.

Creating a session using this structure triggers several requirements on the runtime regarding swapchain images. See the specification of XrSwapchainImageOpenGLKHR for details.

The required window system configuration define to expose this structure type is XR_USE_PLATFORM_WAYLAND.

12.18.5. Swapchain Images

The XrSwapchainImageOpenGLKHR structure is defined as:

```
// Provided by XR_KHR_opengl_enable
typedef struct XrSwapchainImageOpenGLKHR {
    XrStructureType    type;
    void*              next;
    uint32_t           image;
} XrSwapchainImageOpenGLKHR;
```

If a given session was created with some `XrGraphicsBindingOpenGL*KHR` graphics binding structure, the following conditions 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

12.18.6. OpenGL Swapchain Flag Bits

All valid XrSwapchainUsageFlags 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.

12.18.7. New Commands

- xrGetOpenGLGraphicsRequirementsKHR

12.18.8. New Structures

- XrGraphicsRequirementsOpenGLKHR

- XrSwapchainImageOpenGLKHR

- Extending XrSessionCreateInfo:

  - XrGraphicsBindingOpenGLWaylandKHR

  - XrGraphicsBindingOpenGLWin32KHR

  - XrGraphicsBindingOpenGLXcbKHR

  - XrGraphicsBindingOpenGLXlibKHR

12.18.9. New Enum Constants

- `XR_KHR_OPENGL_ENABLE_EXTENSION_NAME`

- `XR_KHR_opengl_enable_SPEC_VERSION`

- Extending XrStructureType:

  - `XR_TYPE_GRAPHICS_BINDING_OPENGL_WAYLAND_KHR`

  - `XR_TYPE_GRAPHICS_BINDING_OPENGL_WIN32_KHR`

  - `XR_TYPE_GRAPHICS_BINDING_OPENGL_XCB_KHR`

  - `XR_TYPE_GRAPHICS_BINDING_OPENGL_XLIB_KHR`

  - `XR_TYPE_GRAPHICS_REQUIREMENTS_OPENGL_KHR`

  - `XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_KHR`

12.18.10. 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

- Revision 5, 2019-01-25 (Robert Menzel)

  - Description updated

- Revision 6, 2019-07-02 (Robert Menzel)

  - Minor fixes

- Revision 7, 2019-07-08 (Rylie Pavlik)

  - Adjusted member name in XCB struct

- Revision 8, 2019-11-28 (Jakob Bornecrantz)

  - Added note about context not allowed to be current in a different thread.

- Revision 9, 2020-08-06 (Bryce Hutchings)

  - Added new `XR_ERROR_GRAPHICS_REQUIREMENTS_CALL_MISSING` error code

- Revision 10, 2021-08-31 (Paulo F. Gomes)

  - Document handling of `XrSwapchainUsageFlags`

- Revision 11, 2025-03-07 (Rylie Pavlik, Collabora, Ltd.)

  - Re-organize, clarify, and make more uniform with other graphics binding extensions.

12.19. XR_KHR_opengl_es_enable

Name String

XR_KHR_opengl_es_enable

Extension Type

Instance extension

Registered Extension Number

25

Revision

9

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0

Last Modified Date

2025-03-07

IP Status

No known IP claims.

Contributors

Mark Young, LunarG
Bryce Hutchings, Microsoft
Paul Pedriana, Oculus
Minmin Gong, Microsoft
Robert Menzel, NVIDIA
Martin Renschler, Qualcomm
Paulo Gomes, Samsung Electronics

12.19.1. Overview

This extension enables the use of the OpenGL ES graphics API in an OpenXR application. Without this extension, an OpenXR application **may** not be able to use any OpenGL ES swapchain images.

This extension provides the mechanisms necessary for an application to generate a valid XrGraphicsBindingOpenGLESAndroidKHR structure in order to create an OpenGL ES-based XrSession. 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. Although not theoretically Android-specific, the OpenGL ES extension is currently tailored for Android. Note that the application

is responsible for creating an OpenGL ES context to be used for rendering. However, the runtime provides the OpenGL ES textures to render into. This extension provides mechanisms for the application to interact with those textures by calling xrEnumerateSwapchainImages and providing XrSwapchainImageOpenGLESKHR structures to populate.

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 supported by this extension, before including the OpenXR platform header `openxr_platform.h`, in all portions of the library or application that interact with the types, values, and functions it defines. The only window system define currently supported by this extension is:

- XR_USE_PLATFORM_ANDROID

> **ℹ** *Note*
>
> This extension does **not** specify requirements for when the supplied context is current in any thread, unlike `XR_KHR_opengl_enable`. This is an oversight in the design of this extension, and fixing it requires either breaking compatibility or a new extension.

12.19.2. Get Graphics Requirements

The xrGetOpenGLESGraphicsRequirementsKHR function is defined as:

```
// Provided by XR_KHR_opengl_es_enable
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.

This call queries OpenGL ES API version requirements for an instance and system. 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`.

The XrGraphicsRequirementsOpenGLESKHR structure is defined as:

```
// Provided by XR_KHR_opengl_es_enable
typedef struct XrGraphicsRequirementsOpenGLESKHR {
    XrStructureType    type;
    void*              next;
    XrVersion          minApiVersionSupported;
    XrVersion          maxApiVersionSupported;
} XrGraphicsRequirementsOpenGLESKHR;
```

- `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 **may** 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

12.19.3. Graphics Binding Structure

These structures are only available when the corresponding `XR_USE_PLATFORM_` window system/platform macro is defined before including `openxr_platform.h`.

The XrGraphicsBindingOpenGLESAndroidKHR structure is defined as:

```
// Provided by XR_KHR_opengl_es_enable
typedef struct XrGraphicsBindingOpenGLESAndroidKHR {
    XrStructureType    type;
    const void*        next;
    EGLDisplay         display;
    EGLConfig          config;
    EGLContext         context;
} XrGraphicsBindingOpenGLESAndroidKHR;
```

- `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`.

To create an OpenGL ES-backed XrSession on Android, the application **can** provide a pointer to an XrGraphicsBindingOpenGLESAndroidKHR structure in the XrSessionCreateInfo::`next` chain when calling xrCreateSession.

Creating a session using this structure triggers several requirements on the runtime regarding swapchain images. See the specification of XrSwapchainImageOpenGLESKHR for details.

The required window system configuration define to expose this structure type is XR_USE_PLATFORM_ANDROID.

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

12.19.4. Swapchain Images

The XrSwapchainImageOpenGLESKHR structure is defined as:

```
// Provided by XR_KHR_opengl_es_enable
typedef struct XrSwapchainImageOpenGLESKHR {
    XrStructureType    type;
    void*              next;
    uint32_t           image;
} XrSwapchainImageOpenGLESKHR;
```

<div align="center">

Member Descriptions

</div>

- `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 some `XrGraphicsBindingOpenGLES*KHR` graphics binding structure, the following conditions 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.

<div align="center">

Valid Usage (Implicit)

</div>

- 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

12.19.5. OpenGL ES Swapchain Flag Bits

All valid XrSwapchainUsageFlags 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.

12.19.6. New Commands

- xrGetOpenGLESGraphicsRequirementsKHR

12.19.7. New Structures

- XrGraphicsRequirementsOpenGLESKHR
- XrSwapchainImageOpenGLESKHR
- Extending XrSessionCreateInfo:
  - XrGraphicsBindingOpenGLESAndroidKHR

12.19.8. New Enum Constants

- XR_KHR_OPENGL_ES_ENABLE_EXTENSION_NAME
- XR_KHR_opengl_es_enable_SPEC_VERSION
- Extending XrStructureType:
  - XR_TYPE_GRAPHICS_BINDING_OPENGL_ES_ANDROID_KHR
  - XR_TYPE_GRAPHICS_REQUIREMENTS_OPENGL_ES_KHR
  - XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_ES_KHR

12.19.9. 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`
- Revision 9, 2025-03-07 (Rylie Pavlik, Collabora, Ltd.)
    - Re-organize, clarify, and make more uniform with other graphics binding extensions, and describe known design quirk.

12.20. 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

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0

Last Modified Date

2021-05-11

IP Status

No known IP claims.

Contributors

Jakob Bornecrantz, Collabora
Rylie 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 (Rylie Pavlik, Collabora, Ltd.)
  - Updated for promotion from MND to KHR

12.21. XR_KHR_visibility_mask

Name String

`XR_KHR_visibility_mask`

Extension Type

Instance extension

Registered Extension Number

32

Revision

2

Ratification Status

Ratified

Extension and Version Dependencies

[OpenXR 1.0](#)

Last Modified Date

2018-07-05

IP Status

No known IP claims.

Contributors

Paul Pedriana, Oculus
Alex Turner, Microsoft

Contacts

Paul Pedriana, Oculus

Overview

This extension support 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 are not visible to the user. A common occurrence in head-mounted VR hardware is that the optical system's frustum does not intersect precisely with the rectangular display it is viewing. As a result, it may be that there are parts of the display that are not 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.

```
// Provided by XR_KHR_visibility_mask
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.

```
// Provided by XR_KHR_visibility_mask
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.

- 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 is defined as:

```
// Provided by XR_KHR_visibility_mask
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.

The XrEventDataVisibilityMaskChangedKHR structure is queued to indicate that a given visibility 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 queued for the application, one per view.

New Functions

The xrGetVisibilityMaskKHR function is defined as:

```
// Provided by XR_KHR_visibility_mask
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 XrVisibilityMaskKHR ::`vertexCapacityInput` or XrVisibilityMaskKHR::`indexCapacityInput` is `0`, the runtime **must** respond as if both fields were set to `0`, returning the vertex count and index count through XrVisibilityMaskKHR ::`vertexCountOutput` or XrVisibilityMaskKHR::`indexCountOutput` respectively. If a view mask for the specified view isn't available, the returned vertex and index counts **must** be `0`.

Issues

Version History

- Revision 1, 2018-07-05 (Paul Pedriana)
  - Initial version.
- Revision 2, 2019-07-15 (Alex Turner)
  - Adjust two-call idiom usage.

12.22. XR_KHR_vulkan_enable

Name String

XR_KHR_vulkan_enable

Extension Type

Instance extension

Registered Extension Number

26

Revision

9

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0

Last Modified Date

2025-03-07

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

12.22.1. Overview

This extension enables the use of the Vulkan graphics API in an OpenXR application. Without this extension, an OpenXR application **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. However, the

runtime provides the Vulkan images to render into. This extension provides mechanisms for the application to interact with those images by calling 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_VULKAN before including the OpenXR platform header openxr_platform.h, in all portions of the library or application that interact with the types, values, and functions it defines.

12.22.2. 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 XrGraphicsBindingVulkanKHR must also be externally synchronized.

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.

12.22.3. 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 xrGetVulkanDeviceExtensionsKHR.

In order to satisfy the VkPhysicalDevice requirements, the application can query xrGetVulkanGraphicsDeviceKHR to identify the correct VkPhysicalDevice.

Populating an XrGraphicsBindingVulkanKHR with a VkInstance, VkDevice, or VkPhysicalDevice that does not meet the requirements outlined by this extension **may** result in undefined behavior by the OpenXR runtime.

The API version, instance extension, device extension and physical device requirements only apply to the VkInstance, VkDevice, and VkPhysicalDevice objects which the application wishes to associate with

an XrGraphicsBindingVulkanKHR.

The xrGetVulkanGraphicsRequirementsKHR function is defined as:

```
// Provided by XR_KHR_vulkan_enable
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

The XrGraphicsRequirementsVulkanKHR structure is defined as:

```
// Provided by XR_KHR_vulkan_enable
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.

Some computer systems 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:

```
// Provided by XR_KHR_vulkan_enable
XrResult xrGetVulkanGraphicsDeviceKHR(
    XrInstance                              instance,
    XrSystemId                              systemId,
    VkInstance                              vkInstance,
    VkPhysicalDevice*                       vkPhysicalDevice);
```

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.
- `vkInstance` is a valid Vulkan `VkInstance`.
- `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`

```
// Provided by XR_KHR_vulkan_enable
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 the Buffer Size Parameters section 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`

```
// Provided by XR_KHR_vulkan_enable
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 the Buffer Size Parameters section 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

12.22.4. Graphics Binding Structure

The XrGraphicsBindingVulkanKHR structure is defined as:

```
// Provided by XR_KHR_vulkan_enable
typedef struct XrGraphicsBindingVulkanKHR {
    XrStructureType    type;
    const void*        next;
    VkInstance         instance;
    VkPhysicalDevice   physicalDevice;
    VkDevice           device;
    uint32_t           queueFamilyIndex;
    uint32_t           queueIndex;
} XrGraphicsBindingVulkanKHR;
```

- `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.

To create a Vulkan-backed XrSession, the application provides a pointer to an XrGraphicsBindingVulkanKHR structure in the XrSessionCreateInfo::`next` chain when calling xrCreateSession.

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

Creating a session using this structure triggers several requirements on the runtime regarding swapchain images. See the specification of XrSwapchainImageVulkanKHR for details. The application **must** externally synchronize the queue referred to by this structure according to Concurrency.

12.22.5. Swapchain Images

The XrSwapchainImageVulkanKHR structure is defined as:

```
// Provided by XR_KHR_vulkan_enable
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 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.

The OpenXR runtime **must** return a texture created in accordance with Vulkan Swapchain Flag Bits.

The OpenXR runtime **must** manage image resource state in accordance with Vulkan Swapchain Image Layout.

> ### 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

12.22.6. Vulkan 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.

XrSwapchainUsageFlagBits	Corresponding Vulkan flag bit
`XR_SWAPCHAIN_USAGE_COLOR_ATTACHMENT_BIT`	`VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`
`XR_SWAPCHAIN_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`	`VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`
`XR_SWAPCHAIN_USAGE_UNORDERED_ACCESS_BIT`	`VK_IMAGE_USAGE_STORAGE_BIT`
`XR_SWAPCHAIN_USAGE_TRANSFER_SRC_BIT`	`VK_IMAGE_USAGE_TRANSFER_SRC_BIT`
`XR_SWAPCHAIN_USAGE_TRANSFER_DST_BIT`	`VK_IMAGE_USAGE_TRANSFER_DST_BIT`
`XR_SWAPCHAIN_USAGE_SAMPLED_BIT`	`VK_IMAGE_USAGE_SAMPLED_BIT`
`XR_SWAPCHAIN_USAGE_MUTABLE_FORMAT_BIT`	`VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT`
`XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_KHR` (Added by XR_KHR_swapchain_usage_input_attachment_bit and only available when that extension is enabled)	`VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`

12.22.7. Vulkan Swapchain Image Layout

If an application waits on a swapchain image by calling xrWaitSwapchainImage in a session created using XrGraphicsBindingVulkanKHR, and that call returns `XR_SUCCESS` or `XR_SESSION_LOSS_PENDING`, then the OpenXR runtime **must** guarantee that the following conditions are true, keeping in mind that the runtime **must** not access the `VkQueue` in xrWaitSwapchainImage:

- 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 XrGraphicsBindingVulkanKHR has ownership of the image.

When an application releases a swapchain image by calling xrReleaseSwapchainImage, in a session created using XrGraphicsBindingVulkanKHR, 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 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.

12.22.8. New Commands

- xrGetVulkanDeviceExtensionsKHR

- xrGetVulkanGraphicsDeviceKHR

- xrGetVulkanGraphicsRequirementsKHR

- xrGetVulkanInstanceExtensionsKHR

12.22.9. New Structures

- XrGraphicsRequirementsVulkanKHR

- XrSwapchainImageVulkanKHR

- Extending XrSessionCreateInfo:

  ◦ XrGraphicsBindingVulkanKHR

12.22.10. New Enum Constants

- `XR_KHR_VULKAN_ENABLE_EXTENSION_NAME`

- `XR_KHR_vulkan_enable_SPEC_VERSION`

- Extending XrStructureType:

  ◦ `XR_TYPE_GRAPHICS_BINDING_VULKAN_KHR`

  ◦ `XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN_KHR`

  ◦ `XR_TYPE_SWAPCHAIN_IMAGE_VULKAN_KHR`

12.22.11. 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

- Revision 8, 2021-01-21 (Rylie Pavlik, Collabora, Ltd.)

  - Document mapping for `XrSwapchainUsageFlags`

- Revision 9, 2025-03-07 (Rylie Pavlik, Collabora, Ltd.)

  - Re-organize, clarify, and make more uniform with other graphics binding extensions.

12.23. XR_KHR_vulkan_enable2

Name String

    `XR_KHR_vulkan_enable2`

Extension Type

    Instance extension

Registered Extension Number

    91

Revision

    3

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0

Last Modified Date

2025-03-07

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

12.23.1. Overview

This extension enables the use of the Vulkan graphics API in an OpenXR application. Without this extension, an OpenXR application **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, the application source code **must** define XR_USE_GRAPHICS_API_VULKAN before including the OpenXR platform header `openxr_platform.h`, in all portions of the library or application that interact with the types, values, and functions it defines.

> ℹ️ *Note*
>
> This extension is intended as an alternative to `XR_KHR_vulkan_enable`, and does not depend on it.

12.23.2. 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:

```
// Provided by XR_KHR_vulkan_enable2
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

The XrGraphicsRequirementsVulkan2KHR structure populated by xrGetVulkanGraphicsRequirements2KHR is defined as:

```
// Provided by XR_KHR_vulkan_enable2
// 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.

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`.

```
// Provided by XR_KHR_vulkan_enable2
XrResult xrCreateVulkanInstanceKHR(
    XrInstance                              instance,
    const XrVulkanInstanceCreateInfoKHR*    createInfo,
    VkInstance*                             vulkanInstance,
    VkResult*                               vulkanResult);
```

The XrVulkanInstanceCreateInfoKHR structure contains the input parameters to xrCreateVulkanInstanceKHR.

```
// Provided by XR_KHR_vulkan_enable2
typedef struct XrVulkanInstanceCreateInfoKHR {
    XrStructureType                 type;
    const void*                     next;
    XrSystemId                      systemId;
    XrVulkanInstanceCreateFlagsKHR  createFlags;
    PFN_vkGetInstanceProcAddr       pfnGetInstanceProcAddr;
    const VkInstanceCreateInfo*     vulkanCreateInfo;
    const VkAllocationCallbacks*    vulkanAllocator;
} XrVulkanInstanceCreateInfoKHR;
```

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
- `systemId` is an XrSystemId handle for the system which will be used to create a session.
- `createFlags` is a bitmask of XrVulkanInstanceCreateFlagBitsKHR
- `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

The XrVulkanInstanceCreateInfoKHR::`createFlags` member is of the following type, and contains a bitwise-OR of zero or more of the bits defined in XrVulkanInstanceCreateFlagBitsKHR.

```
typedef XrFlags64 XrVulkanInstanceCreateFlagsKHR;
```

Valid bits for XrVulkanInstanceCreateFlagsKHR are defined by XrVulkanInstanceCreateFlagBitsKHR.

```
// Flag bits for XrVulkanInstanceCreateFlagsKHR
```

There are currently no Vulkan instance creation flag bits defined. This is reserved for future use.

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 VkPhysicalDevice that is compatible with the OpenXR implementation when xrGetVulkanGraphicsDevice2KHR is invoked. The application will use this VkPhysicalDevice to interact with the OpenXR runtime.

```
// Provided by XR_KHR_vulkan_enable2
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.

The XrVulkanGraphicsDeviceGetInfoKHR structure contains the input parameters to xrCreateVulkanInstanceKHR.

```
// Provided by XR_KHR_vulkan_enable2
typedef struct XrVulkanGraphicsDeviceGetInfoKHR {
    XrStructureType    type;
    const void*        next;
    XrSystemId         systemId;
    VkInstance         vulkanInstance;
} XrVulkanGraphicsDeviceGetInfoKHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `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 XrVulkanInstanceCreateInfoKHR ::`pfnGetInstanceProcAddr`.

```
// Provided by XR_KHR_vulkan_enable2
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 XrVulkanInstanceCreateInfoKHR::pfnGetInstanceProcAddr.

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.

```
// Provided by XR_KHR_vulkan_enable2
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

- type is the XrStructureType of this structure.

- next is NULL or a pointer to the next structure in a structure chain.

- systemId is an XrSystemId handle for the system which will be used to create a session.

- createFlags is a bitmask of XrVulkanDeviceCreateFlagBitsKHR

- 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.

If the vulkanPhysicalDevice parameter does not match the output of xrGetVulkanGraphicsDeviceKHR, then the runtime **must** return XR_ERROR_HANDLE_INVALID.

- 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

XrVulkanDeviceCreateFlagsKHR specify details of device creation. The XrVulkanDeviceCreateInfoKHR::`createFlags` member is of the following type, and contains a bitwise-OR of zero or more of the bits defined in XrVulkanDeviceCreateFlagBitsKHR.

```
typedef XrFlags64 XrVulkanDeviceCreateFlagsKHR;
```

Valid bits for XrVulkanDeviceCreateFlagsKHR are defined by XrVulkanDeviceCreateFlagBitsKHR.

```
// Flag bits for XrVulkanDeviceCreateFlagsKHR
```

There are currently no Vulkan device creation flag bits defined. This is reserved for future use.

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.

```
// Provided by XR_KHR_vulkan_enable2
// 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 the following items replicate the implicit valid usage for 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

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.

12.23.3. 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.

12.23.4. 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:

```
// Provided by XR_KHR_vulkan_enable2
// 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 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 the following items replicate the implicit valid usage for 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

---

Swapchain Image Layout

If an application waits on a swapchain image by calling xrWaitSwapchainImage in a session created using XrGraphicsBindingVulkan2KHR, and that call returns `XR_SUCCESS` or `XR_SESSION_LOSS_PENDING`, then the OpenXR runtime **must** guarantee that the following conditions are true, keeping in mind that the runtime **must** not access the `VkQueue` in xrWaitSwapchainImage:

- 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.

XrSwapchainUsageFlagBits	Corresponding Vulkan flag bit
XR_SWAPCHAIN_USAGE_COLOR_ATTACHMENT_BIT	VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
XR_SWAPCHAIN_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT	VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
XR_SWAPCHAIN_USAGE_UNORDERED_ACCESS_BIT	VK_IMAGE_USAGE_STORAGE_BIT
XR_SWAPCHAIN_USAGE_TRANSFER_SRC_BIT	VK_IMAGE_USAGE_TRANSFER_SRC_BIT
XR_SWAPCHAIN_USAGE_TRANSFER_DST_BIT	VK_IMAGE_USAGE_TRANSFER_DST_BIT
XR_SWAPCHAIN_USAGE_SAMPLED_BIT	VK_IMAGE_USAGE_SAMPLED_BIT
XR_SWAPCHAIN_USAGE_MUTABLE_FORMAT_BIT	VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT
XR_SWAPCHAIN_USAGE_INPUT_ATTACHMENT_BIT_KHR (Added by XR_KHR_swapchain_usage_input_attachment_bit and only available when that extension is enabled)	VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT

12.23.5. 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 (Rylie Pavlik, Collabora, Ltd.)

    ◦ Document mapping for `XrSwapchainUsageFlags`

- Revision 3, 2025-03-07 (Rylie Pavlik, Collabora, Ltd.)

    ◦ Clarify and make more uniform with other graphics binding extensions.

12.24. XR_KHR_vulkan_swapchain_format_list

Name String

`XR_KHR_vulkan_swapchain_format_list`

Extension Type

Instance extension

Registered Extension Number

15

Revision

5

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0
and
`XR_KHR_vulkan_enable`
or
OpenXR 1.0
and
`XR_KHR_vulkan_enable2`

Last Modified Date

> 2024-11-13

IP Status

> No known IP claims.

Contributors

> Paul Pedriana, Oculus
>
> Dan Ginsburg, Valve
>
> Jakob Bornecrantz, NVIDIA

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 an application enables and uses XR_KHR_vulkan_enable2 as the graphics binding extension, 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:

```
XR_TYPE_VULKAN_SWAPCHAIN_FORMAT_LIST_CREATE_INFO_KHR
```

New Enums

New Structures

```
// Provided by XR_KHR_vulkan_swapchain_format_list
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 (Rylie Pavlik, Collabora, Ltd.)

- Fix reference to the mutable-format bit in Vulkan.
- Revision 5, 2024-11-13 (Jakob Bornecrantz, NVIDIA)
  - Fix correct Vulkan enable extension being referenced.
  - Clarify `XR_KHR_vulkan_enable2` being used by the app.

12.25. 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

Ratification Status

Ratified

Extension and Version Dependencies

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.

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

New Functions

To convert from a Windows performance counter time stamp to XrTime, call:

```
// Provided by XR_KHR_win32_convert_performance_counter_time
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.

To convert from `XrTime` to a Windows performance counter time stamp, call:

```
// Provided by XR_KHR_win32_convert_performance_counter_time
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

12.26. XR_EXT_conformance_automation

Name String

XR_EXT_conformance_automation

Extension Type

Instance extension

Registered Extension Number

48

Revision

3

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0

Last Modified Date

2021-04-14

IP Status

No known IP claims.

Contributors

Lachlan Ford, Microsoft
Rylie Pavlik, Collabora

Overview

The XR_EXT_conformance_automation allows conformance test and runtime developers to provide hints to the underlying runtime as to what input the test is expecting. This enables runtime authors to automate the testing of their runtime conformance. This is useful for achieving rapidly iterative runtime development whilst maintaining conformance for runtime releases.

This extension provides the following capabilities:

- The ability to toggle the active state of an input device.
- The ability to set the state of an input device button or other input component.
- The ability to set the location of the input device.

Applications **may** call these functions at any time. The runtime **must** do its best to honor the request of applications calling these functions, however it does not guarantee that any state change will be reflected immediately, at all, or with the exact value that was requested. Applications are thus advised to wait for the state change to be observable and to not assume that the value they requested will be the value observed. If any of the functions of this extension are called, control over input **must** be removed from the physical hardware of the system.

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

New Functions

```
// Provided by XR_EXT_conformance_automation
XrResult xrSetInputDeviceActiveEXT(
    XrSession                              session,
    XrPath                                 interactionProfile,
    XrPath                                 topLevelPath,
    XrBool32                               isActive);
```

Parameter Descriptions

- `session` is the XrSession to set the input device state in.

- `interactionProfile` is the path representing the interaction profile of the input device (e.g. */interaction_profiles/khr/simple_controller*).

- `topLevelPath` is the path representing the input device (e.g. */user/hand/left*).

- `isActive` is the requested activation state of the input device.

Valid Usage

- `session` **must** be a valid session handle.
- `topLevelPath` **must** be a valid top level path.

Valid Usage (Implicit)

- The `XR_EXT_conformance_automation` extension **must** be enabled prior to calling xrSetInputDeviceActiveEXT
- `session` **must** be a valid XrSession handle

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_PATH_UNSUPPORTED`
- `XR_ERROR_PATH_INVALID`

```
// Provided by XR_EXT_conformance_automation
XrResult xrSetInputDeviceStateBoolEXT(
    XrSession                               session,
    XrPath                                  topLevelPath,
    XrPath                                  inputSourcePath,
    XrBool32                                state);
```

Parameter Descriptions

- `session` is the XrSession to set the input device state in.
- `topLevelPath` is the path representing the input device (e.g. */user/hand/left*).
- `inputSourcePath` is the full path of the input component for which we wish to set the state for (e.g. */user/hand/left/input/select/click*).
- `state` is the requested boolean state of the input device.

Valid Usage

- `session` **must** be a valid session handle.
- `topLevelPath` **must** be a valid top level path.
- `inputSourcePath` **must** be a valid input source path.

Valid Usage (Implicit)

- The `XR_EXT_conformance_automation` extension **must** be enabled prior to calling xrSetInputDeviceStateBoolEXT
- `session` **must** be a valid XrSession handle

Return Codes

```
// Provided by XR_EXT_conformance_automation
XrResult xrSetInputDeviceStateFloatEXT(
    XrSession                           session,
    XrPath                              topLevelPath,
    XrPath                              inputSourcePath,
    float                               state);
```

Parameter Descriptions

- session is the XrSession to set the input device state in.

- topLevelPath is the path representing the input device (e.g. */user/hand/left*).

- inputSourcePath is the full path of the input component for which we wish to set the state for (e.g. */user/hand/left/input/trigger/value*).

- state is the requested float state of the input device.

Valid Usage

- `session` **must** be a valid session handle.

- `topLevelPath` **must** be a valid top level path.

- `inputSourcePath` **must** be a valid input source path.

Valid Usage (Implicit)

- The `XR_EXT_conformance_automation` extension **must** be enabled prior to calling xrSetInputDeviceStateFloatEXT

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

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_PATH_UNSUPPORTED`

- `XR_ERROR_PATH_INVALID`

```
// Provided by XR_EXT_conformance_automation
XrResult xrSetInputDeviceStateVector2fEXT(
    XrSession                           session,
    XrPath                              topLevelPath,
    XrPath                              inputSourcePath,
    XrVector2f                          state);
```

Parameter Descriptions

- `session` is the XrSession to set the input device state in.

- `topLevelPath` is the path representing the input device (e.g. */user/hand/left*).

- `inputSourcePath` is the full path of the input component for which we wish to set the state for (e.g. */user/hand/left/input/thumbstick*).

- `state` is the requested two-dimensional state of the input device.

Valid Usage

- `session` **must** be a valid session handle.

- `topLevelPath` **must** be a valid top level path.

- `inputSourcePath` **must** be a valid input source path.

Valid Usage (Implicit)

- The `XR_EXT_conformance_automation` extension **must** be enabled prior to calling xrSetInputDeviceStateVector2fEXT

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

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_PATH_UNSUPPORTED

- XR_ERROR_PATH_INVALID

```
// Provided by XR_EXT_conformance_automation
XrResult xrSetInputDeviceLocationEXT(
    XrSession                               session,
    XrPath                                  topLevelPath,
    XrPath                                  inputSourcePath,
    XrSpace                                 space,
    XrPosef                                 pose);
```

Parameter Descriptions

- session is the XrSession to set the input device state in.

- topLevelPath is the path representing the input device (e.g. */user/hand/left*).

- inputSourcePath is the full path of the input component for which we wish to set the pose for (e.g. */user/hand/left/input/grip/pose*).

- pose is the requested pose state of the input device.

New Function Pointers

Issues

None

Version History

- Revision 1, 2019-10-01 (Lachlan Ford)
  - Initial draft
- Revision 2, 2021-03-04 (Rylie Pavlik)
  - Correct errors in function parameter documentation.
- Revision 3, 2021-04-14 (Rylie Pavlik)
  - Fix missing error code

12.27. XR_EXT_dpad_binding

Name String

XR_EXT_dpad_binding

Extension Type

Instance extension

Registered Extension Number

79

Revision

1

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0
and
XR_KHR_binding_modification

Last Modified Date

2022-04-20

IP Status

No known IP claims.

Contributors

Joe Ludwig, Valve

Keith Bradner, Valve

Rune Berg, Valve

Nathan Nuber, Valve

Jakob Bornecrantz, Collabora

Rylie Pavlik, Collabora

Jules Blok, Epic Games

Overview

This extension allows the application to bind one or more digital actions to a trackpad or thumbstick as though it were a dpad by defining additional component paths to suggest bindings for. The behavior of this dpad-like mapping may be customized using XrInteractionProfileDpadBindingEXT.

Applications **must** also enable the `XR_KHR_binding_modification` extension that this builds on top of.

New Component Paths

When this extension is enabled, a runtime **must** accept otherwise-valid suggested bindings that refer to the following component paths added to certain existing input subpaths.

- For a given interaction profile,
  - For each input subpath valid in that interaction profile that has identifier *trackpad* but without a component specified (i.e. *.../input/trackpad* or *.../input/trackpad_*<location>), a runtime **must** accept the following components appended to that path in a suggested binding:
    - *.../dpad_up*
    - *.../dpad_down*
    - *.../dpad_left*
    - *.../dpad_right*
    - *.../dpad_center*
  - For each input subpath valid in that interaction profile that has identifier *thumbstick* but without a component specified (i.e. *.../input/thumbstick* or *.../input/thumbstick_*<location>), a runtime **must** accept the following components appended to that path in a suggested binding:
    - *.../dpad_up*
    - *.../dpad_down*
    - *.../dpad_left*
    - *.../dpad_right*

While a runtime **may** ignore accepted suggested bindings, and **may** use their contents as suggestions for automatic remapping when not obeying them, this extension defines interpretations the runtime **must** make in the case that a suggested binding using one of these paths is being obeyed.

An application **can** pass XrInteractionProfileDpadBindingEXT in the XrBindingModificationsKHR

`::bindingModifications` array associated with a suggested binding to customize the behavior of this mapping in the case that suggested bindings are being obeyed, and to provide remapping hints in other cases. If no XrInteractionProfileDpadBindingEXT structure is present in XrBindingModificationsKHR::`bindingModifications` for a given action set and component-less input subpath, the runtime **must** behave as if one were passed with the following values:

- `forceThreshold` = 0.5
- `forceThresholdReleased` = 0.4
- `centerRegion` = 0.5
- `wedgeAngle` = ½ π
- `isSticky` = `XR_FALSE`
- `onHaptic` = `NULL`
- `offHaptic` = `NULL`

For the purposes of description, the (-1, 1) ranges of the x and y components of trackpad and thumbstick inputs are depicted in this extension as if their scale were equal between axes. However, this is **not** required by this extension: while their numeric scale is treated as equal, their physical scale **may** not be.

Each of the component paths defined by this extension behave as boolean inputs. The center component *…/dpad_center* (only present when the path identifier is *trackpad*) **must** not be active at the same time as any other dpad component. For the other components, zero, one, or (depending on the `wedgeAngle`) two of them **may** be active at any time, though only adjacent components on a single logical dpad may be active simultaneously. For example, *…/dpad_down* and *…/dpad_left* are adjacent, and thus **may** be active simultaneously, while *…/dpad_up* and *…/dpad_down* are not adjacent and **must** not be active simultaneously.

> *Note*
>
> If `wedgeAngle` > ½ π, it is possible for two components referring to adjacent directions (excluding *…/dpad_center*) to be active at the same time, as the directional regions overlap. If `wedgeAngle` < ½ π, there are wedges between directional regions that correspond to no dpad component.

The following components are defined by possibly-overlapping truncated wedges pointing away from 0, 0 in x, y input space, with their angular size of XrInteractionProfileDpadBindingEXT::`wedgeAngle` centered around the indicated direction.

- *…/dpad_up*: direction (0, 1)
- *…/dpad_down*: direction (0, -1)
- *…/dpad_left*: direction (-1, 0)
- *…/dpad_right*: direction (1, 0)

Typical values for `wedgeAngle` are ½ π (or 90°) for regions that do not overlap or ¾ π (or 135°) for regions are evenly divided between the exclusive region for one cardinal direction and the overlap with neighboring regions.

Each of these regions are truncated by an arc to exclude the area within a radius of XrInteractionProfileDpadBindingEXT::`centerRegion` away from 0, 0. When used with an input path with an identifier of *trackpad*, the area within this radius corresponds to the *…/dpad_center* component. When used with an input path with an identifier of *thumbstick*, the area within this radius is a region where all dpad components **must** be inactive.
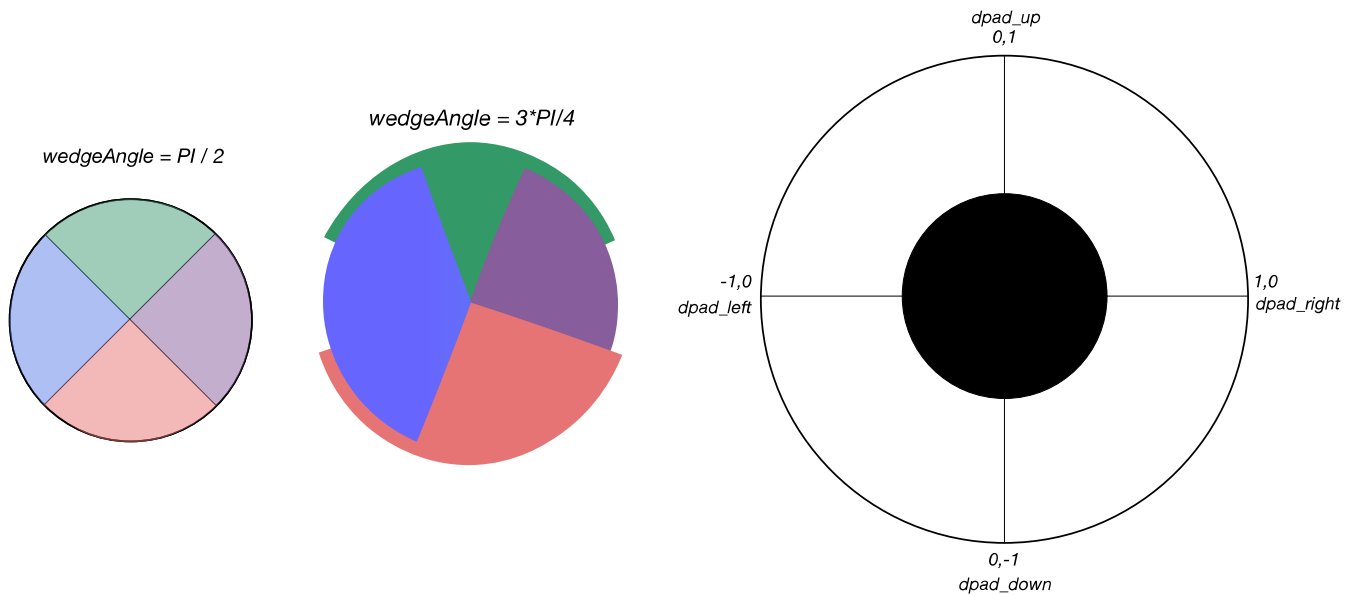
Figure 8. Wedge Angles

Behavior

For both the *trackpad* and *thumbstick* input identifiers, there are conditions that **must** be true for any dpad component to report active. If these conditions are true, the selection of which component or components are active, if any, takes place.

- Activation of a dpad component when appended to an input path with identifier *trackpad* on the values of the *…/x* and *…/y* components, as well as on an overall activation state. If the overall state is inactive, the runtime **must** treat all corresponding dpad components as inactive.

  ○ If the component *…/click* is also valid for the trackpad, the overall activation state is equal to the value of the *…/click*.

  ○ If the component *…/click* is **not** valid for the trackpad, but the component *…/force* **is** valid, the overall activation state depends on the value of that *…/force* component, as well as the previous overall activation state for hysteresis. The *…/force* component value hysteresis thresholds for overall activation are XrInteractionProfileDpadBindingEXT::`forceThreshold` and `forceThresholdReleased`. More explicitly:

    ▪ If the previous overall state was inactive, the current overall state **must** be active if and only if the value of the *…/force* component is greater than or equal to `forceThreshold`.

- If the previous overall state was active, the current state **must** be inactive if and only if the value of the .../*force* component is strictly less than `forceThresholdReleased`.

- Activation of a dpad component when appended to an input path with identifier *thumbstick* depends only on the value of the .../*x* and .../*y* components of that input.

  ◦ If the thumbstick *x* and *y* values correspond to a deflection from center of less than `centerRegion`, all dpad components **must** be reported as inactive.

Hysteresis is desirable to avoid an unintentional, rapid toggling between the active and inactive state that can occur when the amount of force applied by the user is very close to the threshold at which the input is considered active. Hysteresis is optional, and is achieved through a difference between `forceThreshold` and `forceThresholdReleased`.

When XrInteractionProfileDpadBindingEXT::`isSticky` is `XR_FALSE`, and the above logic indicates that some dpad component is active, a runtime obeying suggested bindings **must** select which dpad components to report as active based solely on the current *x*, *y* values.

If XrInteractionProfileDpadBindingEXT::`isSticky` is `XR_TRUE`, the region(s) to be made active **must** be latched when the above logic begins to indicate that some dpad component is active, and the *x* and *y* values are within at least one region. The latched region(s) **must** continue to be reported as active until the activation logic indicates that all dpad components **must** be inactive. The latched region(s) remain active even if the input leaves that region or enters another region.

The runtime **must** latch the *x* and *y* values, and thus the region or regions (in the case of overlapping dpad component wedges), when the sticky activation toggle becomes true. The latched regions **must** continue to be true until the input returns to the center region (for a thumbstick) or is released (for a trackpad). In this way, sticky dpads maintain their selected region across touch/click transitions.

Examples for isSticky == `XR_TRUE`

- Trackpad example: If the user clicks a trackpad in the .../*dpad_up* region, then (while clicked) slides their finger to the .../*dpad_down* region, .../*dpad_up* will remain true.

- Thumbstick example: If the user presses up on the thumbstick and activates the .../*dpad_up* region, then slides the thumbstick around to the .../*dpad_down* region without crossing the `centerRegion`, .../*dpad_up* is the virtual input that will be true.

- Thumbstick example: If the user presses up on the thumbstick and activates the .../*dpad_up* region, then slides the thumbstick directly down and through the region specified by `centerRegion` to .../*dpad_down.* Initially .../*dpad_up* will activate. Then when the thumbstick enters the `centerRegion` it will deactivate. Finally, when entering the .../*dpad_down* region .../*dpad_down* will activate.

New Structures

The XrInteractionProfileDpadBindingEXT structure is defined as:

```
// Provided by XR_EXT_dpad_binding
typedef struct XrInteractionProfileDpadBindingEXT {
    XrStructureType                type;
    const void*                    next;
    XrPath                         binding;
    XrActionSet                    actionSet;
    float                          forceThreshold;
    float                          forceThresholdReleased;
    float                          centerRegion;
    float                          wedgeAngle;
    XrBool32                       isSticky;
    const XrHapticBaseHeader*      onHaptic;
    const XrHapticBaseHeader*      offHaptic;
} XrInteractionProfileDpadBindingEXT;
```

- `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.

- `binding` is the input path used for the specified actions in the suggested binding list to be used as a dpad. E.g. path:/user/hand/right/input/thumbstick

- `actionSet` is the action set for which this dpad will be active. The implementation **must** use the parameters from this structure for any actions from this action set that are bound to one of the dpad subpaths for this input path.

- `forceThreshold` a number in the half-open range (0, 1] representing the force value threshold at or above which (≥) a dpad input will transition from inactive to active.

- `forceThresholdReleased` a number in the half-open range (0, 1] representing the force value threshold strictly below which (<) a dpad input will transition from active to inactive.

- `centerRegion` defines the center region of the thumbstick or trackpad. This is the radius, in the input value space, of a logically circular region in the center of the input, in the range (0, 1).

- `wedgeAngle` indicates the angle in radians of each direction region and is a value in the half-open range [0, π).

- `isSticky` indicates that the implementation will latch the first region that is activated and continue to indicate that the binding for that region is true until the user releases the input underlying the virtual dpad.

- `onHaptic` is the haptic output that the runtime **must** trigger when the binding changes from false to true. If this field is `NULL`, the runtime **must** not trigger any haptic output on the threshold. This field **can** point to any supported sub-type of XrHapticBaseHeader.

- `offHaptic` is the haptic output that the runtime **must** trigger when the binding changes from true to false. If this field is NULL, the runtime **must** not trigger any haptic output on the threshold. This field **can** point to any supported sub-type of XrHapticBaseHeader.

The XrInteractionProfileDpadBindingEXT structure is an input struct that defines how to use any two-axis input to provide dpad-like functionality to the application. The struct **must** be added for each input that should be treated as a dpad to the XrBindingModificationsKHR::`bindingModifications` array in the XrBindingModificationsKHR structure (See `XR_KHR_binding_modification` extension).

Runtimes are free to ignore any of the fields when not obeying the bindings, but **may** use it for automatic rebindings of actions.

The implementation **must** return `XR_ERROR_VALIDATION_FAILURE` from xrSuggestInteractionProfileBindings if any of the following are true:

- `forceThreshold` or `forceThresholdReleased` are outside the half-open range (0, 1]

- `forceThreshold` < `forceThresholdReleased`

- `centerRegion` is outside the exclusive range (0, 1)

- `wedgeAngle` outside the half-open range [0, π)

If more than one XrInteractionProfileDpadBindingEXT is provided for the same input identifier, including top level path (e.g. */user/hand/left/input/thumbstick*), and two or more of them specify the same actionset, the runtime **must** return `XR_ERROR_VALIDATION_FAILURE`. If the same input identifier, including top level path, is used for more than one action set, in addition to inputs being suppressed by higher priority action sets, haptic events from dpads are also suppressed.

For example, a Valve Index controller binding with a "Walking" action set can have a dpad on each of:

- left thumbstick

- right thumbstick

- left trackpad

- right trackpad

Another action set can also have a dpad active on each of those inputs, and they can have different settings. If both action sets are active, the higher priority one trumps the lower priority one, and the lower priority one is suppressed.

> **Valid Usage (Implicit)**
>
> - The `XR_EXT_dpad_binding` extension **must** be enabled prior to using XrInteractionProfileDpadBindingEXT
>
> - `type` **must** be `XR_TYPE_INTERACTION_PROFILE_DPAD_BINDING_EXT`
>
> - `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
>
> - `actionSet` **must** be a valid XrActionSet handle
>
> - If `onHaptic` is not `NULL`, `onHaptic` **must** be a pointer to a valid XrHapticBaseHeader-based structure. See also: XrHapticVibration
>
> - If `offHaptic` is not `NULL`, `offHaptic` **must** be a pointer to a valid XrHapticBaseHeader-based structure. See also: XrHapticVibration

New Functions

Issues

- What if an interaction profile is added that contains a *trackpad* identifier, for which there is neither a *.../click* or a *.../force* component?

  - Equivalent logic would apply to whatever component is available to distinguish action from inaction.

- Is zero a valid wedge angle? Is π?

  - Yes, though it is mostly useless, as it makes the directional regions empty in size and thus impossible to activate. The user could only activate *.../dpad_center* on a *trackpad* identifier. π is not a valid wedge angle because that would imply being able to activate three adjacent directions, of which two must be opposite. In practice, the sensors underlying these inputs make it effectively impossible to input an exact floating point value.

Example

The following sample code shows how to create dpad bindings using this extension.

```
1    // Create dpad paths
2    XrPath pathThumbstick, pathDpadUp, pathDpadDown;
3    xrStringToPath( pInstance, "/user/hand/left/input/thumbstick", &pathThumbstick);
4    xrStringToPath( pInstance, "/user/hand/left/input/thumbstick/dpad_up", &pathDpadUp
  );
5    xrStringToPath( pInstance, "/user/hand/left/input/thumbstick/dpad_down",
  &pathDpadDown );
6
7    // Set dpad binding modifiers
8    XrInteractionProfileDpadBindingEXT xrDpadModification {
  XR_TYPE_INTERACTION_PROFILE_DPAD_BINDING_EXT };
9    xrDpadModification.actionSet = xrActionSet_Main;
10    xrDpadModification.binding = pathThumbstick;
11    xrDpadModification.centerRegion = 0.25f;
12    xrDpadModification.wedgeAngle = 2.0f;
13    // A gap between these next two members creates hysteresis, to avoid rapid
  toggling
14    xrDpadModification.forceThreshold = 0.8f;
15    xrDpadModification.forceThresholdReleased = 0.2f;
16
17    // Add dpad binding modifiers to binding modifications vector
18    std::vector< XrInteractionProfileDpadBindingEXT > vBindingModifs;
19    vBindingModifs.push_back( xrDpadModification );
20
21    std::vector< XrBindingModificationBaseHeaderKHR* > vBindingModifsBase;
22    for ( XrInteractionProfileDpadBindingEXT &modif : vBindingModifs )
23    {
24        vBindingModifsBase.push_back( reinterpret_cast<
  XrBindingModificationBaseHeaderKHR* >( &modif) );
25    }
26
27    XrBindingModificationsKHR xrBindingModifications {
  XR_TYPE_BINDING_MODIFICATIONS_KHR };
28    xrBindingModifications.bindingModifications = vBindingModifsBase.data();
29    xrBindingModifications.bindingModificationCount = ( uint32_t )vBindingModifsBase
```

```
    .size();
30
31      // Set dpad input path as suggested binding for an action
32      XrActionSuggestedBinding xrActionBindingTeleport, xrActionBindingMenu;
33
34      xrActionBindingTeleport.action = xrAction_Teleport;
35      xrActionBindingTeleport.binding = pathDpadUp;
36
37      xrActionBindingMenu.action = xrAction_Menu;
38      xrActionBindingMenu.binding = pathDpadDown;
39
40      std::vector< XrActionSuggestedBinding > vActionBindings;
41      vActionBindings.push_back( xrActionBindingTeleport );
42      vActionBindings.push_back( xrActionBindingMenu );
43
44      // Create interaction profile/controller path
45      XrPath xrInteractionProfilePath;
46      xrStringToPath( pInstance, "/interaction_profiles/valve/index_controller",
&xrInteractionProfilePath );
47
48      // Set suggested binding to interaction profile
49      XrInteractionProfileSuggestedBinding xrInteractionProfileSuggestedBinding {
XR_TYPE_INTERACTION_PROFILE_SUGGESTED_BINDING };
50      xrInteractionProfileSuggestedBinding.interactionProfile =
xrInteractionProfilePath;
51      xrInteractionProfileSuggestedBinding.suggestedBindings = vActionBindings.data();
52      xrInteractionProfileSuggestedBinding.countSuggestedBindings = ( uint32_t
)vActionBindings.size();
53
54      // Set binding modifications to interaction profile's suggested binding
55      xrInteractionProfileSuggestedBinding.next = &xrBindingModifications;
56
57      // Finally, suggest interaction profile bindings to runtime
58      xrSuggestInteractionProfileBindings( pInstance,
&xrInteractionProfileSuggestedBinding );
```

Version History

- Revision 1, 2022-02-18 (Rune Berg)

    ◦ Initial extension description

12.28. XR_EXT_eye_gaze_interaction

Name String

```
XR_EXT_eye_gaze_interaction
```

Extension Type

Instance extension

Registered Extension Number

31

Revision

2

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0

Last Modified Date

2020-02-20

IP Status

No known IP claims.

Contributors

Denny Rönngren, Tobii
Yin Li, Microsoft
Alex Turner, Microsoft
Paul Pedriana, Oculus
Rémi Arnaud, Varjo
Blake Taylor, Magic Leap
Lachlan Ford, Microsoft
Cass Everitt, Oculus

Overview

This extension provides an XrPath for getting eye gaze input from an eye tracker to enable eye gaze interactions.

The intended use for this extension is to provide:

- system properties to inform if eye gaze interaction is supported by the current device.

- an XrPath for real time eye tracking that exposes an accurate and precise eye gaze pose to be used to enable eye gaze interactions.

- a structure XrEyeGazeSampleTimeEXT that allows for an application to retrieve more information regarding the eye tracking samples.

With these building blocks, an application can discover if the XR runtime has access to an eye tracker,

bind the eye gaze pose to the action system, determine if the eye tracker is actively tracking the users eye gaze, and use the eye gaze pose as an input signal to build eye gaze interactions.

12.28.1. Eye tracker

An eye tracker is a sensory device that tracks eyes and accurately maps what the user is looking at. The main purpose of this extension is to provide accurate and precise eye gaze for the application.

Eye tracking data can be sensitive personal information and is closely linked to personal privacy and integrity. It is strongly recommended that applications that store or transfer eye tracking data always ask the user for active and specific acceptance to do so.

If a runtime supports a permission system to control application access to the eye tracker, then the runtime **must** set the `isActive` field to `XR_FALSE` on the supplied XrActionStatePose structure, and **must** clear `XR_SPACE_LOCATION_POSITION_TRACKED_BIT`, `XR_SPACE_LOCATION_POSITION_VALID_BIT`, `XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT` and `XR_SPACE_LOCATION_ORIENTATION_VALID_BIT` when locating using the tracked space until the application has been allowed access to the eye tracker. When the application access has been allowed, the runtime **may** set `isActive` on the supplied XrActionStatePose structure to `XR_TRUE` and **may** set `XR_SPACE_LOCATION_POSITION_TRACKED_BIT`, `XR_SPACE_LOCATION_POSITION_VALID_BIT` `XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT` and `XR_SPACE_LOCATION_ORIENTATION_VALID_BIT` when locating using the tracked space.

12.28.2. Device enumeration

When the eye gaze input extension is enabled an application **may** pass in a XrSystemEyeGazeInteractionPropertiesEXT structure in next chain structure when calling xrGetSystemProperties to acquire information about the connected eye tracker.

The runtime **must** populate the XrSystemEyeGazeInteractionPropertiesEXT structure with the relevant information to the XrSystemProperties returned by the xrGetSystemProperties call.

```
// Provided by XR_EXT_eye_gaze_interaction
typedef struct XrSystemEyeGazeInteractionPropertiesEXT {
    XrStructureType    type;
    void*              next;
    XrBool32           supportsEyeGazeInteraction;
} XrSystemEyeGazeInteractionPropertiesEXT;
```

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.

- `supportsEyeGazeInteraction` the runtime **must** set this value to `XR_TRUE` when eye gaze sufficient for use cases such as aiming or targeting is supported by the current device, otherwise the runtime **must** set this to `XR_FALSE`.

Valid Usage (Implicit)

- The `XR_EXT_eye_gaze_interaction` extension **must** be enabled prior to using XrSystemEyeGazeInteractionPropertiesEXT

- `type` **must** be `XR_TYPE_SYSTEM_EYE_GAZE_INTERACTION_PROPERTIES_EXT`

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

12.28.3. Eye gaze input

This extension exposes a new interaction profile path */interaction_profiles/ext/eye_gaze_interaction* that is valid for the user path

- */user/eyes_ext*

with supported input subpath

- *…/input/gaze_ext/pose*

Note

The interaction profile path */interaction_profiles/ext/eye_gaze_interaction* defined here does not follow current rules for interaction profile names. If this extension were introduced today, it would be called */interaction_profiles/ext/eye_gaze_interaction_ext*, to allow for modifications when promoted to a KHR extension or the core specification.

The eye gaze pose is natively oriented with +Y up, +X to the right, and -Z forward and not gravity-aligned, similar to the `XR_REFERENCE_SPACE_TYPE_VIEW`. The eye gaze pose may originate from a point positioned between the user's eyes. At any point of time both the position and direction of the eye pose is tracked or untracked. This means that the runtime **must** set both `XR_SPACE_LOCATION_POSITION_TRACKED_BIT` and `XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT` or clear both `XR_SPACE_LOCATION_POSITION_TRACKED_BIT` and `XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT`.

One particularity for eye trackers compared to most other spatial input is that the runtime may not have the capability to predict or interpolate eye gaze poses. Runtimes that cannot predict or interpolate eye gaze poses **must** clamp the gaze pose requested in the xrLocateSpace call to the value nearest to `time` requested in the call. To allow for an application to reason about high accuracy eye tracking, the application **can** chain in an XrEyeGazeSampleTimeEXT to the next pointer of the XrSpaceLocation structure passed into the xrLocateSpace call. The runtime **must** set `time` in the XrEyeGazeSampleTimeEXT structure to the clamped, predicted or interpolated time. The application **should** inspect the `time` field to understand when in time the pose is expressed. The `time` field **may** be in the future if a runtime can predict gaze poses. The runtime **must** set the `time` field to 0 if the sample time is not available.

When the runtime provides a nominal eye gaze pose, the `XR_SPACE_LOCATION_POSITION_TRACKED_BIT` **must** be set if the eye otherwise has a fully-tracked pose relative to the other space. A runtime **can** provide a sub-nominal eye-gaze pose but **must** then clear the `XR_SPACE_LOCATION_POSITION_TRACKED_BIT`. An application can expect that a nominal eye gaze pose can be used for use cases such as aiming or targeting, while a sub-nominal eye gaze pose has degraded performance and should not be relied on for all input scenarios. Applications should be very careful when using sub-nominal eye gaze pose, since the behavior can vary considerably for different users and manufacturers, and some manufacturers **may** not provide sub-nominal eye gaze pose at all.

With current technology, some eye trackers **may** need to undergo an explicit calibration routine to provide a nominal accurate and precise eye gaze pose. If the eye tracker is in an uncalibrated state when the first call to xrSyncActions is made with an eye gaze action enabled, then the runtime **should** request eye tracker calibration from the user if it has not yet been requested.

```
// Provided by XR_EXT_eye_gaze_interaction
typedef struct XrEyeGazeSampleTimeEXT {
    XrStructureType    type;
    void*              next;
    XrTime             time;
} XrEyeGazeSampleTimeEXT;
```

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.

- `time` is when in time the eye gaze pose is expressed.

12.28.4. Sample code

The following example code shows how to bind the eye pose to the action system.

```
extern XrInstance instance;
extern XrSession session;
extern XrPosef pose_identity;

// Create action set
XrActionSetCreateInfo actionSetInfo{XR_TYPE_ACTION_SET_CREATE_INFO};
strcpy(actionSetInfo.actionSetName, "gameplay");
strcpy(actionSetInfo.localizedActionSetName, "Gameplay");
actionSetInfo.priority = 0;
XrActionSet gameplayActionSet;
CHK_XR(xrCreateActionSet(instance, &actionSetInfo, &gameplayActionSet));

// Create user intent action
XrActionCreateInfo actionInfo{XR_TYPE_ACTION_CREATE_INFO};
strcpy(actionInfo.actionName, "user_intent");
actionInfo.actionType = XR_ACTION_TYPE_POSE_INPUT;
strcpy(actionInfo.localizedActionName, "User Intent");
XrAction userIntentAction;
CHK_XR(xrCreateAction(gameplayActionSet, &actionInfo, &userIntentAction));

// Create suggested bindings
XrPath eyeGazeInteractionProfilePath;
CHK_XR(xrStringToPath(instance, "/interaction_profiles/ext/eye_gaze_interaction",
&eyeGazeInteractionProfilePath));

XrPath gazePosePath;
CHK_XR(xrStringToPath(instance, "/user/eyes_ext/input/gaze_ext/pose", &gazePosePath));

XrActionSuggestedBinding bindings;
bindings.action = userIntentAction;
bindings.binding = gazePosePath;

XrInteractionProfileSuggestedBinding suggestedBindings
```

```
{XR_TYPE_INTERACTION_PROFILE_SUGGESTED_BINDING};
suggestedBindings.interactionProfile = eyeGazeInteractionProfilePath;
suggestedBindings.suggestedBindings = &bindings;
suggestedBindings.countSuggestedBindings = 1;
CHK_XR(xrSuggestInteractionProfileBindings(instance, &suggestedBindings));

XrSessionActionSetsAttachInfo attachInfo{XR_TYPE_SESSION_ACTION_SETS_ATTACH_INFO};
attachInfo.countActionSets = 1;
attachInfo.actionSets = &gameplayActionSet;
CHK_XR(xrAttachSessionActionSets(session, &attachInfo));

XrActionSpaceCreateInfo createActionSpaceInfo{XR_TYPE_ACTION_SPACE_CREATE_INFO};
createActionSpaceInfo.action = userIntentAction;
createActionSpaceInfo.poseInActionSpace = pose_identity;
XrSpace gazeActionSpace;
CHK_XR(xrCreateActionSpace(session, &createActionSpaceInfo, &gazeActionSpace));

XrReferenceSpaceCreateInfo createReferenceSpaceInfo{XR_TYPE_REFERENCE_SPACE_CREATE_INFO};
createReferenceSpaceInfo.referenceSpaceType = XR_REFERENCE_SPACE_TYPE_LOCAL;
createReferenceSpaceInfo.poseInReferenceSpace = pose_identity;
XrSpace localReferenceSpace;
CHK_XR(xrCreateReferenceSpace(session, &createReferenceSpaceInfo, &localReferenceSpace));

while(true)
{
  XrActiveActionSet activeActionSet{gameplayActionSet, XR_NULL_PATH};
  XrTime time;

  XrActionsSyncInfo syncInfo{XR_TYPE_ACTIONS_SYNC_INFO};
  syncInfo.countActiveActionSets = 1;
  syncInfo.activeActionSets = &activeActionSet;
  CHK_XR(xrSyncActions(session, &syncInfo));

  XrActionStatePose actionStatePose{XR_TYPE_ACTION_STATE_POSE};
  XrActionStateGetInfo getActionStateInfo{XR_TYPE_ACTION_STATE_GET_INFO};
  getActionStateInfo.action = userIntentAction;
  CHK_XR(xrGetActionStatePose(session, &getActionStateInfo, &actionStatePose));

  if(actionStatePose.isActive){
    XrEyeGazeSampleTimeEXT eyeGazeSampleTime{XR_TYPE_EYE_GAZE_SAMPLE_TIME_EXT};
    XrSpaceLocation gazeLocation{XR_TYPE_SPACE_LOCATION, &eyeGazeSampleTime};
    CHK_XR(xrLocateSpace(gazeActionSpace, localReferenceSpace, time, &gazeLocation));

    // Do things
  }
}
```

Version History

- Revision 1, 2020-02-20 (Denny Rönngren)
    - Initial version
- Revision 2, 2022-05-27 (Bryce Hutchings)
    - Remove error-prone `XrEyeGazeSampleTimeEXT` validation requirement

12.29. XR_EXT_future

Name String

   `XR_EXT_future`

Extension Type

   Instance extension

Registered Extension Number

   470

Revision

   1

Ratification Status

   Ratified

Extension and Version Dependencies

   OpenXR 1.0

Contributors

   Bryce Hutchings, Microsoft
   Andreas Selvik, Meta
   Ron Bessems, Magic Leap
   Yin Li, Microsoft Corporation
   Baolin Fu, ByteDance
   Cass Everitt, Meta Platforms
   Charlton Rodda, Collabora
   Jakob Bornecrantz, NVIDIA
   John Kearney, Meta Platforms
   Jonathan Wright, Meta Platforms
   Jun Yan, ByteDance
   Junyi Wang, ByteDance
   Karthik Kadappan, Magic Leap
   Natalie Fleury, Meta Platforms
   Nathan Nuber, Valve

Nikita Lutsenko, Meta Platforms

Robert Blenkinsopp, Ultraleap

Rylie Pavlik, Collabora

Tim Mowrer, Meta Platforms

Wenlin Mao, Meta Platforms

Will Fu, ByteDance

Zhipeng Liu, ByteDance

12.29.1. Overview

In XR systems there are certain operations that are long running and do not reasonably complete within a normal frame loop. This extension introduces the concept of a *future* which supports creation of asynchronous (async) functions for such long running operations. This extension does not include any asynchronous operations: it is expected that other extensions will use these *futures* and their associated conventions in this extension to define their asynchronous operations.

An XrFutureEXT represents the future result of an asynchronous operation, comprising an XrResult and possibly additional outputs. Long running operations immediately return an XrFutureEXT when started, letting the application poll the state of the future, and get the result once ready by calling a "complete"-function.

12.29.2. Getting a future

The XrFutureEXT basetype is defined as:

```
// Provided by XR_EXT_future
XR_DEFINE_OPAQUE_64(XrFutureEXT)
```

Asynchronous functions return an XrFutureEXT token as a placeholder for a value that will be returned later. An XrFutureEXT returned by a successful call to a function starting an asynchronous operation **should** normally start in the XR_FUTURE_STATE_PENDING_EXT state, but **may** skip directly to XR_FUTURE_STATE_READY_EXT if the result is immediately available.

The value XR_NULL_FUTURE_EXT, numerically equal to 0, is never a valid XrFutureEXT value.

Note that an XrFutureEXT token is neither a handle nor an atom type (such as XrPath). It belongs to a new category and is defined as an opaque 64-bit value. See Future Scope for details on the scope and lifecycle of a future.

Style note: Functions that return an XrFutureEXT **should** be named with the suffix "Async", e.g. xrPerformLongTaskAsync. This function **must** not set the XrFutureEXT to XR_NULL_FUTURE_EXT when the function returns XR_SUCCESS.

12.29.3. Waiting for a future to become ready

The xrPollFutureEXT function is defined as:

```
// Provided by XR_EXT_future
XrResult xrPollFutureEXT(
    XrInstance                                  instance,
    const XrFuturePollInfoEXT*                  pollInfo,
    XrFuturePollResultEXT*                      pollResult);
```

Parameter Descriptions

- `instance` is an XrInstance handle

- `pollInfo` is a pointer to an XrFuturePollInfoEXT structure.

- `pollResult` is a pointer to an XrFuturePollResultEXT structure to be populated on a successful call.

Applications **can** use this function to check the current state of a future, typically while waiting for the async operation to complete and the future to become "ready" to complete.

> *Note*
>
> Each XrFutureEXT value **must** be externally synchronized by the application when calling completion, polling, and cancellation functions, and when destroying the associated handle.

Valid Usage (Implicit)

- The `XR_EXT_future` extension **must** be enabled prior to calling xrPollFutureEXT

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

- `pollInfo` **must** be a pointer to a valid XrFuturePollInfoEXT structure

- `pollResult` **must** be a pointer to an XrFuturePollResultEXT structure

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_FUTURE_INVALID_EXT

The XrFuturePollInfoEXT structure is defined as:

```
// Provided by XR_EXT_future
typedef struct XrFuturePollInfoEXT {
    XrStructureType    type;
    const void*        next;
    XrFutureEXT        future;
} XrFuturePollInfoEXT;
```

Member Descriptions

- type is the XrStructureType of this structure.

- next is NULL or a pointer to the next structure in a structure chain.

- future is the XrFutureEXT future being polled.

An XrFuturePollInfoEXT structure is used to pass future to xrPollFutureEXT.

Valid Usage (Implicit)

- The XR_EXT_future extension **must** be enabled prior to using XrFuturePollInfoEXT

- type **must** be XR_TYPE_FUTURE_POLL_INFO_EXT

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

The XrFuturePollResultEXT structure is defined as:

```
// Provided by XR_EXT_future
typedef struct XrFuturePollResultEXT {
    XrStructureType     type;
    void*               next;
    XrFutureStateEXT    state;
} XrFuturePollResultEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain.
- `state` is the XrFutureStateEXT of the XrFutureEXT passed to xrPollFutureEXT.

An XrFuturePollResultEXT structure is used to return the result of xrPollFutureEXT.

Valid Usage (Implicit)

- The `XR_EXT_future` extension **must** be enabled prior to using XrFuturePollResultEXT
- `type` **must** be `XR_TYPE_FUTURE_POLL_RESULT_EXT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain

12.29.4. Completing a Future

Extensions that provide async functions returning a future **should** also provide a matching completion function to "complete" the future in order to return the result of the asynchronous operation. This function **should** be named with the suffix "Complete" replacing the "Async" suffix, e.g. `xrPerformLongTaskComplete` is a suitable completion function name corresponding to `xrPerformLongTaskAsync`.

A completion function **must** populate a structure that **must** be based on XrFutureCompletionBaseHeaderEXT to return the result of the asynchronous operation. Such a structure **may** be `static_cast` to and from XrFutureCompletionBaseHeaderEXT, allowing generic handling of the asynchronous operation results as well as polymorphic output from such an operation. The XrResult returned from a completion function **must** not be used to return the result of the asynchronous operation. Instead, the XrResult returned from a completion function **must** indicate both whether the completion function was called correctly, and if the completion of the future succeeded.

For instance, a completion function returning `XR_ERROR_HANDLE_INVALID` means that a handle passed to the completion function was invalid, not that a handle associated with the asynchronous operation is invalid. Note that `XR_SUCCESS` **should** be returned from the completion function even if the asynchronous operation itself was a failure; that failure is indicated in XrFutureCompletionBaseHeaderEXT::`futureResult` rather than the return value of the completion function.

When a completion function is called with a future that is in the `XR_FUTURE_STATE_PENDING_EXT` state, the runtime **must** return `XR_ERROR_FUTURE_PENDING_EXT`.

The XrResult of the asynchronous operation **must** be returned in the `futureResult` of the return structure extending XrFutureCompletionBaseHeaderEXT. Completion functions which only need to return an XrResult **may** populate the XrFutureCompletionEXT structure provided by this extension as their output structure.

Once a completion function is called on a future with a valid output structure and returns `XR_SUCCESS`, the future is considered **completed**, and therefore **invalidated**. Any usage of this future thereafter **must** return `XR_ERROR_FUTURE_INVALID_EXT`.

Passing a completed future to any function accepting futures **must** return `XR_ERROR_FUTURE_INVALID_EXT`.

The runtime **may** release any resources associated with an XrFutureEXT once the future has been completed or invalidated.

> *Note*
>
> Each XrFutureEXT value **must** be externally synchronized by the application when calling completion, polling, and cancellation functions, and when destroying the associated handle.

The XrFutureCompletionBaseHeaderEXT structure is defined as:

```
// Provided by XR_EXT_future
typedef struct XrFutureCompletionBaseHeaderEXT {
    XrStructureType    type;
    void*              next;
    XrResult           futureResult;
} XrFutureCompletionBaseHeaderEXT;
```

XrFutureCompletionBaseHeaderEXT is a base header for the result of a future completion function.

The XrFutureCompletionEXT structure is defined as:

```
// Provided by XR_EXT_future
typedef struct XrFutureCompletionEXT {
    XrStructureType    type;
    void*              next;
    XrResult           futureResult;
} XrFutureCompletionEXT;
```

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `futureResult` is XrResult of the async operation associated with future passed to the completion function.

This is a minimal implementation of XrFutureCompletionBaseHeaderEXT, containing only the fields present in the base header structure. It is intended for use by asynchronous operations that do not have other outputs or return values beyond an XrResult value, as the output parameter of their completion function.

Valid Usage (Implicit)

- The `XR_EXT_future` extension **must** be enabled prior to using XrFutureCompletionEXT

- `type` **must** be `XR_TYPE_FUTURE_COMPLETION_EXT`

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

- `futureResult` **must** be a valid XrResult value

12.29.5. Two-Call Idiom in Asynchronous Operations

OpenXR uses a two-call idiom for interfaces that return arrays or buffers of variable size. Asynchronous operations returning such an array or buffer similarly use the structure style of that two-call idiom, with small modifications to the typical completion function conventions to account for this pattern.

For completion functions returning an array or buffer using the two-call idiom, the future **must** be marked as **completed** if the output array size is sufficient for all elements of the data and was thus populated by the completion function. If the output array size is not sufficient, the runtime **must** not mark the future as completed nor invalidated.

For an array of zero data elements, this means the first call to the two-call idiom completion function **must** mark the future as **completed** and invalidated, even if the array is a `NULL` pointer. If XrFutureCompletionBaseHeaderEXT::`futureResult` is a failure the runtime **must invalidate** the future after the first call, and any further usage of this future **must** return `XR_ERROR_FUTURE_INVALID_EXT`.

For non-zero output arrays where XrFutureCompletionBaseHeaderEXT::`futureResult` is not a failure, XrFutureCompletionBaseHeaderEXT::`futureResult` **must** be identical for both calls to the completion function.

This definition allows asynchronous operations to return dynamically sized outputs by using the two-

[call idiom](#) in a familiar way.

12.29.6. Cancelling a future

The [xrCancelFutureEXT](#) function is defined as:

```
// Provided by XR_EXT_future
XrResult xrCancelFutureEXT(
    XrInstance                                  instance,
    const XrFutureCancelInfoEXT*                cancelInfo);
```

Parameter Descriptions

- `instance` is an [XrInstance](#) handle

- `cancelInfo` is a pointer to an [XrFutureCancelInfoEXT](#) structure.

This function cancels the future and signals that the async operation is not required. After a future has been cancelled any functions using this future **must** return `XR_ERROR_FUTURE_INVALID_EXT`.

A runtime **may** stop the asynchronous operation associated with a future after an app has cancelled it.

> *Note*
>
> Each [XrFutureEXT](#) value **must** be externally synchronized by the application when calling completion, polling, and cancellation functions, or destroying the associated handle.

Valid Usage (Implicit)

- The `XR_EXT_future` extension **must** be enabled prior to calling [xrCancelFutureEXT](#)

- `instance` **must** be a valid [XrInstance](#) handle

- `cancelInfo` **must** be a pointer to a valid [XrFutureCancelInfoEXT](#) structure

Thread Safety

- Access to the `future` member of the `cancelInfo` parameter **must** be externally synchronized

The XrFutureCancelInfoEXT structure is defined as:

```
// Provided by XR_EXT_future
typedef struct XrFutureCancelInfoEXT {
    XrStructureType    type;
    const void*        next;
    XrFutureEXT        future;
} XrFutureCancelInfoEXT;
```

Member Descriptions

- type is the XrStructureType of this structure.
- next is NULL or a pointer to the next structure in a structure chain.
- future is XrFutureEXT to cancel.

An XrFutureCancelInfoEXT describes which future to cancel.

Valid Usage (Implicit)

- The XR_EXT_future extension **must** be enabled prior to using XrFutureCancelInfoEXT
- type **must** be XR_TYPE_FUTURE_CANCEL_INFO_EXT
- next **must** be NULL or a valid pointer to the next structure in a structure chain

12.29.7. XrFutureEXT Lifecycle

The XrFutureStateEXT enumerates the possible future lifecycle states:

```
// Provided by XR_EXT_future
typedef enum XrFutureStateEXT {
    XR_FUTURE_STATE_PENDING_EXT = 1,
    XR_FUTURE_STATE_READY_EXT = 2,
    XR_FUTURE_STATE_MAX_ENUM_EXT = 0x7FFFFFFF
} XrFutureStateEXT;
```

Enumerant Descriptions

- XR_FUTURE_STATE_PENDING_EXT. The state of a future that is waiting for the async operation to conclude. This is typically the initial state of a future returned from an async function.

- XR_FUTURE_STATE_READY_EXT. The state of a future when the result of the async operation is ready. The application **can** retrieve the result by calling the associated completion function.
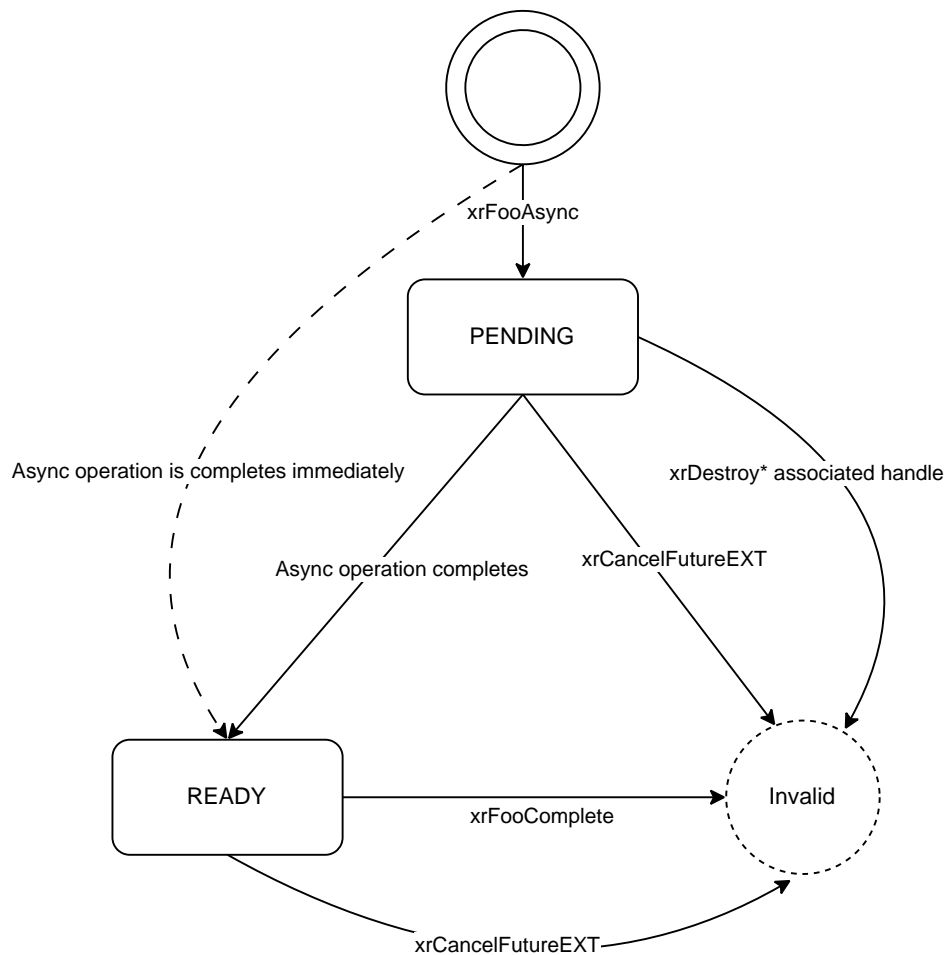
Figure 9. `XrFutureEXT` *Lifecycle*

A future that is not invalidated (or completed) **may** be in one of two states, `Pending` and `Ready`, represented by `XR_FUTURE_STATE_PENDING_EXT` and `XR_FUTURE_STATE_READY_EXT` respectively.

- When successfully returned from an async function the future starts out as `Pending`. In this state the future **may** be polled, but **must** not be passed to a completion function. Applications **should** wait for the future to become ready and keep polling the state of the future. If a pending future is passed to the associated completion function, it **must** return `XR_ERROR_FUTURE_PENDING_EXT`.

- Once the asynchronous operation succeeds or fails, the state of the future moves to `Ready`. In the ready state the future **may** be "Completed" with the `Complete` function. See Completing a Future.

- After being successfully completed, the future becomes invalidated if the completion function returns a success code, and in the case of two-call idioms, the array was not `NULL`.

- After a call to xrCancelFutureEXT, the future becomes invalidated immediately and any resources associated with it **may** be freed (including handles)

- When the associated handle is destroyed, the futures become invalidated. See Future Scope.

A future returned from an async function **must** be in either the state `XR_FUTURE_STATE_PENDING_EXT` or `XR_FUTURE_STATE_READY_EXT`. A runtime **may** skip the `Pending` state and go directly to `Ready` if the result is immediately available.

12.29.8. Future Scope

An XrFutureEXT is scoped to the "associated handle" of the future. The associated handle is the handle passed to the asynchronous operation that returns the XrFutureEXT. When the associated handle is destroyed, the runtime **must** invalidate the future and **may** free any associated resources.

> *Note*
>
> For example, for a hypothetical async function xrGetFooAsync(Session session, XrFooGetInfo info, XrFutureEXT* future) then XrSession is the associated handle, and if the app calls xrDestroySession(⋯) the returned future becomes invalid.
>
> Likewise, for xrRequestBar(BarGenerator barGenerator, XrBarGenerateInfo info, XrFutureEXT* future), the hypothetical BarGenerator is the associated handle that scopes the future.

12.29.9. Extension Guidelines for Asynchronous Functions

Extensions exposing asynchronous functions using XR_EXT_future **should** follow the following patterns:

1. Functions returning a future **should** use the suffix "Async", prior to an author/vendor tag if applicable. For example:

   - xrGetFooAsync(⋯)

   - xrRequestBarAsyncKHR(⋯)

   - xrCreateObjectAsyncVENDOR(⋯)

2. The name of the future out parameter **should** be future. For example:

   - xrGetFooAsync(⋯, XrFutureEXT* future)

   - xrRequestBarAsyncKHR(⋯, XrFutureEXT* future)

   - xrCreateObjectAsyncVENDOR(⋯, XrFutureEXT* future)

3. Functions completing a future **should** match the name of the function returning the future, but with "Complete" rather than "Async" as the suffix. This is a deviation from the normal pattern in OpenXR, if "complete" is considered to be the verb; however this provides for a useful sorting order keeping the "Async" and "Complete" functions adjacent, and fits the pattern of using suffixes for asynchronous functions. The completion function **must** use the same handle type as the corresponding async function and the runtime **must** return XR_ERROR_HANDLE_INVALID if the handle value passed to the completion function is different from the value passed to the async function that returned the future. For example:

   - xrGetFooComplete(⋯)

   - xrRequestBarCompleteKHR(⋯),

   - xrCreateObjectCompleteVENDOR(⋯)

4. The output structure used in the "Complete" function **should** extend

XrFutureCompletionBaseHeaderEXT (starting with `type`, `next`, and `futureResult` fields).

5. If an operation requires more than the basic XrFutureCompletionEXT output, the output structure populated by the "Complete" function **should** be named based on the function that returned the future, with the suffix "Completion". For example:

   ◦ `xrGetFooComplete` populates `XrGetFooCompletion`

   ◦ `xrRequestBarComplete` populates `XrRequestBarCompletionKHR`

   ◦ `xrCreateObjectCompleteVENDOR` populates `XrCreateObjectCompletionVENDOR`

6. The `XrFutureEXT` parameter in the "Complete" function **should** be named `future`. For example:

   ◦ `xrGetFooComplete(…, XrFutureEXT future)`

   ◦ `xrRequestBarCompleteKHR(…, XrFutureEXT future)`

   ◦ `xrCreateObjectCompleteVENDOR(…, XrFutureEXT future)`

7. The parameter with the completion structure **should** be named `completion`. e.g.

   ◦ `xrGetFooComplete(…, XrFutureEXT future, XrGetFooCompletion* completion)`

   ◦ `xrRequestBarCompleteKHR(…, XrFutureEXT future, XrRequestBarCompletionKHR* completion)`

   ◦ `xrCreateObjectCompleteVENDOR(…, XrFutureEXT future, XrCreateObjectCompletionVENDOR* completion)`

12.29.10. Asynchronous function patterns

`xrCreate` **functions**

```
/***************************/
/* Foo extension definition */
/***************************/
typedef void *XrFoo; // Handle definition
typedef struct XrFooObjectCreateInfo {
  XrStructureType type;
  const void *next;
} XrFooObjectCreateInfo;
#define XR_TYPE_FOO_OBJECT_CREATE_INFO ((XrStructureType)1100092000U)

// extends struct XrFutureCompletionBaseHeader using "parentstruct"
typedef struct XrFooObjectCreateCompletionEXT {
  XrStructureType type;
  void *XR_MAY_ALIAS next;
  XrResult futureResult;
  XrFoo foo;
} XrFooObjectCreateCompletionEXT;
#define XR_TYPE_FOO_OBJECT_CREATE_COMPLETION ((XrStructureType)1100092001U)

typedef XrResult(XRAPI_PTR *PFN_xrCreateFooObjectAsync)(
```

```
    XrSession session, const XrFooObjectCreateInfo *createInfo,
    XrFutureEXT *future);
typedef XrResult(XRAPI_PTR *PFN_xrCreateFooObjectComplete)(
    XrSession session, XrFutureEXT future,
    XrFooObjectCreateCompletionEXT *completion);

/************************/
/* End Foo definition   */
/************************/

PFN_xrCreateFooObjectAsync xrCreateFooObjectAsync; // previously initialized
PFN_xrCreateFooObjectComplete
    xrCreateFooObjectComplete;      // previously initialized
PFN_xrPollFutureEXT xrPollFutureEXT; // previously initialized
XrInstance instance;                 // previously initialized
XrSession session;                   // previously initialized

XrFutureEXT futureFooObject;
XrResult result;

XrFooObjectCreateInfo createInfo{XR_TYPE_FOO_OBJECT_CREATE_INFO};
result = xrCreateFooObjectAsync(session, &createInfo, &futureFooObject);
CHK_XR(result);

bool keepLooping = true;
bool futureReady = false;
while (keepLooping) {
  XrFuturePollInfoEXT pollInfo{XR_TYPE_FUTURE_POLL_INFO_EXT};
  XrFuturePollResultEXT pollResult{XR_TYPE_FUTURE_POLL_RESULT_EXT};
  pollInfo.future = futureFooObject;
  CHK_XR(xrPollFutureEXT(instance, &pollInfo, &pollResult));

  if (pollResult.state == XR_FUTURE_STATE_READY_EXT) {
    futureReady = true;
    keepLooping = false;
  } else {
    // sleep(10);
  }
}

if (futureReady) {
  XrFooObjectCreateCompletionEXT completion{
      XR_TYPE_FOO_OBJECT_CREATE_COMPLETION};
  result = xrCreateFooObjectComplete(session, futureFooObject, &completion);
  CHK_XR(result);                  // Result of the complete function
  CHK_XR(completion.futureResult); // Return code of the create function
  // completion.fooObject is now valid and may be used!
}
```

```
/***************************/
/* Foo extension definition */
/***************************/
typedef struct XrFooObjectCreateInfo {
  XrStructureType type;
  const void *next;
} XrFooObjectCreateInfo;
#define XR_TYPE_FOO_OBJECTS_CREATE_INFO ((XrStructureType)1100092002U)

// extends struct XrFutureCompletionBaseHeader using "parentstruct"
typedef struct XrFooObjectsCreateCompletionEXT {
  XrStructureType type;
  void *next;
  XrResult futureResult;
  uint32_t elementCapacityInput;
  uint32_t elementCapacityOutput;
  float *elements;
} XrFooObjectsCreateCompletionEXT;
#define XR_TYPE_FOO_OBJECTS_CREATE_COMPLETION ((XrStructureType)1100092003U)

typedef XrResult(XRAPI_PTR *PFN_xrCreateFooObjectsAsync)(
    XrSession session, const XrFooObjectCreateInfo *createInfo,
    XrFutureEXT *future);
typedef XrResult(XRAPI_PTR *PFN_xrCreateFooObjectsComplete)(
    XrSession session, XrFutureEXT future,
    XrFooObjectsCreateCompletionEXT *completion);

/************************/
/* End Foo definition   */
/************************/

PFN_xrCreateFooObjectsAsync xrCreateFooObjectsAsync; // previously initialized
PFN_xrCreateFooObjectsComplete
    xrCreateFooObjectsComplete;      // previously initialized
PFN_xrPollFutureEXT xrPollFutureEXT; // previously initialized
XrInstance instance;                 // previously initialized
XrSession session;                   // previously initialized

XrFutureEXT futureFooObjects;
XrResult result;

XrFooObjectCreateInfo createInfo{XR_TYPE_FOO_OBJECTS_CREATE_INFO};
result = xrCreateFooObjectsAsync(session, &createInfo, &futureFooObjects);
CHK_XR(result);
```

```
  bool keepLooping = true;
  bool futureReady = false;
  while (keepLooping) {
    XrFuturePollInfoEXT pollInfo{XR_TYPE_FUTURE_POLL_INFO_EXT};
    XrFuturePollResultEXT pollResult{XR_TYPE_FUTURE_POLL_RESULT_EXT};
    pollInfo.future = futureFooObjects;
    CHK_XR(xrPollFutureEXT(instance, &pollInfo, &pollResult));

    if (pollResult.state == XR_FUTURE_STATE_READY_EXT) {
      futureReady = true;
      keepLooping = false;
    } else {
      // sleep(10);
    }
  }

  if (futureReady) {
    XrFooObjectsCreateCompletionEXT completion{
        XR_TYPE_FOO_OBJECTS_CREATE_COMPLETION};
    result = xrCreateFooObjectsComplete(session, futureFooObjects, &completion);
    CHK_XR(result); // Result of the complete function
    CHK_XR(completion.futureResult);

    // If elementCapacityOutput is 0, then the future is now complete / invalid
    if (completion.elementCapacityOutput != 0) {
      std::vector<float> floatValues(completion.elementCapacityOutput);
      completion.elementCapacityInput = (uint32_t)floatValues.size();
      completion.elements = floatValues.data();

      result = xrCreateFooObjectsComplete(session, futureFooObjects, &completion);
      CHK_XR(result); // Result of the complete function
    }
  }

  // completion.elements has now been filled with values by the runtime.
```

Sample code

```
/****************************************/
/* Slow Foo extension definition */
/****************************************/
// extends struct XrFutureCompletionBaseHeader using "parentstruct"
typedef struct XrSlowFooCompletionEXT {
  XrStructureType type;
  void *XR_MAY_ALIAS next;
  XrResult futureResult;
```

```
    float foo;
} XrSlowFooCompletionEXT;
#define XR_TYPE_SLOW_FOO_COMPLETION_EXT ((XrStructureType)1100092005U)

typedef struct XrSlowFooInfoEXT {
  XrStructureType type;
  void *XR_MAY_ALIAS next;
} XrSlowFooInfoEXT;
#define XR_TYPE_SLOW_FOO_INFO_EXT ((XrStructureType)1100092006U)

typedef XrResult(XRAPI_PTR *PFN_xrSlowFooAsyncEXT)(XrSession session,
                                                   XrSlowFooInfoEXT slowFooInfo,
                                                   XrFutureEXT *future);
typedef XrResult(XRAPI_PTR *PFN_xrSlowFooCompleteEXT)(
    XrSession session, XrFutureEXT future, XrSlowFooCompletionEXT *completion);


/*******************************************/
/* End Slow Foo extension definition */
/*******************************************/

class MyGame {
  void OnSlowFooRequest() {
    if (m_slowFooFuture == XR_NULL_FUTURE_EXT) {
      // Make initial request.
      XrSlowFooInfoEXT fooInfo{XR_TYPE_SLOW_FOO_INFO_EXT};
      XrResult result = xrSlowFooAsyncEXT(session, fooInfo, &m_slowFooFuture);
      CHK_XR(result);
    }
  }

  void OnGameTickOrSomeOtherReoccurringFunction() {

    // Check if a future is outstanding
    if (m_slowFooFuture == XR_NULL_FUTURE_EXT) {
      return;
    }

    // Poll for state of future
    XrFuturePollInfoEXT pollInfo{XR_TYPE_FUTURE_POLL_INFO_EXT};
    XrFuturePollResultEXT pollResult{XR_TYPE_FUTURE_POLL_RESULT_EXT};
    pollInfo.future = m_slowFooFuture;
    CHK_XR(xrPollFutureEXT(instance, &pollInfo, &pollResult));

    if (pollResult.state == XR_FUTURE_STATE_READY_EXT) {
      // Complete the future, consuming the result
      XrSlowFooCompletionEXT completion{XR_TYPE_SLOW_FOO_COMPLETION_EXT};
      XrResult result =
          xrSlowFooCompleteEXT(session, m_slowFooFuture, &completion);
```

```
      // Check XrResult from the completion function
      CHK_XR(result);
      // Check XrResult from the async operation
      CHK_XR(completion.futureResult);
      m_fooValue = completion.foo;
      m_slowFooFuture = XR_NULL_FUTURE_EXT;
    }
  }

  XrFutureEXT m_slowFooFuture{XR_NULL_FUTURE_EXT};
  float m_fooValue{0.0f};

  PFN_xrSlowFooAsyncEXT xrSlowFooAsyncEXT;         // previously initialized
  PFN_xrSlowFooCompleteEXT xrSlowFooCompleteEXT; // previously initialized
  PFN_xrPollFutureEXT xrPollFutureEXT;             // previously initialized
  XrInstance instance;                             // previously initialized
  XrSession session;                               // previously initialized
};
```

Multi-threaded code

```
class MyThreadedGame {

  MyThreadedGame() {
    // Start the thread
    m_processThread = std::thread(&MyThreadedGame::ThreadFunction, this);
    StartSlowFooRequest();
  }

  ~MyThreadedGame() {
    // all functions using futures must be synchronized.
    CancelSlowFooRequestFuture();
    m_abort = true;
    m_processThread.join();
  }

  void StartSlowFooRequest() {
    std::unique_lock<std::mutex> lock(m_mutex);
    if (m_slowFooFuture == XR_NULL_FUTURE_EXT) {
      // Make initial request.
      XrSlowFooInfoEXT fooInfo{XR_TYPE_SLOW_FOO_INFO_EXT};
      XrResult result = xrSlowFooAsyncEXT(session, fooInfo, &m_slowFooFuture);
      CHK_XR(result);
    }
  }
```

```cpp
void CancelSlowFooRequestFuture() {
  std::unique_lock<std::mutex> lock(m_mutex);
  if (m_slowFooFuture != XR_NULL_FUTURE_EXT) {
    XrFutureCancelInfoEXT cancel_info{XR_TYPE_FUTURE_CANCEL_INFO_EXT};
    cancel_info.future = m_slowFooFuture;
    xrCancelFutureEXT(instance, &cancel_info);
    m_slowFooFuture = XR_NULL_FUTURE_EXT;
  }
}

void CheckFooRequestCompletion() {

  std::unique_lock<std::mutex> lock(m_mutex);
  // Check if a future is outstanding
  if (m_slowFooFuture == XR_NULL_FUTURE_EXT) {
    return;
  }

  // Poll for state of future
  XrFuturePollInfoEXT pollInfo{XR_TYPE_FUTURE_POLL_INFO_EXT};
  XrFuturePollResultEXT pollResult{XR_TYPE_FUTURE_POLL_RESULT_EXT};
  pollInfo.future = m_slowFooFuture;
  CHK_XR(xrPollFutureEXT(instance, &pollInfo, &pollResult));

  if (pollResult.state == XR_FUTURE_STATE_READY_EXT) {
    // Complete the future, consuming the result
    XrSlowFooCompletionEXT completion{XR_TYPE_SLOW_FOO_COMPLETION_EXT};
    XrResult result =
        xrSlowFooCompleteEXT(session, m_slowFooFuture, &completion);
    // Check XrResult from the completion function
    CHK_XR(result);
    // Check XrResult from the async operation
    CHK_XR(completion.futureResult);
    m_fooValue = completion.foo;
    m_slowFooFuture = XR_NULL_FUTURE_EXT;

    // Do something with the foo value.
  }
}

void ThreadFunction() {
  while (!m_abort) {
    // other logic here

    CheckFooRequestCompletion();

    // sleep if needed.
  }
```

```
    }

    XrFutureEXT m_slowFooFuture{XR_NULL_FUTURE_EXT};
    float m_fooValue{0.0f};
    bool m_abort{false};
    std::mutex m_mutex;
    std::thread m_processThread;
};
```

New Base Types

- XrFutureEXT

New Functions

- xrPollFutureEXT

- xrCancelFutureEXT

New Structures

- XrFutureCompletionEXT

- XrFutureCompletionBaseHeaderEXT

- XrFuturePollInfoEXT

- XrFuturePollResultEXT

- XrFutureCancelInfoEXT

New Enum Constants

- XR_NULL_FUTURE_EXT

XrStructureType enumeration is extended with:

- XR_TYPE_FUTURE_CANCEL_INFO_EXT

- XR_TYPE_FUTURE_POLL_INFO_EXT

- XR_TYPE_FUTURE_POLL_RESULT_EXT

- XR_TYPE_FUTURE_COMPLETION_EXT

XrResult enumeration is extended with:

- XR_ERROR_FUTURE_PENDING_EXT

- XR_ERROR_FUTURE_INVALID_EXT

Issues

- Should there be a state for completed functions that is separate from "invalid"?
  - Resolved.
  - Answer: No. This would force an implementing runtime to remember old futures forever. In order to allow implementations that delete all associated data about a future after completion, we cannot differentiate between a future that never existed and one that was completed. Similarly, invalidated/completed is not formally a "state" for futures in the final API.

Version History

- Revision 1, 2023-02-14 (Andreas Løve Selvik, Meta Platforms and Ron Bessems, Magic Leap)
  - Initial extension description

12.30. XR_EXT_hand_interaction

Name String

    `XR_EXT_hand_interaction`

Extension Type

    Instance extension

Registered Extension Number

    303

Revision

    1

Ratification Status

    Ratified

Extension and Version Dependencies

    OpenXR 1.0

API Interactions

- Interacts with `XR_EXT_palm_pose`

Contributors

    Yin Li, Microsoft
    Alex Turner, Microsoft
    Casey Meekhof, Microsoft
    Lachlan Ford, Microsoft
    Eric Provencher, Unity Technologies
    Bryan Dube, Unity Technologies
    Peter Kuhn, Unity Technologies
    Tanya Li, Unity Technologies

Jakob Bornecrantz, Collabora

Jonathan Wright, Meta Platforms

Federico Schliemann, Meta Platforms

Andreas Loeve Selvik, Meta Platforms

Nathan Nuber, Valve

Joe Ludwig, Valve

Rune Berg, Valve

Adam Harwood, Ultraleap

Robert Blenkinsopp, Ultraleap

Paulo Gomes, Samsung Electronics

Ron Bessems, Magic Leap

Bastiaan Olij, Godot Engine

12.30.1. Overview

This extension defines four commonly used action poses for all user hand interaction profiles including both hand tracking devices and motion controller devices.

This extension also introduces a new interaction profile specifically designed for hand tracking devices to input through the OpenXR action system. Though, for runtimes with controller inputs, the runtime **should** also provide this interaction profile through action mappings from the controller inputs, so that an application whose suggested action bindings solely depending on this hand interaction profile is usable on such runtimes as well.

12.30.2. Action poses for hand interactions

The following four action poses (i.e. "pinch," "poke," "aim," and "grip") enable a hand and finger interaction model, whether the tracking inputs are provided by a hand tracking device or a motion controller device.

The runtime **must** support all of the following action subpaths on all interaction profiles that are valid for the user paths of */user/hand/left* and */user/hand/right*, including those interaction profiles enabled through extensions.

- *…/input/aim/pose*

- *…/input/grip/pose*

- *…/input/pinch_ext/pose*

- *…/input/poke_ext/pose*

Aim pose

The *…/input/aim/pose* is designed for interacting with objects out of arm's reach. For example, using a virtual laser pointer to aim at a virtual button on the wall is an interaction suited to the "aim" pose.

This is the same "aim" pose defined in Standard pose identifiers. Every tracked controller profile

already supports this pose.
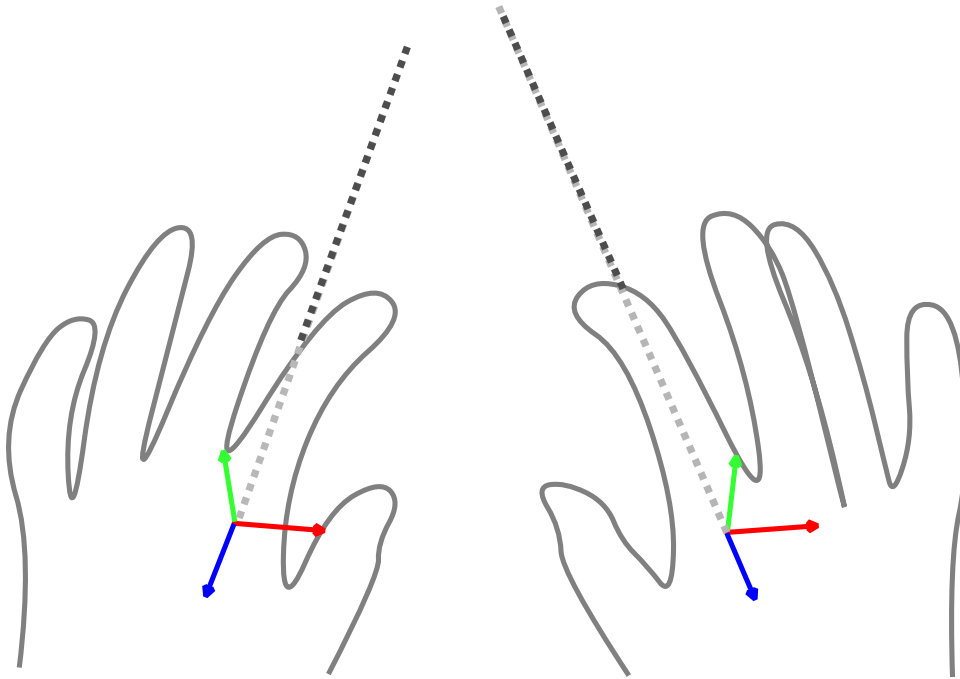
Figure 10. Example aim pose.

Position

The position of an "aim" pose is typically in front of the user's hand and moves together with the corresponding hand, so that the user is able to easily see the aiming ray cast to the target in the world and adjust for aim.

Orientation

The orientation of an "aim" pose is typically stabilized so that it is suitable to render an aiming ray emerging from the user's hand pointing into the world.

The -Z direction is the forward direction of the aiming gesture, that is, where the aiming ray is pointing at.

The +Y direction is a runtime defined direction based on the hand tracking device or ergonomics of the controller in the user's hand. It is typically pointing up in the world when the user is performing the aiming gesture naturally forward with a hand or controller in front of the user body.

The +X direction is orthogonal to +Y and +Z using the right-hand rule.

When targeting an object out of arm's reach, the runtime **may** optimize the "aim" pose stability for pointing at a target, therefore the rotation of the "aim" pose **may** account for forearm or shoulder motion as well as hand rotation. Hence, the "aim" pose **may** not always rigidly attach to the user's hand rotation. If the application desires to rotate the targeted remote object in place, it **should** use the rotation of the "grip" pose instead of "aim" pose, as if the user is remotely holding the object and rotating it.

Grip pose

The *.../input/grip/pose* is designed for holding an object with a full hand grip gesture, for example, grasping and pushing a door's handle or holding and swinging a sword.

This is the same "grip" pose defined in Standard pose identifiers. Every tracked controller profile already supports this pose.

The runtime **should** optimize the "grip" pose orientation so that it stabilizes large virtual objects held in the user's hand.
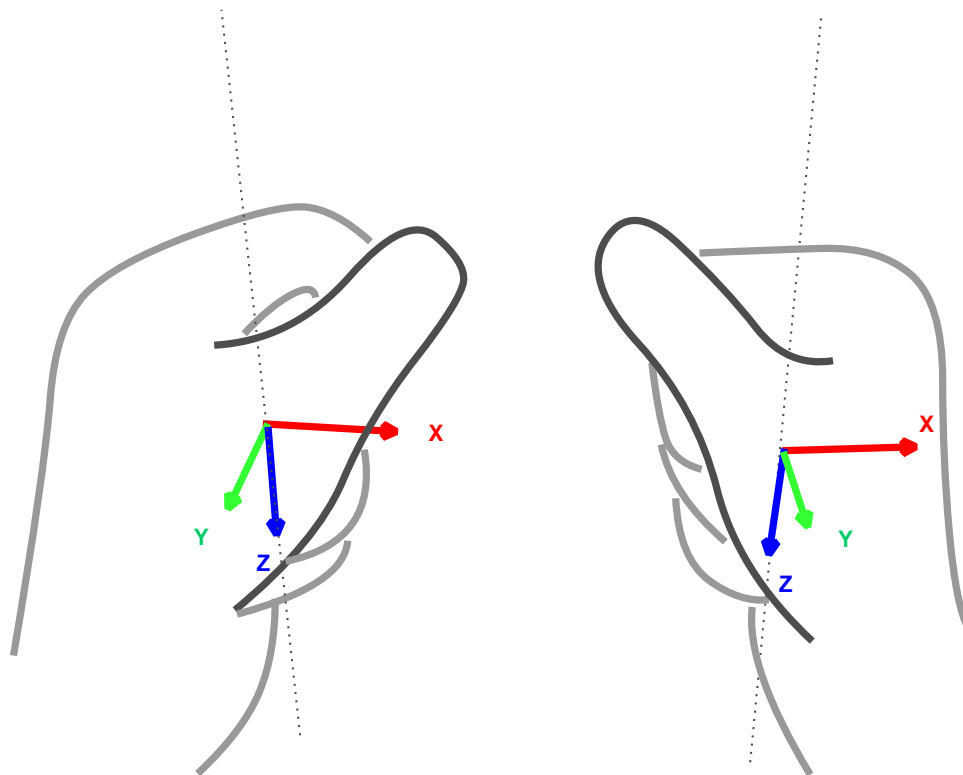
Figure 11. Example grip pose.

Position

The position of the "grip" pose is at the centroid of the user's palm when the user makes a fist or holds a tube-like object in the hand.

Orientation

The orientation of the "grip" pose **may** be used to render a virtual object held in the hand, for example, holding the grip of a virtual sword.

The Z axis of the grip pose goes through the center of the user's curled fingers when the user makes a fist or holds a controller, and the -Z direction (forward) goes from the little finger to the index finger.

When the user completely opens their hand to form a flat 5-finger pose and the palms face each other, the ray that is normal to the user's palms defines the X axis. The +X direction points away from the

palm of the left hand and into the palm of the right hand. That is to say, in the described pose, the +X direction points to the user's right for both hands. To further illustrate: if the user is holding a stick by making a fist with each hand in front of the body and pointing the stick up, the +X direction points to the user's right for both hands.

The +Y direction is orthogonal to +Z and +X using the right-hand rule.

Pinch pose

The *.../input/pinch_ext/pose* is designed for interacting with a small object within arm's reach using a finger and thumb with a "pinch" gesture. For example, turning a key to open a lock or moving the knob on a slider control are interactions suited to the "pinch" pose.

The runtime **should** stabilize the "pinch" pose while the user is performing the "pinch" gesture.
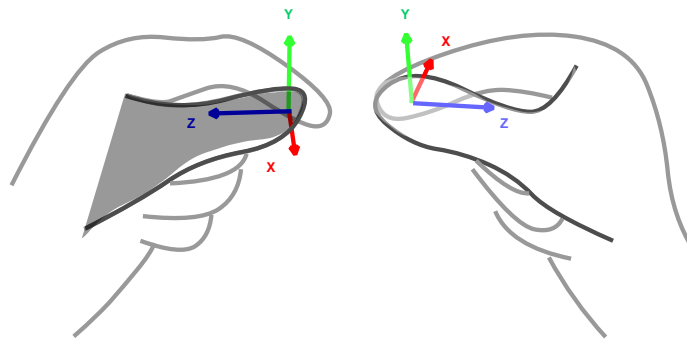
Figure 12. Example pinch pose.

Position

When the input is provided by a hand tracking device, the position of the "pinch" pose is typically where the index and thumb fingertips will touch each other for a "pinch" gesture.

The runtime **may** provide the "pinch" pose using any finger based on the current user's preference for accessibility support. An application typically designs the "pinch" pose interaction assuming the "pinch" is performed using the index finger and thumb.

When the input is provided by a motion controller device, the position of the "pinch" pose is typically based on a fixed offset from the grip pose in front of the controller, where the user **can** naturally interact with a small object. The runtime **should** avoid obstructing the "pinch" pose with the physical profile of the motion controller.

Orientation

The "pinch" pose orientation **must** rotate together with the hand rotation.
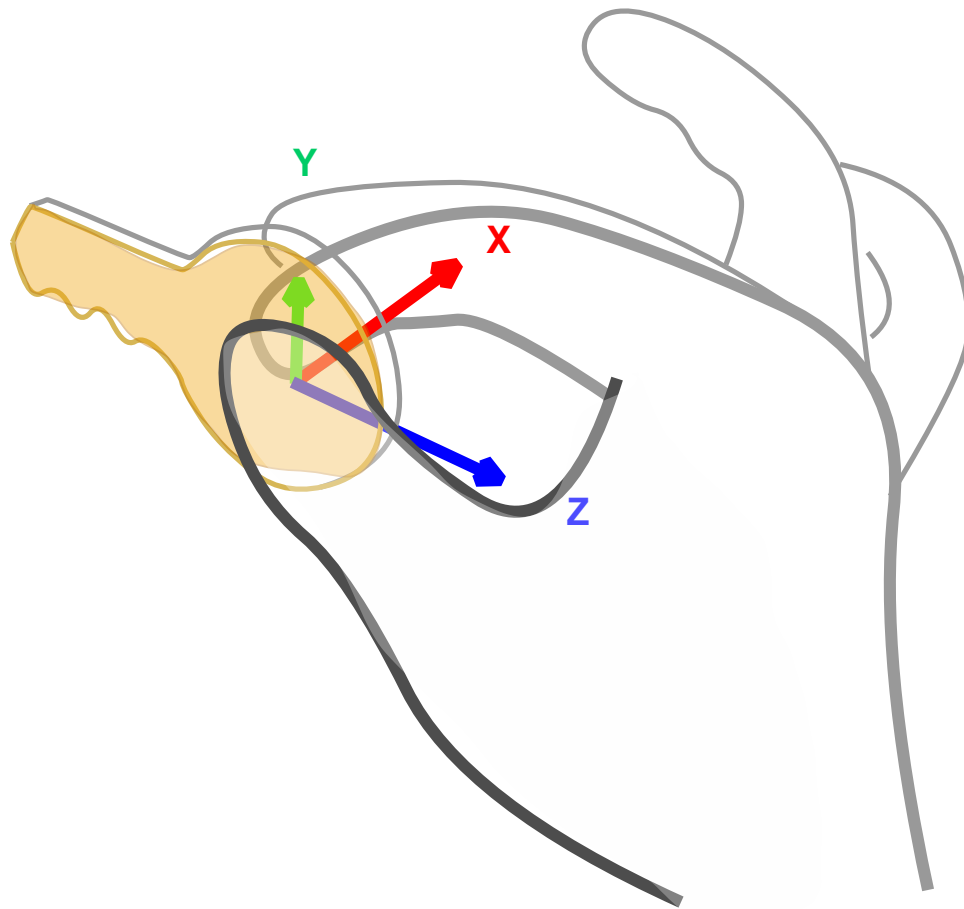
Figure 13. Example pinch orientation on right hand.

The "pinch" pose's orientation **may** be used to render a virtual object being held by a "pinch" gesture, for example, holding a key as illustrated in picture above.

If this virtual key is within a plane as illustrated in the above picture, the Y and Z axes of the "pinch" pose are within this plane.

The +Z axis is the backward direction of the "pinch" pose, typically the direction from the "pinch" position pointing to the mid point of thumb and finger proximal joints.

When the user puts both hands in front of the body at the same height, palms facing each other and fingers pointing forward, then performs a "pinch" gesture with both hands, the +Y direction for both hands **should** be roughly pointing up.

The X direction follows the right-hand rule using the Z and Y axes.

If the input is provided by a motion controller device, the orientation of the "pinch" pose is typically based on a fixed-rotation offset from the "grip" pose orientation that roughly follows the above definition when the user is holding the controller naturally.

Poke pose

The *.../input/poke_ext/pose* is designed for interactions using a fingertip to touch and push a small

object. For example, pressing a push button with a fingertip, swiping to scroll a browser view, or typing on a virtual keyboard are interactions suited to the "poke" pose.

The application **may** use the "poke" pose as a point to interact with virtual objects, and this pose is typically enough for simple interactions.

The application **may** also use a volumetric representation of a "poke" gesture using a sphere combined with the "poke" pose. The center of such a sphere is located the distance of one radius in the +Z direction of the "poke" pose, such that the "poke" pose falls on the surface of the sphere and the sphere models the shape of the fingertip.
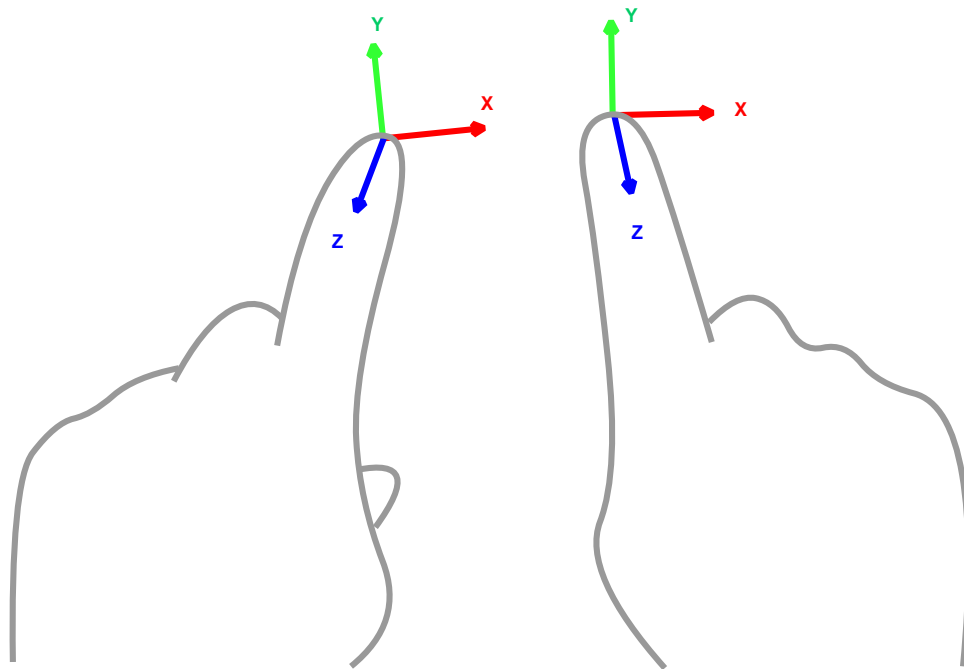
Figure 14. Example poke pose.

Position

When input is provided by a hand tracking device, the position of the "poke" pose is at the surface of the extended index fingertip. The runtime **may** provide the "poke" pose using other fingers for accessibility support.

When input is provided by a motion controller, the position of the "poke" pose is typically based on a fixed offset from the "grip" pose in front of the controller, where touching and pushing a small object feels natural using the controller. The runtime **should** avoid obstructing the "poke" pose with the physical profile of the motion controller.

Orientation

The +Y direction of the "poke" pose is the up direction in the world when the user is extending the index finger forward with palm facing down. When using a motion controller, +Y matches the up direction in the world when the user extends the index finger forward while holding the controller with palm facing down.

The +Z direction points from the fingertip towards the knuckle and parallel to the index finger distal bone, i.e. backwards when the user is holding a controller naturally in front of the body and pointing index finger forward.

The +X direction is orthogonal to +Y and +Z using the right-hand rule.

The "poke" pose **must** rotate together with the tip of the finger or the controller's "grip" pose.

12.30.3. The interaction profile for hand tracking devices

The hand interaction profile is designed for runtimes which provide hand inputs using hand tracking devices instead of controllers with triggers or buttons. This allows hand tracking devices to provide commonly used gestures and action poses to the OpenXR action system.

In addition to hand tracking devices, runtimes with controller inputs **should** also implement this interaction profile through action bindings, so that an application whose suggested action bindings solely depending on this hand interaction profile is usable on such runtimes as well.

Interaction profile path:

- */interaction_profiles/ext/hand_interaction_ext*

Valid for top level user path:

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

Supported component paths:

- *…/input/aim/pose*
- *…/input/grip/pose*
- *…/input/pinch_ext/pose*
- *…/input/poke_ext/pose*
- *…/input/pinch_ext/value*
- *…/input/pinch_ext/ready_ext*
- *…/input/aim_activate_ext/value*
- *…/input/aim_activate_ext/ready_ext*
- *…/input/grasp_ext/value*
- *…/input/grasp_ext/ready_ext*

> **ℹ**
> *Note*
>
> When the runtime supports `XR_VERSION_1_1` and use of OpenXR 1.1 is requested by the application, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

> **ℹ**
> *Note*
>
> When the `XR_KHR_maintenance1` extension is available and enabled, this interaction profile **must** also support
>
> - *…/input/grip_surface/pose*

This interaction profile supports the above four action poses, as well as the following three groups of action inputs.

Pinch action

This interaction profile supports *…/input/pinch_ext/value* and *…/input/pinch_ext/ready_ext* actions.

The *…/input/pinch_ext/value* is a 1D analog input component indicating the extent which the user is bringing their finger and thumb together to perform a "pinch" gesture.

The *…/input/pinch_ext/value* **can** be used as either a boolean or float action type, where the value `XR_TRUE` or `1.0f` represents that the finger and thumb are touching each other.

The *…/input/pinch_ext/value* **must** be at value `0.0f` or `XR_FALSE` when the hand is in a natural and relaxed open state without the user making any extra effort.

The *…/input/pinch_ext/value* **should** be linear to the distance between the finger and thumb tips when they are in the range to change "pinch" value from 0 to 1.

The *…/input/pinch_ext/ready_ext* is a boolean input, where the value `XR_TRUE` indicates that the fingers used to perform the "pinch" gesture are properly tracked by the hand tracking device and the hand shape is observed to be ready to perform or is performing a "pinch" gesture.

The *…/input/pinch_ext/value* **must** be `0.0f` or `XR_FALSE` when the *…/input/pinch_ext/ready_ext* is `XR_FALSE`.

The runtime **may** drive the input of the "pinch" gesture using any finger with the thumb to support accessibility.

Aim activate action

This interaction profile supports *…/input/aim_activate_ext/value* and *…/input/aim_activate_ext/ready_ext* actions.

The *…/input/aim_activate_ext/value* is a 1D analog input component indicating that the user activated the action on the target that the user is pointing at with the aim pose.

The "aim_activate" gesture is runtime defined, and it **should** be chosen so that the "aim" pose tracking is stable and usable for pointing at a distant target while the gesture is being performed.

The .../input/aim_activate_ext/value **can** be used as either a boolean or float action type, where the value XR_TRUE or 1.0f represents that the aimed-at target is being fully interacted with.

The .../input/aim_activate_ext/ready_ext is a boolean input, where the value XR_TRUE indicates that the fingers to perform the "aim_activate" gesture are properly tracked by the hand tracking device and the hand shape is observed to be ready to perform or is performing an "aim_activate" gesture.

The .../input/aim_activate_ext/value **must** be 0.0f or XR_FALSE when the .../input/aim_activate_ext/ready_ext is XR_FALSE.

Grasp action

This interaction profile supports .../input/grasp_ext/value action.

The .../input/grasp_ext/value is a 1D analog input component indicating that the user is making a fist.

The .../input/grasp_ext/value **can** be used as either a boolean or float action type, where the value XR_TRUE or 1.0f represents that the fist is tightly closed.

The .../input/grasp_ext/value **must** be at value 0.0f or XR_FALSE when the hand is in a natural and relaxed open state without the user making any extra effort.

The .../input/grasp_ext/ready_ext is a boolean input, where the value XR_TRUE indicates that the hand performing the grasp action is properly tracked by the hand tracking device and it is observed to be ready to perform or is performing the grasp action.

The .../input/grasp_ext/value **must** be 0.0f or XR_FALSE when the .../input/grasp_ext/ready_ext is XR_FALSE.

Hand interaction gestures overlap

The values of the above "pinch", "grasp", and "aim_activate" input actions **may** not be mutually exclusive when the input is provided by a hand tracking device. The application **should** not assume these actions are distinctively activated as action inputs provided by buttons or triggers on a controller. The application **should** suggest action bindings considering the intent of the action and their paired action pose.

Using hand interaction profile with controllers

The runtimes with controller inputs **should** support the /interaction_profiles/ext/hand_interaction_ext profile using input mapping, so that applications **can** solely rely on the /interaction_profiles/ext/hand_interaction_ext profile to build XR experiences.

If the application desires to further customize the action poses with more flexible use of controller interaction profiles, the application **can** also provide action binding suggestions of controller profile using specific buttons or triggers to work together with the commonly used four action poses.

Typical usages of action poses with hand or controller profiles

- The *.../input/grip/pose* is typically used for holding a large object in the user's hand. When using a hand interaction profile, it is typically paired with *.../input/grasp_ext/value* for the user to directly manipulate an object held in a hand. When using a controller interaction profile, the "grip" pose is typically paired with a "squeeze" button or trigger that gives the user the sense of tightly holding an object.

- The *.../input/pinch_ext/pose* is typically used for directly manipulating a small object using the pinch gesture. When using a hand interaction profile, it is typically paired with the *.../input/pinch_ext/value* gesture. When using a controller interaction profile, it is typically paired with a trigger manipulated with the index finger, which typically requires curling the index finger and applying pressure with the fingertip.

- The *.../input/poke_ext/pose* is typically used for contact-based interactions using the motion of the hand or fingertip. It typically does not pair with other hand gestures or buttons on the controller. The application typically uses a sphere collider with the "poke" pose to visualize the pose and detect touch with a virtual object.

- The *.../input/aim/pose* is typically used for aiming at objects out of arm's reach. When using a hand interaction profile, it is typically paired with *.../input/aim_activate_ext/value* to optimize aiming ray stability while performing the gesture. When using a controller interaction profile, the "aim" pose is typically paired with a trigger or a button for aim and fire operations.

- Because controllers are typically mapping buttons or triggers for the above hand interaction values, they typically report `XR_TRUE` for their corresponding *.../ready_ext* action. This is because the buttons and triggers are always prepared and capable of receiving actions.

New Object Types

New Flag Types

New Enum Constants

New Enums

New Structures

New Functions

Issues

Version History

- Revision 1, 2021-08-06 (Yin Li)

- Initial extension description

12.31. XR_EXT_hand_joints_motion_range

Name String

XR_EXT_hand_joints_motion_range

Extension Type

Instance extension

Registered Extension Number

81

Revision

1

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0
and
XR_EXT_hand_tracking

Last Modified Date

2021-04-15

IP Status

No known IP claims.

Contributors

Joe van den Heuvel, Valve
Rune Berg, Valve
Joe Ludwig, Valve
Jakob Bornecrantz, Collabora

Overview

This extension augments the XR_EXT_hand_tracking extension to enable applications to request that the XrHandJointLocationsEXT returned by xrLocateHandJointsEXT should return hand joint locations conforming to a range of motion specified by the application.

The application **must** enable the XR_EXT_hand_tracking extension in order to use this extension.

New Object Types

New Flag Types

New Enum Constants

New Enums

The XrHandJointsMotionRangeEXT describes the hand joints' range of motion returned by xrLocateHandJointsEXT.

Runtimes **must** support both `XR_HAND_JOINTS_MOTION_RANGE_CONFORMING_TO_CONTROLLER_EXT` and `XR_HAND_JOINTS_MOTION_RANGE_UNOBSTRUCTED_EXT` for each controller interaction profile that supports hand joint data.

```
// Provided by XR_EXT_hand_joints_motion_range
typedef enum XrHandJointsMotionRangeEXT {
    XR_HAND_JOINTS_MOTION_RANGE_UNOBSTRUCTED_EXT = 1,
    XR_HAND_JOINTS_MOTION_RANGE_CONFORMING_TO_CONTROLLER_EXT = 2,
    XR_HAND_JOINTS_MOTION_RANGE_MAX_ENUM_EXT = 0x7FFFFFFF
} XrHandJointsMotionRangeEXT;
```

<div style="text-align:center">

Enumerant Descriptions

</div>

- `XR_HAND_JOINTS_MOTION_RANGE_UNOBSTRUCTED_EXT` This option refers to the range of motion of a human hand, without any obstructions. Input systems that obstruct the movement of the user's hand (e.g.: a held controller preventing the user from making a fist) or have only limited ability to track finger positions **must** use the information available to them to emulate an unobstructed range of motion.

- `XR_HAND_JOINTS_MOTION_RANGE_CONFORMING_TO_CONTROLLER_EXT` This option refers to the range of motion of the hand joints taking into account any physical limits imposed by the controller itself. This will tend to be the most accurate pose compared to the user's actual hand pose, but might not allow a closed fist for example.

  - If the current interaction profile represents a controller, or other device that obstructs the hand, the implementation **must** return joint locations conforming to the shape of that device. If the current interaction profile is being emulated by a different physical controller, the implementation **may** return joint locations conforming to the shape of either the current interaction profile or the actual physical controller.

  - If the current interaction profile does not represent a controller, the implementation **must** return joint locations based on the unobstructed joint locations.

New Structures

The XrHandJointsMotionRangeInfoEXT is a structure that an application **can** chain in XrHandJointsLocateInfoEXT to request the joint motion range specified by the handJointsMotionRange field.

Runtimes **must** return the appropriate joint locations depending on the handJointsMotionRange field and the currently active interaction profile.

```
// Provided by XR_EXT_hand_joints_motion_range
typedef struct XrHandJointsMotionRangeInfoEXT {
    XrStructureType             type;
    const void*                 next;
    XrHandJointsMotionRangeEXT    handJointsMotionRange;
} XrHandJointsMotionRangeInfoEXT;
```

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.

- handJointsMotionRange is an XrHandJointsMotionRangeEXT that defines the hand joint range of motion the application wants.

Valid Usage (Implicit)

- The XR_EXT_hand_joints_motion_range extension **must** be enabled prior to using XrHandJointsMotionRangeInfoEXT

- type **must** be XR_TYPE_HAND_JOINTS_MOTION_RANGE_INFO_EXT

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

- handJointsMotionRange **must** be a valid XrHandJointsMotionRangeEXT value

New Functions

Issues

Version History

- Revision 1, 2021-04-15 (Rune Berg)
  - Initial extension description

12.32. XR_EXT_hand_tracking

Name String

`XR_EXT_hand_tracking`

Extension Type

Instance extension

Registered Extension Number

52

Revision

4

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0

Last Modified Date

2021-04-15

IP Status

No known IP claims.

Contributors

Yin Li, Microsoft
Lachlan Ford, Microsoft
Alex Turner, Microsoft
Bryce Hutchings, Microsoft
Cass Everitt, Oculus
Blake Taylor, Magic Leap
Joe van den Heuvel, Valve
Rune Berg, Valve
Valerie Benson, Ultraleap
Rylie Pavlik, Collabora

12.32.1. Overview

This extension enables applications to locate the individual joints of hand tracking inputs. It enables applications to render hands in XR experiences and interact with virtual objects using hand joints.

12.32.2. Inspect system capability

An application **can** inspect whether the system is capable of hand tracking input by extending the XrSystemProperties with XrSystemHandTrackingPropertiesEXT structure when calling xrGetSystemProperties.

```
// Provided by XR_EXT_hand_tracking
typedef struct XrSystemHandTrackingPropertiesEXT {
    XrStructureType    type;
    void*              next;
    XrBool32           supportsHandTracking;
} XrSystemHandTrackingPropertiesEXT;
```

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.
- supportsHandTracking is an XrBool32, indicating if current system is capable of hand tracking input.

Valid Usage (Implicit)

- The XR_EXT_hand_tracking extension **must** be enabled prior to using XrSystemHandTrackingPropertiesEXT
- type **must** be XR_TYPE_SYSTEM_HAND_TRACKING_PROPERTIES_EXT
- next **must** be NULL or a valid pointer to the next structure in a structure chain

If a runtime returns XR_FALSE for supportsHandTracking, the runtime **must** return XR_ERROR_FEATURE_UNSUPPORTED from xrCreateHandTrackerEXT.

12.32.3. Create a hand tracker handle

The XrHandTrackerEXT handle represents the resources for hand tracking of the specific hand.

```
XR_DEFINE_HANDLE(XrHandTrackerEXT)
```

An application creates separate XrHandTrackerEXT handles for left and right hands. This handle can be used to locate hand joints using xrLocateHandJointsEXT function.

A hand tracker provides joint locations with an unobstructed range of motion of an empty human hand.

> **i** *Note*
>
> This behavior can be modified by the XR_EXT_hand_joints_motion_range extension

An application can create an XrHandTrackerEXT handle using xrCreateHandTrackerEXT function.

```
// Provided by XR_EXT_hand_tracking
XrResult xrCreateHandTrackerEXT(
    XrSession                                   session,
    const XrHandTrackerCreateInfoEXT*           createInfo,
    XrHandTrackerEXT*                           handTracker);
```

Parameter Descriptions

- session is an XrSession in which the hand tracker will be active.
- createInfo is the XrHandTrackerCreateInfoEXT used to specify the hand tracker.
- handTracker is the returned XrHandTrackerEXT handle.

Valid Usage (Implicit)

- The XR_EXT_hand_tracking extension **must** be enabled prior to calling xrCreateHandTrackerEXT
- session **must** be a valid XrSession handle
- createInfo **must** be a pointer to a valid XrHandTrackerCreateInfoEXT structure
- handTracker **must** be a pointer to an XrHandTrackerEXT handle

If the system does not support hand tracking, runtime **must** return XR_ERROR_FEATURE_UNSUPPORTED from xrCreateHandTrackerEXT. In this case, the runtime **must** return XR_FALSE for XrSystemHandTrackingPropertiesEXT::supportsHandTracking when the function xrGetSystemProperties is called, so that the application **can** avoid creating a hand tracker.

The XrHandTrackerCreateInfoEXT structure describes the information to create an XrHandTrackerEXT handle.

```
// Provided by XR_EXT_hand_tracking
typedef struct XrHandTrackerCreateInfoEXT {
    XrStructureType      type;
    const void*          next;
    XrHandEXT            hand;
    XrHandJointSetEXT    handJointSet;
} XrHandTrackerCreateInfoEXT;
```

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.

- `hand` is an XrHandEXT which describes which hand the tracker is tracking.

- `handJointSet` is an XrHandJointSetEXT describe the set of hand joints to retrieve.

Valid Usage (Implicit)

- The `XR_EXT_hand_tracking` extension **must** be enabled prior to using XrHandTrackerCreateInfoEXT

- `type` **must** be `XR_TYPE_HAND_TRACKER_CREATE_INFO_EXT`

- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrHandTrackingDataSourceInfoEXT

- `hand` **must** be a valid XrHandEXT value

- `handJointSet` **must** be a valid XrHandJointSetEXT value

The XrHandEXT describes which hand the XrHandTrackerEXT is tracking.

```
// Provided by XR_EXT_hand_tracking
typedef enum XrHandEXT {
    XR_HAND_LEFT_EXT = 1,
    XR_HAND_RIGHT_EXT = 2,
    XR_HAND_MAX_ENUM_EXT = 0x7FFFFFFF
} XrHandEXT;
```

Enumerant Descriptions

- `XR_HAND_LEFT_EXT` specifies the hand tracker will be tracking the user's left hand.

- `XR_HAND_RIGHT_EXT` specifies the hand tracker will be tracking the user's right hand.

The XrHandJointSetEXT enum describes the set of hand joints to track when creating an XrHandTrackerEXT.

```
// Provided by XR_EXT_hand_tracking
typedef enum XrHandJointSetEXT {
    XR_HAND_JOINT_SET_DEFAULT_EXT = 0,
    XR_HAND_JOINT_SET_MAX_ENUM_EXT = 0x7FFFFFFF
} XrHandJointSetEXT;
```

Enumerant Descriptions

- XR_HAND_JOINT_SET_DEFAULT_EXT indicates that the created XrHandTrackerEXT tracks the set of hand joints described by XrHandJointEXT enum, i.e. the xrLocateHandJointsEXT function returns an array of joint locations with the count of XR_HAND_JOINT_COUNT_EXT and can be indexed using XrHandJointEXT.

xrDestroyHandTrackerEXT function releases the handTracker and the underlying resources when finished with hand tracking experiences.

```
// Provided by XR_EXT_hand_tracking
XrResult xrDestroyHandTrackerEXT(
    XrHandTrackerEXT                            handTracker);
```

Parameter Descriptions

- handTracker is an XrHandTrackerEXT previously created by xrCreateHandTrackerEXT.

Valid Usage (Implicit)

- The XR_EXT_hand_tracking extension **must** be enabled prior to calling xrDestroyHandTrackerEXT
- handTracker **must** be a valid XrHandTrackerEXT handle

Thread Safety

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

12.32.4. Locate hand joints

The xrLocateHandJointsEXT function locates an array of hand joints to a base space at given time.

```
// Provided by XR_EXT_hand_tracking
XrResult xrLocateHandJointsEXT(
    XrHandTrackerEXT                            handTracker,
    const XrHandJointsLocateInfoEXT*            locateInfo,
    XrHandJointLocationsEXT*                    locations);
```

Parameter Descriptions

- `handTracker` is an XrHandTrackerEXT previously created by xrCreateHandTrackerEXT.

- `locateInfo` is a pointer to XrHandJointsLocateInfoEXT describing information to locate hand joints.

- `locations` is a pointer to XrHandJointLocationsEXT receiving the returned hand joint locations.

Valid Usage (Implicit)

- The `XR_EXT_hand_tracking` extension **must** be enabled prior to calling xrLocateHandJointsEXT

- `handTracker` **must** be a valid XrHandTrackerEXT handle

- `locateInfo` **must** be a pointer to a valid XrHandJointsLocateInfoEXT structure

- `locations` **must** be a pointer to an XrHandJointLocationsEXT structure

The XrHandJointsLocateInfoEXT structure describes the information to locate hand joints.

```
// Provided by XR_EXT_hand_tracking
typedef struct XrHandJointsLocateInfoEXT {
    XrStructureType    type;
    const void*        next;
    XrSpace            baseSpace;
    XrTime             time;
} XrHandJointsLocateInfoEXT;
```

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.

- baseSpace is an XrSpace within which the returned hand joint locations will be represented.

- time is an XrTime at which to locate the hand joints.

XrHandJointLocationsEXT structure returns the state of the hand joint locations.

```
// Provided by XR_EXT_hand_tracking
typedef struct XrHandJointLocationsEXT {
    XrStructureType           type;
    void*                     next;
    XrBool32                  isActive;
    uint32_t                  jointCount;
    XrHandJointLocationEXT*   jointLocations;
} XrHandJointLocationsEXT;
```

The application **must** allocate the memory for the output array `jointLocations` that can contain at least `jointCount` of XrHandJointLocationEXT.

The application **must** set `jointCount` as described by the XrHandJointSetEXT when creating the XrHandTrackerEXT otherwise the runtime **must** return `XR_ERROR_VALIDATION_FAILURE`.

The runtime **must** return `jointLocations` representing the range of motion of a human hand, without any obstructions. Input systems that obstruct the movement of the user's hand (e.g.: a held controller

preventing the user from making a fist) or that have only limited ability to track finger positions **must** use the information available to them to emulate an unobstructed range of motion.

The runtime **must** update the `jointLocations` array ordered so that the application can index elements using the corresponding hand joint enum (e.g. XrHandJointEXT) as described by XrHandJointSetEXT when creating the XrHandTrackerEXT. For example, when the XrHandTrackerEXT is created with `XR_HAND_JOINT_SET_DEFAULT_EXT`, the application **must** set the `jointCount` to `XR_HAND_JOINT_COUNT_EXT`, and the runtime **must** fill the `jointLocations` array ordered so that it may be indexed by the XrHandJointEXT enum.

If the returned `isActive` is true, the runtime **must** return all joint locations with both `XR_SPACE_LOCATION_POSITION_VALID_BIT` and `XR_SPACE_LOCATION_ORIENTATION_VALID_BIT` set. Although, in this case, some joint space locations **may** be untracked (i.e. `XR_SPACE_LOCATION_POSITION_TRACKED_BIT` or `XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT` is unset).

If the returned `isActive` is false, it indicates the hand tracker did not detect the hand input or the application lost input focus. In this case, the runtime **must** return all `jointLocations` with neither `XR_SPACE_LOCATION_POSITION_VALID_BIT` nor `XR_SPACE_LOCATION_ORIENTATION_VALID_BIT` set.

> ### Valid Usage (Implicit)
>
> - The XR_EXT_hand_tracking extension **must** be enabled prior to using XrHandJointLocationsEXT
>
> - `type` **must** be `XR_TYPE_HAND_JOINT_LOCATIONS_EXT`
>
> - `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrHandJointVelocitiesEXT, XrHandTrackingDataSourceStateEXT
>
> - `jointLocations` **must** be a pointer to an array of `jointCount` XrHandJointLocationEXT structures
>
> - The `jointCount` parameter **must** be greater than `0`

XrHandJointLocationEXT structure describes the position, orientation, and radius of a hand joint.

```
// Provided by XR_EXT_hand_tracking
typedef struct XrHandJointLocationEXT {
    XrSpaceLocationFlags    locationFlags;
    XrPosef                 pose;
    float                   radius;
} XrHandJointLocationEXT;
```

Member Descriptions

- `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 a hand joint within the reference frame of the corresponding XrHandJointsLocateInfoEXT::`baseSpace`.

- `radius` is a `float` value radius of the corresponding joint in units of meters.

If the returned `locationFlags` has `XR_SPACE_LOCATION_POSITION_VALID_BIT` set, the returned radius **must** be a positive value.

If the returned `locationFlags` has `XR_SPACE_LOCATION_POSITION_VALID_BIT` unset, the returned radius value is undefined and should be avoided.

Valid Usage (Implicit)

- The XR_EXT_hand_tracking extension **must** be enabled prior to using XrHandJointLocationEXT

- `locationFlags` **must** be 0 or a valid combination of XrSpaceLocationFlagBits values

The application can chain an XrHandJointVelocitiesEXT structure to the `next` pointer of XrHandJointLocationsEXT when calling xrLocateHandJointsEXT to retrieve the hand joint velocities.

```
// Provided by XR_EXT_hand_tracking
typedef struct XrHandJointVelocitiesEXT {
    XrStructureType           type;
    void*                     next;
    uint32_t                  jointCount;
    XrHandJointVelocityEXT*   jointVelocities;
} XrHandJointVelocitiesEXT;
```

- `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.

- `jointCount` is a `uint32_t` describing the number of elements in `jointVelocities` array.

- `jointVelocities` is an array of XrHandJointVelocityEXT receiving the returned hand joint velocities.

The application **must** allocate the memory for the output array `jointVelocities` that can contain at least `jointCount` of XrHandJointVelocityEXT.

The application **must** input `jointCount` as described by the XrHandJointSetEXT when creating the XrHandTrackerEXT. Otherwise, the runtime **must** return `XR_ERROR_VALIDATION_FAILURE`.

The runtime **must** update the `jointVelocities` array in the order so that the application can index elements using the corresponding hand joint enum (e.g. XrHandJointEXT) as described by the XrHandJointSetEXT when creating the XrHandTrackerEXT. For example, when the XrHandTrackerEXT is created with `XR_HAND_JOINT_SET_DEFAULT_EXT`, the application **must** set the `jointCount` to `XR_HAND_JOINT_COUNT_EXT`, and the returned `jointVelocities` array **must** be ordered to be indexed by enum XrHandJointEXT enum.

If the returned XrHandJointLocationsEXT::`isActive` is false, it indicates the hand tracker did not detect a hand input or the application lost input focus. In this case, the runtime **must** return all `jointVelocities` with neither `XR_SPACE_VELOCITY_LINEAR_VALID_BIT` nor `XR_SPACE_VELOCITY_ANGULAR_VALID_BIT` set.

If an XrHandJointVelocitiesEXT structure is chained to XrHandJointLocationsEXT::`next`, the returned XrHandJointLocationsEXT::`isActive` is true, and the velocity is observed or can be calculated by the runtime, the runtime **must** fill in the linear velocity of each hand joint within the reference frame of XrHandJointsLocateInfoEXT::`baseSpace` and set the `XR_SPACE_VELOCITY_LINEAR_VALID_BIT`. Similarly, if an XrHandJointVelocitiesEXT structure is chained to XrHandJointLocationsEXT::`next`, the returned XrHandJointLocationsEXT::`isActive` is true, and the *angular velocity* is observed or can be calculated by the runtime, the runtime **must** fill in the angular velocity of each joint within the reference frame of XrHandJointsLocateInfoEXT::`baseSpace` and set the `XR_SPACE_VELOCITY_ANGULAR_VALID_BIT`.

XrHandJointVelocityEXT structure describes the linear and angular velocity of a hand joint.

```
// Provided by XR_EXT_hand_tracking
typedef struct XrHandJointVelocityEXT {
    XrSpaceVelocityFlags    velocityFlags;
    XrVector3f              linearVelocity;
    XrVector3f              angularVelocity;
} XrHandJointVelocityEXT;
```

Member Descriptions

- `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 the hand joint with respect to and expressed in the reference frame of the corresponding XrHandJointsLocateInfoEXT::`baseSpace`, in units of meters per second.

- `angularVelocity` is the relative angular velocity of the hand joint with respect to the corresponding XrHandJointsLocateInfoEXT::`baseSpace`. The vector's direction is expressed in the reference frame of the corresponding XrHandJointsLocateInfoEXT::`baseSpace` and is parallel to the rotational axis of the hand joint. The vector's magnitude is the relative angular speed of the hand joint in radians per second. The vector follows the right-hand rule for torque/rotation.

12.32.5. Example code for locating hand joints

The following example code demonstrates how to locate all hand joints relative to a world space.

```
XrInstance instance; // previously initialized
XrSystemId systemId; // previously initialized
XrSession session;   // previously initialized
XrSpace worldSpace;  // previously initialized, e.g. from
                     // XR_REFERENCE_SPACE_TYPE_LOCAL


// Inspect hand tracking system properties
XrSystemHandTrackingPropertiesEXT handTrackingSystemProperties{
    XR_TYPE_SYSTEM_HAND_TRACKING_PROPERTIES_EXT};
XrSystemProperties systemProperties{XR_TYPE_SYSTEM_PROPERTIES,
                                    &handTrackingSystemProperties};
CHK_XR(xrGetSystemProperties(instance, systemId, &systemProperties));
if (!handTrackingSystemProperties.supportsHandTracking) {
    // The system does not support hand tracking
    return;
}

// Get function pointer for xrCreateHandTrackerEXT
PFN_xrCreateHandTrackerEXT pfnCreateHandTrackerEXT;
CHK_XR(xrGetInstanceProcAddr(instance, "xrCreateHandTrackerEXT",
                             reinterpret_cast<PFN_xrVoidFunction*>(
                             &pfnCreateHandTrackerEXT)));

// Create a hand tracker for left hand that tracks default set of hand joints.
XrHandTrackerEXT leftHandTracker{};
{
    XrHandTrackerCreateInfoEXT createInfo{XR_TYPE_HAND_TRACKER_CREATE_INFO_EXT};
    createInfo.hand = XR_HAND_LEFT_EXT;
    createInfo.handJointSet = XR_HAND_JOINT_SET_DEFAULT_EXT;
    CHK_XR(pfnCreateHandTrackerEXT(session, &createInfo, &leftHandTracker));
}

// Allocate buffers to receive joint location and velocity data before frame
// loop starts
```

```cpp
XrHandJointLocationEXT jointLocations[XR_HAND_JOINT_COUNT_EXT];
XrHandJointVelocityEXT jointVelocities[XR_HAND_JOINT_COUNT_EXT];

XrHandJointVelocitiesEXT velocities{XR_TYPE_HAND_JOINT_VELOCITIES_EXT};
velocities.jointCount = XR_HAND_JOINT_COUNT_EXT;
velocities.jointVelocities = jointVelocities;

XrHandJointLocationsEXT locations{XR_TYPE_HAND_JOINT_LOCATIONS_EXT};
locations.next = &velocities;
locations.jointCount = XR_HAND_JOINT_COUNT_EXT;
locations.jointLocations = jointLocations;

// Get function pointer for xrLocateHandJointsEXT
PFN_xrLocateHandJointsEXT pfnLocateHandJointsEXT;
CHK_XR(xrGetInstanceProcAddr(instance, "xrLocateHandJointsEXT",
                             reinterpret_cast<PFN_xrVoidFunction*>(
                             &pfnLocateHandJointsEXT)));
while (1) {
    // ...
    // For every frame in frame loop
    // ...
    XrFrameState frameState; // previously returned from xrWaitFrame
    const XrTime time = frameState.predictedDisplayTime;

    XrHandJointsLocateInfoEXT locateInfo{XR_TYPE_HAND_JOINTS_LOCATE_INFO_EXT};
    locateInfo.baseSpace = worldSpace;
    locateInfo.time = time;

    CHK_XR(pfnLocateHandJointsEXT(leftHandTracker, &locateInfo, &locations));

    if (locations.isActive) {
        // The returned joint location array can be directly indexed with
        // XrHandJointEXT enum.
        const XrPosef &indexTipInWorld =
            jointLocations[XR_HAND_JOINT_INDEX_TIP_EXT].pose;
        const XrPosef &thumbTipInWorld =
            jointLocations[XR_HAND_JOINT_THUMB_TIP_EXT].pose;

        // using the returned radius and velocity of index finger tip.
        const float indexTipRadius =
            jointLocations[XR_HAND_JOINT_INDEX_TIP_EXT].radius;
        const XrHandJointVelocityEXT &indexTipVelocity =
            jointVelocities[XR_HAND_JOINT_INDEX_TIP_EXT];
    }
}
```

12.32.6. Conventions of hand joints

This extension defines 26 joints for hand tracking: 4 joints for the thumb finger, 5 joints for the other four fingers, and the wrist and palm of the hands.
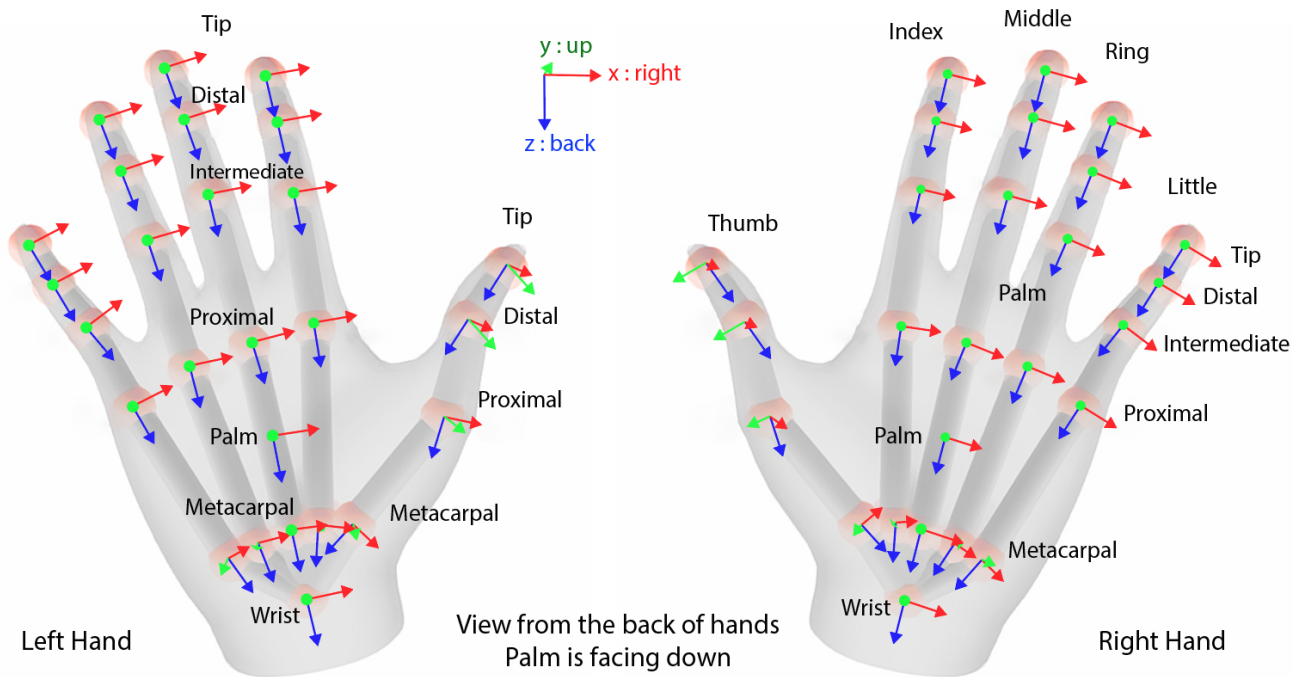
```
// Provided by XR_EXT_hand_tracking
typedef enum XrHandJointEXT {
    XR_HAND_JOINT_PALM_EXT = 0,
    XR_HAND_JOINT_WRIST_EXT = 1,
    XR_HAND_JOINT_THUMB_METACARPAL_EXT = 2,
    XR_HAND_JOINT_THUMB_PROXIMAL_EXT = 3,
    XR_HAND_JOINT_THUMB_DISTAL_EXT = 4,
    XR_HAND_JOINT_THUMB_TIP_EXT = 5,
    XR_HAND_JOINT_INDEX_METACARPAL_EXT = 6,
    XR_HAND_JOINT_INDEX_PROXIMAL_EXT = 7,
    XR_HAND_JOINT_INDEX_INTERMEDIATE_EXT = 8,
    XR_HAND_JOINT_INDEX_DISTAL_EXT = 9,
    XR_HAND_JOINT_INDEX_TIP_EXT = 10,
    XR_HAND_JOINT_MIDDLE_METACARPAL_EXT = 11,
    XR_HAND_JOINT_MIDDLE_PROXIMAL_EXT = 12,
    XR_HAND_JOINT_MIDDLE_INTERMEDIATE_EXT = 13,
    XR_HAND_JOINT_MIDDLE_DISTAL_EXT = 14,
    XR_HAND_JOINT_MIDDLE_TIP_EXT = 15,
    XR_HAND_JOINT_RING_METACARPAL_EXT = 16,
    XR_HAND_JOINT_RING_PROXIMAL_EXT = 17,
    XR_HAND_JOINT_RING_INTERMEDIATE_EXT = 18,
    XR_HAND_JOINT_RING_DISTAL_EXT = 19,
    XR_HAND_JOINT_RING_TIP_EXT = 20,
    XR_HAND_JOINT_LITTLE_METACARPAL_EXT = 21,
    XR_HAND_JOINT_LITTLE_PROXIMAL_EXT = 22,
    XR_HAND_JOINT_LITTLE_INTERMEDIATE_EXT = 23,
    XR_HAND_JOINT_LITTLE_DISTAL_EXT = 24,
    XR_HAND_JOINT_LITTLE_TIP_EXT = 25,
    XR_HAND_JOINT_MAX_ENUM_EXT = 0x7FFFFFFF
} XrHandJointEXT;
```

The finger joints, except the tips, are named after the corresponding bone at the further end of the bone from the finger tips. The joint's orientation is defined at a fully opened hand pose facing down as in the above picture.

> **Note**
>
> Many applications and game engines use names to identify joints rather than using indices. If possible, applications should use the joint name part of the XrHandJointEXT enum plus a hand identifier to help prevent joint name clashes (e.g. Index_Metacarpal_L, Thumb_Tip_R). Using consistent names increases the portability of assets between applications and engines. Including the hand in the identifier prevents ambiguity when both hands are used in the same skeleton, such as when they are combined with additional joints to form a full body skeleton.

The backward (+Z) direction is parallel to the corresponding bone and points away from the finger tip.

The up (+Y) direction is pointing out of the back of and perpendicular to the corresponding finger nail at the fully opened hand pose. The X direction is perpendicular to Y and Z and follows the right hand rule.

The wrist joint is located at the pivot point of the wrist which is location invariant when twisting hand without moving the forearm. The backward (+Z) direction is parallel to the line from wrist joint to middle finger metacarpal joint, and points away from the finger tips. The up (+Y) direction points out towards back of hand and perpendicular to the skin at wrist. The X direction is perpendicular to the Y and Z directions and follows the right hand rule.

The palm joint is located at the center of the middle finger's metacarpal bone. The backward (+Z) direction is parallel to the middle finger's metacarpal bone, and points away from the finger tips. The up (+Y) direction is perpendicular to palm surface and pointing towards the back of the hand. The X direction is perpendicular to the Y and Z directions and follows the right hand rule.

The radius of each joint is the distance from the joint to the skin in meters. The application can use a sphere at the joint location with joint radius for collision detection for interactions, such as pushing a virtual button using the index finger tip.

For example, suppose the radius of the palm joint is `r` then the app **can** offset `{0, -r, 0}` to palm joint location to get the surface of hand palm center, or offset `{0, r, 0}` to get the back surface of the hand.

Note that the palm joint for the hand tracking is not the same as .../*input/grip/pose* when hand tracking is provided by controller tracking. A "grip" pose is located at the center of the controller handle when user is holding a controller, outside of the user's hand. A "palm" pose is located at the center of middle finger metacarpal bone which is inside the user's hand.

```
// Provided by XR_EXT_hand_tracking
#define XR_HAND_JOINT_COUNT_EXT 26
```

XR_HAND_JOINT_COUNT_EXT defines the number of hand joint enumerants defined in XrHandJointEXT

New Object Types

- XrHandTrackerEXT

New Flag Types

New Enum Constants

- XR_HAND_JOINT_COUNT_EXT

XrObjectType enumeration is extended with:

- `XR_OBJECT_TYPE_HAND_TRACKER_EXT`

XrStructureType enumeration is extended with:

- `XR_TYPE_SYSTEM_HAND_TRACKING_PROPERTIES_EXT`
- `XR_TYPE_HAND_TRACKER_CREATE_INFO_EXT`
- `XR_TYPE_HAND_JOINTS_LOCATE_INFO_EXT`
- `XR_TYPE_HAND_JOINT_LOCATIONS_EXT`
- `XR_TYPE_HAND_JOINT_VELOCITIES_EXT`

New Enums

- XrHandEXT
- XrHandJointEXT
- XrHandJointSetEXT

New Structures

- XrSystemHandTrackingPropertiesEXT
- XrHandTrackerCreateInfoEXT
- XrHandJointsLocateInfoEXT
- XrHandJointLocationEXT
- XrHandJointVelocityEXT
- XrHandJointLocationsEXT
- XrHandJointVelocitiesEXT

New Functions

- xrCreateHandTrackerEXT
- xrDestroyHandTrackerEXT
- xrLocateHandJointsEXT

Issues

Version History

- Revision 1, 2019-09-16 (Yin LI)
  - Initial extension description
- Revision 2, 2020-04-20 (Yin LI)
  - Replace hand joint spaces to locate hand joints function.

- Revision 3, 2021-04-13 (Rylie Pavlik, Rune Berg)
  - Fix example code to properly use `xrGetInstanceProcAddr`.
  - Add recommended bone names
- Revision 4, 2021-04-15 (Rune Berg)
  - Clarify that use of this extension produces an unobstructed hand range of motion.

12.33. XR_EXT_hand_tracking_data_source

Name String

`XR_EXT_hand_tracking_data_source`

Extension Type

Instance extension

Registered Extension Number

429

Revision

1

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0
and
`XR_EXT_hand_tracking`

Last Modified Date

2023-01-23

IP Status

No known IP claims.

Contributors

Jakob Bornecrantz, Collabora
John Kearney, Meta
Robert Memmott, Meta
Andreas Selvik, Meta
Yin Li, Microsoft
Robert Blenkinsopp, Ultraleap
Nathan Nuber, Valve

Contacts

John Kearney, Meta

Overview

This extension augments the XR_EXT_hand_tracking extension.

Runtimes **may** support a variety of data sources for hand joint data for XR_EXT_hand_tracking, and some runtimes and devices **may** use joint data from multiple sources. This extension allows an application and the runtime to communicate about and make use of those data sources in a cooperative manner.

This extension allows the application to specify the data sources that it wants data from when creating a hand tracking handle, and allows the runtime to specify the currently active data source.

The application **must** enable the XR_EXT_hand_tracking extension in order to use this extension.

The XrHandTrackingDataSourceEXT enum describes a hand tracking data source when creating an XrHandTrackerEXT handle.

```
// Provided by XR_EXT_hand_tracking_data_source
typedef enum XrHandTrackingDataSourceEXT {
    XR_HAND_TRACKING_DATA_SOURCE_UNOBSTRUCTED_EXT = 1,
    XR_HAND_TRACKING_DATA_SOURCE_CONTROLLER_EXT = 2,
    XR_HAND_TRACKING_DATA_SOURCE_MAX_ENUM_EXT = 0x7FFFFFFF
} XrHandTrackingDataSourceEXT;
```

The application **can** use XrHandTrackingDataSourceEXT with XrHandTrackingDataSourceInfoEXT when calling xrCreateHandTrackerEXT to tell the runtime all supported data sources for the application for the hand tracking inputs.

The application **can** use it with XrHandTrackingDataSourceStateEXT when calling xrLocateHandJointsEXT to inspect what data source the runtime used for the returned hand joint locations.

If the XR_EXT_hand_joints_motion_range extension is supported by the runtime and the data source is XR_HAND_TRACKING_DATA_SOURCE_CONTROLLER_EXT, then it is expected that application will use that extension when retrieving hand joint poses.

- `XR_HAND_TRACKING_DATA_SOURCE_UNOBSTRUCTED_EXT` - This data source value indicates that the hand tracking data source supports using individual fingers and joints separately. Examples of such sources include optical hand tracking, data gloves, or motion capture devices.

- `XR_HAND_TRACKING_DATA_SOURCE_CONTROLLER_EXT` - This data source value indicates that the hand tracking data source is a motion controller. The runtime **must** not supply this data source if the controller providing the data is not actively held in the user's hand, but **may** still provide data if the runtime is unable to detect if the controller is not in the user's hand, or a user selected policy changes this behavior. Unless specified otherwise by another extension, data returned from `XR_HAND_TRACKING_DATA_SOURCE_CONTROLLER_EXT` **must** behave as `XR_HAND_JOINTS_MOTION_RANGE_UNOBSTRUCTED_EXT`.

The XrHandTrackingDataSourceInfoEXT structure is defined as:

```
// Provided by XR_EXT_hand_tracking_data_source
typedef struct XrHandTrackingDataSourceInfoEXT {
    XrStructureType              type;
    const void*                  next;
    uint32_t                     requestedDataSourceCount;
    XrHandTrackingDataSourceEXT* requestedDataSources;
} XrHandTrackingDataSourceInfoEXT;
```

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `requestedDataSourceCount` is the number of elements in the `requestedDataSources` array.

- `requestedDataSources` is an array of XrHandTrackingDataSourceEXT that the application accepts.

The XrHandTrackingDataSourceInfoEXT is a structure that an application **can** chain to XrHandTrackerCreateInfoEXT::`next` to specify the hand tracking data sources that the application accepts.

Because the hand tracking device **may** change during a running session, the runtime **may** return a valid XrHandTrackerEXT handle even if there is no currently active hand tracking device or the active device does not safisty any or all data sources requested by the applications's call to xrCreateHandTrackerEXT. The runtime **may** instead return `XR_ERROR_FEATURE_UNSUPPORTED` from

xrCreateHandTrackerEXT, if for example the runtime believes it will never be able to satisfy the request.

If any value in `requestedDataSources` is duplicated, the runtime **must** return `XR_ERROR_VALIDATION_FAILURE` from the call to xrCreateHandTrackerEXT. If `requestedDataSourceCount` is `0`, the runtime **must** return `XR_ERROR_VALIDATION_FAILURE` from the call to xrCreateHandTrackerEXT.

Valid Usage (Implicit)

- The `XR_EXT_hand_tracking_data_source` extension **must** be enabled prior to using XrHandTrackingDataSourceInfoEXT

- `type` **must** be `XR_TYPE_HAND_TRACKING_DATA_SOURCE_INFO_EXT`

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

- If `requestedDataSourceCount` is not `0`, `requestedDataSources` **must** be a pointer to an array of `requestedDataSourceCount` XrHandTrackingDataSourceEXT values

---

The XrHandTrackingDataSourceStateEXT structure is defined as:

```
// Provided by XR_EXT_hand_tracking_data_source
typedef struct XrHandTrackingDataSourceStateEXT {
    XrStructureType             type;
    void*                       next;
    XrBool32                    isActive;
    XrHandTrackingDataSourceEXT dataSource;
} XrHandTrackingDataSourceStateEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `isActive` indicating there is an active data source

- `dataSource` indicating the data source that was used to generate the hand tracking joints.

XrHandTrackingDataSourceStateEXT is a structure that an application **can** chain to XrHandJointLocationsEXT::`next` when calling xrLocateHandJointsEXT to retrieve the data source of the currently active hand tracking device.

When the returned `isActive` is `XR_FALSE`, it indicates the currently active hand tracking device does not

support any of the requested data sources. In these cases, the runtime **must** also return no valid tracking locations for hand joints from this xrLocateHandJointsEXT function.

If the tracker was not created with XrHandTrackingDataSourceInfoEXT chained to XrHandTrackerCreateInfoEXT::`next`, then the runtime **must** return `XR_ERROR_VALIDATION_FAILURE`, if XrHandTrackingDataSourceStateEXT is passed in the call to xrLocateHandJointsEXT.

If there is an active hand tracking device that is one of the specified XrHandTrackingDataSourceInfoEXT::`requestedDataSources`, the runtime **must** set `isActive` to `XR_TRUE`. When the runtime sets `isActive` to `XR_TRUE`, the runtime **must** set `dataSource` indicate the active data source. The runtime **must** return a `dataSource` that is a subset of the XrHandTrackingDataSourceInfoEXT::`requestedDataSources` when creating the corresponding hand tracker.

Valid Usage (Implicit)

- The `XR_EXT_hand_tracking_data_source` extension **must** be enabled prior to using XrHandTrackingDataSourceStateEXT

- `type` **must** be `XR_TYPE_HAND_TRACKING_DATA_SOURCE_STATE_EXT`

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

- `dataSource` **must** be a valid XrHandTrackingDataSourceEXT value

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with: * `XR_TYPE_HAND_TRACKING_DATA_SOURCE_INFO_EXT` * `XR_TYPE_HAND_TRACKING_DATA_SOURCE_STATE_EXT`

New Enums

- XrHandTrackingDataSourceEXT

New Structures

- XrHandTrackingDataSourceInfoEXT

- XrHandTrackingDataSourceStateEXT

New Functions

Issues

1. Should this extension require `XR_HAND_JOINTS_MOTION_RANGE_CONFORMING_TO_CONTROLLER_EXT` if the

data source is `XR_HAND_TRACKING_DATA_SOURCE_CONTROLLER_EXT` and `XR_EXT_hand_joints_motion_range` is not enabled?

RESOLVED: Yes.

It should not be required. We expect that a key use of the data from this extension will be replicating data hand tracking joint data for social purposes. For that use-case, the data returned in the style of `XR_HAND_JOINTS_MOTION_RANGE_UNOBSTRUCTED_EXT` is more appropriate.

This is consistent with `XR_EXT_hand_tracking` extension which requires that the `jointLocations` represent `the range of motion of a human hand, without any obstructions`.

2. Should XrHandTrackingDataSourceInfoEXT include an `isActive` member or can it use `isActive` from XrHandJointLocationsEXT?

RESOLVED: Yes.

Yes; XrHandTrackingDataSourceInfoEXT needs to include the `isActive` member and cannot use the `isActive` from XrHandJointLocationsEXT as the meaning of these members is different.

The `isActive` member of XrHandTrackingDataSourceStateEXT allows the runtime to describe if the tracking device is active. XrHandTrackingDataSourceStateEXT::`isActive` describes if the tracking device is actively tracking. It is possible for a data source to be active but not actively tracking and we want to represent if the device is active in this extension.

Version History

- Revision 1, 2023-01-23 (John Kearney)
  - Initial extension description

12.34. XR_EXT_performance_settings

Name String

    `XR_EXT_performance_settings`

Extension Type

    Instance extension

Registered Extension Number

    16

Revision

    4

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0

Last Modified Date

2021-04-14

IP Status

No known IP claims.

Contributors

Armelle Laine, Qualcomm Technologies Inc, on behalf of Qualcomm Innovation Center, Inc
Rylie Pavlik, Collabora

12.34.1. Overview

This extension defines an API for the application to give performance hints to the runtime and for the runtime to send performance related notifications back to the application. This allows both sides to dial in a suitable compromise between needed CPU and GPU performance, thermal sustainability and a consistent good user experience throughout the session.

The goal is to render frames consistently, in time, under varying system load without consuming more energy than necessary.

In summary, the APIs allow:

- setting performance level hints
- receiving performance related notifications

12.34.2. Setting Performance Levels Hints

Performance level hint definition

The XR performance level hints for a given hardware system are expressed as a level XrPerfSettingsLevelEXT for each of the XR-critical processing domains XrPerfSettingsDomainEXT (currently defined is a CPU and a GPU domain):

```
// Provided by XR_EXT_performance_settings
typedef enum XrPerfSettingsDomainEXT {
    XR_PERF_SETTINGS_DOMAIN_CPU_EXT = 1,
    XR_PERF_SETTINGS_DOMAIN_GPU_EXT = 2,
    XR_PERF_SETTINGS_DOMAIN_MAX_ENUM_EXT = 0x7FFFFFFF
} XrPerfSettingsDomainEXT;
```

```
// Provided by XR_EXT_performance_settings
typedef enum XrPerfSettingsLevelEXT {
    XR_PERF_SETTINGS_LEVEL_POWER_SAVINGS_EXT = 0,
    XR_PERF_SETTINGS_LEVEL_SUSTAINED_LOW_EXT = 25,
    XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT = 50,
    XR_PERF_SETTINGS_LEVEL_BOOST_EXT = 75,
    XR_PERF_SETTINGS_LEVEL_MAX_ENUM_EXT = 0x7FFFFFFF
} XrPerfSettingsLevelEXT;
```

This extension defines platform-independent level hints:

- XR_PERF_SETTINGS_LEVEL_POWER_SAVINGS_EXT is used by the application to indicate that it enters a non-XR section (head-locked / static screen), during which power savings are to be prioritized. Consistent XR compositing, consistent frame rendering, and low latency are not needed.

- XR_PERF_SETTINGS_LEVEL_SUSTAINED_LOW_EXT is used by the application to indicate that it enters a low and stable complexity section, during which reducing power is more important than occasional late rendering frames. With such a hint, the XR Runtime still strives for consistent XR compositing (no tearing) within a thermally sustainable range(*), but is allowed to take measures to reduce power, such as increasing latencies or reducing headroom.

- XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT is used by the application to indicate that it enters a high or dynamic complexity section, during which the XR Runtime strives for consistent XR compositing and frame rendering within a thermally sustainable range(*).

- XR_PERF_SETTINGS_LEVEL_BOOST_EXT is used to indicate that the application enters a section with very high complexity, during which the XR Runtime is allowed to step up beyond the thermally sustainable range. As not thermally sustainable, this level is meant to be used for short-term durations (< 30 seconds).

(*) If the application chooses one of the two sustainable levels (XR_PERF_SETTINGS_LEVEL_SUSTAINED_LOW_EXT or XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT), the device may still run into thermal limits under non-nominal circumstances (high room temperature, additional background loads, extended device operation) and therefore the application should also in the sustainable modes be prepared to react to performance notifications (in particular XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT and XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT in the thermal sub-domain, see Notification level definition).

The XR Runtime shall select `XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT` as the default hint if the application does not provide any. The function to call for setting performance level hints is xrPerfSettingsSetPerformanceLevelEXT.

```
// Provided by XR_EXT_performance_settings
XrResult xrPerfSettingsSetPerformanceLevelEXT(
    XrSession                                   session,
    XrPerfSettingsDomainEXT                     domain,
    XrPerfSettingsLevelEXT                      level);
```

Example of using the short-term boost level hint

For a limited amount of time, both the Mobile and PC systems can provide a higher level of performance than is thermally sustainable. It is desirable to make this extra computational power available for short complex scenes, then go back to a sustainable lower level. This section describes means for the application developer to apply settings directing the runtime to boost performance for a short-term duration.

The application developer must pay attention to keep these boost periods very short and carefully monitor the side effects, which may vary a lot between different hardware systems.

```
 1 extern XrInstance instance; ①
 2 extern XrSession session;
 3
 4 // Get function pointer for xrPerfSettingsSetPerformanceLevelEXT
 5 PFN_xrPerfSettingsSetPerformanceLevelEXT pfnPerfSettingsSetPerformanceLevelEXT;
 6 CHK_XR(xrGetInstanceProcAddr(instance, "xrPerfSettingsSetPerformanceLevelEXT",
 7                              (PFN_xrVoidFunction*)(
 8                              &pfnPerfSettingsSetPerformanceLevelEXT)));
 9
10 // before entering the high complexity section
11 pfnPerfSettingsSetPerformanceLevelEXT(session, XR_PERF_SETTINGS_DOMAIN_CPU_EXT,
   XR_PERF_SETTINGS_LEVEL_BOOST_EXT); ②
12 pfnPerfSettingsSetPerformanceLevelEXT(session, XR_PERF_SETTINGS_DOMAIN_GPU_EXT,
   XR_PERF_SETTINGS_LEVEL_BOOST_EXT);
13
14 // entering the high complexity section
15 // ... running
16 // end of the high complexity section
17
18 pfnPerfSettingsSetPerformanceLevelEXT(session, XR_PERF_SETTINGS_DOMAIN_CPU_EXT,
   XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT); ③
19 pfnPerfSettingsSetPerformanceLevelEXT(session, XR_PERF_SETTINGS_DOMAIN_GPU_EXT,
   XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT);
```

① we assume that `instance` and `session` are initialized and their handles are available

② setting performance level to `XR_PERF_SETTINGS_LEVEL_BOOST_EXT` on both CPU and GPU domains

③ going back to the sustainable `XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT`

Example of using the sustained low level hint for the CPU domain

```
 1 extern XrInstance instance; ①
 2 extern XrSession session;
 3
 4 // Get function pointer for xrPerfSettingsSetPerformanceLevelEXT
 5 PFN_xrPerfSettingsSetPerformanceLevelEXT pfnPerfSettingsSetPerformanceLevelEXT;
 6 CHK_XR(xrGetInstanceProcAddr(instance, "xrPerfSettingsSetPerformanceLevelEXT",
 7                             (PFN_xrVoidFunction*)(
 8                             &pfnPerfSettingsSetPerformanceLevelEXT)));
 9
10 // before entering a low CPU complexity section
11 pfnPerfSettingsSetPerformanceLevelEXT(session, XR_PERF_SETTINGS_DOMAIN_CPU_EXT,
   XR_PERF_SETTINGS_LEVEL_SUSTAINED_LOW_EXT);
12 pfnPerfSettingsSetPerformanceLevelEXT(session, XR_PERF_SETTINGS_DOMAIN_GPU_EXT,
   XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT); ②
13
14 // entering the low complexity section
15 // ... running
16 // end of the low complexity section
17
18 pfnPerfSettingsSetPerformanceLevelEXT(session, XR_PERF_SETTINGS_DOMAIN_CPU_EXT,
   XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT); ③
```

① we assume that `instance` and `session` are initialized and their handles are available

② the developer may choose to only reduce CPU domain and keep the GPU domain at `XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT`

③ going back to the sustainable `XR_PERF_SETTINGS_LEVEL_SUSTAINED_HIGH_EXT` for CPU

12.34.3. Receiving Performance Related Notifications

The XR runtime shall provide performance related notifications to the application in the following situations:

- the compositing performance within the runtime has reached a new level, either improved or degraded from the previous one (`subDomain` is set to `XR_PERF_SETTINGS_SUB_DOMAIN_COMPOSITING_EXT`)

- the application rendering performance has reached a new level, either improved or degraded from the previous one (`subDomain` is set to `XR_PERF_SETTINGS_SUB_DOMAIN_RENDERING_EXT`)

- the temperature of the device has reached a new level, either improved or degraded from the previous one (`subDomain` is set to `XR_PERF_SETTINGS_SUB_DOMAIN_THERMAL_EXT`).

When degradation is observed, the application **should** take measures reducing its workload, helping the compositing or rendering `subDomain` to meet their deadlines, or the thermal `subDomain` to avoid or stop throttling. When improvement is observed, the application can potentially rollback some of its mitigations.

```
// Provided by XR_EXT_performance_settings
typedef struct XrEventDataPerfSettingsEXT {
    XrStructureType                     type;
    const void*                         next;
    XrPerfSettingsDomainEXT             domain;
    XrPerfSettingsSubDomainEXT          subDomain;
    XrPerfSettingsNotificationLevelEXT  fromLevel;
    XrPerfSettingsNotificationLevelEXT  toLevel;
} XrEventDataPerfSettingsEXT;
```

```
// Provided by XR_EXT_performance_settings
typedef enum XrPerfSettingsSubDomainEXT {
    XR_PERF_SETTINGS_SUB_DOMAIN_COMPOSITING_EXT = 1,
    XR_PERF_SETTINGS_SUB_DOMAIN_RENDERING_EXT = 2,
    XR_PERF_SETTINGS_SUB_DOMAIN_THERMAL_EXT = 3,
    XR_PERF_SETTINGS_SUB_DOMAIN_MAX_ENUM_EXT = 0x7FFFFFFF
} XrPerfSettingsSubDomainEXT;
```

Compositing Sub-Domain

One of the major functions the runtime shall provide is the timely compositing of the submitted layers in the background. The runtime has to share the CPU and GPU system resources for this operation with the application. Since this is extremely time sensitive - the head room is only a few milliseconds - the runtime may have to ask the application via notifications to cooperate and relinquish some usage of the indicated resource (CPU or GPU domain). Performance issues in this area that the runtime notices are notified to the application with the subDomain set to XR_PERF_SETTINGS_SUB_DOMAIN_COMPOSITING_EXT.

Rendering Sub-Domain

The application submits rendered layers to the runtime for compositing. Performance issues in this area that the runtime notices (i.e. missing submission deadlines) are notified to the application with the subDomain set to XR_PERF_SETTINGS_SUB_DOMAIN_RENDERING_EXT.

Thermal Sub-Domain

XR applications run at a high-performance level during long periods of time, across a game or an entire movie session. As form factors shrink, especially on mobile solutions, the risk of reaching die thermal runaway or reaching the limits on skin and battery temperatures increases. When thermal limits are reached, the device mitigates the heat generation leading to severe performance reductions, which greatly affects user experience (dropped frames, high latency).

Better than dropping frames when it is too late, pro-active measures from the application should be encouraged.

The performance notification with the `subDomain` set to `XR_PERF_SETTINGS_SUB_DOMAIN_THERMAL_EXT` provides an early warning allowing the application to take mitigation actions.

Notification level definition

The levels are defined as follows:

```
// Provided by XR_EXT_performance_settings
typedef enum XrPerfSettingsNotificationLevelEXT {
    XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT = 0,
    XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT = 25,
    XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT = 75,
    XR_PERF_SETTINGS_NOTIFICATION_LEVEL_MAX_ENUM_EXT = 0x7FFFFFFF
} XrPerfSettingsNotificationLevelEXT;
```

- `XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT` notifies that the sub-domain has reached a level where no further actions other than currently applied are necessary.

- `XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT` notifies that the sub-domain has reached an early warning level where the application should start proactive mitigation actions with the goal to return to the `XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT` level.

- `XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT` notifies that the sub-domain has reached a critical level with significant performance degradation. The application should take drastic mitigation action.

The above definitions summarize the broad interpretation of the notification levels, however sub-domain specific definitions of each level and their transitions are specified below:

- `XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT`

  - For the compositing sub-domain, `XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT` indicates that the composition headroom is consistently being met with sufficient margin.
    Getting into `XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT` from `XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT` indicates that the composition headroom was consistently **met with sufficient margin during a sufficient time period**.

  - For the rendering sub-domain, `XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT` indicates that frames are being submitted in time to be used by the compositor.
    Getting into `XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT` from `XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT` indicates that **during a sufficient time period**, **none** of the due layers was **too late** to be picked up by the compositor.

  - For the thermal sub-domain, `XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT` indicates that the current load should be sustainable in the near future.
    Getting into `XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT` from `XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT` indicates that the runtime does not presuppose any further temperature mitigation action on the application side, other than the current ones.

- `XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT`

- For the compositing sub-domain, `XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT` indicates that the compositing headroom of the current frame was met but the margin is considered insufficient by the runtime, and the application **should** reduce its workload in the notified domain to solve this problem.

  Getting into `XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT` from `XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT` indicates that the compositing deadline was **not missed during a sufficient time period**.

- For the rendering sub-domain, `XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT` indicates that at least one layer is regularly late to be picked up by the compositor, resulting in a degraded user experience, and that the application should take action to consistently provide frames in a more timely manner.

  Getting into `XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT` from `XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT` indicates that **the runtime has stopped any of its own independent actions** which are tied to the `XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT` level.

- For the thermal sub-domain, the `XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT` indicates that the runtime expects the device to overheat under the current load, and that the application should take mitigating action in order to prevent thermal throttling.

  Getting into `XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT` from `XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT` indicates that **the underlying system thermal throttling has stopped**.

- `XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT`

  - For the compositing sub-domain, `XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT` indicates that composition can no longer be maintained under the current workload. The runtime may take independent action that will interfere with the application (e.g. limiting the framerate, ignoring submitted layers, or shutting down the application) in order to correct this problem.

  - For the rendering sub-domain, `XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT` indicates that at least one layer is **too often** late to be picked up by the compositor, and consequently the runtime may take independent action that will interfere with the application (e.g. informing the user that the application is not responding, displaying a tracking environment in order to maintain user orientation).

  - For the thermal sub-domain, `XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT` indicates that the **underlying system is taking measures, such as thermal throttling** to reduce the temperature, impacting the XR experience..

Leaving `XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT` indicates that any mitigating actions by the runtime (e.g. down-clocking the device to stay within thermal limits) have ended.

Performance Settings API Reference

xrPerfSettingsSetPerformanceLevelEXT

```
// Provided by XR_EXT_performance_settings
XrResult xrPerfSettingsSetPerformanceLevelEXT(
    XrSession                                   session,
    XrPerfSettingsDomainEXT                     domain,
    XrPerfSettingsLevelEXT                      level);
```

Parameter Descriptions

- `session` is a valid XrSession handle.

- `domain`: the processing domain for which the level hint is applied

- `level`: the level hint to be applied

Valid Usage (Implicit)

- The `XR_EXT_performance_settings` extension **must** be enabled prior to calling xrPerfSettingsSetPerformanceLevelEXT

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

- `domain` **must** be a valid XrPerfSettingsDomainEXT value

- `level` **must** be a valid XrPerfSettingsLevelEXT value

Refer to Performance level hint definition for the definition of the level enumerations.

XrEventDataPerformanceSettingsEXT

The XrEventDataPerfSettingsEXT structure is defined as:

```
// Provided by XR_EXT_performance_settings
typedef struct XrEventDataPerfSettingsEXT {
    XrStructureType                     type;
    const void*                         next;
    XrPerfSettingsDomainEXT             domain;
    XrPerfSettingsSubDomainEXT          subDomain;
    XrPerfSettingsNotificationLevelEXT  fromLevel;
    XrPerfSettingsNotificationLevelEXT  toLevel;
} XrEventDataPerfSettingsEXT;
```

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.

- `domain` : processing domain in which a threshold has been crossed

- `subDomain` : system area in which a threshold has been crossed

- `fromLevel` : enumerated notification level which has been exited

- `toLevel` : enumerated notification level which has been entered

Valid Usage (Implicit)

- The `XR_EXT_performance_settings` extension **must** be enabled prior to using XrEventDataPerfSettingsEXT

- `type` **must** be `XR_TYPE_EVENT_DATA_PERF_SETTINGS_EXT`

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

```
// Provided by XR_EXT_performance_settings
typedef enum XrPerfSettingsDomainEXT {
    XR_PERF_SETTINGS_DOMAIN_CPU_EXT = 1,
    XR_PERF_SETTINGS_DOMAIN_GPU_EXT = 2,
    XR_PERF_SETTINGS_DOMAIN_MAX_ENUM_EXT = 0x7FFFFFFF
} XrPerfSettingsDomainEXT;
```

```
// Provided by XR_EXT_performance_settings
typedef enum XrPerfSettingsSubDomainEXT {
    XR_PERF_SETTINGS_SUB_DOMAIN_COMPOSITING_EXT = 1,
    XR_PERF_SETTINGS_SUB_DOMAIN_RENDERING_EXT = 2,
    XR_PERF_SETTINGS_SUB_DOMAIN_THERMAL_EXT = 3,
    XR_PERF_SETTINGS_SUB_DOMAIN_MAX_ENUM_EXT = 0x7FFFFFFF
} XrPerfSettingsSubDomainEXT;
```

```
// Provided by XR_EXT_performance_settings
typedef enum XrPerfSettingsNotificationLevelEXT {
    XR_PERF_SETTINGS_NOTIF_LEVEL_NORMAL_EXT = 0,
    XR_PERF_SETTINGS_NOTIF_LEVEL_WARNING_EXT = 25,
    XR_PERF_SETTINGS_NOTIF_LEVEL_IMPAIRED_EXT = 75,
    XR_PERF_SETTINGS_NOTIFICATION_LEVEL_MAX_ENUM_EXT = 0x7FFFFFFF
} XrPerfSettingsNotificationLevelEXT;
```

Version History

- Revision 1, 2017-11-30 (Armelle Laine)
- Revision 2, 2021-04-13 (Rylie Pavlik)
    - Correctly show function pointer retrieval in sample code
    - Fix sample code callouts
- Revision 3, 2021-04-14 (Rylie Pavlik)
    - Fix missing error code
- Revision 4, 2022-10-26 (Rylie Pavlik)
    - Update XML markup to correct the generated valid usage

12.35. XR_EXT_spatial_anchor

Name String

XR_EXT_spatial_anchor

Extension Type

Instance extension

Registered Extension Number

763

Revision

1

Ratification Status

Ratified

Extension and Version Dependencies

XR_EXT_spatial_entity

Contributors

Nihav Jain, Google

Natalie Fleury, Meta

Yuichi Taguchi, Meta

Ron Bessems, Meta

Yin Li, Microsoft

Jimmy Alamparambil, ByteDance

Zhipeng Liu, ByteDance

Jun Yan, ByteDance

12.35.1. Overview

This extension builds on `XR_EXT_spatial_entity` and allows applications to create spatial anchors, which are arbitrary points in the user's physical environment that will then be tracked by the runtime. The runtime **should** then adjust the position and orientation of the anchor's origin over time as needed, independent of all other spaces & anchors, to ensure that it maintains its original mapping to the real world.

An anchor that tracks a given position and orientation within an XrSpatialContextEXT is represented as a spatial entity with (or "that has") the `XR_SPATIAL_COMPONENT_TYPE_ANCHOR_EXT` component.

12.35.2. Benefit of using anchors

As the runtime's understanding of the user's physical environment updates throughout the lifetime of an XrSpatialContextEXT, virtual objects **may** appear to drift away from where they were placed by the application, which impacts the application's realism and the quality of the user's experience. By creating an anchor close to where a virtual object is placed, and then always rendering that virtual object relative to its anchor, an application **can** ensure that each virtual object appears to stay at the same position and orientation in the physical environment. Also, unlike certain reference spaces, anchors are unaffected by system-level recentering.

12.35.3. Runtime support

If the runtime supports spatial anchors, it **must** indicate this by enumerating `XR_SPATIAL_CAPABILITY_ANCHOR_EXT` in xrEnumerateSpatialCapabilitiesEXT.

12.35.4. Configuration

The XrSpatialCapabilityConfigurationAnchorEXT structure is defined as:

```
// Provided by XR_EXT_spatial_anchor
typedef struct XrSpatialCapabilityConfigurationAnchorEXT {
    XrStructureType                    type;
    const void*                        next;
    XrSpatialCapabilityEXT             capability;
    uint32_t                           enabledComponentCount;
    const XrSpatialComponentTypeEXT*   enabledComponents;
} XrSpatialCapabilityConfigurationAnchorEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `capability` is an XrSpatialCapabilityEXT.

- `enabledComponentCount` is a `uint32_t` describing the count of elements in the `enabledComponents` array.

- `enabledComponents` is a pointer to an array of XrSpatialComponentTypeEXT.

Applications **can** enable the `XR_SPATIAL_CAPABILITY_ANCHOR_EXT` spatial capability by including a pointer to an XrSpatialCapabilityConfigurationAnchorEXT structure in XrSpatialContextCreateInfoEXT ::capabilityConfigs.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` if `capability` is not `XR_SPATIAL_CAPABILITY_ANCHOR_EXT`.

Valid Usage (Implicit)

- The `XR_EXT_spatial_anchor` extension **must** be enabled prior to using XrSpatialCapabilityConfigurationAnchorEXT

- `type` **must** be `XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_ANCHOR_EXT`

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

- `capability` **must** be a valid XrSpatialCapabilityEXT value

- `enabledComponents` **must** be a pointer to an array of `enabledComponentCount` valid XrSpatialComponentTypeEXT values

- The `enabledComponentCount` parameter **must** be greater than `0`

12.35.5. Guaranteed Components

A runtime that supports `XR_SPATIAL_CAPABILITY_ANCHOR_EXT` **must** provide the following spatial components as guaranteed components of all entities created or discovered by this capability and **must** enumerate them in xrEnumerateSpatialCapabilityComponentTypesEXT:

- `XR_SPATIAL_COMPONENT_TYPE_ANCHOR_EXT`

Anchor Component

Component Data

The `XR_SPATIAL_COMPONENT_TYPE_ANCHOR_EXT` uses XrPosef for its data which provides the position and orientation of the anchor.

Component List Structure to Query Data

The XrSpatialComponentAnchorListEXT structure is defined as:

```
// Provided by XR_EXT_spatial_anchor
typedef struct XrSpatialComponentAnchorListEXT {
    XrStructureType    type;
    void*              next;
    uint32_t           locationCount;
    XrPosef*           locations;
} XrSpatialComponentAnchorListEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain.
- `locationCount` is a `uint32_t` describing the count of elements in the `locations` array.
- `locations` is an array of XrPosef.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` from xrQuerySpatialComponentDataEXT if XrSpatialComponentAnchorListEXT is in the XrSpatialComponentDataQueryResultEXT::next chain but `XR_SPATIAL_COMPONENT_TYPE_ANCHOR_EXT` is not included in XrSpatialComponentDataQueryConditionEXT ::componentTypes.

The runtime **must** return `XR_ERROR_SIZE_INSUFFICIENT` from xrQuerySpatialComponentDataEXT if `locationCount` is less than XrSpatialComponentDataQueryResultEXT::entityIdCountOutput.

Valid Usage (Implicit)

- The `XR_EXT_spatial_anchor` extension **must** be enabled prior to using XrSpatialComponentAnchorListEXT

- `type` **must** be `XR_TYPE_SPATIAL_COMPONENT_ANCHOR_LIST_EXT`

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

- `locations` **must** be a pointer to an array of `locationCount` XrPosef structures

- The `locationCount` parameter **must** be greater than `0`

Configuration

If `XR_SPATIAL_COMPONENT_TYPE_ANCHOR_EXT` is enumerated in XrSpatialCapabilityComponentTypesEXT::`componentTypes` for some capability, an application **can** enable it by including the enumerant in the XrSpatialCapabilityConfigurationBaseHeaderEXT::`enabledComponents` list of the XrSpatialCapabilityConfigurationBaseHeaderEXT derived structure of the capability that supports this component.

This component does not require any special configuration to be included in the XrSpatialCapabilityConfigurationBaseHeaderEXT::`next` chain.

12.35.6. Creating a Spatial Anchor

The xrCreateSpatialAnchorEXT function is defined as:

```
// Provided by XR_EXT_spatial_anchor
XrResult xrCreateSpatialAnchorEXT(
    XrSpatialContextEXT                         spatialContext,
    const XrSpatialAnchorCreateInfoEXT*         createInfo,
    XrSpatialEntityIdEXT*                       anchorEntityId,
    XrSpatialEntityEXT*                         anchorEntity);
```

> ## Parameter Descriptions
>
> - `spatialContext` is an XrSpatialContextEXT previously created using xrCreateSpatialContextAsyncEXT.
>
> - `createInfo` is a pointer to an XrSpatialAnchorCreateInfoEXT.
>
> - `anchorEntityId` is a pointer to an XrSpatialEntityIdEXT in which the anchor entity's ID is returned.
>
> - `anchorEntity` is a pointer to an XrSpatialEntityEXT in which the anchor entity's handle is returned.

The application **can** create a spatial anchor by using xrCreateSpatialAnchorEXT.

To get updated component data for an anchor, pass the value populated in `anchorEntity` into the XrSpatialUpdateSnapshotCreateInfoEXT::`entities` when creating a snapshot. The application **can** use `anchorEntityId` to uniquely identify this anchor in the XrSpatialComponentDataQueryResultEXT ::`entityIds` array when using xrQuerySpatialComponentDataEXT.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` from xrCreateSpatialAnchorEXT if `XR_SPATIAL_CAPABILITY_ANCHOR_EXT` was not configured for `spatialContext`. See Configuration for how to configure an XrSpatialContextEXT for the `XR_SPATIAL_CAPABILITY_ANCHOR_EXT` capability.

The anchor represented by `anchorEntity` is only valid for the lifetime of `spatialContext`, or until the application calls xrDestroySpatialEntityEXT on it, whichever comes first. Other extensions **may** offer functions to persist this newly created anchor across multiple XrSession or to share it across process boundaries with other applications.

A newly created anchor, until destroyed, **must** be discoverable in its parent spatial context. This means that the runtime **must** include `anchorEntityId` in the snapshot created using xrCreateSpatialDiscoverySnapshotAsyncEXT for `spatialContext` if the anchor matches the discovery criteria set in XrSpatialDiscoverySnapshotCreateInfoEXT. The newly created anchor **may** also be discoverable in other spatial contexts configured with `XR_SPATIAL_CAPABILITY_ANCHOR_EXT`, although with a different XrSpatialEntityIdEXT since a particular XrSpatialEntityIdEXT is unique to its XrSpatialContextEXT.

Valid Usage (Implicit)

- The `XR_EXT_spatial_anchor` extension **must** be enabled prior to calling xrCreateSpatialAnchorEXT

- `spatialContext` **must** be a valid XrSpatialContextEXT handle

- `createInfo` **must** be a pointer to a valid XrSpatialAnchorCreateInfoEXT structure

- `anchorEntityId` **must** be a pointer to an XrSpatialEntityIdEXT value

- `anchorEntity` **must** be a pointer to an XrSpatialEntityEXT handle

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`

- `XR_ERROR_TIME_INVALID`

- `XR_ERROR_POSE_INVALID`

The XrSpatialAnchorCreateInfoEXT structure is defined as:

```
// Provided by XR_EXT_spatial_anchor
typedef struct XrSpatialAnchorCreateInfoEXT {
    XrStructureType    type;
    const void*        next;
    XrSpace            baseSpace;
    XrTime             time;
    XrPosef            pose;
} XrSpatialAnchorCreateInfoEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `baseSpace` is the XrSpace in which `pose` is applied.

- `time` is the XrTime at which `baseSpace` is located (and `pose` is applied).

- `pose` is the location for the anchor entity.

Valid Usage (Implicit)

- The `XR_EXT_spatial_anchor` extension **must** be enabled prior to using XrSpatialAnchorCreateInfoEXT

- `type` **must** be `XR_TYPE_SPATIAL_ANCHOR_CREATE_INFO_EXT`

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

- `baseSpace` **must** be a valid XrSpace handle

12.35.7. Query Anchor Pose

After the anchor is created, the runtime **should** then adjust its position and orientation over time relative to other spaces in order to maintain the best possible alignment to its original real-world location, even if that changes the anchor's relationship to the original XrSpatialAnchorCreateInfoEXT ::`baseSpace` used to initialize it.

The application **can** use xrCreateSpatialUpdateSnapshotEXT with the anchor's XrSpatialEntityEXT to create a new XrSpatialSnapshotEXT and then query the `XR_SPATIAL_COMPONENT_TYPE_ANCHOR_EXT` component from that snapshot using xrQuerySpatialComponentDataEXT. The application **can** add XrSpatialComponentAnchorListEXT to XrSpatialComponentDataQueryResultEXT::`next` to retrieve the latest location data for the anchors.

The runtime **may** set the tracking state of a newly created anchor to

`XR_SPATIAL_ENTITY_TRACKING_STATE_PAUSED_EXT`. The application **must** only read the anchor entity's state provided in XrSpatialComponentDataQueryResultEXT::`entityStates` and the entity's anchor component data if the tracking state is `XR_SPATIAL_ENTITY_TRACKING_STATE_TRACKING_EXT`.

12.35.8. Guidelines For Using Anchors

- Each anchor's pose adjusts independent of any other anchor or space. Separately anchored virtual objects **may** shift or rotate relative to each other, breaking the spatial hierarchy in cases where these virtual objects are expected to stay in place relative to each other. For such cases, the application **should** reuse the same anchor for all virtual objects that do not move relative to each other.

- Application **should** destroy any XrSpatialEntityEXT handles for anchors that are no longer being used in order to free up the resources the runtime **may** be using to track those anchors.

12.35.9. Example Code

Configure Anchor Capability

The following example demonstrates how to configure the anchor capability when creating a spatial context.

```cpp
// Create a spatial spatial context
XrSpatialContextEXT spatialContext{};
{

  std::vector<XrSpatialComponentTypeEXT> enabledComponents = {
    XR_SPATIAL_COMPONENT_TYPE_ANCHOR_EXT,
  };

  XrSpatialCapabilityConfigurationAnchorEXT
anchorConfig{XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_ANCHOR_EXT};
  anchorConfig.capability = XR_SPATIAL_CAPABILITY_ANCHOR_EXT;
  anchorConfig.enabledComponentCount = enabledComponents.size();
  anchorConfig.enabledComponents = enabledComponents.data();

  std::array<XrSpatialCapabilityConfigurationBaseHeaderEXT*, 1> capabilityConfigs = {
    reinterpret_cast<XrSpatialCapabilityConfigurationBaseHeaderEXT*>(&anchorConfig),
  };

  XrSpatialContextCreateInfoEXT
spatialContextCreateInfo{XR_TYPE_SPATIAL_CONTEXT_CREATE_INFO_EXT};
  spatialContextCreateInfo.capabilityConfigCount = capabilityConfigs.size();
  spatialContextCreateInfo.capabilityConfigs = capabilityConfigs.data();
  XrFutureEXT createContextFuture;
  CHK_XR(xrCreateSpatialContextAsyncEXT(session, &spatialContextCreateInfo,
&createContextFuture));

  waitUntilReady(createContextFuture);

  XrCreateSpatialContextCompletionEXT
completion{XR_TYPE_CREATE_SPATIAL_CONTEXT_COMPLETION_EXT};
  CHK_XR(xrCreateSpatialContextCompleteEXT(session, createContextFuture, &completion));
  if (completion.futureResult != XR_SUCCESS) {
    return;
  }

  spatialContext = completion.spatialContext;
}

// ...
// Create spatial anchors and get their latest pose in the frame loop.
// ...

CHK_XR(xrDestroySpatialContextEXT(spatialContext));
```

Create Spatial Anchor & Get Its Location

The following example demonstrates how to create a spatial anchor & gets its pose every frame.

```
XrSpatialAnchorCreateInfoEXT createInfo{XR_TYPE_SPATIAL_ANCHOR_CREATE_INFO_EXT};
createInfo.baseSpace = localSpace;
createInfo.time = predictedDisplayTime;
createInfo.pose = {{0, 0, 0, 1}, {1, 1, 1}};

XrSpatialEntityIdEXT spatialAnchorEntityId;
XrSpatialEntityEXT spatialAnchorEntity;
CHK_XR(xrCreateSpatialAnchorEXT(spatialContext, &createInfo, &spatialAnchorEntityId,
&spatialAnchorEntity));

auto updateAnchorLocation = [&](XrTime time) {
  // We want to get updated data for all components of the entities, so skip specifying
componentTypes.
  XrSpatialUpdateSnapshotCreateInfoEXT
snapshotCreateInfo{XR_TYPE_SPATIAL_UPDATE_SNAPSHOT_CREATE_INFO_EXT};
  snapshotCreateInfo.entityCount = 1;
  snapshotCreateInfo.entities = &spatialAnchorEntity;
  snapshotCreateInfo.baseSpace = localSpace;
  snapshotCreateInfo.time = time;

  XrSpatialSnapshotEXT snapshot;
  CHK_XR(xrCreateSpatialUpdateSnapshotEXT(spatialContext, &snapshotCreateInfo,
&snapshot));

  // Query for the entities that have the anchor component on them.
  std::array<XrSpatialComponentTypeEXT, 1> componentsToQuery
{XR_SPATIAL_COMPONENT_TYPE_ANCHOR_EXT};
  XrSpatialComponentDataQueryConditionEXT
queryCond{XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_CONDITION_EXT};
  queryCond.componentTypeCount = componentsToQuery.size();
  queryCond.componentTypes = componentsToQuery.data();

  XrSpatialComponentDataQueryResultEXT
queryResult{XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_RESULT_EXT};
  CHK_XR(xrQuerySpatialComponentDataEXT(snapshot, &queryCond, &queryResult));

  std::vector<XrSpatialEntityIdEXT> entityIds(queryResult.entityIdCountOutput);
  std::vector<XrSpatialEntityTrackingStateEXT>
entityStates(queryResult.entityIdCountOutput);
  queryResult.entityIdCapacityInput = entityIds.size();
  queryResult.entityIds = entityIds.data();
  queryResult.entityStateCapacityInput = entityStates.size();
  queryResult.entityStates = entityStates.data();
```

```
  // query for the pose data
  std::vector<XrPosef> locations(queryResult.entityIdCountOutput);
  XrSpatialComponentAnchorListEXT
locationList{XR_TYPE_SPATIAL_COMPONENT_ANCHOR_LIST_EXT};
  locationList.locationCount = locations.size();
  locationList.locations = locations.data();
  queryResult.next = &locationList;

  CHK_XR(xrQuerySpatialComponentDataEXT(snapshot, &queryCond, &queryResult));

  for (int32_t i = 0; i < queryResult.entityIdCountOutput; ++i) {
    if (entityStates[i] == XR_SPATIAL_ENTITY_TRACKING_STATE_TRACKING_EXT) {
      // Pose for entity entityIds[i] is locations[i].
    }
  }

  CHK_XR(xrDestroySpatialSnapshotEXT(snapshot));
};


while (1) {
  // ...
  // For every frame in frame loop
  // ...

  XrFrameState frameState;  // previously returned from xrWaitFrame
  const XrTime time = frameState.predictedDisplayTime;

  updateAnchorLocation(time);

  // ...
  // Finish frame loop
  // ...
}

CHK_XR(xrDestroySpatialEntityEXT(spatialAnchorEntity));
```

12.35.10. New Commands

- xrCreateSpatialAnchorEXT

12.35.11. New Structures

- XrSpatialAnchorCreateInfoEXT

- XrSpatialCapabilityConfigurationAnchorEXT

- Extending XrSpatialComponentDataQueryResultEXT:
  - XrSpatialComponentAnchorListEXT

12.35.12. New Enum Constants

- `XR_EXT_SPATIAL_ANCHOR_EXTENSION_NAME`

- `XR_EXT_spatial_anchor_SPEC_VERSION`

- Extending XrSpatialCapabilityEXT:
  - `XR_SPATIAL_CAPABILITY_ANCHOR_EXT`

- Extending XrSpatialComponentTypeEXT:
  - `XR_SPATIAL_COMPONENT_TYPE_ANCHOR_EXT`

- Extending XrStructureType:
  - `XR_TYPE_SPATIAL_ANCHOR_CREATE_INFO_EXT`

  - `XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_ANCHOR_EXT`

  - `XR_TYPE_SPATIAL_COMPONENT_ANCHOR_LIST_EXT`

12.35.13. Issues

12.35.14. Version History

- Revision 1, 2024-07-10 (Nihav Jain, Google)
  - Initial extension description

12.36. XR_EXT_spatial_entity

Name String

> `XR_EXT_spatial_entity`

Extension Type

> Instance extension

Registered Extension Number

> 741

Revision

> 1

Ratification Status

> Ratified

Extension and Version Dependencies

> `XR_EXT_future`

Contributors

> Nihav Jain, Google
> Jared Finder, Google
> Natalie Fleury, Meta
> Yuichi Taguchi, Meta
> Ron Bessems, Meta
> Yin Li, Microsoft
> Karthik Kadappan, Magic Leap
> Jimmy Alamparambil, ByteDance
> Zhipeng Liu, ByteDance
> Jun Yan, ByteDance

12.36.1. Overview

This extension introduces the concepts and foundations for scene understanding and spatial reasoning in OpenXR. This unifies several related but distinct areas of functionality, which are enumerated, configured, and interacted with in a broadly uniform way as defined by this extension. As this extension lacks concrete definitions of any one of these functional areas, the formal specification text tends to be somewhat abstract. Examples included in this extension specification text refers at times to functionality defined in a forthcoming or hypothetical related extension for the purpose of illustration, without inherently limiting or specifying such additional functionality.

The broad pieces of this extension are the following:

- Spatial entities: The functionality is centered around *entities*, which provide very little functionality on their own.

- Spatial components: These entities have *components* associated with them that provide data and behaviors.

- Spatial component types: Each spatial component is of a specific *component type*, and any given *entity* has at most a single *component* of any given *component type*.

- Spatial context: All spatial entity interaction occurs in a *context* after an initialization and configuration phase.

- Spatial capabilities: Spatial entity manipulation is broadly provided by *capabilities*. A capability is some unit of functionality, for example (without limitation) application-defined anchors, plane detection, or image tracking. Each *capability* is typically defined in a separate extension (enabled at instance creation as usual) and is enabled for a specific *context* at the time of creation.

- Each *capability* is associated with a set of *component types* for which *components* are present on every entity exposed by that *capability*. The extension defining a *capability* specifies which *component types* are mandatory for the capability ("guaranteed"), while that same extension or others **may** specify **optional** *component types* provided by some potential implementations. Any

number of *capabilities* might provide entities with components of a given *component type*, which are uniformly usable no matter the *capability* that produced it.

- Spatial capability features: Further, some capabilities require configuration, and thus are parameterized by *capability features*.

This extension provides a mechanism for enumerating the *components* provided by each *capability* supported on the current system, both the mandatory and any **optional** *components*.

As some implementations **may** require different degrees of parameterization for *capabilities*, this extension provides a mechanism for enumerating the supported *capability features* associated with a given *capability* in the current system.

This extension also defines several common *components* expected to be used across a wide range of *capabilities*.

12.36.2. Spatial Entity

Spatial entities are entities that exist in some space, that have various associated data organized into components. They **may** be any of the following:

- Physical (e.g. planar surfaces like walls and floors, objects like chairs and bookcases, etc.)

- Virtual (e.g. content placed and shared by another application or user),

- App-defined (e.g. application marking an area as the "living room" or "kitchen", or marking a point as the location to place the TV etc.)

Things which are exposed via the action system, like controllers or eye gaze, are not intended to be modeled as spatial entities.

Spatial entities in OpenXR are modeled as an *Entity-Component system*. Each spatial entity has a set of components, and each component provides a unique set of data and behaviors for that entity.

Spatial entities are represented by either an `XrSpatialEntityIdEXT` atom or an `XrSpatialEntityEXT` handle, details of which are provided in the Spatial Entity Representations section.

12.36.3. Spatial Component Types

A spatial entity has one or more components which provide data or behaviors for that entity. See Common Components for some common components defined by this extension.

```
// Provided by XR_EXT_spatial_entity
typedef enum XrSpatialComponentTypeEXT {
    XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT = 1,
    XR_SPATIAL_COMPONENT_TYPE_BOUNDED_3D_EXT = 2,
    XR_SPATIAL_COMPONENT_TYPE_PARENT_EXT = 3,
    XR_SPATIAL_COMPONENT_TYPE_MESH_3D_EXT = 4,
  // Provided by XR_EXT_spatial_plane_tracking
    XR_SPATIAL_COMPONENT_TYPE_PLANE_ALIGNMENT_EXT = 1000741000,
  // Provided by XR_EXT_spatial_plane_tracking
    XR_SPATIAL_COMPONENT_TYPE_MESH_2D_EXT = 1000741001,
  // Provided by XR_EXT_spatial_plane_tracking
    XR_SPATIAL_COMPONENT_TYPE_POLYGON_2D_EXT = 1000741002,
  // Provided by XR_EXT_spatial_plane_tracking
    XR_SPATIAL_COMPONENT_TYPE_PLANE_SEMANTIC_LABEL_EXT = 1000741003,
  // Provided by XR_EXT_spatial_marker_tracking
    XR_SPATIAL_COMPONENT_TYPE_MARKER_EXT = 1000743000,
  // Provided by XR_EXT_spatial_anchor
    XR_SPATIAL_COMPONENT_TYPE_ANCHOR_EXT = 1000762000,
  // Provided by XR_EXT_spatial_persistence
    XR_SPATIAL_COMPONENT_TYPE_PERSISTENCE_EXT = 1000763000,
    XR_SPATIAL_COMPONENT_TYPE_MAX_ENUM_EXT = 0x7FFFFFFF
} XrSpatialComponentTypeEXT;
```

The XrSpatialComponentTypeEXT enumeration identifies the different types of components that the runtime **may** support.

Not all component types listed are provided by this extension on its own: some require additional extensions to be enabled at instance creation time, as documented.

The enumerants have the following values:

Enum	Description
XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT	Component that provides the 2D bounds for a spatial entity. Corresponding list structure is XrSpatialComponentBounded2DListEXT; Corresponding data structure is XrSpatialBounded2DDataEXT
XR_SPATIAL_COMPONENT_TYPE_BOUNDED_3D_EXT	Component that provides the 3D bounds for a spatial entity. Corresponding list structure is XrSpatialComponentBounded3DListEXT; Corresponding data structure is XrBoxf

Enum	Description
XR_SPATIAL_COMPONENT_TYPE_PARENT_EXT	Component that provides the XrSpatialEntityIdEXT of the parent for a spatial entity. Corresponding list structure is XrSpatialComponentParentListEXT; Corresponding data structure is XrSpatialEntityIdEXT
XR_SPATIAL_COMPONENT_TYPE_MESH_3D_EXT	Component that provides a 3D mesh for a spatial entity. Corresponding list structure is XrSpatialComponentMesh3DListEXT; Corresponding data structure is XrSpatialMeshDataEXT
XR_SPATIAL_COMPONENT_TYPE_PLANE_ALIGNMENT_EXT	Component that provides the plane alignment enum for a spatial entity. Corresponding list structure is XrSpatialComponentPlaneAlignmentListEXT; Corresponding data structure is XrSpatialPlaneAlignmentEXT (Added by the XR_EXT_spatial_plane_tracking extension)
XR_SPATIAL_COMPONENT_TYPE_MESH_2D_EXT	Component that provides a 2D mesh for a spatial entity. Corresponding list structure is XrSpatialComponentMesh2DListEXT; Corresponding data structure is XrSpatialMeshDataEXT (Added by the XR_EXT_spatial_plane_tracking extension)
XR_SPATIAL_COMPONENT_TYPE_POLYGON_2D_EXT	Component that provides a 2D boundary polygon for a spatial entity. Corresponding list structure is XrSpatialComponentPolygon2DListEXT; Corresponding data structure is XrSpatialPolygon2DDataEXT (Added by the XR_EXT_spatial_plane_tracking extension)
XR_SPATIAL_COMPONENT_TYPE_PLANE_SEMANTIC_LABEL_EXT	Component that provides a semantic label for a plane. Corresponding list structure is XrSpatialComponentPlaneSemanticLabelListEXT; Corresponding data structure is XrSpatialPlaneSemanticLabelEXT (Added by the XR_EXT_spatial_plane_tracking extension)

Enum	Description
`XR_SPATIAL_COMPONENT_TYPE_MARKER_EXT`	A component describing the marker type, id and location. Corresponding list structure is XrSpatialComponentMarkerListEXT; Corresponding data structure is XrSpatialMarkerDataEXT (Added by the `XR_EXT_spatial_marker_tracking` extension)
`XR_SPATIAL_COMPONENT_TYPE_ANCHOR_EXT`	Component that provides the location for an anchor. Corresponding list structure is XrSpatialComponentAnchorListEXT; Corresponding data structure is XrPosef (Added by the `XR_EXT_spatial_anchor` extension)
`XR_SPATIAL_COMPONENT_TYPE_PERSISTENCE_EXT`	Component that provides the persisted UUID for a spatial entity. Corresponding list structure is XrSpatialComponentPersistenceListEXT; Corresponding data structure is XrSpatialPersistenceDataEXT (Added by the `XR_EXT_spatial_persistence` extension)

12.36.4. Spatial Capabilities and Setup

Spatial capabilities define a runtime's abilities to discover entities that have a guaranteed set of components on them. Applications enable the components of a spatial capability when creating the XrSpatialContextEXT, and the runtime in turn **must** provide only the enabled components on discovered entities. e.g. If a runtime reports that one of the components for a given capability is "semantic labels", it means the application **can** enable semantic labels via the configuration for that capability and the runtime **must** only provide the semantic label component if it is configured.

```
// Provided by XR_EXT_spatial_entity
typedef enum XrSpatialCapabilityEXT {
  // Provided by XR_EXT_spatial_plane_tracking
    XR_SPATIAL_CAPABILITY_PLANE_TRACKING_EXT = 1000741000,
  // Provided by XR_EXT_spatial_marker_tracking
    XR_SPATIAL_CAPABILITY_MARKER_TRACKING_QR_CODE_EXT = 1000743000,
  // Provided by XR_EXT_spatial_marker_tracking
    XR_SPATIAL_CAPABILITY_MARKER_TRACKING_MICRO_QR_CODE_EXT = 1000743001,
  // Provided by XR_EXT_spatial_marker_tracking
    XR_SPATIAL_CAPABILITY_MARKER_TRACKING_ARUCO_MARKER_EXT = 1000743002,
  // Provided by XR_EXT_spatial_marker_tracking
    XR_SPATIAL_CAPABILITY_MARKER_TRACKING_APRIL_TAG_EXT = 1000743003,
  // Provided by XR_EXT_spatial_anchor
    XR_SPATIAL_CAPABILITY_ANCHOR_EXT = 1000762000,
    XR_SPATIAL_CAPABILITY_MAX_ENUM_EXT = 0x7FFFFFFF
} XrSpatialCapabilityEXT;
```

The XrSpatialCapabilityEXT enumeration identifies the different types of capabilities that the runtime **may** support.

The xrEnumerateSpatialCapabilitiesEXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrEnumerateSpatialCapabilitiesEXT(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    uint32_t                                    capabilityCapacityInput,
    uint32_t*                                   capabilityCountOutput,
    XrSpatialCapabilityEXT*                     capabilities);
```

Parameter Descriptions

- `instance` is a handle to an XrInstance.

- `systemId` is the XrSystemId whose spatial capabilities will be enumerated.

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

- `capabilityCountOutput` is the number of capabilities, or the required capacity in the case that `capabilityCapacityInput` is insufficient.

- `capabilities` is an array of XrSpatialCapabilityEXT. It **can** be `NULL` if `capabilityCapacityInput` is 0.

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

The application **can** enumerate the list of spatial capabilities supported by a given XrSystemId using xrEnumerateSpatialCapabilitiesEXT.

The runtime **must** not enumerate the spatial capabilities whose extension is not enabled for `instance`.

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to calling xrEnumerateSpatialCapabilitiesEXT

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

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

- If `capabilityCapacityInput` is not 0, `capabilities` **must** be a pointer to an array of capabilityCapacityInput XrSpatialCapabilityEXT values

The xrEnumerateSpatialCapabilityComponentTypesEXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrEnumerateSpatialCapabilityComponentTypesEXT(
    XrInstance                              instance,
    XrSystemId                              systemId,
    XrSpatialCapabilityEXT                  capability,
    XrSpatialCapabilityComponentTypesEXT*   capabilityComponents);
```

Parameter Descriptions

- `instance` is a handle to an XrInstance.

- `systemId` is the XrSystemId whose spatial capability components will be enumerated.

- `capability` is the XrSpatialCapabilityEXT for which the components will be enumerated.

- `capabilityComponents` is a pointer to an XrSpatialCapabilityComponentTypesEXT, which is an input/output structure in which an application-allocated array is populated.

This function enumerates the component types that the given capability provides on its entities in the system as configured.

The application **can** use the component types enumerated in XrSpatialCapabilityComponentTypesEXT::`componentTypes` to understand the full set of components that the `systemId` supports for `capability` and **can** use this list to determine what a valid configuration for `capability` is when creating an

[XrSpatialContextEXT](#) for it.

The runtime **must** return `XR_ERROR_SPATIAL_CAPABILITY_UNSUPPORTED_EXT` if `capability` is not enumerated by [xrEnumerateSpatialCapabilitiesEXT](#).

The runtime **must** not enumerate the spatial component types whose extension is not enabled for `instance`.

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to calling [xrEnumerateSpatialCapabilityComponentTypesEXT](#)

- `instance` **must** be a valid [XrInstance](#) handle

- `capability` **must** be a valid [XrSpatialCapabilityEXT](#) value

- `capabilityComponents` **must** be a pointer to an [XrSpatialCapabilityComponentTypesEXT](#) 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_SIZE_INSUFFICIENT`

- `XR_ERROR_SYSTEM_INVALID`

- `XR_ERROR_SPATIAL_CAPABILITY_UNSUPPORTED_EXT`

The [XrSpatialCapabilityComponentTypesEXT](#) structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrSpatialCapabilityComponentTypesEXT {
    XrStructureType             type;
    void*                       next;
    uint32_t                    componentTypeCapacityInput;
    uint32_t                    componentTypeCountOutput;
    XrSpatialComponentTypeEXT*  componentTypes;
} XrSpatialCapabilityComponentTypesEXT;
```

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.

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

- `componentTypeCountOutput` is the number of component types, or the required capacity in the case that `componentTypeCapacityInput` is insufficient.

- `componentTypes` is an array of XrSpatialComponentTypeEXT. It **can** be `NULL` if `componentTypeCapacityInput` is 0.

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

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to using XrSpatialCapabilityComponentTypesEXT

- `type` **must** be `XR_TYPE_SPATIAL_CAPABILITY_COMPONENT_TYPES_EXT`

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

- If `componentTypeCapacityInput` is not `0`, `componentTypes` **must** be a pointer to an array of `componentTypeCapacityInput` XrSpatialComponentTypeEXT values

```
// Provided by XR_EXT_spatial_entity
typedef enum XrSpatialCapabilityFeatureEXT {
  // Provided by XR_EXT_spatial_marker_tracking
    XR_SPATIAL_CAPABILITY_FEATURE_MARKER_TRACKING_FIXED_SIZE_MARKERS_EXT = 1000743000,
  // Provided by XR_EXT_spatial_marker_tracking
    XR_SPATIAL_CAPABILITY_FEATURE_MARKER_TRACKING_STATIC_MARKERS_EXT = 1000743001,
    XR_SPATIAL_CAPABILITY_FEATURE_MAX_ENUM_EXT = 0x7FFFFFFF
} XrSpatialCapabilityFeatureEXT;
```

Some capabilities have parameters exposed to the application to configure how the component data is computed by the runtime. These dimensions of parameterization/configurability are known as capability features. E.g. for an image tracking capability, a runtime **may** support a feature for the application to specify whether the tracked images are stationary or not.

Providing this information to the runtime via a configuration structure **must** not change the set of component types present on the associated entities, e.g. on the tracked image. However, the runtime **may** be able to optimize e.g. the tracking abilities of the image tracking capability and provide a better experience to the application.

Such features are represented by XrSpatialCapabilityFeatureEXT and the application enumerates them by using xrEnumerateSpatialCapabilityFeaturesEXT.

Each capability feature has a corresponding configuration structure to enable it. Such configuration structures **must** be chained to XrSpatialCapabilityConfigurationBaseHeaderEXT::next of the corresponding capability.

The xrEnumerateSpatialCapabilityFeaturesEXT function is defines as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrEnumerateSpatialCapabilityFeaturesEXT(
    XrInstance                                  instance,
    XrSystemId                                  systemId,
    XrSpatialCapabilityEXT                      capability,
    uint32_t                                    capabilityFeatureCapacityInput,
    uint32_t*                                   capabilityFeatureCountOutput,
    XrSpatialCapabilityFeatureEXT*              capabilityFeatures);
```

- `instance` is a handle to an XrInstance.

- `systemId` is the XrSystemId whose spatial capability components will be enumerated.

- `capability` is the XrSpatialCapabilityEXT for which the features will be enumerated.

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

- `capabilityFeatureCountOutput` is the number of features, or the required capacity in the case that `capabilityFeatureCapacityInput` is insufficient.

- `capabilityFeatures` is an array of XrSpatialCapabilityFeatureEXT. It **can** be `NULL` if `capabilityFeatureCapacityInput` is 0.

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

The application discovers the features supported by a given system for a XrSpatialCapabilityEXT by using xrEnumerateSpatialCapabilityFeaturesEXT.

For capabilities that have features exposed, the application selects the feature or features to enable and provides the corresponding configuration structure in the next chain of the capability configuration structures in XrSpatialContextCreateInfoEXT::`capabilityConfigs`.

If `capability` is not a capability enumerated by xrEnumerateSpatialCapabilitiesEXT, the runtime **must** return `XR_ERROR_SPATIAL_CAPABILITY_UNSUPPORTED_EXT`.

The runtime **must** not enumerate the spatial capability features whose extension is not enabled for `instance`.

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to calling xrEnumerateSpatialCapabilityFeaturesEXT

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

- `capability` **must** be a valid XrSpatialCapabilityEXT value

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

- If `capabilityFeatureCapacityInput` is not `0`, `capabilityFeatures` **must** be a pointer to an array of `capabilityFeatureCapacityInput` XrSpatialCapabilityFeatureEXT values

<div style="border: 1px solid #ccc; padding: 1em;">

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

- XR_ERROR_SPATIAL_CAPABILITY_UNSUPPORTED_EXT

</div>

12.36.5. Spatial Context

Create a spatial context

```
// Provided by XR_EXT_spatial_entity
XR_DEFINE_HANDLE(XrSpatialContextEXT)
```

The XrSpatialContextEXT handle represents the resources for discovering and updating some number of spatial entities in the environment of the user. Application **can** use this handle to discover and update spatial entities using other functions in this extension.

The xrCreateSpatialContextAsyncEXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrCreateSpatialContextAsyncEXT(
    XrSession                                   session,
    const XrSpatialContextCreateInfoEXT*        createInfo,
    XrFutureEXT*                                future);
```

<div style="border: 1px solid; padding: 1em;">

Parameter Descriptions

- `session` is an XrSession in which the spatial context will be active.

- `createInfo` is the XrSpatialContextCreateInfoEXT used to specify the spatial context parameters.

- `future` is a pointer to an XrFutureEXT.

</div>

The application **can** create an XrSpatialContextEXT handle by:

- Providing XrSpatialCapabilityConfigurationBaseHeaderEXT derived structures in XrSpatialContextCreateInfoEXT::`capabilityConfigs` to enable capabilities and enable components for that capability.

- Configuring the capabilities themselves with the corresponding configuration structures of its XrSpatialCapabilityFeatureEXT.

The runtime **must** return `XR_ERROR_SPATIAL_CAPABILITY_CONFIGURATION_INVALID_EXT` if XrSpatialContextCreateInfoEXT::capabilityConfigCount is 0. A spatial context handle needs at least one capability.

The runtime **must** return `XR_ERROR_SPATIAL_CAPABILITY_UNSUPPORTED_EXT` if any capability in the XrSpatialContextCreateInfoEXT::`capabilityConfigs` array is not enumerated by xrEnumerateSpatialCapabilitiesEXT.

The runtime **must** return `XR_ERROR_SPATIAL_CAPABILITY_CONFIGURATION_INVALID_EXT` if any XrSpatialCapabilityConfigurationBaseHeaderEXT::`enabledComponentCount` in XrSpatialContextCreateInfoEXT::`capabilityConfigs` is 0. A capability configuration is incomplete without a list of component types to enable for that capability.

The runtime **must** return `XR_ERROR_SPATIAL_COMPONENT_UNSUPPORTED_FOR_CAPABILITY_EXT` if any component type listed in XrSpatialCapabilityConfigurationBaseHeaderEXT::`enabledComponents` is not enumerated for XrSpatialCapabilityConfigurationBaseHeaderEXT::`capability` in xrEnumerateSpatialCapabilityComponentTypesEXT.

If any of the structures in the next chain of XrSpatialContextCreateInfoEXT::`capabilityConfigs` corresponds to an XrSpatialCapabilityFeatureEXT that is not enumerated for that capability in xrEnumerateSpatialCapabilityFeaturesEXT, the runtime **must** ignore that XrSpatialCapabilityFeatureEXT structure.

The runtime **must** return `XR_ERROR_SPATIAL_CAPABILITY_CONFIGURATION_INVALID_EXT` if XrSpatialContextCreateInfoEXT::`capabilityConfigs` contains multiple structures with the same XrSpatialCapabilityConfigurationBaseHeaderEXT::`capability`.

To ensure optimal use of system resources, the runtime **may** use the configurations provided in XrSpatialContextCreateInfoEXT array to prepare itself for spatial requests to come in. For example, a

runtime that supports plane tracking capability **may** only begin its plane tracking pipeline if a spatial context handle containing the plane tracking capability is created by the application. If the configured capabilities have a long warm-up time, calls to xrCreateSpatialDiscoverySnapshotAsyncEXT **may** result in an empty snapshot. Application **can** wait for XrEventDataSpatialDiscoveryRecommendedEXT before using xrCreateSpatialDiscoverySnapshotAsyncEXT to be sure that the underlying tracking services have warmed up.

If a runtime enforces a permission system to control application access to the spatial capabilities being configured for the XrSpatialContextEXT, then the runtime **must** return `XR_ERROR_PERMISSION_INSUFFICIENT` if those permissions have not been granted to this application.

This function starts an asynchronous operation and creates a corresponding XrFutureEXT, usable with xrPollFutureEXT and related functions. The return value of this function only indicates whether the parameters were acceptable to schedule the asynchronous operation. The corresponding completion function is xrCreateSpatialContextCompleteEXT, usable when a future from this function is in the READY state, with outputs populated by that function in the completion structure XrCreateSpatialContextCompletionEXT.

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to calling xrCreateSpatialContextAsyncEXT

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

- `createInfo` **must** be a pointer to a valid XrSpatialContextCreateInfoEXT structure

- `future` **must** be a pointer to an XrFutureEXT value

The XrSpatialContextCreateInfoEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrSpatialContextCreateInfoEXT {
    XrStructureType                                         type;
    const void*                                             next;
    uint32_t                                                capabilityConfigCount;
    const XrSpatialCapabilityConfigurationBaseHeaderEXT* const*    capabilityConfigs;
} XrSpatialContextCreateInfoEXT;
```

- `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.

- `capabilityConfigCount` is a `uint32_t` describing the count of elements in the `capabilityConfigs` array.

- `capabilityConfigs` is a pointer to an array of XrSpatialCapabilityConfigurationBaseHeaderEXT pointers.

The XrSpatialContextCreateInfoEXT structure describes the information to create an XrSpatialContextEXT handle.

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to using XrSpatialContextCreateInfoEXT

- `type` **must** be `XR_TYPE_SPATIAL_CONTEXT_CREATE_INFO_EXT`

- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrSpatialContextPersistenceConfigEXT

- `capabilityConfigs` **must** be a pointer to an array of `capabilityConfigCount` valid XrSpatialCapabilityConfigurationBaseHeaderEXT-based structures. See also: XrSpatialCapabilityConfigurationAprilTagEXT, XrSpatialCapabilityConfigurationArucoMarkerEXT, XrSpatialCapabilityConfigurationMicroQrCodeEXT, XrSpatialCapabilityConfigurationPlaneTrackingEXT, XrSpatialCapabilityConfigurationQrCodeEXT

- The `capabilityConfigCount` parameter **must** be greater than `0`

The XrSpatialCapabilityConfigurationBaseHeaderEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrSpatialCapabilityConfigurationBaseHeaderEXT {
    XrStructureType                     type;
    const void*                         next;
    XrSpatialCapabilityEXT              capability;
    uint32_t                            enabledComponentCount;
    const XrSpatialComponentTypeEXT*    enabledComponents;
} XrSpatialCapabilityConfigurationBaseHeaderEXT;
```

This structure is not directly used in the API but instead its child structures **can** be used with XrSpatialContextCreateInfoEXT to configure spatial capabilities.

Member Descriptions

- type is the XrStructureType of this structure.

- next is NULL or a pointer to the next structure in a structure chain.

- capability is an XrSpatialCapabilityEXT.

- enabledComponentCount is a uint32_t describing the count of elements in the enabledComponents array.

- enabledComponents is a pointer to an array of XrSpatialComponentTypeEXT.

The runtime **must** return XR_ERROR_SPATIAL_CAPABILITY_UNSUPPORTED_EXT if capability is not enumerated by xrEnumerateSpatialCapabilitiesEXT. The runtime **must** return XR_ERROR_SPATIAL_COMPONENT_UNSUPPORTED_FOR_CAPABILITY_EXT if any component type listed in enabledComponents is not enumerated for capability in xrEnumerateSpatialCapabilityComponentTypesEXT.

Valid Usage (Implicit)

- The XR_EXT_spatial_entity extension **must** be enabled prior to using XrSpatialCapabilityConfigurationBaseHeaderEXT

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

- capability **must** be a valid XrSpatialCapabilityEXT value

- enabledComponents **must** be a pointer to an array of enabledComponentCount valid XrSpatialComponentTypeEXT values

- The enabledComponentCount parameter **must** be greater than 0

The xrCreateSpatialContextCompleteEXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrCreateSpatialContextCompleteEXT(
    XrSession                               session,
    XrFutureEXT                             future,
    XrCreateSpatialContextCompletionEXT*    completion);
```

Parameter Descriptions

- session is the XrSession previously passed to xrCreateSpatialContextAsyncEXT::session.

- future is the XrFutureEXT received from xrCreateSpatialContextAsyncEXT.

- completion is a pointer to an XrCreateSpatialContextCompletionEXT.

xrCreateSpatialContextCompleteEXT completes the asynchronous operation started by xrCreateSpatialContextAsyncEXT. The runtime **must** return XR_ERROR_FUTURE_PENDING_EXT if future is not in ready state. The runtime **must** return XR_ERROR_FUTURE_INVALID_EXT if future has already been completed or cancelled.

Valid Usage (Implicit)

- The XR_EXT_spatial_entity extension **must** be enabled prior to calling xrCreateSpatialContextCompleteEXT

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

- completion **must** be a pointer to an XrCreateSpatialContextCompletionEXT 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_OUT_OF_MEMORY

- XR_ERROR_LIMIT_REACHED

- XR_ERROR_FUTURE_PENDING_EXT

- XR_ERROR_FUTURE_INVALID_EXT

The XrCreateSpatialContextCompletionEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrCreateSpatialContextCompletionEXT {
    XrStructureType        type;
    void*                  next;
    XrResult               futureResult;
    XrSpatialContextEXT    spatialContext;
} XrCreateSpatialContextCompletionEXT;
```

Member Descriptions

- type is the XrStructureType of this structure.

- next is NULL or a pointer to the next structure in a structure chain.

- futureResult is the XrResult of the spatial context creation operation.

- spatialContext is an XrSpatialContextEXT created using the data and configuration in xrCreateSpatialContextAsyncEXT::createInfo.

`futureResult` values:

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_OUT_OF_MEMORY`
- `XR_ERROR_LIMIT_REACHED`
- `XR_ERROR_SPATIAL_CAPABILITY_CONFIGURATION_INVALID_EXT`

If `futureResult` is a success code, `spatialContext` **must** be valid. If `spatialContext` is valid, it remains so only within the lifecycle of xrCreateSpatialContextAsyncEXT::`session` or until the application destroys the `spatialContext` with xrDestroySpatialContextEXT, whichever comes first.

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to using XrCreateSpatialContextCompletionEXT
- `type` **must** be `XR_TYPE_CREATE_SPATIAL_CONTEXT_COMPLETION_EXT`
- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain
- `futureResult` **must** be a valid XrResult value
- If `spatialContext` is not `XR_NULL_HANDLE`, `spatialContext` **must** be a valid XrSpatialContextEXT handle

Destroy the spatial context

The xrDestroySpatialContextEXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrDestroySpatialContextEXT(
    XrSpatialContextEXT                          spatialContext);
```

<div style="text-align:center">

Parameter Descriptions

</div>

- spatialContext is an XrSpatialContextEXT previously created by xrCreateSpatialContextAsyncEXT.

The application **can** call xrDestroySpatialContextEXT function to release the spatialContext handle and the underlying resources when finished with spatial entity discovery and update tasks. If there is no other valid XrSpatialContextEXT that was created with the same spatial capabilities as spatialContext, this call serves as a suggestion to the runtime to disable the tracking services required for those capabilities to save system resources.

<div style="text-align:center">

Valid Usage (Implicit)

</div>

- The XR_EXT_spatial_entity extension **must** be enabled prior to calling xrDestroySpatialContextEXT
- spatialContext **must** be a valid XrSpatialContextEXT handle

<div style="text-align:center">

Thread Safety

</div>

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

<div style="text-align:center">

Return Codes

</div>

Success

- XR_SUCCESS

Failure

- XR_ERROR_FUNCTION_UNSUPPORTED
- XR_ERROR_RUNTIME_FAILURE
- XR_ERROR_HANDLE_INVALID

12.36.6. Spatial Entity Representations

Spatial Entity ID

```
// Provided by XR_EXT_spatial_entity
XR_DEFINE_ATOM(XrSpatialEntityIdEXT)
```

XrSpatialEntityIdEXT is used to represent any kind of entity discovered by the runtime in the spatial environment of the user. An XrSpatialEntityIdEXT is valid for the XrSpatialContextEXT in which it is discovered, and the runtime **must** not reuse the same XrSpatialEntityIdEXT for different entities within the same XrSpatialContextEXT. Also, the runtime **must** not reuse the same XrSpatialEntityIdEXT across multiple XrSpatialContextEXT within the same XrSession regardless of whether it represents the same entity or different ones.

```
// Provided by XR_EXT_spatial_entity
#define XR_NULL_SPATIAL_ENTITY_ID_EXT 0
```

XR_NULL_SPATIAL_ENTITY_ID_EXT is a reserved value representing an invalid XrSpatialEntityIdEXT. It **may** be passed to and returned from API functions only when specifically allowed.

Spatial Entity Handle

```
// Provided by XR_EXT_spatial_entity
XR_DEFINE_HANDLE(XrSpatialEntityEXT)
```

The XrSpatialEntityEXT handle represents a spatial entity. An application **can** create such a handle to express its interest in a specific entity to the runtime.

Create Spatial Entity Handle from ID

The xrCreateSpatialEntityFromIdEXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrCreateSpatialEntityFromIdEXT(
    XrSpatialContextEXT                         spatialContext,
    const XrSpatialEntityFromIdCreateInfoEXT*   createInfo,
    XrSpatialEntityEXT*                         spatialEntity);
```

The application **can** use xrCreateSpatialEntityFromIdEXT to create an XrSpatialEntityEXT handle which is a reference to an entity that exists in the user's environment.

The runtime **must** return `XR_ERROR_SPATIAL_ENTITY_ID_INVALID_EXT` if XrSpatialEntityFromIdCreateInfoEXT::`entityId` is not a valid ID for `spatialContext`.

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

- XR_ERROR_SPATIAL_ENTITY_ID_INVALID_EXT

The XrSpatialEntityFromIdCreateInfoEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrSpatialEntityFromIdCreateInfoEXT {
    XrStructureType        type;
    const void*            next;
    XrSpatialEntityIdEXT   entityId;
} XrSpatialEntityFromIdCreateInfoEXT;
```

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.

- entityId is the XrSpatialEntityIdEXT of the entity that the application wants to create a handle for.

Destroy Spatial Entity Handle

The xrDestroySpatialEntityEXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrDestroySpatialEntityEXT(
    XrSpatialEntityEXT                          spatialEntity);
```

The application **can** use xrDestroySpatialEntityEXT to release the `spatialEntity` handle when it is no longer interested in the entity referenced by this handle.

12.36.7. Spatial Snapshot

```
// Provided by XR_EXT_spatial_entity
XR_DEFINE_HANDLE(XrSpatialSnapshotEXT)
```

The application **can** create spatial snapshots for the purpose of discovering spatial entities or for updating its information about known spatial entities. The XrSpatialSnapshotEXT handle represents the immutable data for the discovered or updated spatial entities and a subset of their components as selected by the application. The spatial snapshot represents a coherent view of the entities and their component data. Once a snapshot is created, the snapshot's data **must** remain constant while the snapshot is valid.

The application **can** create any number of snapshots it wants but **must** be mindful of the memory being allocated for each new snapshot and **must** destroy the snapshots once it no longer needs them.

Create discovery snapshot

The xrCreateSpatialDiscoverySnapshotAsyncEXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrCreateSpatialDiscoverySnapshotAsyncEXT(
    XrSpatialContextEXT                          spatialContext,
    const XrSpatialDiscoverySnapshotCreateInfoEXT* createInfo,
    XrFutureEXT*                                 future);
```

The application **can** discover spatial entities by creating a discovery snapshot by using xrCreateSpatialDiscoverySnapshotAsyncEXT.

This function starts an asynchronous operation and creates a corresponding XrFutureEXT, usable with xrPollFutureEXT and related functions. The return value of this function only indicates whether the parameters were acceptable to schedule the asynchronous operation. The corresponding completion function is xrCreateSpatialDiscoverySnapshotCompleteEXT, usable when a future from this function is in the READY state, with outputs populated by that function in the completion structure XrCreateSpatialDiscoverySnapshotCompletionEXT.

The application **can** submit multiple discovery snapshot creation requests without needing to wait for the previous one to be completed. The runtime **may** process and complete the snapshot creation in any order. The runtime **may** delay the completion of the discovery snapshot creation to throttle the application if it needs to reduce the use of system resources due to power, thermal or other policies of the device.

The application **can** use XrSpatialDiscoverySnapshotCreateInfoEXT::`componentTypes` to filter the list of entities and the components whose data the runtime **must** include in the snapshot. If the application provides a valid list of spatial component types in XrSpatialDiscoverySnapshotCreateInfoEXT ::`componentTypes`, then the runtime **must** only include spatial entities in the snapshot that have at least one of the components provided in XrSpatialDiscoverySnapshotCreateInfoEXT::`componentTypes`. Also, the runtime **must** only include data for only those components in the snapshot.

The runtime **must** return `XR_ERROR_SPATIAL_COMPONENT_NOT_ENABLED_EXT` if any of the XrSpatialComponentTypeEXT in XrSpatialDiscoverySnapshotCreateInfoEXT::`componentTypes` are not enabled for the spatial capabilities passed to XrSpatialContextCreateInfoEXT::`capabilityConfigs` when creating `spatialContext`.

If the application does not provide a list of spatial component types in XrSpatialDiscoverySnapshotCreateInfoEXT::`componentTypes`, the runtime **must** include all the spatial entities in the snapshot that have the set of components which are enumerated in XrSpatialCapabilityConfigurationBaseHeaderEXT::enabledComponents for the capabilities configured for `spatialContext`. The runtime **must** include the data for all the enabled components of the capabilities configured for `spatialContext`.

If XrEventDataReferenceSpaceChangePending is queued before the completion of `future`, and XrEventDataReferenceSpaceChangePending::`poseValid` is false, then the runtime **may** either create an

XrSpatialSnapshotEXT that has no entities in it or set the XrSpatialEntityTrackingStateEXT of the entities that are no longer locatable in XrCreateSpatialDiscoverySnapshotCompletionInfoEXT ::baseSpace at XrCreateSpatialDiscoverySnapshotCompletionInfoEXT::time to XR_SPATIAL_ENTITY_TRACKING_STATE_PAUSED_EXT or XR_SPATIAL_ENTITY_TRACKING_STATE_STOPPED_EXT. The runtime **must** not set XrCreateSpatialContextCompletionEXT::futureResult to an error code because of XrEventDataReferenceSpaceChangePending.

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to calling xrCreateSpatialDiscoverySnapshotAsyncEXT

- `spatialContext` **must** be a valid XrSpatialContextEXT handle

- `createInfo` **must** be a pointer to a valid XrSpatialDiscoverySnapshotCreateInfoEXT structure

- `future` **must** be a pointer to an XrFutureEXT 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

- XR_ERROR_SPATIAL_COMPONENT_NOT_ENABLED_EXT

The XrSpatialDiscoverySnapshotCreateInfoEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrSpatialDiscoverySnapshotCreateInfoEXT {
    XrStructureType                     type;
    const void*                         next;
    uint32_t                            componentTypeCount;
    const XrSpatialComponentTypeEXT*    componentTypes;
} XrSpatialDiscoverySnapshotCreateInfoEXT;
```

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.

- `componentTypeCount` is a `uint32_t` describing the count of elements in the `componentTypes` array.

- `componentTypes` is an array of XrSpatialComponentTypeEXT.

The XrSpatialDiscoverySnapshotCreateInfoEXT structure describes the information to create an XrSpatialSnapshotEXT handle when discovering spatial entities.

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to using XrSpatialDiscoverySnapshotCreateInfoEXT

- `type` **must** be `XR_TYPE_SPATIAL_DISCOVERY_SNAPSHOT_CREATE_INFO_EXT`

- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrSpatialDiscoveryPersistenceUuidFilterEXT, XrSpatialFilterTrackingStateEXT

- If `componentTypeCount` is not `0`, `componentTypes` **must** be a pointer to an array of `componentTypeCount` valid XrSpatialComponentTypeEXT values

The xrCreateSpatialDiscoverySnapshotCompleteEXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrCreateSpatialDiscoverySnapshotCompleteEXT(
    XrSpatialContextEXT                         spatialContext,
    const XrCreateSpatialDiscoverySnapshotCompletionInfoEXT*
createSnapshotCompletionInfo,
    XrCreateSpatialDiscoverySnapshotCompletionEXT* completion);
```

- `spatialContext` is the XrSpatialContextEXT previously passed to xrCreateSpatialDiscoverySnapshotAsyncEXT::`spatialContext`.

- `createSnapshotCompletionInfo` is a pointer to an XrCreateSpatialDiscoverySnapshotCompletionInfoEXT which provides info about the discovery snapshot creation request completion.

- `completion` is a pointer to an XrCreateSpatialDiscoverySnapshotCompletionEXT.

xrCreateSpatialDiscoverySnapshotCompleteEXT completes the asynchronous operation started by xrCreateSpatialDiscoverySnapshotAsyncEXT. The runtime **must** return `XR_ERROR_FUTURE_PENDING_EXT` if XrCreateSpatialDiscoverySnapshotCompletionInfoEXT::`future` is not in ready state. The runtime **must** return `XR_ERROR_FUTURE_INVALID_EXT` if XrCreateSpatialDiscoverySnapshotCompletionInfoEXT::`future` has already been completed or cancelled.

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to calling xrCreateSpatialDiscoverySnapshotCompleteEXT

- `spatialContext` **must** be a valid XrSpatialContextEXT handle

- `createSnapshotCompletionInfo` **must** be a pointer to a valid XrCreateSpatialDiscoverySnapshotCompletionInfoEXT structure

- `completion` **must** be a pointer to an XrCreateSpatialDiscoverySnapshotCompletionEXT 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_OUT_OF_MEMORY

- XR_ERROR_LIMIT_REACHED

- XR_ERROR_TIME_INVALID

- XR_ERROR_FUTURE_PENDING_EXT

- XR_ERROR_FUTURE_INVALID_EXT

The XrCreateSpatialDiscoverySnapshotCompletionInfoEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrCreateSpatialDiscoverySnapshotCompletionInfoEXT {
    XrStructureType    type;
    const void*        next;
    XrSpace            baseSpace;
    XrTime             time;
    XrFutureEXT        future;
} XrCreateSpatialDiscoverySnapshotCompletionInfoEXT;
```

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `baseSpace` is the XrSpace in which all the locations of the discovery XrSpatialSnapshotEXT will be located.

- `time` is the XrTime at which all the locations of the discovery XrSpatialSnapshotEXT will be located.

- `future` is the XrFutureEXT received from xrCreateSpatialDiscoverySnapshotAsyncEXT

The locations in the various component data included in the created snapshot will be represented in `baseSpace`, located at `time`.

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to using XrCreateSpatialDiscoverySnapshotCompletionInfoEXT

- `type` **must** be `XR_TYPE_CREATE_SPATIAL_DISCOVERY_SNAPSHOT_COMPLETION_INFO_EXT`

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

- `baseSpace` **must** be a valid XrSpace handle

The XrCreateSpatialDiscoverySnapshotCompletionEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrCreateSpatialDiscoverySnapshotCompletionEXT {
    XrStructureType         type;
    void*                   next;
    XrResult                futureResult;
    XrSpatialSnapshotEXT    snapshot;
} XrCreateSpatialDiscoverySnapshotCompletionEXT;
```

Discovery Recommendation Event

The XrEventDataSpatialDiscoveryRecommendedEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrEventDataSpatialDiscoveryRecommendedEXT {
    XrStructureType        type;
    const void*            next;
    XrSpatialContextEXT    spatialContext;
} XrEventDataSpatialDiscoveryRecommendedEXT;
```

Member Descriptions

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

- **next** is NULL or a pointer to the next structure in a structure chain.

- **spatialContext** is the XrSpatialContextEXT for which discovery is being recommended by the runtime.

The application **can** retrieve this event by using xrPollEvent. The application **can** avoid excessive calls to xrCreateSpatialDiscoverySnapshotAsyncEXT to discover spatial entities by waiting for this event. If the application creates multiple discovery snapshots with the same XrSpatialDiscoverySnapshotCreateInfoEXT between two XrEventDataSpatialDiscoveryRecommendedEXT events, the resultant snapshots **may** contain the same entities and therefore the snapshot creation and data queries would be wasteful.

Waiting for this event to create a new discovery snapshot ensures that the application is not overloading the system with discovery requests for which the runtime **may** not return any new data and helps avoid the risk of overusing the system resources, and getting throttled due to power or thermal policies of the device. This also helps create parity between runtimes that are discovering spatial entities on the fly with live tracking and runtimes which are providing spatial entities off of a previously recorded state (where the runtime **may** queue the discovery recommendation event only once for each XrSpatialContextEXT).

The runtime **must** not queue this event for notifying the application about changes or adjustments made to the component data of existing spatial entities. The application **can** use the xrCreateSpatialUpdateSnapshotEXT to keep track of component data updates for the spatial entities it is interested in.

A runtime **may** queue a discovery recommendation event without waiting for the application to first call xrCreateSpatialDiscoverySnapshotAsyncEXT. For example, a runtime **may** base the decision of queueing the discovery recommendation event on the configuration of the XrSpatialContextEXT, its own understanding of the environment around the user (discovery of new entities or loss of existing ones), or for hinting an appropriate discovery request cadence to the application so as not to overload the system resources. The runtime **may** choose to never queue this event for an XrSpatialContextEXT if no entities are found in the user's environment throughout the lifetime of that XrSpatialContextEXT.

The runtime **must** not queue this event for a given spatialContext until the application completes its

creation by using xrCreateSpatialContextCompleteEXT.

After the application calls xrDestroySpatialContextEXT, the runtime **must** not queue any more discovery recommendation events for that spatial context nor return any such events for that context from xrPollEvent.

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to using XrEventDataSpatialDiscoveryRecommendedEXT

- `type` **must** be `XR_TYPE_EVENT_DATA_SPATIAL_DISCOVERY_RECOMMENDED_EXT`

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

---

Query Component Data

The xrQuerySpatialComponentDataEXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrQuerySpatialComponentDataEXT(
    XrSpatialSnapshotEXT                       snapshot,
    const XrSpatialComponentDataQueryConditionEXT* queryCondition,
    XrSpatialComponentDataQueryResultEXT*      queryResult);
```

Parameter Descriptions

- `snapshot` is an XrSpatialSnapshotEXT previously provided by xrCreateSpatialDiscoverySnapshotCompleteEXT or xrCreateSpatialUpdateSnapshotEXT.

- `queryCondition` is a pointer to an XrSpatialComponentDataQueryConditionEXT structure.

- `queryResult` is a pointer to an XrSpatialComponentDataQueryResultEXT structure.

The application **can** use xrQuerySpatialComponentDataEXT to query the component data of the entities in the snapshot by attaching a list structure to XrSpatialComponentDataQueryResultEXT::`next` corresponding to each XrSpatialComponentTypeEXT in XrSpatialComponentDataQueryConditionEXT ::`componentTypes`.

If the application attaches a list structure to XrSpatialComponentDataQueryResultEXT::`next` that does not correspond to any of the components listed in XrSpatialComponentDataQueryConditionEXT ::`componentTypes`, the runtime **must** return `XR_ERROR_VALIDATION_FAILURE`.

The application **can** choose to attach the list structures corresponding to only a subset of components listed in XrSpatialComponentDataQueryConditionEXT::componentTypes. The application **can** choose to omit the list structures altogether if it only wishes to know the ids and tracking state of the spatial entities that satisfy the queryCondition. The runtime **must** not treat the absence of list structures from the XrSpatialComponentDataQueryResultEXT::next chain as a failure.

If XrEventDataReferenceSpaceChangePending is queued and XrEventDataReferenceSpaceChangePending::changeTime elapsed while the application is querying component data from an XrSpatialSnapshotEXT, the application **may** use the event data to adjusted poses accordingly.

The runtime **must** populate XrSpatialComponentDataQueryResultEXT::entityIds only with entities that have all the components specified in XrSpatialComponentDataQueryConditionEXT::componentTypes. If XrSpatialComponentDataQueryConditionEXT::componentTypeCount is 0, the runtime **must** populate queryResult with all the entities (and their tracking states) that are in the snapshot. If additional query conditions are added to XrSpatialComponentDataQueryConditionEXT::next, the runtime **must** treat those as an "AND" with the component types availability i.e. the runtime **must** populate XrSpatialComponentDataQueryResultEXT::entityIds only with entities that satisfy all of the provided conditions. The runtime **must** populate the component data in the list structures in the same order as the entities in XrSpatialComponentDataQueryResultEXT::entityIds i.e. the component data at a given index in the list structure array **must** correspond to the entity at the same index.

If the tracking state for an entity is not XR_SPATIAL_ENTITY_TRACKING_STATE_TRACKING_EXT, the runtime **must** not change the data at the index corresponding to that entity in the array contained in the list structures attached to XrSpatialComponentDataQueryResultEXT.

Valid Usage (Implicit)

- The XR_EXT_spatial_entity extension **must** be enabled prior to calling xrQuerySpatialComponentDataEXT
- snapshot **must** be a valid XrSpatialSnapshotEXT handle
- queryCondition **must** be a pointer to a valid XrSpatialComponentDataQueryConditionEXT structure
- queryResult **must** be a pointer to an XrSpatialComponentDataQueryResultEXT structure

---

As an example the application creates an XrSpatialSnapshotEXT which contains 5 entities, where -

- Entity 1 and 2 have components `XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT` and `XR_SPATIAL_COMPONENT_TYPE_PARENT_EXT`
- Entity 3 and 4 have components `XR_SPATIAL_COMPONENT_TYPE_BOUNDED_3D_EXT` and `XR_SPATIAL_COMPONENT_TYPE_MESH_3D_EXT`
- Entity 5 has components `XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT` and `XR_SPATIAL_COMPONENT_TYPE_MESH_3D_EXT`.

XrSpatialEntityIdEXT	1	2	3	4	5
Bounded2D	Yes	Yes	No	No	Yes
Parent	Yes	Yes	No	No	No
Mesh3D	No	No	Yes	Yes	Yes
Bounded3D	No	No	Yes	Yes	No

Figure 15. Example snapshot

xrQuerySpatialComponentDataEXT on the above snapshot with `XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT` listed in the query condition will result in entity #1, #2, and #5 being returned to the application and the application **can** attach an array of

[XrSpatialBounded2DDataEXT](#) as part of the [XrSpatialComponentBounded2DListEXT](#) structure to the next chain of [XrSpatialComponentDataQueryResultEXT](#) to get the bounded2D data.
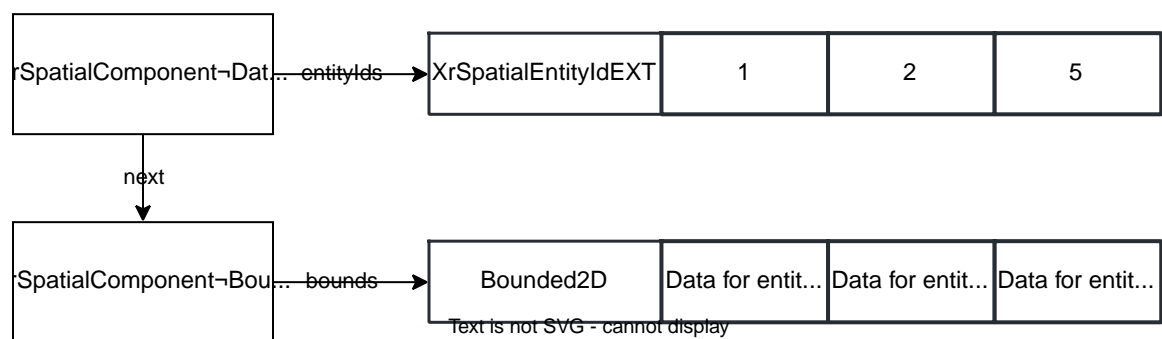
Figure 16. Example query result

[xrQuerySpatialComponentDataEXT](#) on the above snapshot with `XR_SPATIAL_COMPONENT_TYPE_BOUNDED_3D_EXT` and `XR_SPATIAL_COMPONENT_TYPE_MESH_3D_EXT` components listed in the query condition will result in entity #3 and #4 being returned to the application and the application **can** attach arrays of [XrBoxf](#) and [XrSpatialMeshDataEXT](#) as part of the [XrSpatialComponentBounded3DListEXT](#) and [XrSpatialComponentMesh3DListEXT](#) structures respectively to the next chain of [XrSpatialComponentDataQueryResultEXT](#) to get the component data.
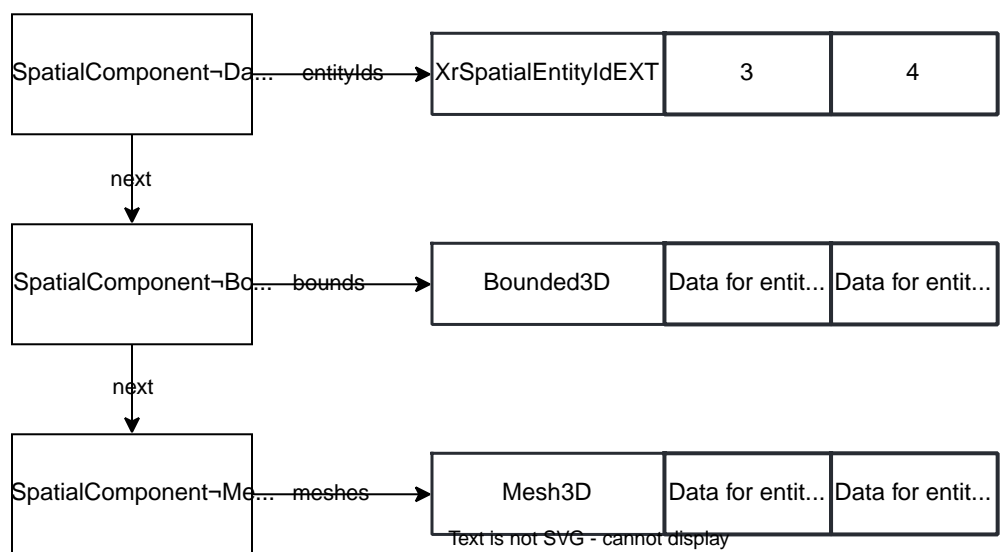
Figure 17. Example query result

The [XrSpatialComponentDataQueryConditionEXT](#) structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrSpatialComponentDataQueryConditionEXT {
    XrStructureType                     type;
    const void*                         next;
    uint32_t                            componentTypeCount;
    const XrSpatialComponentTypeEXT*    componentTypes;
} XrSpatialComponentDataQueryConditionEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `componentTypeCount` is a `uint32_t` describing the count of elements in the `componentTypes` array.

- `componentTypes` is an array of XrSpatialComponentTypeEXT for which to get the data from the snapshot.

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to using XrSpatialComponentDataQueryConditionEXT

- `type` **must** be `XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_CONDITION_EXT`

- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrSpatialDiscoveryPersistenceUuidFilterEXT, XrSpatialFilterTrackingStateEXT

- If `componentTypeCount` is not `0`, `componentTypes` **must** be a pointer to an array of `componentTypeCount` valid XrSpatialComponentTypeEXT values

The XrSpatialComponentDataQueryResultEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrSpatialComponentDataQueryResultEXT {
    XrStructureType                     type;
    void*                               next;
    uint32_t                            entityIdCapacityInput;
    uint32_t                            entityIdCountOutput;
    XrSpatialEntityIdEXT*               entityIds;
    uint32_t                            entityStateCapacityInput;
    uint32_t                            entityStateCountOutput;
    XrSpatialEntityTrackingStateEXT*    entityStates;
} XrSpatialComponentDataQueryResultEXT;
```

Member Descriptions

- type is the XrStructureType of this structure.

- next is NULL or a pointer to the next structure in a structure chain.

- entityIdCapacityInput is the capacity of the entityIds array, or 0 to indicate a request to retrieve the required capacity.

- entityIdCountOutput is the number of XrSpatialEntityIdEXT in entityIds, or the required capacity in the case that entityIdCapacityInput is insufficient.

- entityIds is an array of XrSpatialEntityIdEXT. It **can** be NULL if entityIdCapacityInput is 0.

- entityStateCapacityInput is the capacity of the entityStates array, or 0 to indicate a request to retrieve the required capacity.

- entityStateCountOutput is the number of XrSpatialEntityTrackingStateEXT in entityStates, or the required capacity in the case that entityStateCapacityInput is insufficient. This **must** always be the same as entityIdCountOutput.

- entityStates is an array of XrSpatialEntityTrackingStateEXT. It **can** be NULL if entityStateCapacityInput is 0.

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

An application **can** use the entityIds with xrCreateSpatialEntityFromIdEXT to create XrSpatialEntityEXT handles for the entities it is interested in getting regular updates for. The application **can** then use these XrSpatialEntityEXT handles with xrCreateSpatialUpdateSnapshotEXT to create an update snapshot that has the runtime's latest known data of the components for the provided entities.

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to using XrSpatialComponentDataQueryResultEXT

- `type` **must** be `XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_RESULT_EXT`

- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrSpatialComponentAnchorListEXT, XrSpatialComponentBounded2DListEXT, XrSpatialComponentBounded3DListEXT, XrSpatialComponentMarkerListEXT, XrSpatialComponentMesh2DListEXT, XrSpatialComponentMesh3DListEXT, XrSpatialComponentParentListEXT, XrSpatialComponentPersistenceListEXT, XrSpatialComponentPlaneAlignmentListEXT, XrSpatialComponentPlaneSemanticLabelListEXT, XrSpatialComponentPolygon2DListEXT

- If `entityIdCapacityInput` is not `0`, `entityIds` **must** be a pointer to an array of `entityIdCapacityInput` `XrSpatialEntityIdEXT` values

- If `entityStateCapacityInput` is not `0`, `entityStates` **must** be a pointer to an array of `entityStateCapacityInput` XrSpatialEntityTrackingStateEXT values

```
// Provided by XR_EXT_spatial_entity
typedef enum XrSpatialEntityTrackingStateEXT {
    XR_SPATIAL_ENTITY_TRACKING_STATE_STOPPED_EXT = 1,
    XR_SPATIAL_ENTITY_TRACKING_STATE_PAUSED_EXT = 2,
    XR_SPATIAL_ENTITY_TRACKING_STATE_TRACKING_EXT = 3,
    XR_SPATIAL_ENTITY_TRACKING_STATE_MAX_ENUM_EXT = 0x7FFFFFFF
} XrSpatialEntityTrackingStateEXT;
```

The XrSpatialEntityTrackingStateEXT enumerates the possible spatial entity tracking states:

The enums have the following meanings:

Enum	Description
`XR_SPATIAL_ENTITY_TRACKING_STATE_STOPPED_EXT`	The runtime has stopped tracking this entity and will never resume tracking it.
`XR_SPATIAL_ENTITY_TRACKING_STATE_PAUSED_EXT`	The runtime has paused tracking this entity but **may** resume tracking it in the future.
`XR_SPATIAL_ENTITY_TRACKING_STATE_TRACKING_EXT`	The runtime is currently tracking this entity and its component data is valid.
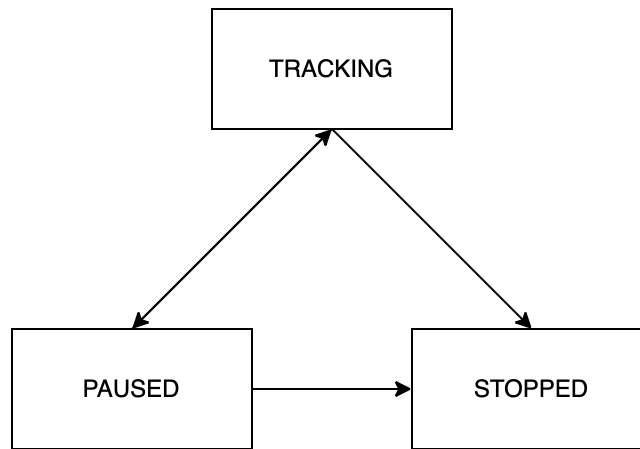
Figure 18. XrSpatialEntityTrackingStateEXT

- The runtime **may** change the state of the spatial entity from `XR_SPATIAL_ENTITY_TRACKING_STATE_TRACKING_EXT` to `XR_SPATIAL_ENTITY_TRACKING_STATE_PAUSED_EXT` if it suspends the tracking of that spatial entity but has the possibility of resuming its tracking in the future. Some examples of when the runtime **may** do this include (but not limited to) if the application loses input focus; or if the given spatial entity is too far from the user to be accurately tracked; or if there are too many entities being tracked and the runtime wants to reduce the cost of tracking. XrSpatialEntityTrackingStateEXT helps the application insulate itself from the different tracking policies of each runtime.

- The runtime **may** change the state of an entity from `XR_SPATIAL_ENTITY_TRACKING_STATE_PAUSED_EXT` to `XR_SPATIAL_ENTITY_TRACKING_STATE_TRACKING_EXT` or `XR_SPATIAL_ENTITY_TRACKING_STATE_STOPPED_EXT`.

- The runtime **must** change the state of the spatial entity from `XR_SPATIAL_ENTITY_TRACKING_STATE_TRACKING_EXT` or `XR_SPATIAL_ENTITY_TRACKING_STATE_PAUSED_EXT` to `XR_SPATIAL_ENTITY_TRACKING_STATE_STOPPED_EXT` if the spatial entity is lost and its tracking will never be recovered or resumed. An example of such a case would be if the device loses tracking, restarts its tracking session but is unable to relocalize in its environment, and therefore treats discovered entities of this tracking session as new entities.

- Once the tracking state of an entity is set to `XR_SPATIAL_ENTITY_TRACKING_STATE_STOPPED_EXT`, the runtime **must** never change it any other state.

- When querying the component data of a spatial entity using xrQuerySpatialComponentDataEXT, the runtime **must** set valid data in the contents of the buffers provided by the application in the next chain of XrSpatialComponentDataQueryResultEXT if the entity state is `XR_SPATIAL_ENTITY_TRACKING_STATE_TRACKING_EXT`. If the entity state is `XR_SPATIAL_ENTITY_TRACKING_STATE_STOPPED_EXT` or `XR_SPATIAL_ENTITY_TRACKING_STATE_PAUSED_EXT`, the runtime **must** not change the content of the buffers.

Two-call idiom for component data

The XrSpatialBufferEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrSpatialBufferEXT {
    XrSpatialBufferIdEXT      bufferId;
    XrSpatialBufferTypeEXT    bufferType;
} XrSpatialBufferEXT;
```

Member Descriptions

- `bufferId` the XrSpatialBufferIdEXT of the buffer data.

- `bufferType` is the XrSpatialBufferTypeEXT to indicate the type of data in `bufferId`. The application **can** use `bufferType` to determine which function to use to retrieve the actual data of the buffer.

Some spatial components have variable-sized data and therefore require using fundamentals-buffer-size-parameters to retrieve their data. In such cases, the spatial component data structure provides an XrSpatialBufferEXT for each variable sized buffer needed in that component's data.

For the same `bufferId`, the runtime **must** provide the same data from one component data query to another, even across one snapshot to another. A different `bufferId` between component data query calls indicates to the application that the data for that component **may** have changed.

Valid Usage (Implicit)

- The XR_EXT_spatial_entity extension **must** be enabled prior to using XrSpatialBufferEXT

- `bufferType` **must** be a valid XrSpatialBufferTypeEXT value

```
// Provided by XR_EXT_spatial_entity
XR_DEFINE_ATOM(XrSpatialBufferIdEXT)
```

XrSpatialBufferIdEXT is used to represent any kind of variable sized data for a spatial component.

The runtime **must** keep the XrSpatialBufferIdEXT and its data in memory for at least the lifecycle of the XrSpatialSnapshotEXT that contains it. The runtime **may** keep the XrSpatialBufferIdEXT and its data in memory for longer than the lifecycle of the XrSpatialSnapshotEXT in order to return the same ID as part of snapshots created later on by the application. For the same XrSpatialBufferIdEXT, the runtime **must** always return the same data via the appropriate `xrGetSpatialBuffer*` function.

```
// Provided by XR_EXT_spatial_entity
typedef enum XrSpatialBufferTypeEXT {
    XR_SPATIAL_BUFFER_TYPE_UNKNOWN_EXT = 0,
    XR_SPATIAL_BUFFER_TYPE_STRING_EXT = 1,
    XR_SPATIAL_BUFFER_TYPE_UINT8_EXT = 2,
    XR_SPATIAL_BUFFER_TYPE_UINT16_EXT = 3,
    XR_SPATIAL_BUFFER_TYPE_UINT32_EXT = 4,
    XR_SPATIAL_BUFFER_TYPE_FLOAT_EXT = 5,
    XR_SPATIAL_BUFFER_TYPE_VECTOR2F_EXT = 6,
    XR_SPATIAL_BUFFER_TYPE_VECTOR3F_EXT = 7,
    XR_SPATIAL_BUFFER_TYPE_MAX_ENUM_EXT = 0x7FFFFFFF
} XrSpatialBufferTypeEXT;
```

The XrSpatialBufferTypeEXT enumeration identifies the different data types of the buffer represented XrSpatialBufferIdEXT.

Enumerant Descriptions

- XR_SPATIAL_BUFFER_TYPE_STRING_EXT. The XrSpatialBufferIdEXT **can** be passed to xrGetSpatialBufferStringEXT to retrieve a string buffer.

- XR_SPATIAL_BUFFER_TYPE_UINT8_EXT. The XrSpatialBufferIdEXT **can** be passed to xrGetSpatialBufferUint8EXT to retrieve a uint8_t buffer.

- XR_SPATIAL_BUFFER_TYPE_UINT16_EXT. The XrSpatialBufferIdEXT **can** be passed to xrGetSpatialBufferUint16EXT to retrieve a uint16_t buffer.

- XR_SPATIAL_BUFFER_TYPE_UINT32_EXT. The XrSpatialBufferIdEXT **can** be passed to xrGetSpatialBufferUint32EXT to retrieve a uint32_t buffer.

- XR_SPATIAL_BUFFER_TYPE_FLOAT_EXT. The XrSpatialBufferIdEXT **can** be passed to xrGetSpatialBufferFloatEXT to retrieve a float buffer.

- XR_SPATIAL_BUFFER_TYPE_VECTOR2F_EXT. The XrSpatialBufferIdEXT **can** be passed to xrGetSpatialBufferVector2fEXT to retrieve an XrVector2f buffer.

- XR_SPATIAL_BUFFER_TYPE_VECTOR3F_EXT. The XrSpatialBufferIdEXT **can** be passed to xrGetSpatialBufferVector3fEXT to retrieve an XrVector3f buffer.

The xrGetSpatialBufferStringEXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrGetSpatialBufferStringEXT(
    XrSpatialSnapshotEXT                        snapshot,
    const XrSpatialBufferGetInfoEXT*            info,
    uint32_t                                    bufferCapacityInput,
    uint32_t*                                   bufferCountOutput,
    char*                                       buffer);
```

Parameter Descriptions

- snapshot is a handle to an XrSpatialSnapshotEXT.

- info holds the information on the buffer to query.

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

- bufferCountOutput is the number of characters, or the required capacity in the case that bufferCapacityInput is insufficient.

- buffer is an array of char. It **can** be NULL if bufferCapacityInput is 0.

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

The application **can** get the data for an XrSpatialBufferEXT provided by a component, where XrSpatialBufferEXT::bufferType is XR_SPATIAL_BUFFER_TYPE_STRING_EXT by using xrGetSpatialBufferStringEXT.

The runtime **must** return XR_ERROR_VALIDATION_FAILURE if the XrSpatialBufferTypeEXT for XrSpatialBufferGetInfoEXT::bufferId is not XR_SPATIAL_BUFFER_TYPE_STRING_EXT.

The runtime **must** return XR_ERROR_SPATIAL_BUFFER_ID_INVALID_EXT if XrSpatialBufferGetInfoEXT ::bufferId does not belong to snapshot.

buffer filled by the runtime **must** be a null-terminated UTF-8 string.

The xrGetSpatialBufferUint8EXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrGetSpatialBufferUint8EXT(
    XrSpatialSnapshotEXT                      snapshot,
    const XrSpatialBufferGetInfoEXT*          info,
    uint32_t                                  bufferCapacityInput,
    uint32_t*                                 bufferCountOutput,
    uint8_t*                                  buffer);
```

<div align="center">

Parameter Descriptions

</div>

- snapshot is a handle to an XrSpatialSnapshotEXT.

- info holds the information on the buffer to query.

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

- bufferCountOutput is the number of elements in the buffer array, or the required capacity in the case that bufferCapacityInput is insufficient.

- buffer is an array of uint8_t. It **can** be NULL if bufferCapacityInput is 0.

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

The application **can** get the data for an XrSpatialBufferEXT provided by a component, where XrSpatialBufferEXT::bufferType is XR_SPATIAL_BUFFER_TYPE_UINT8_EXT by using xrGetSpatialBufferUint8EXT.

The runtime **must** return XR_ERROR_VALIDATION_FAILURE if the XrSpatialBufferTypeEXT for XrSpatialBufferGetInfoEXT::bufferId is not XR_SPATIAL_BUFFER_TYPE_UINT8_EXT.

The runtime **must** return XR_ERROR_SPATIAL_BUFFER_ID_INVALID_EXT if XrSpatialBufferGetInfoEXT ::bufferId does not belong to snapshot.

<div align="center">

Valid Usage (Implicit)

</div>

- The XR_EXT_spatial_entity extension **must** be enabled prior to calling xrGetSpatialBufferUint8EXT

- snapshot **must** be a valid XrSpatialSnapshotEXT handle

- info **must** be a pointer to a valid XrSpatialBufferGetInfoEXT 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 uint8_t values

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_SIZE_INSUFFICIENT

- XR_ERROR_SPATIAL_BUFFER_ID_INVALID_EXT

The xrGetSpatialBufferUint16EXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrGetSpatialBufferUint16EXT(
    XrSpatialSnapshotEXT                        snapshot,
    const XrSpatialBufferGetInfoEXT*            info,
    uint32_t                                    bufferCapacityInput,
    uint32_t*                                   bufferCountOutput,
    uint16_t*                                   buffer);
```

The application **can** get the data for an XrSpatialBufferEXT provided by a component, where XrSpatialBufferEXT::`bufferType` is `XR_SPATIAL_BUFFER_TYPE_UINT16_EXT` by using xrGetSpatialBufferUint16EXT.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` if the XrSpatialBufferTypeEXT for XrSpatialBufferGetInfoEXT::`bufferId` is not `XR_SPATIAL_BUFFER_TYPE_UINT16_EXT`.

The runtime **must** return `XR_ERROR_SPATIAL_BUFFER_ID_INVALID_EXT` if XrSpatialBufferGetInfoEXT::`bufferId` does not belong to `snapshot`.

The xrGetSpatialBufferUint32EXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrGetSpatialBufferUint32EXT(
    XrSpatialSnapshotEXT                        snapshot,
    const XrSpatialBufferGetInfoEXT*            info,
    uint32_t                                    bufferCapacityInput,
    uint32_t*                                   bufferCountOutput,
    uint32_t*                                   buffer);
```

- `snapshot` is a handle to an XrSpatialSnapshotEXT.

- `info` holds the information on the buffer to query.

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

- `bufferCountOutput` is the number of elements in the `buffer` array, or the required capacity in the case that `bufferCapacityInput` is insufficient.

- `buffer` is an array of `uint32_t`. It **can** be `NULL` if `bufferCapacityInput` is 0.

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

The application **can** get the data for an XrSpatialBufferEXT provided by a component, where XrSpatialBufferEXT::`bufferType` is `XR_SPATIAL_BUFFER_TYPE_UINT32_EXT` by using xrGetSpatialBufferUint32EXT.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` if the XrSpatialBufferTypeEXT for XrSpatialBufferGetInfoEXT::`bufferId` is not `XR_SPATIAL_BUFFER_TYPE_UINT32_EXT`.

The runtime **must** return `XR_ERROR_SPATIAL_BUFFER_ID_INVALID_EXT` if XrSpatialBufferGetInfoEXT ::`bufferId` does not belong to `snapshot`.

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to calling xrGetSpatialBufferUint32EXT

- `snapshot` **must** be a valid XrSpatialSnapshotEXT handle

- `info` **must** be a pointer to a valid XrSpatialBufferGetInfoEXT 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` `uint32_t` values

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_SIZE_INSUFFICIENT

- XR_ERROR_SPATIAL_BUFFER_ID_INVALID_EXT

The xrGetSpatialBufferFloatEXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrGetSpatialBufferFloatEXT(
    XrSpatialSnapshotEXT                        snapshot,
    const XrSpatialBufferGetInfoEXT*            info,
    uint32_t                                    bufferCapacityInput,
    uint32_t*                                   bufferCountOutput,
    float*                                      buffer);
```

The application **can** get the data for an XrSpatialBufferEXT provided by a component, where XrSpatialBufferEXT::`bufferType` is `XR_SPATIAL_BUFFER_TYPE_FLOAT_EXT` by using xrGetSpatialBufferFloatEXT.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` if the XrSpatialBufferTypeEXT for XrSpatialBufferGetInfoEXT::`bufferId` is not `XR_SPATIAL_BUFFER_TYPE_FLOAT_EXT`.

The runtime **must** return `XR_ERROR_SPATIAL_BUFFER_ID_INVALID_EXT` if XrSpatialBufferGetInfoEXT ::`bufferId` does not belong to `snapshot`.

The xrGetSpatialBufferVector2fEXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrGetSpatialBufferVector2fEXT(
    XrSpatialSnapshotEXT                        snapshot,
    const XrSpatialBufferGetInfoEXT*            info,
    uint32_t                                    bufferCapacityInput,
    uint32_t*                                   bufferCountOutput,
    XrVector2f*                                 buffer);
```

Parameter Descriptions

- `snapshot` is a handle to an XrSpatialSnapshotEXT.

- `info` holds the information on the buffer to query.

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

- `bufferCountOutput` is the number of elements in the `buffer` array, or the required capacity in the case that `bufferCapacityInput` is insufficient.

- `buffer` is an array of XrVector2f. It **can** be `NULL` if `bufferCapacityInput` is 0.

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

The application **can** get the data for an XrSpatialBufferEXT provided by a component, where XrSpatialBufferEXT::`bufferType` is `XR_SPATIAL_BUFFER_TYPE_VECTOR2F_EXT` by using xrGetSpatialBufferVector2fEXT.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` if the XrSpatialBufferTypeEXT for XrSpatialBufferGetInfoEXT::`bufferId` is not `XR_SPATIAL_BUFFER_TYPE_VECTOR2F_EXT`.

The runtime **must** return `XR_ERROR_SPATIAL_BUFFER_ID_INVALID_EXT` if XrSpatialBufferGetInfoEXT ::`bufferId` does not belong to `snapshot`.

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to calling xrGetSpatialBufferVector2fEXT

- `snapshot` **must** be a valid XrSpatialSnapshotEXT handle

- `info` **must** be a pointer to a valid XrSpatialBufferGetInfoEXT 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` XrVector2f structures

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_SIZE_INSUFFICIENT

- XR_ERROR_SPATIAL_BUFFER_ID_INVALID_EXT

The xrGetSpatialBufferVector3fEXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrGetSpatialBufferVector3fEXT(
    XrSpatialSnapshotEXT                        snapshot,
    const XrSpatialBufferGetInfoEXT*            info,
    uint32_t                                    bufferCapacityInput,
    uint32_t*                                   bufferCountOutput,
    XrVector3f*                                 buffer);
```

- `snapshot` is a handle to an XrSpatialSnapshotEXT.

- `info` holds the information on the buffer to query.

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

- `bufferCountOutput` is the number of elements in the `buffer` array, or the required capacity in the case that `bufferCapacityInput` is insufficient.

- `buffer` is an array of XrVector3f. It **can** be `NULL` if `bufferCapacityInput` is 0.

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

The application **can** get the data for an XrSpatialBufferEXT provided by a component, where XrSpatialBufferEXT::`bufferType` is `XR_SPATIAL_BUFFER_TYPE_VECTOR3F_EXT` by using xrGetSpatialBufferVector3fEXT.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` if the XrSpatialBufferTypeEXT for XrSpatialBufferGetInfoEXT::`bufferId` is not `XR_SPATIAL_BUFFER_TYPE_VECTOR3F_EXT`.

The runtime **must** return `XR_ERROR_SPATIAL_BUFFER_ID_INVALID_EXT` if XrSpatialBufferGetInfoEXT ::`bufferId` does not belong to `snapshot`.

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to calling xrGetSpatialBufferVector3fEXT

- `snapshot` **must** be a valid XrSpatialSnapshotEXT handle

- `info` **must** be a pointer to a valid XrSpatialBufferGetInfoEXT 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` XrVector3f structures

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_SIZE_INSUFFICIENT

- XR_ERROR_SPATIAL_BUFFER_ID_INVALID_EXT

The XrSpatialBufferGetInfoEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrSpatialBufferGetInfoEXT {
    XrStructureType        type;
    const void*            next;
    XrSpatialBufferIdEXT   bufferId;
} XrSpatialBufferGetInfoEXT;
```

Member Descriptions

- type is the XrStructureType of this structure.

- next is NULL or a pointer to the next structure in a structure chain.

- bufferId an XrSpatialBufferIdEXT for the buffer whose data to retrieve.

```
// Provided by XR_EXT_spatial_entity
#define XR_NULL_SPATIAL_BUFFER_ID_EXT 0
```

XR_NULL_SPATIAL_BUFFER_ID_EXT is a reserved value representing an invalid `XrSpatialBufferIdEXT`. It **may** be passed to and returned from API functions only when specifically allowed.

Create Update Snapshot

The xrCreateSpatialUpdateSnapshotEXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrCreateSpatialUpdateSnapshotEXT(
    XrSpatialContextEXT                         spatialContext,
    const XrSpatialUpdateSnapshotCreateInfoEXT* createInfo,
    XrSpatialSnapshotEXT*                       snapshot);
```

The application **can** use xrCreateSpatialUpdateSnapshotEXT to create a snapshot and get the latest component data for specific entities as known by the runtime. Applications **can** provide the XrSpatialEntityEXT handles and the component types they are interested in when creating the snapshot.

The application **can** use XrSpatialUpdateSnapshotCreateInfoEXT::`componentTypes` to filter the list of

components whose data **must** be included in the snapshot. If the application provides a valid list of spatial component types in XrSpatialUpdateSnapshotCreateInfoEXT::componentTypes, then the runtime **must** only include spatial entities in the snapshot that have at least one of the components provided in XrSpatialUpdateSnapshotCreateInfoEXT::componentTypes. Also, the runtime **must** only include data for those components in the snapshot.

The runtime **must** return XR_ERROR_SPATIAL_COMPONENT_NOT_ENABLED_EXT if any of the XrSpatialComponentTypeEXT in XrSpatialUpdateSnapshotCreateInfoEXT::componentTypes are not enabled for the spatial capabilities passed to XrSpatialContextCreateInfoEXT::capabilityConfigs when creating spatialContext.

If the application does not provide a list of spatial component types in XrSpatialUpdateSnapshotCreateInfoEXT::componentTypes, the runtime **must** include all the spatial entities listed in XrSpatialUpdateSnapshotCreateInfoEXT::entities in the snapshot and it **must** include the data for all the enabled components of the capabilities configured for spatialContext.

The application **can** create any number of snapshots it wants but **must** be mindful of the memory being allocated for each new snapshot and **must** destroy the snapshots once it no longer needs them.

Valid Usage (Implicit)

- The XR_EXT_spatial_entity extension **must** be enabled prior to calling xrCreateSpatialUpdateSnapshotEXT

- spatialContext **must** be a valid XrSpatialContextEXT handle

- createInfo **must** be a pointer to a valid XrSpatialUpdateSnapshotCreateInfoEXT structure

- snapshot **must** be a pointer to an XrSpatialSnapshotEXT handle

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

- XR_ERROR_TIME_INVALID

- XR_ERROR_SPATIAL_COMPONENT_NOT_ENABLED_EXT

The XrSpatialUpdateSnapshotCreateInfoEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrSpatialUpdateSnapshotCreateInfoEXT {
    XrStructureType                   type;
    const void*                       next;
    uint32_t                          entityCount;
    const XrSpatialEntityEXT*         entities;
    uint32_t                          componentTypeCount;
    const XrSpatialComponentTypeEXT*  componentTypes;
    XrSpace                           baseSpace;
    XrTime                            time;
} XrSpatialUpdateSnapshotCreateInfoEXT;
```

- `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.

- `entityCount` is a `uint32_t` describing the count of elements in the `entities` array.

- `entities` is an array of XrSpatialEntityEXT for which the runtime **must** include the component data in the snapshot.

- `componentTypeCount` is a `uint32_t` describing the count of elements in the `componentTypes` array.

- `componentTypes` is an array of XrSpatialComponentTypeEXT for which the runtime **must** include the data in the snapshot.

- `baseSpace` is the XrSpace relative to which all the locations of the update XrSpatialSnapshotEXT will be located.

- `time` is the XrTime at which all the locations of the update XrSpatialSnapshotEXT will be located.

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to using XrSpatialUpdateSnapshotCreateInfoEXT

- `type` **must** be XR_TYPE_SPATIAL_UPDATE_SNAPSHOT_CREATE_INFO_EXT

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

- `entities` **must** be a pointer to an array of `entityCount` valid XrSpatialEntityEXT handles

- If `componentTypeCount` is not `0`, `componentTypes` **must** be a pointer to an array of `componentTypeCount` valid XrSpatialComponentTypeEXT values

- `baseSpace` **must** be a valid XrSpace handle

- The `entityCount` parameter **must** be greater than `0`

- Both of `baseSpace` and the elements of `entities` **must** have been created, allocated, or retrieved from the same XrSession

Destroy snapshot

The xrDestroySpatialSnapshotEXT function is defined as:

```
// Provided by XR_EXT_spatial_entity
XrResult xrDestroySpatialSnapshotEXT(
    XrSpatialSnapshotEXT                       snapshot);
```

Parameter Descriptions

- snapshot   is   an   XrSpatialSnapshotEXT   previously   provided   by
  xrCreateSpatialDiscoverySnapshotCompleteEXT or xrCreateSpatialUpdateSnapshotEXT.

The application **can** call xrDestroySpatialSnapshotEXT to destroy the XrSpatialSnapshotEXT handle
and the resources associated with it.

Valid Usage (Implicit)

- The   XR_EXT_spatial_entity   extension   **must**   be   enabled   prior   to   calling
  xrDestroySpatialSnapshotEXT

- snapshot **must** be a valid XrSpatialSnapshotEXT handle

Thread Safety

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

Return Codes

Success

- XR_SUCCESS

Failure

- XR_ERROR_FUNCTION_UNSUPPORTED

- XR_ERROR_RUNTIME_FAILURE

- XR_ERROR_HANDLE_INVALID

12.36.8. Common Components

Bounded 2D

Component data

The XrSpatialBounded2DDataEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrSpatialBounded2DDataEXT {
    XrPosef        center;
    XrExtent2Df    extents;
} XrSpatialBounded2DDataEXT;
```

Member Descriptions

- center is an XrPosef defining the geometric center of the bounded 2D component.

- extents is extents of the bounded 2D component along the x-axis (extents.width), y-axis (extents.height), centered on center.

The extents of the XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT refer to the entity's size in the x-y plane of the plane's coordinate system. A plane with a position of {0, 0, 0}, rotation of {0, 0, 0, 1} (no rotation), and an extent of {1, 1} refers to a 1 meter x 1 meter plane centered at {0, 0, 0} with its front face normal vector pointing towards the +Z direction in the component's space.
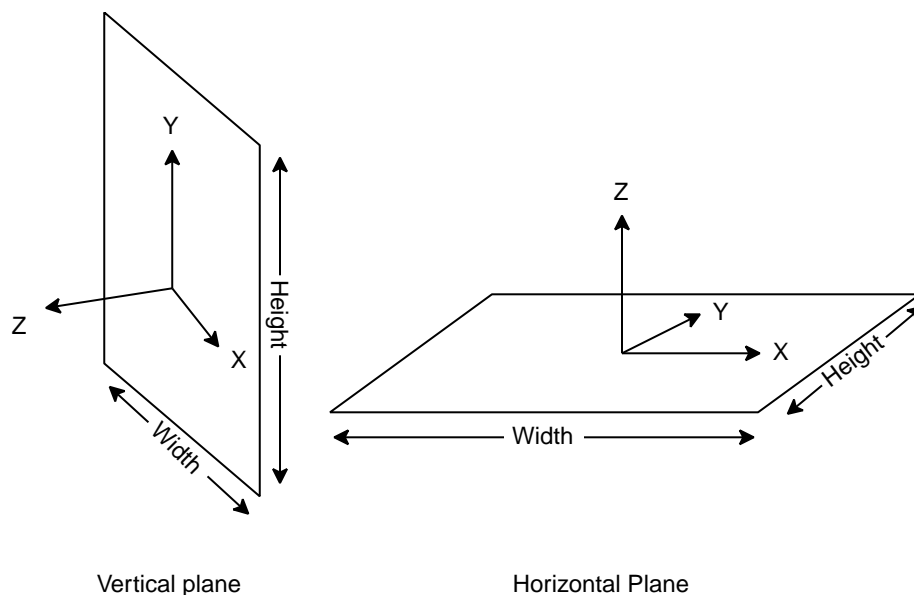
Figure 19. Bounded2D Component Coordinate System

> **Note**
>
> OpenXR uses an X-Y plane with +Z as the plane normal but other APIs **may** use an X-Z plane with +Y as the plane normal. The X-Y plane **can** be converted to an X-Z plane by rotating -π/2 radians around the +X axis.

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to using XrSpatialBounded2DDataEXT

Component list structure to query data

The XrSpatialComponentBounded2DListEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrSpatialComponentBounded2DListEXT {
    XrStructureType             type;
    void*                       next;
    uint32_t                    boundCount;
    XrSpatialBounded2DDataEXT*  bounds;
} XrSpatialComponentBounded2DListEXT;
```

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.

- `boundCount` is a `uint32_t` describing the count of elements in the `bounds` array.

- `bounds` is an array of XrSpatialBounded2DDataEXT.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` from xrQuerySpatialComponentDataEXT if XrSpatialComponentBounded2DListEXT is in the next chain of XrSpatialComponentDataQueryResultEXT::`next` but XrSpatialComponentDataQueryConditionEXT::componentTypeCount is not zero and `XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT` is not included in XrSpatialComponentDataQueryConditionEXT::`componentTypes`.

The runtime **must** return `XR_ERROR_SIZE_INSUFFICIENT` from xrQuerySpatialComponentDataEXT if `boundCount` is less than XrSpatialComponentDataQueryResultEXT::`entityIdCountOutput`.

If xrQuerySpatialComponentDataEXT::snapshot was created from xrCreateSpatialDiscoverySnapshotCompleteEXT, then the runtime **must** provide XrSpatialBounded2DDataEXT::center in XrCreateSpatialDiscoverySnapshotCompletionInfoEXT ::baseSpace and XrCreateSpatialDiscoverySnapshotCompletionInfoEXT::time.

If xrQuerySpatialComponentDataEXT::snapshot was created from xrCreateSpatialUpdateSnapshotEXT, then the runtime **must** provide XrSpatialBounded2DDataEXT::center in XrSpatialUpdateSnapshotCreateInfoEXT::baseSpace and XrSpatialUpdateSnapshotCreateInfoEXT::time.

Valid Usage (Implicit)

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to using XrSpatialComponentBounded2DListEXT

- `type` **must** be `XR_TYPE_SPATIAL_COMPONENT_BOUNDED_2D_LIST_EXT`

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

- `bounds` **must** be a pointer to an array of `boundCount` XrSpatialBounded2DDataEXT structures

- The `boundCount` parameter **must** be greater than `0`

Configuration

If `XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT` is enumerated in XrSpatialCapabilityComponentTypesEXT::componentTypes for some capability, the application **can** enable it by including the enum value in the XrSpatialCapabilityConfigurationBaseHeaderEXT ::enabledComponents list. This component does not require any special configuration to be included in the next chain of XrSpatialCapabilityConfigurationBaseHeaderEXT.

Bounded 3D

Component data

`XR_SPATIAL_COMPONENT_TYPE_BOUNDED_3D_EXT` uses XrBoxf for its data.

Component list structure to query data

The XrSpatialComponentBounded3DListEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrSpatialComponentBounded3DListEXT {
    XrStructureType    type;
    void*              next;
    uint32_t           boundCount;
    XrBoxf*            bounds;
} XrSpatialComponentBounded3DListEXT;
```

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.

- `boundCount` is a `uint32_t` describing the count of elements in the `bounds` array.

- `bounds` is an array of XrBoxf.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` from xrQuerySpatialComponentDataEXT if XrSpatialComponentBounded3DListEXT is in the next chain of XrSpatialComponentDataQueryResultEXT::`next` but XrSpatialComponentDataQueryConditionEXT::componentTypeCount is not zero and `XR_SPATIAL_COMPONENT_TYPE_BOUNDED_3D_EXT` is not included in XrSpatialComponentDataQueryConditionEXT::`componentTypes`.

The runtime **must** return `XR_ERROR_SIZE_INSUFFICIENT` from xrQuerySpatialComponentDataEXT if `boundCount` is less than XrSpatialComponentDataQueryResultEXT::`entityIdCountOutput`.

If xrQuerySpatialComponentDataEXT::`snapshot` was created from xrCreateSpatialDiscoverySnapshotCompleteEXT, then the runtime **must** provide XrBoxf::`center` in XrCreateSpatialDiscoverySnapshotCompletionInfoEXT::`baseSpace` at XrCreateSpatialDiscoverySnapshotCompletionInfoEXT::`time`.

If xrQuerySpatialComponentDataEXT::`snapshot` was created from xrCreateSpatialUpdateSnapshotEXT, then the runtime **must** provide XrBoxf::`center` in XrSpatialUpdateSnapshotCreateInfoEXT::`baseSpace` at XrSpatialUpdateSnapshotCreateInfoEXT::`time`.

Configuration

If `XR_SPATIAL_COMPONENT_TYPE_BOUNDED_3D_EXT` is enumerated in XrSpatialCapabilityComponentTypesEXT::`componentTypes` for some capability, the application **can** enable it by including the enum in the XrSpatialCapabilityConfigurationBaseHeaderEXT ::`enabledComponents` list. This component does not require any special configuration to be included in the next chain of XrSpatialCapabilityConfigurationBaseHeaderEXT.

Parent

Component data

`XR_SPATIAL_COMPONENT_TYPE_PARENT_EXT` uses `XrSpatialEntityIdEXT` for its data.

Component list structure to query data

The XrSpatialComponentParentListEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrSpatialComponentParentListEXT {
    XrStructureType         type;
    void*                   next;
    uint32_t                parentCount;
    XrSpatialEntityIdEXT*   parents;
} XrSpatialComponentParentListEXT;
```

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` from xrQuerySpatialComponentDataEXT if XrSpatialComponentParentListEXT is in the next chain of XrSpatialComponentDataQueryResultEXT ::`next` but XrSpatialComponentDataQueryConditionEXT::componentTypeCount is not zero and `XR_SPATIAL_COMPONENT_TYPE_PARENT_EXT` is not included in XrSpatialComponentDataQueryConditionEXT ::`componentTypes`.

The runtime **must** return `XR_ERROR_SIZE_INSUFFICIENT` from xrQuerySpatialComponentDataEXT if `parentCount` is less than XrSpatialComponentDataQueryResultEXT::`entityIdCountOutput`.

Configuration

If `XR_SPATIAL_COMPONENT_TYPE_PARENT_EXT` is enumerated in XrSpatialCapabilityComponentTypesEXT ::`componentTypes` for some capability, the application **can** enable it by including the enum in the XrSpatialCapabilityConfigurationBaseHeaderEXT::`enabledComponents` list. This component does not require any special configuration to be included in the next chain of XrSpatialCapabilityConfigurationBaseHeaderEXT.

Component data

The XrSpatialMeshDataEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrSpatialMeshDataEXT {
    XrPosef             origin;
    XrSpatialBufferEXT  vertexBuffer;
    XrSpatialBufferEXT  indexBuffer;
} XrSpatialMeshDataEXT;
```

Member Descriptions

- `origin` is an XrPosef defining the origin of the mesh. All vertices of the mesh **must** be relative to this origin.

- `vertexBuffer` is an XrSpatialBufferEXT that provides the ID for a buffer that represents the vertex buffer of the entity this component is on. The position of vertices **must** be relative to `origin`.

- `indexBuffer` is an XrSpatialBufferEXT that provides the ID for a buffer that represents an array of triangle indices, specifying the indices of the mesh vertices in the `vertexBuffer`. The triangle indices **must** be returned in counter-clockwise order and three indices denote one triangle.

The component type using XrSpatialMeshDataEXT **must** specify the XrSpatialBufferTypeEXT of the `vertexBuffer` and `indexBuffer`.

Valid Usage (Implicit)

- The XR_EXT_spatial_entity extension **must** be enabled prior to using XrSpatialMeshDataEXT

- `vertexBuffer` **must** be a valid XrSpatialBufferEXT structure

- `indexBuffer` **must** be a valid XrSpatialBufferEXT structure

Component list structure to query data

The XrSpatialComponentMesh3DListEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrSpatialComponentMesh3DListEXT {
    XrStructureType         type;
    void*                   next;
    uint32_t                meshCount;
    XrSpatialMeshDataEXT*   meshes;
} XrSpatialComponentMesh3DListEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `meshCount` is a `uint32_t` describing the count of elements in the `meshes` array.

- `meshes` is an array of XrSpatialMeshDataEXT.

The application **can** query the mesh 3D component of the spatial entities in an XrSpatialSnapshotEXT by adding `XR_SPATIAL_COMPONENT_TYPE_MESH_3D_EXT` in XrSpatialComponentDataQueryConditionEXT ::`componentTypes` and adding XrSpatialComponentMesh3DListEXT to the next pointer chain of XrSpatialComponentDataQueryResultEXT.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` from xrQuerySpatialComponentDataEXT if XrSpatialComponentMesh3DListEXT is in the next chain of XrSpatialComponentDataQueryResultEXT ::`next` but XrSpatialComponentDataQueryConditionEXT::componentTypeCount is not zero and `XR_SPATIAL_COMPONENT_TYPE_MESH_3D_EXT` is not included in XrSpatialComponentDataQueryConditionEXT::`componentTypes`.

The runtime **must** return `XR_ERROR_SIZE_INSUFFICIENT` from xrQuerySpatialComponentDataEXT if `meshCount` is less than XrSpatialComponentDataQueryResultEXT::`entityIdCountOutput`.

For the XrSpatialMeshDataEXT filled out by the runtime in the `meshes` array, the XrSpatialBufferEXT ::`bufferType` for XrSpatialMeshDataEXT::`vertexBuffer` **must** be `XR_SPATIAL_BUFFER_TYPE_VECTOR3F_EXT` and XrSpatialBufferEXT::`bufferType` for XrSpatialMeshDataEXT::`indexBuffer` **must** be `XR_SPATIAL_BUFFER_TYPE_UINT32_EXT`.

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to using XrSpatialComponentMesh3DListEXT

- `type` **must** be `XR_TYPE_SPATIAL_COMPONENT_MESH_3D_LIST_EXT`

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

- `meshes` **must** be a pointer to an array of `meshCount` XrSpatialMeshDataEXT structures

- The `meshCount` parameter **must** be greater than `0`

Configuration

If `XR_SPATIAL_COMPONENT_TYPE_MESH_3D_EXT` is enumerated in XrSpatialCapabilityComponentTypesEXT ::`componentTypes` for some capability, the application **can** enable it by including the enum in the XrSpatialCapabilityConfigurationBaseHeaderEXT::`enabledComponents` list of the XrSpatialCapabilityConfigurationBaseHeaderEXT derived structure of the capability that supports this component.

This component does not require any special configuration to be included in the next chain of XrSpatialCapabilityConfigurationBaseHeaderEXT.

12.36.9. Tracking state filters

The XrSpatialFilterTrackingStateEXT structure is defined as:

```
// Provided by XR_EXT_spatial_entity
typedef struct XrSpatialFilterTrackingStateEXT {
    XrStructureType                    type;
    const void*                        next;
    XrSpatialEntityTrackingStateEXT    trackingState;
} XrSpatialFilterTrackingStateEXT;
```

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `trackingState` is the XrSpatialEntityTrackingStateEXT for which the application wants to apply the filter.

The application **can** use XrSpatialFilterTrackingStateEXT in the next chain of XrSpatialDiscoverySnapshotCreateInfoEXT to scope the discovery to only those entities whose tracking state is `trackingState`.

The application **can** use XrSpatialFilterTrackingStateEXT in the next chain of XrSpatialComponentDataQueryConditionEXT to scope the component data query from a snapshot only to entities whose tracking state is `trackingState`.

<div align="center">

Valid Usage (Implicit)

</div>

- The `XR_EXT_spatial_entity` extension **must** be enabled prior to using XrSpatialFilterTrackingStateEXT

- `type` **must** be `XR_TYPE_SPATIAL_FILTER_TRACKING_STATE_EXT`

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

- `trackingState` **must** be a valid XrSpatialEntityTrackingStateEXT value

---

12.36.10. Example code

Application Usage

Applications typically use the spatial entity extension in the following pattern:

- An application first enumerates the spatial capabilities of the system using xrEnumerateSpatialCapabilitiesEXT. It then inspects the returned array of XrSpatialCapabilityEXT and enumerates the components and features supported for each of those capabilities by using xrEnumerateSpatialCapabilityComponentTypesEXT and xrEnumerateSpatialCapabilityFeaturesEXT respectively. This gives the application a full picture of the components that it **can** enable and the configurations the capability accepts.

- The application then creates one or many XrSpatialContextEXT handles with specific spatial capability configurations, wherein the configurations enable & configure a specific capability in the spatial context, and enable & configure components for those capabilities.

- For each XrSpatialContextEXT, the application waits to receive XrEventDataSpatialDiscoveryRecommendedEXT events for that XrSpatialContextEXT before using xrCreateSpatialDiscoverySnapshotAsyncEXT to discover spatial entities. Once this async operation is complete, the application receives a XrSpatialSnapshotEXT handle.

- The application queries for the entities and the component data included in this XrSpatialSnapshotEXT by using xrQuerySpatialComponentDataEXT. The application reads the latest component data of the queried entities from structures attached to the next chain of XrSpatialComponentDataQueryResultEXT if the entity state is `XR_SPATIAL_ENTITY_TRACKING_STATE_TRACKING_EXT`.

- If there are specific entities that the application identifies as interesting and wants to get updates

for over time, it creates XrSpatialEntityEXT handles for those entities by using xrCreateSpatialEntityFromIdEXT. The application gets updates for such interesting entities by using xrCreateSpatialUpdateSnapshotEXT and use the same xrQuerySpatialComponentDataEXT function on the newly created XrSpatialSnapshotEXT to get the latest component data for those entities.

Discover spatial entities & query component data

The following example code demonstrates how to discover spatial entities for capability "Foo" query its component data.

```
/***************************/
/* Capability definition   */
/***************************/
// Foo capability has the following components -
// - XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT
#define XR_SPATIAL_CAPABILITY_FOO ((XrSpatialCapabilityEXT)1000740000U)

#define XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_FOO_EXT ((XrStructureType)1000740000U)
// Derives from XrSpatialCapabilityConfigurationBaseHeaderEXT
typedef struct XrSpatialCapabilityConfigurationFooEXT {
  XrStructureType                 type;
  const void* XR_MAY_ALIAS        next;
  XrSpatialCapabilityEXT          capability;
  uint32_t                        enabledComponentCount;
  const XrSpatialComponentTypeEXT*    enabledComponents;
} XrSpatialCapabilityConfigurationFooEXT;

/*****************************/
/* End capability definition */
/*****************************/

auto waitUntilReady = [](XrFutureEXT future) {
  XrFuturePollInfoEXT pollInfo{XR_TYPE_FUTURE_POLL_INFO_EXT};
  XrFuturePollResultEXT pollResult{XR_TYPE_FUTURE_POLL_RESULT_EXT};
  pollInfo.future = future;
  do {
    // sleep(1);
    CHK_XR(xrPollFutureEXT(instance, &pollInfo, &pollResult));
  } while (pollResult.state != XR_FUTURE_STATE_READY_EXT);
};

// Create a spatial spatial context
XrSpatialContextEXT spatialContext{};
{
  const std::array<XrSpatialComponentTypeEXT, 1> enabledComponents = {
    XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT,
  };
```

```
  // Configure Foo capability for the spatial context
  XrSpatialCapabilityConfigurationFooEXT
fooConfig{XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_FOO_EXT};
  fooConfig.capability = XR_SPATIAL_CAPABILITY_FOO;
  fooConfig.enabledComponentCount = enabledComponents.size();
  fooConfig.enabledComponents = enabledComponents.data();

  std::vector<XrSpatialCapabilityConfigurationBaseHeaderEXT*> capabilityConfigs;

capabilityConfigs.push_back(reinterpret_cast<XrSpatialCapabilityConfigurationBaseHeaderEX
T*>(&fooConfig));

  XrSpatialContextCreateInfoEXT
spatialContextCreateInfo{XR_TYPE_SPATIAL_CONTEXT_CREATE_INFO_EXT};
  spatialContextCreateInfo.capabilityConfigCount = capabilityConfigs.size();
  spatialContextCreateInfo.capabilityConfigs = capabilityConfigs.data();
  XrFutureEXT createContextFuture;
  CHK_XR(xrCreateSpatialContextAsyncEXT(session, &spatialContextCreateInfo,
&createContextFuture));

  waitUntilReady(createContextFuture);

  XrCreateSpatialContextCompletionEXT
completion{XR_TYPE_CREATE_SPATIAL_CONTEXT_COMPLETION_EXT};
  CHK_XR(xrCreateSpatialContextCompleteEXT(session, createContextFuture, &completion));
  if (completion.futureResult != XR_SUCCESS) {
    return;
  }

  spatialContext = completion.spatialContext;
}

auto discoverSpatialEntities = [&](XrSpatialContextEXT spatialContext, XrTime time) {
  // We want to look for entities that have the following components.
  std::array<XrSpatialComponentTypeEXT, 1> snapshotComponents
{XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT};

  XrSpatialDiscoverySnapshotCreateInfoEXT
snapshotCreateInfo{XR_TYPE_SPATIAL_DISCOVERY_SNAPSHOT_CREATE_INFO_EXT};
  snapshotCreateInfo.componentTypeCount = snapshotComponents.size();
  snapshotCreateInfo.componentTypes = snapshotComponents.data();
  XrFutureEXT future = XR_NULL_FUTURE_EXT;
  CHK_XR(xrCreateSpatialDiscoverySnapshotAsyncEXT(spatialContext, &snapshotCreateInfo,
&future));

  waitUntilReady(future);
```

```
    XrCreateSpatialDiscoverySnapshotCompletionInfoEXT
completionInfo{XR_TYPE_CREATE_SPATIAL_DISCOVERY_SNAPSHOT_COMPLETION_INFO_EXT};
    completionInfo.baseSpace = localSpace;
    completionInfo.time = time;
    completionInfo.future = future;

    XrCreateSpatialDiscoverySnapshotCompletionEXT
completion{XR_TYPE_CREATE_SPATIAL_DISCOVERY_SNAPSHOT_COMPLETION_EXT};
    CHK_XR(xrCreateSpatialDiscoverySnapshotCompleteEXT(spatialContext, &completionInfo,
&completion));
    if (completion.futureResult == XR_SUCCESS) {
        // Query for the bounded2d component data
        XrSpatialComponentTypeEXT componentToQuery =
XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT;
        XrSpatialComponentDataQueryConditionEXT
queryCond{XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_CONDITION_EXT};
        queryCond.componentTypes = &componentToQuery;

        XrSpatialComponentDataQueryResultEXT
queryResult{XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_RESULT_EXT};
        queryResult.entityIdCapacityInput = 0;
        queryResult.entityIds = nullptr;
        queryResult.entityStateCapacityInput = 0;
        queryResult.entityStates = nullptr;
        CHK_XR(xrQuerySpatialComponentDataEXT(completion.snapshot, &queryCond,
&queryResult));

        std::vector<XrSpatialEntityIdEXT> entityIds(queryResult.entityIdCountOutput);
        std::vector<XrSpatialEntityTrackingStateEXT>
entityStates(queryResult.entityStateCountOutput);
        queryResult.entityIdCapacityInput = entityIds.size();
        queryResult.entityIds = entityIds.data();
        queryResult.entityStateCapacityInput = entityStates.size();
        queryResult.entityStates = entityStates.data();

        std::vector<XrSpatialBounded2DDataEXT> bounded2d(queryResult.entityIdCountOutput);
        XrSpatialComponentBounded2DListEXT
boundsList{XR_TYPE_SPATIAL_COMPONENT_BOUNDED_2D_LIST_EXT};
        boundsList.boundCount = bounded2d.size();
        boundsList.bounds = bounded2d.data();
        queryResult.next = &boundsList;

        CHK_XR(xrQuerySpatialComponentDataEXT(completion.snapshot, &queryCond,
&queryResult));

        for (int32_t i = 0; i < queryResult.entityIdCountOutput; ++i) {
            if (entityStates[i] == XR_SPATIAL_ENTITY_TRACKING_STATE_TRACKING_EXT) {
                // 2D extents for entity entityIds[i] is bounded2d[i].extents.
```

```
        }
      }

      CHK_XR(xrDestroySpatialSnapshotEXT(completion.snapshot));
    }
};

while (1) {
  // ...
  // For every frame in frame loop
  // ...

  XrFrameState frameState;  // previously returned from xrWaitFrame
  const XrTime time = frameState.predictedDisplayTime;

  // Poll for the XR_TYPE_EVENT_DATA_SPATIAL_DISCOVERY_RECOMMENDED_EXT event
  XrEventDataBuffer event = {XR_TYPE_EVENT_DATA_BUFFER};
  XrResult result = xrPollEvent(instance, &event);
  if (result == XR_SUCCESS) {
      switch (event.type) {
          case XR_TYPE_EVENT_DATA_SPATIAL_DISCOVERY_RECOMMENDED_EXT: {
              const XrEventDataSpatialDiscoveryRecommendedEXT& eventdata =
                  *reinterpret_cast<XrEventDataSpatialDiscoveryRecommendedEXT*>(&event);
              // Discover spatial entities for the context that we recceived the
"discovery
              // recommended" event for.
              discoverSpatialEntities(eventdata.spatialContext, time);
              break;
          }
      }
  }

  // ...
  // Finish frame loop
  // ...
}

CHK_XR(xrDestroySpatialContextEXT(spatialContext));
```

Query buffer data

The following example code demonstrates how to get the data of a component that provides an
XrSpatialBufferEXT.

```
/***************************/
/* Component definition    */
```

```
/***************************/
// Foo component that provides an XrVector3f buffer
#define XR_SPATIAL_COMPONENT_TYPE_FOO_EXT ((XrSpatialComponentTypeEXT)1000740000U)

#define XR_TYPE_SPATIAL_COMPONENT_FOO_LIST_EXT ((XrStructureType)1000740000U)

// XrSpatialComponentFooListEXT extends XrSpatialComponentDataQueryResultEXT
typedef struct XrSpatialComponentFooListEXT {
    XrStructureType                 type;
    void* XR_MAY_ALIAS              next;
    uint32_t                        fooCount;
    XrSpatialBufferEXT*             foo;
} XrSpatialComponentFooListEXT;

/*****************************/
/* End Component definition  */
/*****************************/

// Query for the foo component data
XrSpatialComponentTypeEXT componentToQuery = XR_SPATIAL_COMPONENT_TYPE_FOO_EXT;
XrSpatialComponentDataQueryConditionEXT
queryCond{XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_CONDITION_EXT};
queryCond.componentTypeCount = 1;
queryCond.componentTypes = &componentToQuery;

XrSpatialComponentDataQueryResultEXT
queryResult{XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_RESULT_EXT};
CHK_XR(xrQuerySpatialComponentDataEXT(snapshot, &queryCond, &queryResult));

std::vector<XrSpatialEntityIdEXT> entityIds(queryResult.entityIdCountOutput);
queryResult.entityIdCapacityInput = entityIds.size();
queryResult.entityIds = entityIds.data();

std::vector<XrSpatialBufferEXT> fooBuffers(queryResult.entityIdCountOutput);
XrSpatialComponentFooListEXT fooList{XR_TYPE_SPATIAL_COMPONENT_FOO_LIST_EXT};
fooList.fooCount = fooBuffers.size();
fooList.foo = fooBuffers.data();
queryResult.next = &fooList;

CHK_XR(xrQuerySpatialComponentDataEXT(snapshot, &queryCond, &queryResult));

for (int32_t i = 0; i < queryResult.entityIdCountOutput; ++i) {
  // foo component data for entity entityIds[i]
  if (fooBuffers[i].bufferType == XR_SPATIAL_BUFFER_TYPE_VECTOR3F_EXT) {
    XrSpatialBufferGetInfoEXT getInfo{XR_TYPE_SPATIAL_BUFFER_GET_INFO_EXT};
    getInfo.bufferId = fooBuffers[i].bufferId;
    uint32_t bufferCountOutput;
    CHK_XR(xrGetSpatialBufferVector3fEXT(snapshot, &getInfo, 0, &bufferCountOutput,
```

```
nullptr));
    std::vector<XrVector3f> vertexBuffer(bufferCountOutput);
    CHK_XR(xrGetSpatialBufferVector3fEXT(snapshot, &getInfo, bufferCountOutput,
&bufferCountOutput, vertexBuffer.data()));

    // XrVertex3f buffer for entity entityIds[i] is now available in vertexBuffer vector.
  }
}
```

12.36.11. Extension guidelines

- If an extension is defining a new XrSpatialComponentTypeEXT which provides additional data about a spatial entity,

  - the extension **must** also define a list structure for that component which allows the application to pass an array to the runtime to fill out with the data for each of the spatial entities that satisfy xrQuerySpatialComponentDataEXT::queryCondition. Some examples of such list structures are XrSpatialComponentParentListEXT and XrSpatialComponentBounded2DListEXT. If the component data size is variable and requires the application to use fundamentals-buffer-size-parameters to query it, the component data **should** provide an XrSpatialBufferEXT for each variable-sized data in the list structure and it **must** specify the XrSpatialBufferTypeEXT for each buffer. Application **can** then query the buffer data using functions defined in Two-call idiom for component data. An example of such a structure is XrSpatialMeshDataEXT which is included XrSpatialComponentMesh3DListEXT.

  - The extension **can** also provide structures that the application **can** chain to XrSpatialComponentDataQueryConditionEXT::next to provide additional filters for the query pertaining to the data of this component.

- Extensions **can** define structures that extend XrSpatialDiscoverySnapshotCreateInfoEXT to provide additional filters for discovery. The filters for creating the snapshot **must** not affect the configuration of the spatial context, but instead are to be used to provide hints to the runtime on what entities and data are to be included in the snapshot as tracked by the current configuration of the spatial context (and therefore the current configuration of the underlying services).

- If an extension defines a new XrSpatialCapabilityEXT,

  - it **should** also specify the list of XrSpatialComponentTypeEXT that the runtimes **must** provide on entities for that capability.

  - it **must** also provide structures derived from XrSpatialCapabilityConfigurationBaseHeaderEXT that will allow the configuration of that capability.

- If an extension defines a new XrSpatialCapabilityFeatureEXT, it **must** also define a corresponding configuration structure that **can** be chained to the next pointer of XrSpatialCapabilityConfigurationBaseHeaderEXT, that the application **can** use to enable the feature when creating an XrSpatialContextEXT.

- An extension defining a new XrSpatialCapabilityEXT **should** follow this template for the

specification -

- ◦ Overview
- ◦ Runtime support
- ◦ Configuration
- ◦ Guaranteed components
- ◦ Example Code

- An extension defining a new `XrSpatialComponentTypeEXT` **should** follow this template for the specification -

  - ◦ Component data
  - ◦ Component list structure to query data
  - ◦ Configuration

- When writing an api that provides the application with a `XrSpatialBufferIdEXT`, it **must** be accompanied with a `XrSpatialBufferTypeEXT` to inform the application what the data type of the buffer is and the application **can** use an appropriate `xrGetSpatialBuffer*` function to retrieve the actual contents of the buffer.

12.36.12. New Macros

- XR_NULL_SPATIAL_BUFFER_ID_EXT
- XR_NULL_SPATIAL_ENTITY_ID_EXT

12.36.13. New Base Types

- `XrSpatialBufferIdEXT`
- `XrSpatialEntityIdEXT`

12.36.14. New Object Types

- XrSpatialContextEXT
- XrSpatialEntityEXT
- XrSpatialSnapshotEXT

12.36.15. New Commands

- xrCreateSpatialContextAsyncEXT
- xrCreateSpatialContextCompleteEXT
- xrCreateSpatialDiscoverySnapshotAsyncEXT
- xrCreateSpatialDiscoverySnapshotCompleteEXT

- xrCreateSpatialEntityFromIdEXT

- xrCreateSpatialUpdateSnapshotEXT

- xrDestroySpatialContextEXT

- xrDestroySpatialEntityEXT

- xrDestroySpatialSnapshotEXT

- xrEnumerateSpatialCapabilitiesEXT

- xrEnumerateSpatialCapabilityComponentTypesEXT

- xrEnumerateSpatialCapabilityFeaturesEXT

- xrGetSpatialBufferFloatEXT

- xrGetSpatialBufferStringEXT

- xrGetSpatialBufferUint16EXT

- xrGetSpatialBufferUint32EXT

- xrGetSpatialBufferUint8EXT

- xrGetSpatialBufferVector2fEXT

- xrGetSpatialBufferVector3fEXT

- xrQuerySpatialComponentDataEXT

12.36.16. New Structures

- XrCreateSpatialContextCompletionEXT

- XrCreateSpatialDiscoverySnapshotCompletionEXT

- XrCreateSpatialDiscoverySnapshotCompletionInfoEXT

- XrEventDataSpatialDiscoveryRecommendedEXT

- XrSpatialBounded2DDataEXT

- XrSpatialBufferEXT

- XrSpatialBufferGetInfoEXT

- XrSpatialCapabilityComponentTypesEXT

- XrSpatialCapabilityConfigurationBaseHeaderEXT

- XrSpatialComponentDataQueryConditionEXT

- XrSpatialComponentDataQueryResultEXT

- XrSpatialContextCreateInfoEXT

- XrSpatialDiscoverySnapshotCreateInfoEXT

- XrSpatialEntityFromIdCreateInfoEXT

- XrSpatialMeshDataEXT

- XrSpatialUpdateSnapshotCreateInfoEXT

- Extending XrSpatialComponentDataQueryResultEXT:

  - XrSpatialComponentBounded2DListEXT

  - XrSpatialComponentBounded3DListEXT

  - XrSpatialComponentMesh3DListEXT

  - XrSpatialComponentParentListEXT

- Extending XrSpatialDiscoverySnapshotCreateInfoEXT, XrSpatialComponentDataQueryConditionEXT:

  - XrSpatialFilterTrackingStateEXT

12.36.17. New Enums

- XrSpatialBufferTypeEXT

- XrSpatialCapabilityEXT

- XrSpatialCapabilityFeatureEXT

- XrSpatialComponentTypeEXT

- XrSpatialEntityTrackingStateEXT

12.36.18. New Enum Constants

- XR_EXT_SPATIAL_ENTITY_EXTENSION_NAME

- XR_EXT_spatial_entity_SPEC_VERSION

- Extending XrObjectType:

  - XR_OBJECT_TYPE_SPATIAL_CONTEXT_EXT

  - XR_OBJECT_TYPE_SPATIAL_ENTITY_EXT

  - XR_OBJECT_TYPE_SPATIAL_SNAPSHOT_EXT

- Extending XrResult:

  - XR_ERROR_SPATIAL_BUFFER_ID_INVALID_EXT

  - XR_ERROR_SPATIAL_CAPABILITY_CONFIGURATION_INVALID_EXT

  - XR_ERROR_SPATIAL_CAPABILITY_UNSUPPORTED_EXT

  - XR_ERROR_SPATIAL_COMPONENT_NOT_ENABLED_EXT

  - XR_ERROR_SPATIAL_COMPONENT_UNSUPPORTED_FOR_CAPABILITY_EXT

  - XR_ERROR_SPATIAL_ENTITY_ID_INVALID_EXT

- Extending XrStructureType:

  - XR_TYPE_CREATE_SPATIAL_CONTEXT_COMPLETION_EXT

- XR_TYPE_CREATE_SPATIAL_DISCOVERY_SNAPSHOT_COMPLETION_EXT

- XR_TYPE_CREATE_SPATIAL_DISCOVERY_SNAPSHOT_COMPLETION_INFO_EXT

- XR_TYPE_EVENT_DATA_SPATIAL_DISCOVERY_RECOMMENDED_EXT

- XR_TYPE_SPATIAL_BUFFER_GET_INFO_EXT

- XR_TYPE_SPATIAL_CAPABILITY_COMPONENT_TYPES_EXT

- XR_TYPE_SPATIAL_COMPONENT_BOUNDED_2D_LIST_EXT

- XR_TYPE_SPATIAL_COMPONENT_BOUNDED_3D_LIST_EXT

- XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_CONDITION_EXT

- XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_RESULT_EXT

- XR_TYPE_SPATIAL_COMPONENT_MESH_3D_LIST_EXT

- XR_TYPE_SPATIAL_COMPONENT_PARENT_LIST_EXT

- XR_TYPE_SPATIAL_CONTEXT_CREATE_INFO_EXT

- XR_TYPE_SPATIAL_DISCOVERY_SNAPSHOT_CREATE_INFO_EXT

- XR_TYPE_SPATIAL_ENTITY_FROM_ID_CREATE_INFO_EXT

- XR_TYPE_SPATIAL_FILTER_TRACKING_STATE_EXT

- XR_TYPE_SPATIAL_UPDATE_SNAPSHOT_CREATE_INFO_EXT

12.36.19. Issues

- Does a single entity always derive from solely a single capability?

  - Resolved

  - Answer: No. It is completely upto the runtime based on its own tracking capabilities and how it wants to represent a detected entity. The spec does not prescribe any particular representation of spatial entity except for the guaranteed components of a given capability to set a minimum expectation. A runtime **may** be able to merge entities detected by separate capabilities and represent them as a single entity with the guaranteed components of all the capabilities that helped identify it. An example of this could be that tables can be detected by both a plane tracking capability and an object tracking capability, with plane tracking providing the XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT component on the entity and object tracking providing XR_SPATIAL_COMPONENT_TYPE_BOUNDED_3D_EXT. A certain runtime **may** provide the table as 2 separate entities, each with their own set of guaranteed components, while certain runtimes **may** provide just 1 entity to represent the table, and have both XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT and XR_SPATIAL_COMPONENT_TYPE_BOUNDED_3D_EXT on the same entity. What is important to note here is that a given spatial entity can have at most a single *component* of any given *component type.* Therefore, if the component data produced by the different capabilities conflicts for a certain entity, the runtime **must** represent them as 2 separate entities.

12.36.20. Version History

- Revision 1, 2024-04-12 (Nihav Jain, Google)
    - Initial extension description

12.37. XR_EXT_spatial_marker_tracking

Name String

    `XR_EXT_spatial_marker_tracking`

Extension Type

    Instance extension

Registered Extension Number

    744

Revision

    1

Ratification Status

    Ratified

Extension and Version Dependencies

    `XR_EXT_spatial_entity`

Contributors

    Ron Bessems, Meta
    Nihav Jain, Google
    Natalie Fleury, Meta
    Yuichi Taguchi, Meta
    Yin Li, Microsoft
    Jimmy Alamparambil, ByteDance
    Zhipeng Liu, ByteDance
    Jun Yan, ByteDance

12.37.1. Overview

This extension builds on `XR_EXT_spatial_entity` and allows applications to detect and track markers in their environment. Markers are 2D codes which **may** include QR Codes, Micro QR Codes, ArUco markers, or AprilTags.

A tracked marker is represented as a spatial entity with (or "that has") the following components:

- `XR_SPATIAL_COMPONENT_TYPE_MARKER_EXT`

- `XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT`

12.37.2. Runtime support

A runtime **must** advertise its support for the various marker tracking capabilities using xrEnumerateSpatialCapabilitiesEXT by listing any of the following capabilities:

- `XR_SPATIAL_CAPABILITY_MARKER_TRACKING_QR_CODE_EXT`

- `XR_SPATIAL_CAPABILITY_MARKER_TRACKING_MICRO_QR_CODE_EXT`

- `XR_SPATIAL_CAPABILITY_MARKER_TRACKING_ARUCO_MARKER_EXT`

- `XR_SPATIAL_CAPABILITY_MARKER_TRACKING_APRIL_TAG_EXT`

12.37.3. Configuration

To enable detection of a marker type the application **must** pass the corresponding configuration structure to xrCreateSpatialContextAsyncEXT.

Marker Type Configurations

QR codes

The XrSpatialCapabilityConfigurationQrCodeEXT structure is defined as:

```
// Provided by XR_EXT_spatial_marker_tracking
typedef struct XrSpatialCapabilityConfigurationQrCodeEXT {
    XrStructureType                  type;
    const void*                      next;
    XrSpatialCapabilityEXT           capability;
    uint32_t                         enabledComponentCount;
    const XrSpatialComponentTypeEXT* enabledComponents;
} XrSpatialCapabilityConfigurationQrCodeEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `capability` **must** be `XR_SPATIAL_CAPABILITY_MARKER_TRACKING_QR_CODE_EXT`.

- `enabledComponentCount` is a `uint32_t` with the number of elements in `enabledComponents`.

- `enabledComponents` is a pointer to an array of components to enable for this capability.

If QR codes are supported, the runtime **must** enable QR Code tracking when an XrSpatialCapabilityConfigurationQrCodeEXT structure is passed in XrSpatialContextCreateInfoEXT ::capabilityConfigs when calling xrCreateSpatialContextAsyncEXT.

The runtime **must** return `XR_ERROR_SPATIAL_CAPABILITY_UNSUPPORTED_EXT` if `XR_SPATIAL_CAPABILITY_MARKER_TRACKING_QR_CODE_EXT` is not enumerated by xrEnumerateSpatialCapabilitiesEXT.

Valid Usage (Implicit)

- The `XR_EXT_spatial_marker_tracking` extension **must** be enabled prior to using XrSpatialCapabilityConfigurationQrCodeEXT

- `type` **must** be `XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_QR_CODE_EXT`

- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrSpatialMarkerSizeEXT, XrSpatialMarkerStaticOptimizationEXT

- `capability` **must** be a valid XrSpatialCapabilityEXT value

- `enabledComponents` **must** be a pointer to an array of `enabledComponentCount` valid XrSpatialComponentTypeEXT values

- The `enabledComponentCount` parameter **must** be greater than `0`

Micro QR codes

The XrSpatialCapabilityConfigurationMicroQrCodeEXT structure is defined as:

```
// Provided by XR_EXT_spatial_marker_tracking
typedef struct XrSpatialCapabilityConfigurationMicroQrCodeEXT {
    XrStructureType                  type;
    const void*                      next;
    XrSpatialCapabilityEXT           capability;
    uint32_t                         enabledComponentCount;
    const XrSpatialComponentTypeEXT* enabledComponents;
} XrSpatialCapabilityConfigurationMicroQrCodeEXT;
```

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `capability` **must** be `XR_SPATIAL_CAPABILITY_MARKER_TRACKING_MICRO_QR_CODE_EXT`.

- `enabledComponentCount` is a `uint32_t` with the number of elements in `enabledComponents`.

- `enabledComponents` is a pointer to an array of components to enable for this capability.

If Micro QR codes are supported, the runtime **must** enable Micro QR Code tracking when an XrSpatialCapabilityConfigurationMicroQrCodeEXT structure is passed in XrSpatialContextCreateInfoEXT::`capabilityConfigs` when calling xrCreateSpatialContextAsyncEXT.

The runtime **must** return `XR_ERROR_SPATIAL_CAPABILITY_UNSUPPORTED_EXT` if `XR_SPATIAL_CAPABILITY_MARKER_TRACKING_MICRO_QR_CODE_EXT` is not enumerated by xrEnumerateSpatialCapabilitiesEXT.

Valid Usage (Implicit)

- The `XR_EXT_spatial_marker_tracking` extension **must** be enabled prior to using XrSpatialCapabilityConfigurationMicroQrCodeEXT

- `type` **must** be `XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_MICRO_QR_CODE_EXT`

- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrSpatialMarkerSizeEXT, XrSpatialMarkerStaticOptimizationEXT

- `capability` **must** be a valid XrSpatialCapabilityEXT value

- `enabledComponents` **must** be a pointer to an array of `enabledComponentCount` valid XrSpatialComponentTypeEXT values

- The `enabledComponentCount` parameter **must** be greater than `0`

ArUco Markers

The XrSpatialCapabilityConfigurationArucoMarkerEXT structure is defined as:

```
// Provided by XR_EXT_spatial_marker_tracking
typedef struct XrSpatialCapabilityConfigurationArucoMarkerEXT {
    XrStructureType                    type;
    const void*                        next;
    XrSpatialCapabilityEXT             capability;
    uint32_t                           enabledComponentCount;
    const XrSpatialComponentTypeEXT*   enabledComponents;
    XrSpatialMarkerArucoDictEXT        arUcoDict;
} XrSpatialCapabilityConfigurationArucoMarkerEXT;
```

Member Descriptions

- type is the XrStructureType of this structure.

- next is NULL or a pointer to the next structure in a structure chain.

- capability **must** be XR_SPATIAL_CAPABILITY_MARKER_TRACKING_ARUCO_MARKER_EXT.

- enabledComponentCount is a uint32_t with the number of elements in enabledComponents.

- enabledComponents is a pointer to an array of components to enable for this capability.

- arUcoDict is the marker dictionary to detect.

If ArUco markers are supported, the runtime **must** enable ArUco marker tracking when an XrSpatialCapabilityConfigurationArucoMarkerEXT structure is passed in XrSpatialContextCreateInfoEXT::capabilityConfigs when calling xrCreateSpatialContextAsyncEXT.

The runtime **must** return XR_ERROR_SPATIAL_CAPABILITY_UNSUPPORTED_EXT from xrCreateSpatialContextAsyncEXT if an XrSpatialCapabilityConfigurationArucoMarkerEXT structure is in XrSpatialContextCreateInfoEXT::capabilityConfigs but XR_SPATIAL_CAPABILITY_MARKER_TRACKING_ARUCO_MARKER_EXT is not enumerated by xrEnumerateSpatialCapabilitiesEXT.

The XrSpatialMarkerArucoDictEXT enumeration is defined as:

```
// Provided by XR_EXT_spatial_marker_tracking
typedef enum XrSpatialMarkerArucoDictEXT {
    XR_SPATIAL_MARKER_ARUCO_DICT_4X4_50_EXT = 1,
    XR_SPATIAL_MARKER_ARUCO_DICT_4X4_100_EXT = 2,
    XR_SPATIAL_MARKER_ARUCO_DICT_4X4_250_EXT = 3,
    XR_SPATIAL_MARKER_ARUCO_DICT_4X4_1000_EXT = 4,
    XR_SPATIAL_MARKER_ARUCO_DICT_5X5_50_EXT = 5,
    XR_SPATIAL_MARKER_ARUCO_DICT_5X5_100_EXT = 6,
    XR_SPATIAL_MARKER_ARUCO_DICT_5X5_250_EXT = 7,
    XR_SPATIAL_MARKER_ARUCO_DICT_5X5_1000_EXT = 8,
    XR_SPATIAL_MARKER_ARUCO_DICT_6X6_50_EXT = 9,
    XR_SPATIAL_MARKER_ARUCO_DICT_6X6_100_EXT = 10,
    XR_SPATIAL_MARKER_ARUCO_DICT_6X6_250_EXT = 11,
    XR_SPATIAL_MARKER_ARUCO_DICT_6X6_1000_EXT = 12,
    XR_SPATIAL_MARKER_ARUCO_DICT_7X7_50_EXT = 13,
    XR_SPATIAL_MARKER_ARUCO_DICT_7X7_100_EXT = 14,
    XR_SPATIAL_MARKER_ARUCO_DICT_7X7_250_EXT = 15,
    XR_SPATIAL_MARKER_ARUCO_DICT_7X7_1000_EXT = 16,
    XR_SPATIAL_MARKER_ARUCO_DICT_MAX_ENUM_EXT = 0x7FFFFFFF
} XrSpatialMarkerArucoDictEXT;
```

Supported predefined ArUco dictionary:

<div style="text-align: center;">

Enumerant Descriptions

</div>

- `XR_SPATIAL_MARKER_ARUCO_DICT_4X4_50_EXT` — 4 by 4 pixel Aruco marker dictionary with 50 IDs.
- `XR_SPATIAL_MARKER_ARUCO_DICT_4X4_100_EXT` — 4 by 4 pixel Aruco marker dictionary with 100 IDs.
- `XR_SPATIAL_MARKER_ARUCO_DICT_4X4_250_EXT` — 4 by 4 pixel Aruco marker dictionary with 250 IDs.
- `XR_SPATIAL_MARKER_ARUCO_DICT_4X4_1000_EXT` — 4 by 4 pixel Aruco marker dictionary with 1000 IDs.
- `XR_SPATIAL_MARKER_ARUCO_DICT_5X5_50_EXT` — 5 by 5 pixel Aruco marker dictionary with 50 IDs.
- `XR_SPATIAL_MARKER_ARUCO_DICT_5X5_100_EXT` — 5 by 5 pixel Aruco marker dictionary with 100 IDs.
- `XR_SPATIAL_MARKER_ARUCO_DICT_5X5_250_EXT` — 5 by 5 pixel Aruco marker dictionary with 250 IDs.
- `XR_SPATIAL_MARKER_ARUCO_DICT_5X5_1000_EXT` — 5 by 5 pixel Aruco marker dictionary with 1000 IDs.
- `XR_SPATIAL_MARKER_ARUCO_DICT_6X6_50_EXT` — 6 by 6 pixel Aruco marker dictionary with 50 IDs.
- `XR_SPATIAL_MARKER_ARUCO_DICT_6X6_100_EXT` — 6 by 6 pixel Aruco marker dictionary with 100 IDs.
- `XR_SPATIAL_MARKER_ARUCO_DICT_6X6_250_EXT` — 6 by 6 pixel Aruco marker dictionary with 250 IDs.
- `XR_SPATIAL_MARKER_ARUCO_DICT_6X6_1000_EXT` — 6 by 6 pixel Aruco marker dictionary with 1000 IDs.
- `XR_SPATIAL_MARKER_ARUCO_DICT_7X7_50_EXT` — 7 by 7 pixel Aruco marker dictionary with 50 IDs.
- `XR_SPATIAL_MARKER_ARUCO_DICT_7X7_100_EXT` — 7 by 7 pixel Aruco marker dictionary with 100 IDs.
- `XR_SPATIAL_MARKER_ARUCO_DICT_7X7_250_EXT` — 7 by 7 pixel Aruco marker dictionary with 250 IDs.
- `XR_SPATIAL_MARKER_ARUCO_DICT_7X7_1000_EXT` — 7 by 7 pixel Aruco marker dictionary with 1000 IDs.

AprilTags

The XrSpatialCapabilityConfigurationAprilTagEXT structure is defined as:

```
// Provided by XR_EXT_spatial_marker_tracking
typedef struct XrSpatialCapabilityConfigurationAprilTagEXT {
    XrStructureType                    type;
    const void*                        next;
    XrSpatialCapabilityEXT             capability;
    uint32_t                           enabledComponentCount;
    const XrSpatialComponentTypeEXT*   enabledComponents;
    XrSpatialMarkerAprilTagDictEXT     aprilDict;
} XrSpatialCapabilityConfigurationAprilTagEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `capability` **must** be `XR_SPATIAL_CAPABILITY_MARKER_TRACKING_APRIL_TAG_EXT`.

- `enabledComponentCount` is a `uint32_t` with the number of elements in `enabledComponents`.

- `enabledComponents` is a pointer to an array of components to enable for this capability.

- `aprilDict` is the marker dictionary to detect.

If AprilTags are supported, the runtime **must** enable AprilTag tracking when an XrSpatialCapabilityConfigurationAprilTagEXT structure is passed in XrSpatialContextCreateInfoEXT ::`capabilityConfigs` when calling xrCreateSpatialContextAsyncEXT.

The runtime **must** return `XR_ERROR_SPATIAL_CAPABILITY_UNSUPPORTED_EXT` from xrCreateSpatialContextAsyncEXT if an XrSpatialCapabilityConfigurationAprilTagEXT structure is in XrSpatialContextCreateInfoEXT::`capabilityConfigs` but `XR_SPATIAL_CAPABILITY_MARKER_TRACKING_APRIL_TAG_EXT` is not enumerated by xrEnumerateSpatialCapabilitiesEXT.

- The `XR_EXT_spatial_marker_tracking` extension **must** be enabled prior to using XrSpatialCapabilityConfigurationAprilTagEXT

- `type` **must** be `XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_APRIL_TAG_EXT`

- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrSpatialMarkerSizeEXT, XrSpatialMarkerStaticOptimizationEXT

- `capability` **must** be a valid XrSpatialCapabilityEXT value

- `enabledComponents` **must** be a pointer to an array of `enabledComponentCount` valid XrSpatialComponentTypeEXT values

- `aprilDict` **must** be a valid XrSpatialMarkerAprilTagDictEXT value

- The `enabledComponentCount` parameter **must** be greater than `0`

The XrSpatialMarkerAprilTagDictEXT enumeration is defined as:

```
// Provided by XR_EXT_spatial_marker_tracking
typedef enum XrSpatialMarkerAprilTagDictEXT {
    XR_SPATIAL_MARKER_APRIL_TAG_DICT_16H5_EXT = 1,
    XR_SPATIAL_MARKER_APRIL_TAG_DICT_25H9_EXT = 2,
    XR_SPATIAL_MARKER_APRIL_TAG_DICT_36H10_EXT = 3,
    XR_SPATIAL_MARKER_APRIL_TAG_DICT_36H11_EXT = 4,
    XR_SPATIAL_MARKER_APRIL_TAG_DICT_MAX_ENUM_EXT = 0x7FFFFFFF
} XrSpatialMarkerAprilTagDictEXT;
```

Supported predefined AprilTag dictionary:

Enumerant Descriptions

- `XR_SPATIAL_MARKER_APRIL_TAG_DICT_16H5_EXT` — 4 by 4 bits, minimum Hamming distance between any two codes = 5, 30 codes.

- `XR_SPATIAL_MARKER_APRIL_TAG_DICT_25H9_EXT` — 5 by 5 bits, minimum Hamming distance between any two codes = 9, 35 codes.

- `XR_SPATIAL_MARKER_APRIL_TAG_DICT_36H10_EXT` — 6 by 6 bits, minimum Hamming distance between any two codes = 10, 2320 codes.

- `XR_SPATIAL_MARKER_APRIL_TAG_DICT_36H11_EXT` — 6 by 6 bits, minimum Hamming distance between any two codes = 11, 587 codes.

Optional Marker Configurations

Applications **should** call xrEnumerateSpatialCapabilityFeaturesEXT to get the list of supported optional features.

See XrSpatialCapabilityFeatureEXT for a complete list of all spatial capability features supported by any extension.

Marker Size

The XrSpatialMarkerSizeEXT structure is defined as:

```
// Provided by XR_EXT_spatial_marker_tracking
typedef struct XrSpatialMarkerSizeEXT {
    XrStructureType    type;
    const void*        next;
    float              markerSideLength;
} XrSpatialMarkerSizeEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `markerSideLength` is the size in meters of all markers.

If `XR_SPATIAL_CAPABILITY_FEATURE_MARKER_TRACKING_FIXED_SIZE_MARKERS_EXT` is enumerated by xrEnumerateSpatialCapabilityFeaturesEXT for a certain capability, and if the application chains XrSpatialMarkerSizeEXT to the corresponding configuration structure of that capability, the runtime **must** assume that all markers detected have width and height of `markerSideLength`. Providing this information to the runtime allows the runtime to return a more accurate pose and size. This structure **must** be linked into the `next` chain of XrSpatialCapabilityConfigurationQrCodeEXT, XrSpatialCapabilityConfigurationMicroQrCodeEXT, XrSpatialCapabilityConfigurationArucoMarkerEXT, or XrSpatialCapabilityConfigurationAprilTagEXT.

Valid Usage (Implicit)

- The `XR_EXT_spatial_marker_tracking` extension **must** be enabled prior to using XrSpatialMarkerSizeEXT

- `type` **must** be `XR_TYPE_SPATIAL_MARKER_SIZE_EXT`

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

Static Marker Optimization

The XrSpatialMarkerStaticOptimizationEXT structure is defined as:

```
// Provided by XR_EXT_spatial_marker_tracking
typedef struct XrSpatialMarkerStaticOptimizationEXT {
    XrStructureType    type;
    const void*        next;
    XrBool32           optimizeForStaticMarker;
} XrSpatialMarkerStaticOptimizationEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `optimizeForStaticMarker` indicates if all markers in the space are expected to not move.

If `XR_SPATIAL_CAPABILITY_FEATURE_MARKER_TRACKING_STATIC_MARKERS_EXT` is enumerated by xrEnumerateSpatialCapabilityFeaturesEXT for a certain capability, and if the application chains XrSpatialMarkerStaticOptimizationEXT to the corresponding configuration structure of that capability, the runtime **must** assume that all markers detected are static if `optimizeForStaticMarker` is set to `XR_TRUE`. This allows the runtime to generate a more accurate pose and size. This structure **must** be linked into the `next` chain of XrSpatialCapabilityConfigurationQrCodeEXT, XrSpatialCapabilityConfigurationMicroQrCodeEXT, XrSpatialCapabilityConfigurationArucoMarkerEXT, or XrSpatialCapabilityConfigurationAprilTagEXT.

Valid Usage (Implicit)

- The `XR_EXT_spatial_marker_tracking` extension **must** be enabled prior to using XrSpatialMarkerStaticOptimizationEXT

- `type` **must** be `XR_TYPE_SPATIAL_MARKER_STATIC_OPTIMIZATION_EXT`

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

12.37.4. Guaranteed Components

A runtime that supports `XR_SPATIAL_CAPABILITY_MARKER_TRACKING_QR_CODE_EXT`, `XR_SPATIAL_CAPABILITY_MARKER_TRACKING_MICRO_QR_CODE_EXT`, `XR_SPATIAL_CAPABILITY_MARKER_TRACKING_ARUCO_MARKER_EXT`, or `XR_SPATIAL_CAPABILITY_MARKER_TRACKING_APRIL_TAG_EXT` **must** provide the following spatial components

as guaranteed components of all entities discovered by those capabilities, and **must** enumerate them in xrEnumerateSpatialCapabilityComponentTypesEXT:

- `XR_SPATIAL_COMPONENT_TYPE_MARKER_EXT`

- `XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT`

Marker Component

Component data

The XrSpatialMarkerDataEXT structure is defined as:

```
// Provided by XR_EXT_spatial_marker_tracking
typedef struct XrSpatialMarkerDataEXT {
    XrSpatialCapabilityEXT    capability;
    uint32_t                  markerId;
    XrSpatialBufferEXT        data;
} XrSpatialMarkerDataEXT;
```

<div>

Member Descriptions

- `capability` is the XrSpatialCapabilityEXT that detected the marker.

- `markerId` is the encoded identifier from the marker. For ArUco markers and AprilTag this field **must** be valid and filled with the encoded ID. For QR codes this field **must** be zero.

- `data` is the buffer ID and type of additional information contained in the marker.

</div>

`XR_SPATIAL_CAPABILITY_MARKER_TRACKING_QR_CODE_EXT` and `XR_SPATIAL_CAPABILITY_MARKER_TRACKING_MICRO_QR_CODE_EXT` support extra data. If `capability` is one of these -

- If the runtime has successfully decoded the data for the marker, it **must** set the `data` buffer type to either `XR_SPATIAL_BUFFER_TYPE_UINT8_EXT` or `XR_SPATIAL_BUFFER_TYPE_STRING_EXT`, depending on the data in the marker. The runtime **must** also set a valid buffer ID in `data` which the application **can** use with the appropriate `xrGetSpatialBuffer*` function to get the data.

- If the runtime has not yet decoded the data of the marker, it **must** set `data` buffer ID to XR_NULL_SPATIAL_BUFFER_ID_EXT and the buffer type to `XR_SPATIAL_BUFFER_TYPE_UNKNOWN_EXT`.

`XR_SPATIAL_CAPABILITY_MARKER_TRACKING_ARUCO_MARKER_EXT` and `XR_SPATIAL_CAPABILITY_MARKER_TRACKING_APRIL_TAG_EXT` do not support extra data and the runtime **must** set the buffer ID of `data` to XR_NULL_SPATIAL_BUFFER_ID_EXT.

- The `XR_EXT_spatial_marker_tracking` extension **must** be enabled prior to using XrSpatialMarkerDataEXT

- `capability` **must** be a valid XrSpatialCapabilityEXT value

- `data` **must** be a valid XrSpatialBufferEXT structure

Component list structure to query data

The XrSpatialComponentMarkerListEXT structure is defined as:

```
// Provided by XR_EXT_spatial_marker_tracking
typedef struct XrSpatialComponentMarkerListEXT {
    XrStructureType           type;
    void*                     next;
    uint32_t                  markerCount;
    XrSpatialMarkerDataEXT*   markers;
} XrSpatialComponentMarkerListEXT;
```

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.

- `markerCount` is the number of elements in the `markers` member.

- `markers` is an array of XrSpatialMarkerDataEXT.

The application **can** query the marker component of the spatial entities in an XrSpatialSnapshotEXT by adding `XR_SPATIAL_COMPONENT_TYPE_MARKER_EXT` in XrSpatialComponentDataQueryConditionEXT ::`componentTypes` and adding XrSpatialComponentMarkerListEXT to the next pointer chain of XrSpatialComponentDataQueryResultEXT.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` from xrQuerySpatialComponentDataEXT if XrSpatialComponentMarkerListEXT is in the next chain of XrSpatialComponentDataQueryResultEXT ::`next` but `XR_SPATIAL_COMPONENT_TYPE_MARKER_EXT` is not included in XrSpatialComponentDataQueryConditionEXT::`componentTypes`.

The runtime **must** return `XR_ERROR_SIZE_INSUFFICIENT` from xrQuerySpatialComponentDataEXT if `markerCount` is less than XrSpatialComponentDataQueryResultEXT::`entityIdCountOutput`.

Configuration

If `XR_SPATIAL_COMPONENT_TYPE_MARKER_EXT` is enumerated in XrSpatialCapabilityComponentTypesEXT ::`componentTypes` for some capability, an application **can** enable it by including the enumerant in the XrSpatialCapabilityConfigurationBaseHeaderEXT::`enabledComponents` list of the XrSpatialCapabilityConfigurationBaseHeaderEXT derived structure of the capability that supports this component.

This component does not require any special configuration to be included in the XrSpatialCapabilityConfigurationBaseHeaderEXT::`next` chain.

Bounded 2D Component

The bounded 2D component provides the center and extents of the marker represented by the entity it is on. See Bounded 2D for more details about the bounded 2D component.

The XrSpatialBounded2DDataEXT::`center` **must** point to the center of the marker. When looking at the front face of the marker, the X-axis **must** point to the right, and the Y-axis **must** point to the top of the marker. The runtime **must** follow the right-handed coordinate system convention thus the Z-axis comes out of the front face of the marker. This means that a marker with a position of {0, 0, 0}, rotation of {0, 0, 0, 1} (no rotation), and an extent of {1, 1} refers to a 1 meter x 1 meter marker centered at {0, 0, 0} with its front face normal vector pointing towards the +Z direction in the component's space.

A representation of the orientation of the marker is shown below.
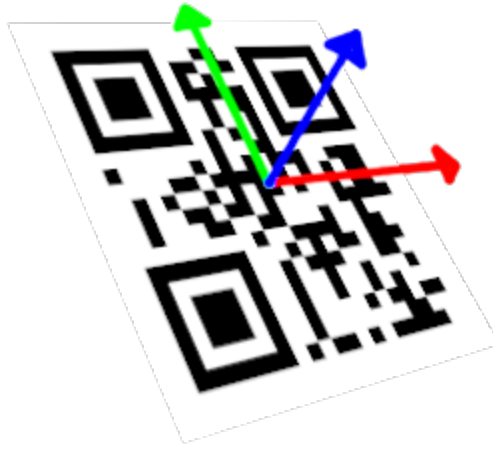
Figure 20. QR code marker with axis

12.37.5. Test Codes

The following codes **must** have their X-Y plane inside the document and the Z-axis pointing at the viewer. The axis origin **must** appear at the center of each marker. The X-axis **must** point to the right, the Y-axis **must** point to the top of the document.
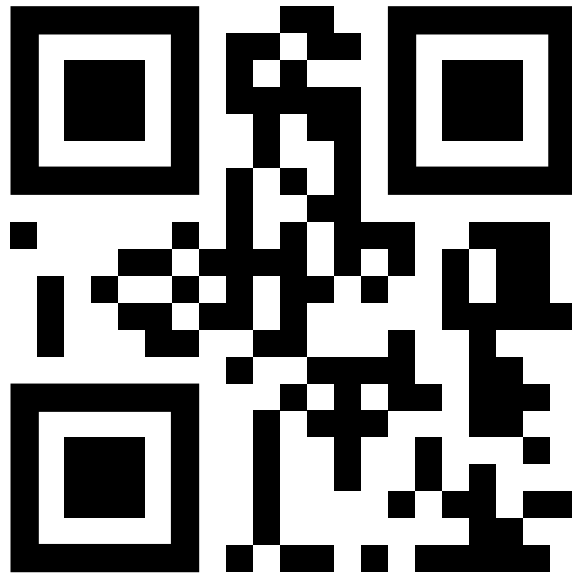
Figure 21. QR code with text 'OpenXR'
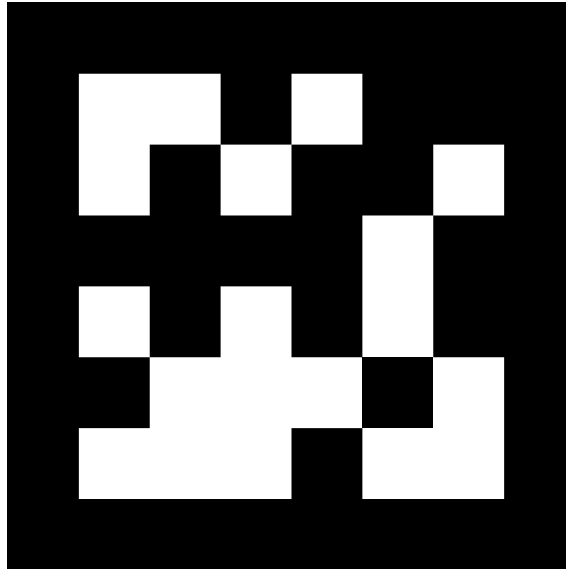
Figure 22. AprilTag `XR_SPATIAL_MARKER_APRIL_TAG_DICT_36H11_EXT` *with ID 42*
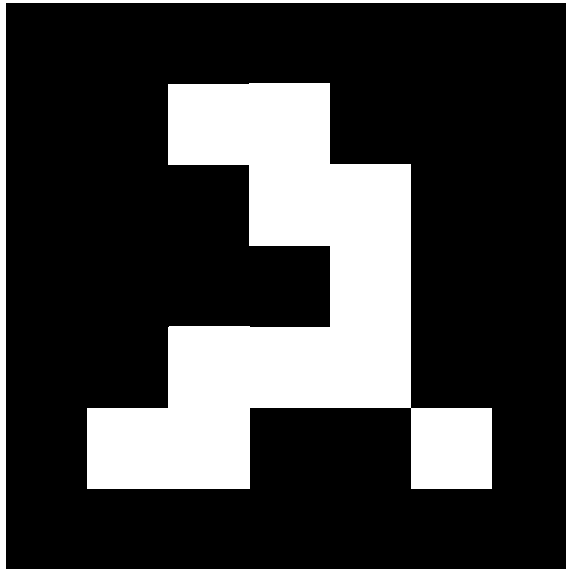
Figure 23. ArUco marker `XR_SPATIAL_MARKER_ARUCO_DICT_5X5_50_EXT` *with ID 43*

12.37.6. Example Code

Configure QR Code Tracking Capability

The following example code demonstrates how to configure the QR code tracking capability when creating a spatial context.

```
// Check if marker tracking capability is supported
uint32_t capabilityCount;
CHK_XR(xrEnumerateSpatialCapabilitiesEXT(instance, systemId, 0, &capabilityCount,
nullptr));
std::vector<XrSpatialCapabilityEXT> capabilities(capabilityCount);
CHK_XR(xrEnumerateSpatialCapabilitiesEXT(instance, systemId, capabilityCount,
&capabilityCount, capabilities.data()));
```

```cpp
if (std::find(capabilities.begin(), capabilities.end(),
XR_SPATIAL_CAPABILITY_MARKER_TRACKING_QR_CODE_EXT) == capabilities.end()) {
  return;
}

uint32_t featureCount = 0;
CHK_XR(xrEnumerateSpatialCapabilityFeaturesEXT(instance, systemId,
XR_SPATIAL_CAPABILITY_MARKER_TRACKING_QR_CODE_EXT, 0, &featureCount, nullptr));
std::vector<XrSpatialCapabilityFeatureEXT> capabilityFeatures(featureCount);
CHK_XR(xrEnumerateSpatialCapabilityFeaturesEXT(instance, systemId,
XR_SPATIAL_CAPABILITY_MARKER_TRACKING_QR_CODE_EXT, featureCount, &featureCount,
capabilityFeatures.data()));

bool supportsFixedMarkerSize = std::find(capabilityFeatures.begin(),
capabilityFeatures.end(),
XR_SPATIAL_CAPABILITY_FEATURE_MARKER_TRACKING_FIXED_SIZE_MARKERS_EXT) !=
capabilityFeatures.end();

// Create a spatial context
XrSpatialContextEXT spatialContext{};

// Enable the 2 guaranteed components of the qr code tracking capability
std::vector<XrSpatialComponentTypeEXT> enabledComponents = {
  XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT,
  XR_SPATIAL_COMPONENT_TYPE_MARKER_EXT,
};
XrSpatialCapabilityConfigurationQrCodeEXT
markerConfiguration{XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_QR_CODE_EXT};
markerConfiguration.capability = XR_SPATIAL_CAPABILITY_MARKER_TRACKING_QR_CODE_EXT;
markerConfiguration.enabledComponentCount =
static_cast<uint32_t>(enabledComponents.size());
markerConfiguration.enabledComponents = enabledComponents.data();

// only chained if features.markerSideLength is true.
XrSpatialMarkerSizeEXT markerSize{XR_TYPE_SPATIAL_MARKER_SIZE_EXT};
markerSize.markerSideLength = 0.10f;
if (supportsFixedMarkerSize) {
  markerConfiguration.next = &markerSize;
}


std::array<XrSpatialCapabilityConfigurationBaseHeaderEXT*, 1> capabilityConfigs = {
  reinterpret_cast<XrSpatialCapabilityConfigurationBaseHeaderEXT*>(&markerConfiguration),
};

XrSpatialContextCreateInfoEXT
spatialContextCreateInfo{XR_TYPE_SPATIAL_CONTEXT_CREATE_INFO_EXT};
```

```
spatialContextCreateInfo.capabilityConfigCount = capabilityConfigs.size();
spatialContextCreateInfo.capabilityConfigs = capabilityConfigs.data();
XrFutureEXT createContextFuture;
CHK_XR(xrCreateSpatialContextAsyncEXT(session, &spatialContextCreateInfo,
&createContextFuture));

waitUntilReady(createContextFuture);

XrCreateSpatialContextCompletionEXT
completion{XR_TYPE_CREATE_SPATIAL_CONTEXT_COMPLETION_EXT};
CHK_XR(xrCreateSpatialContextCompleteEXT(session, createContextFuture, &completion));
if (completion.futureResult != XR_SUCCESS) {
  return;
}

spatialContext = completion.spatialContext;

// ...
// Discovery entities with the spatial context
// ...

CHK_XR(xrDestroySpatialContextEXT(spatialContext));
```

Discover Spatial Entities & Query Component Data

The following example code demonstrates how to discover spatial entities for a context configured
with XR_SPATIAL_CAPABILITY_MARKER_TRACKING_QR_CODE_EXT and query its component data.

```
XrFutureEXT future = XR_NULL_FUTURE_EXT;

// We want to look for entities that have the following components.
std::vector<XrSpatialComponentTypeEXT> snapshotComponents = {
  XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT,
  XR_SPATIAL_COMPONENT_TYPE_MARKER_EXT,
};

auto discoverSpatialEntities = [&](XrSpatialContextEXT spatialContext, XrTime time) {
  XrSpatialDiscoverySnapshotCreateInfoEXT
snapshotCreateInfo{XR_TYPE_SPATIAL_DISCOVERY_SNAPSHOT_CREATE_INFO_EXT};
  snapshotCreateInfo.componentTypeCount = snapshotComponents.size();
  snapshotCreateInfo.componentTypes = snapshotComponents.data();
  CHK_XR(xrCreateSpatialDiscoverySnapshotAsyncEXT(spatialContext, &snapshotCreateInfo,
&future));

  waitUntilReady(future);
```

```
    XrCreateSpatialDiscoverySnapshotCompletionInfoEXT
completionInfo{XR_TYPE_CREATE_SPATIAL_DISCOVERY_SNAPSHOT_COMPLETION_INFO_EXT};
  completionInfo.baseSpace = localSpace;
  completionInfo.time = time;
  completionInfo.future = future;

  XrCreateSpatialDiscoverySnapshotCompletionEXT
completion{XR_TYPE_CREATE_SPATIAL_DISCOVERY_SNAPSHOT_COMPLETION_EXT};
  CHK_XR(xrCreateSpatialDiscoverySnapshotCompleteEXT(spatialContext, &completionInfo,
&completion));
  if (completion.futureResult == XR_SUCCESS) {

    // Query for the bounded2D and marker component data
    XrSpatialComponentDataQueryConditionEXT
queryCond{XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_CONDITION_EXT};
    queryCond.componentTypeCount = snapshotComponents.size();
    queryCond.componentTypes = snapshotComponents.data();

    XrSpatialComponentDataQueryResultEXT
queryResult{XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_RESULT_EXT};
    CHK_XR(xrQuerySpatialComponentDataEXT(completion.snapshot, &queryCond,
&queryResult));

    std::vector<XrSpatialEntityIdEXT> entityIds(queryResult.entityIdCountOutput);
    std::vector<XrSpatialEntityTrackingStateEXT>
entityStates(queryResult.entityIdCountOutput);
    queryResult.entityIdCapacityInput = entityIds.size();
    queryResult.entityIds = entityIds.data();
    queryResult.entityStateCapacityInput = entityStates.size();
    queryResult.entityStates = entityStates.data();

    std::vector<XrSpatialBounded2DDataEXT> bounded2D(queryResult.entityIdCountOutput);
    XrSpatialComponentBounded2DListEXT
bounded2DList{XR_TYPE_SPATIAL_COMPONENT_BOUNDED_2D_LIST_EXT};
    bounded2DList.boundCount = bounded2D.size();
    bounded2DList.bounds = bounded2D.data();
    queryResult.next = &bounded2DList;

    std::vector<XrSpatialMarkerDataEXT> markers;
    XrSpatialComponentMarkerListEXT
markerList{XR_TYPE_SPATIAL_COMPONENT_MARKER_LIST_EXT};
    markers.resize(queryResult.entityIdCountOutput);
    markerList.markerCount = markers.size();
    markerList.markers = markers.data();
    bounded2DList.next = &markerList;

    CHK_XR(xrQuerySpatialComponentDataEXT(completion.snapshot, &queryCond,
&queryResult));
```

```
      for (int32_t i = 0; i < queryResult.entityIdCountOutput; ++i) {
        if (entityStates[i] != XR_SPATIAL_ENTITY_TRACKING_STATE_TRACKING_EXT) {
          continue;
        }

        // 2D bounds for entity entityIds[i] is bounded2D[i].extents centered on
bounded2D[i].center.

        if (markers[i].capability == XR_SPATIAL_CAPABILITY_MARKER_TRACKING_QR_CODE_EXT) {
          // Check if marker data has been decoded.
          if (markers[i].data.bufferId != XR_NULL_SPATIAL_BUFFER_ID_EXT) {
            if (markers[i].data.bufferType == XR_SPATIAL_BUFFER_TYPE_STRING_EXT) {
              // Qr Code data can be queried using
              // XrSpatialBufferGetInfoEXT getInfo{XR_TYPE_SPATIAL_BUFFER_GET_INFO_EXT};
              // info.bufferId = markers[i].data.bufferId;
              // xrGetSpatialBufferStringEXT(completion.snapshot, &getInfo, ...)
            } else if (markers[i].data.bufferType == XR_SPATIAL_BUFFER_TYPE_UINT8_EXT) {
              // Qr Code data can be queried using
              // XrSpatialBufferGetInfoEXT getInfo{XR_TYPE_SPATIAL_BUFFER_GET_INFO_EXT};
              // info.bufferId = markers[i].data.bufferId;
              // xrGetSpatialBufferUint8(completion.snapshot, &getInfo, ...)
            }
          }
        }
      }

    CHK_XR(xrDestroySpatialSnapshotEXT(completion.snapshot));
  }
};

while (1) {
  // ...
  // For every frame in frame loop
  // ...

  XrFrameState frameState;  // previously returned from xrWaitFrame
  const XrTime time = frameState.predictedDisplayTime;

  // Poll for the XR_TYPE_EVENT_DATA_SPATIAL_DISCOVERY_RECOMMENDED_EXT event
  XrEventDataBuffer event = {XR_TYPE_EVENT_DATA_BUFFER};
  XrResult result = xrPollEvent(instance, &event);
  if (result == XR_SUCCESS) {
    switch (event.type) {
      case XR_TYPE_EVENT_DATA_SPATIAL_DISCOVERY_RECOMMENDED_EXT: {
        const XrEventDataSpatialDiscoveryRecommendedEXT& eventdata =
          *reinterpret_cast<XrEventDataSpatialDiscoveryRecommendedEXT*>(&event);
        // Discover spatial entities for the context that we received the
```

```
"discovery
               // recommended" event for.
               discoverSpatialEntities(eventdata.spatialContext, time);
               break;
            }
        }
    }

    // ...
    // Finish frame loop
    // ...
}
```

12.37.7. New Structures

- XrSpatialCapabilityConfigurationAprilTagEXT

- XrSpatialCapabilityConfigurationArucoMarkerEXT

- XrSpatialCapabilityConfigurationMicroQrCodeEXT

- XrSpatialCapabilityConfigurationQrCodeEXT

- XrSpatialMarkerDataEXT

- Extending   XrSpatialCapabilityConfigurationArucoMarkerEXT,
  XrSpatialCapabilityConfigurationAprilTagEXT,   XrSpatialCapabilityConfigurationQrCodeEXT,
  XrSpatialCapabilityConfigurationMicroQrCodeEXT:

  - XrSpatialMarkerSizeEXT

  - XrSpatialMarkerStaticOptimizationEXT

- Extending XrSpatialComponentDataQueryResultEXT:

  - XrSpatialComponentMarkerListEXT

12.37.8. New Enums

- XrSpatialMarkerAprilTagDictEXT

- XrSpatialMarkerArucoDictEXT

12.37.9. New Enum Constants

- XR_EXT_SPATIAL_MARKER_TRACKING_EXTENSION_NAME

- XR_EXT_spatial_marker_tracking_SPEC_VERSION

- Extending XrSpatialCapabilityEXT:

  - XR_SPATIAL_CAPABILITY_MARKER_TRACKING_APRIL_TAG_EXT

  - XR_SPATIAL_CAPABILITY_MARKER_TRACKING_ARUCO_MARKER_EXT

- XR_SPATIAL_CAPABILITY_MARKER_TRACKING_MICRO_QR_CODE_EXT

- XR_SPATIAL_CAPABILITY_MARKER_TRACKING_QR_CODE_EXT

- Extending XrSpatialCapabilityFeatureEXT:

  - XR_SPATIAL_CAPABILITY_FEATURE_MARKER_TRACKING_FIXED_SIZE_MARKERS_EXT

  - XR_SPATIAL_CAPABILITY_FEATURE_MARKER_TRACKING_STATIC_MARKERS_EXT

- Extending XrSpatialComponentTypeEXT:

  - XR_SPATIAL_COMPONENT_TYPE_MARKER_EXT

- Extending XrStructureType:

  - XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_APRIL_TAG_EXT

  - XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_ARUCO_MARKER_EXT

  - XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_MICRO_QR_CODE_EXT

  - XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_QR_CODE_EXT

  - XR_TYPE_SPATIAL_COMPONENT_MARKER_LIST_EXT

  - XR_TYPE_SPATIAL_MARKER_SIZE_EXT

  - XR_TYPE_SPATIAL_MARKER_STATIC_OPTIMIZATION_EXT

12.37.10. Version History

- Revision 1, 2024-07-29 (Ron Bessems, Meta)

  - Initial extension description

12.38. XR_EXT_spatial_persistence

Name String

XR_EXT_spatial_persistence

Extension Type

Instance extension

Registered Extension Number

764

Revision

1

Ratification Status

Ratified

Extension and Version Dependencies

> XR_EXT_spatial_entity
>
> and
>
> XR_EXT_future

Contributors

> Nihav Jain, Google
>
> Jared Finder, Google
>
> Natalie Fleury, Meta
>
> Yuichi Taguchi, Meta
>
> Ron Bessems, Meta
>
> Yin Li, Microsoft
>
> Jimmy Alamparambil, ByteDance
>
> Zhipeng Liu, ByteDance
>
> Jun Yan, ByteDance

12.38.1. Overview

This extension allows applications to discover and correlate spatial entities across application sessions, OpenXR sessions and multiple OpenXR spatial contexts within a session. The XR_EXT_spatial_entity extension established that an entity within an XrSpatialContextEXT is represented by an XrSpatialEntityIdEXT. This extension extends on that concept by establishing that an entity, if persisted, is represented by an XrUuid across application and OpenXR sessions i.e. an application **can** use the XrUuid provided by this extension to identify an entity across sessions. This extension also provides useful overlaps with the XR_EXT_spatial_entity extension to discover persisted entities in the user's environment and the ability to query their component data.

12.38.2. Spatial Persistence Context

Create a spatial persistence context

```
// Provided by XR_EXT_spatial_persistence
XR_DEFINE_HANDLE(XrSpatialPersistenceContextEXT)
```

The XrSpatialPersistenceContextEXT handle represents the connection to a persistent spatial entity storage.

The xrCreateSpatialPersistenceContextAsyncEXT function is defined as:

```
// Provided by XR_EXT_spatial_persistence
XrResult xrCreateSpatialPersistenceContextAsyncEXT(
    XrSession                                   session,
    const XrSpatialPersistenceContextCreateInfoEXT* createInfo,
    XrFutureEXT*                                future);
```

Parameter Descriptions

- session is an XrSession in which the spatial persistence context will be active.

- createInfo is the XrSpatialPersistenceContextCreateInfoEXT used to specify the spatial persistence context parameters.

- future is a pointer to an XrFutureEXT.

An application **can** create an XrSpatialPersistenceContextEXT handle using the xrCreateSpatialPersistenceContextAsyncEXT function and configure the scope of the persistence context in createInfo.

The runtime **must** return XR_ERROR_SPATIAL_PERSISTENCE_SCOPE_UNSUPPORTED_EXT if XrSpatialPersistenceContextCreateInfoEXT::scope is not enumerated by xrEnumerateSpatialPersistenceScopesEXT.

If a runtime enforces a permission system to control application access to the persistence storage represented by XrSpatialPersistenceContextEXT, then the runtime **must** return XR_ERROR_PERMISSION_INSUFFICIENT if those permissions have not been granted to this application.

Valid Usage (Implicit)

- The XR_EXT_spatial_persistence extension **must** be enabled prior to calling xrCreateSpatialPersistenceContextAsyncEXT

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

- createInfo **must** be a pointer to a valid XrSpatialPersistenceContextCreateInfoEXT structure

- future **must** be a pointer to an XrFutureEXT value

The XrSpatialPersistenceContextCreateInfoEXT structure is defined as:

```
// Provided by XR_EXT_spatial_persistence
typedef struct XrSpatialPersistenceContextCreateInfoEXT {
    XrStructureType                  type;
    const void*                      next;
    XrSpatialPersistenceScopeEXT     scope;
} XrSpatialPersistenceContextCreateInfoEXT;
```

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.

- scope is an XrSpatialPersistenceScopeEXT defining the storage scope of the persistence context.

The XrSpatialPersistenceContextCreateInfoEXT structure describes the information to create an XrSpatialPersistenceContextEXT handle.

```
// Provided by XR_EXT_spatial_persistence
typedef enum XrSpatialPersistenceScopeEXT {
    XR_SPATIAL_PERSISTENCE_SCOPE_SYSTEM_MANAGED_EXT = 1,
  // Provided by XR_EXT_spatial_persistence_operations
    XR_SPATIAL_PERSISTENCE_SCOPE_LOCAL_ANCHORS_EXT = 1000781000,
    XR_SPATIAL_PERSISTENCE_SCOPE_MAX_ENUM_EXT = 0x7FFFFFFF
} XrSpatialPersistenceScopeEXT;
```

The XrSpatialPersistenceScopeEXT enumeration identifies the different types of persistence context scopes.

The enums have the following meanings:

Enum	Description
`XR_SPATIAL_PERSISTENCE_SCOPE_SYSTEM_MANAGED_EXT`	Provides the application with read-only access (i.e. application cannot modify the store associated with this scope) to spatial entities persisted and managed by the system. The application **can** use the UUID in the persistence component for this scope to correlate entities across spatial contexts and device reboots.
`XR_SPATIAL_PERSISTENCE_SCOPE_LOCAL_ANCHORS_EXT`	Persistence operations and data access is limited to spatial anchors, on the same device, for the same user and same app (Added by the `XR_EXT_spatial_persistence_operations` extension)

The xrEnumerateSpatialPersistenceScopesEXT function is defined as:

```
// Provided by XR_EXT_spatial_persistence
XrResult xrEnumerateSpatialPersistenceScopesEXT(
    XrInstance                               instance,
    XrSystemId                               systemId,
    uint32_t                                 persistenceScopeCapacityInput,
    uint32_t*                                persistenceScopeCountOutput,
    XrSpatialPersistenceScopeEXT*            persistenceScopes);
```

Parameter Descriptions

- instance is a handle to an XrInstance.

- systemId is the XrSystemId whose spatial persistence scopes will be enumerated.

- persistenceScopeCapacityInput is the capacity of the persistenceScopes array, or 0 to indicate a request to retrieve the required capacity.

- persistenceScopeCountOutput is the number of persistence scopes, or the required capacity in the case that persistenceScopeCapacityInput is insufficient.

- persistenceScopes is an array of XrSpatialPersistenceScopeEXT. It **can** be NULL if persistenceScopeCapacityInput is 0.

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

The application **can** enumerate the list of spatial persistence scopes supported by a given XrSystemId using xrEnumerateSpatialPersistenceScopesEXT.

Valid Usage (Implicit)

- The XR_EXT_spatial_persistence extension **must** be enabled prior to calling xrEnumerateSpatialPersistenceScopesEXT

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

- persistenceScopeCountOutput **must** be a pointer to a uint32_t value

- If persistenceScopeCapacityInput is not 0, persistenceScopes **must** be a pointer to an array of persistenceScopeCapacityInput XrSpatialPersistenceScopeEXT values

The xrCreateSpatialPersistenceContextCompleteEXT function is defined as:

```
// Provided by XR_EXT_spatial_persistence
XrResult xrCreateSpatialPersistenceContextCompleteEXT(
    XrSession                                   session,
    XrFutureEXT                                 future,
    XrCreateSpatialPersistenceContextCompletionEXT* completion);
```

Parameter Descriptions

- session is the XrSession previously passed to xrCreateSpatialPersistenceContextAsyncEXT ::session.

- future is the XrFutureEXT received from xrCreateSpatialPersistenceContextAsyncEXT.

- completion is a pointer to an XrCreateSpatialPersistenceContextCompletionEXT.

xrCreateSpatialPersistenceContextCompleteEXT completes the asynchronous operation started by xrCreateSpatialPersistenceContextAsyncEXT. The runtime **must** return XR_ERROR_FUTURE_PENDING_EXT if future is not in ready state. The runtime **must** return XR_ERROR_FUTURE_INVALID_EXT if future has already been completed or cancelled.

Valid Usage (Implicit)

- The `XR_EXT_spatial_persistence` extension **must** be enabled prior to calling xrCreateSpatialPersistenceContextCompleteEXT

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

- `completion` **must** be a pointer to an XrCreateSpatialPersistenceContextCompletionEXT 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_OUT_OF_MEMORY`

- `XR_ERROR_LIMIT_REACHED`

- `XR_ERROR_FUTURE_PENDING_EXT`

- `XR_ERROR_FUTURE_INVALID_EXT`

The XrCreateSpatialPersistenceContextCompletionEXT structure is defined as:

```
// Provided by XR_EXT_spatial_persistence
typedef struct XrCreateSpatialPersistenceContextCompletionEXT {
    XrStructureType                       type;
    void*                                 next;
    XrResult                              futureResult;
    XrSpatialPersistenceContextResultEXT  createResult;
    XrSpatialPersistenceContextEXT        persistenceContext;
} XrCreateSpatialPersistenceContextCompletionEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `futureResult` is the XrResult of the spatial persistence context creation operation.

- `createResult` is the XrSpatialPersistenceContextResultEXT of the spatial persistence context creation operation.

- `persistenceContext` is an XrSpatialPersistenceContextEXT created using the data and configuration in xrCreateSpatialPersistenceContextAsyncEXT::`createInfo`.

Future Return Codes

`futureResult` values:

Success

- `XR_SUCCESS`

- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_RUNTIME_FAILURE`

- `XR_ERROR_INSTANCE_LOST`

- `XR_ERROR_SESSION_LOST`

- `XR_ERROR_OUT_OF_MEMORY`

- `XR_ERROR_LIMIT_REACHED`

- `XR_ERROR_PERMISSION_INSUFFICIENT`

If `futureResult` and `createResult` are both success codes, `persistenceContext` **must** be valid. If `persistenceContext` is valid, it **must** remain so within the lifecycle of xrCreateSpatialPersistenceContextAsyncEXT::`session` or until the application uses

xrDestroySpatialPersistenceContextEXT with `persistenceContext`, whichever comes first.

The runtime **must** set `createResult` only if `futureResult` is a success code.

<div style="border:1px solid #ccc; padding:1em;">

Valid Usage (Implicit)

- The `XR_EXT_spatial_persistence` extension **must** be enabled prior to using XrCreateSpatialPersistenceContextCompletionEXT

- `type` **must** be `XR_TYPE_CREATE_SPATIAL_PERSISTENCE_CONTEXT_COMPLETION_EXT`

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

- `futureResult` **must** be a valid XrResult value

- `createResult` **must** be a valid XrSpatialPersistenceContextResultEXT value

- `persistenceContext` **must** be a valid XrSpatialPersistenceContextEXT handle

</div>

```
// Provided by XR_EXT_spatial_persistence
typedef enum XrSpatialPersistenceContextResultEXT {
    XR_SPATIAL_PERSISTENCE_CONTEXT_RESULT_SUCCESS_EXT = 0,
  // Provided by XR_EXT_spatial_persistence_operations
    XR_SPATIAL_PERSISTENCE_CONTEXT_RESULT_ENTITY_NOT_TRACKING_EXT = -1000781001,
  // Provided by XR_EXT_spatial_persistence_operations
    XR_SPATIAL_PERSISTENCE_CONTEXT_RESULT_PERSIST_UUID_NOT_FOUND_EXT = -1000781002,
    XR_SPATIAL_PERSISTENCE_CONTEXT_RESULT_MAX_ENUM_EXT = 0x7FFFFFFF
} XrSpatialPersistenceContextResultEXT;
```

The XrSpatialPersistenceContextResultEXT enumeration identifies the different types of result codes for a persistence operation. Failures during persistence operations are not always in control of the application and this enumeration is used for conveying such cases.

The enums have the following meanings:

Enum	Description
`XR_SPATIAL_PERSISTENCE_CONTEXT_RESULT_SUCCESS_EXT`	The persistence context operation was a success.
`XR_SPATIAL_PERSISTENCE_CONTEXT_RESULT_ENTITY_NOT_TRACKING_EXT`	The persistence operation failed because the entity could not be tracked by the runtime. (Added by the XR_EXT_spatial_persistence_operations extension)

Enum	Description
`XR_SPATIAL_PERSISTENCE_CONTEXT_RESULT_PERSIST_UUID_NOT_FOUND_EXT`	The provided persist UUID was not found in the storage. (Added by the `XR_EXT_spatial_persistence_operations` extension)

Destroy the spatial persistence context

The xrDestroySpatialPersistenceContextEXT function is defined as:

```
// Provided by XR_EXT_spatial_persistence
XrResult xrDestroySpatialPersistenceContextEXT(
    XrSpatialPersistenceContextEXT              persistenceContext);
```

Parameter Descriptions

- `persistenceContext` is an XrSpatialPersistenceContextEXT previously created by xrCreateSpatialPersistenceContextAsyncEXT.

The application **can** use xrDestroySpatialPersistenceContextEXT to release the `persistenceContext` handle when it is finished with spatial persistence tasks.

The runtime **must** not destroy the underlying resources for `persistenceContext` when xrDestroySpatialPersistenceContextEXT is called if there are any valid XrSpatialContextEXT handles that `persistenceContext` was linked to via XrSpatialContextPersistenceConfigEXT. This is because the persistence context's resources are still used by the spatial context for discovering persisted entities. Destroying the persistence context handle in such a situation only removes the application's access to these resources.

The resources for a destroyed XrSpatialPersistenceContextEXT **must** be freed when all the XrSpatialContextEXT handles the persistence context was linked to are destroyed.

Valid Usage (Implicit)

- The `XR_EXT_spatial_persistence` extension **must** be enabled prior to calling xrDestroySpatialPersistenceContextEXT

- `persistenceContext` **must** be a valid XrSpatialPersistenceContextEXT handle

12.38.3. Discover persisted entities

Persistence component

Persisted spatial entities have the persistence component on them which the runtime **must** include in the discovery and update snapshots if `XR_SPATIAL_COMPONENT_TYPE_PERSISTENCE_EXT` is enabled during the creation of XrSpatialContextEXT and included in XrSpatialDiscoverySnapshotCreateInfoEXT ::`componentTypes` or XrSpatialUpdateSnapshotCreateInfoEXT::`componentTypes`.

Component Data

The XrSpatialPersistenceDataEXT structure is defined as:

```
// Provided by XR_EXT_spatial_persistence
typedef struct XrSpatialPersistenceDataEXT {
    XrUuid                       persistUuid;
    XrSpatialPersistenceStateEXT    persistState;
} XrSpatialPersistenceDataEXT;
```

> **Valid Usage (Implicit)**
>
> - The `XR_EXT_spatial_persistence` extension **must** be enabled prior to using XrSpatialPersistenceDataEXT
>
> - persistState **must** be a valid XrSpatialPersistenceStateEXT value

```
// Provided by XR_EXT_spatial_persistence
typedef enum XrSpatialPersistenceStateEXT {
    XR_SPATIAL_PERSISTENCE_STATE_LOADED_EXT = 1,
    XR_SPATIAL_PERSISTENCE_STATE_NOT_FOUND_EXT = 2,
    XR_SPATIAL_PERSISTENCE_STATE_MAX_ENUM_EXT = 0x7FFFFFFF
} XrSpatialPersistenceStateEXT;
```

The XrSpatialPersistenceStateEXT enumeration identifies the different states of the persisted uuid.

The enums have the following meanings:

Enum	Description
XR_SPATIAL_PERSISTENCE_STATE_LOADED_EXT	The persisted UUID has been successfully loaded from the storage.
XR_SPATIAL_PERSISTENCE_STATE_NOT_FOUND_EXT	The persisted UUID was not found in the storage and was either removed from it or never was in it.

Component list structure to query data

The XrSpatialComponentPersistenceListEXT structure is defined as:

```
// Provided by XR_EXT_spatial_persistence
typedef struct XrSpatialComponentPersistenceListEXT {
    XrStructureType                type;
    void*                          next;
    uint32_t                       persistDataCount;
    XrSpatialPersistenceDataEXT*   persistData;
} XrSpatialComponentPersistenceListEXT;
```

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.

- `persistDataCount` is a `uint32_t` describing the count of elements in the `persistData` array.

- `persistData` is an array of XrSpatialPersistenceDataEXT.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` from xrQuerySpatialComponentDataEXT if XrSpatialComponentPersistenceListEXT is in the next chain of XrSpatialComponentDataQueryResultEXT::`next` but `XR_SPATIAL_COMPONENT_TYPE_PERSISTENCE_EXT` is not included in XrSpatialComponentDataQueryConditionEXT::`componentTypes`.

The runtime **must** return `XR_ERROR_SIZE_INSUFFICIENT` from xrQuerySpatialComponentDataEXT if `persistDataCount` is less than XrSpatialComponentDataQueryResultEXT::`entityIdCountOutput`.

Unlike the other components, the runtime **must** set the data for `XR_SPATIAL_COMPONENT_TYPE_PERSISTENCE_EXT` regardless of the XrSpatialEntityTrackingStateEXT.

Valid Usage (Implicit)

- The `XR_EXT_spatial_persistence` extension **must** be enabled prior to using XrSpatialComponentPersistenceListEXT

- `type` **must** be `XR_TYPE_SPATIAL_COMPONENT_PERSISTENCE_LIST_EXT`

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

- `persistData` **must** be a pointer to an array of `persistDataCount` XrSpatialPersistenceDataEXT structures

- The `persistDataCount` parameter **must** be greater than `0`

Configuration

If `XR_SPATIAL_COMPONENT_TYPE_PERSISTENCE_EXT` is enumerated in XrSpatialCapabilityComponentTypesEXT::`componentTypes` for some capability, the application **can** enable it by including the enum in the XrSpatialCapabilityConfigurationBaseHeaderEXT ::`enabledComponents` list. This component does not require any special configuration to be included in the next chain of XrSpatialCapabilityConfigurationBaseHeaderEXT. If the application is including `XR_SPATIAL_COMPONENT_TYPE_PERSISTENCE_EXT` in the enabled component list, it **must** also include XrSpatialContextPersistenceConfigEXT in the next chain of XrSpatialContextCreateInfoEXT otherwise the runtime **must** return `XR_ERROR_SPATIAL_CAPABILITY_CONFIGURATION_INVALID_EXT` from xrCreateSpatialContextAsyncEXT.

Configure spatial context with persistence contexts

The XrSpatialContextPersistenceConfigEXT structure is defined as:

```
// Provided by XR_EXT_spatial_persistence
typedef struct XrSpatialContextPersistenceConfigEXT {
    XrStructureType                      type;
    const void*                          next;
    uint32_t                             persistenceContextCount;
    const XrSpatialPersistenceContextEXT*   persistenceContexts;
} XrSpatialContextPersistenceConfigEXT;
```

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.

- persistenceContextCount is a uint32_t describing the count of elements in the persistenceContexts array.

- persistenceContexts is an an array of XrSpatialPersistenceContextEXT.

An application **can** add XrSpatialContextPersistenceConfigEXT to the next chain of XrSpatialContextCreateInfoEXT. This will configure the created XrSpatialContextEXT with persistenceContexts and allow the application to discover the spatial entities persisted in the storage represented by the XrSpatialPersistenceContextEXT handles in persistenceContexts.

Valid Usage (Implicit)

- The XR_EXT_spatial_persistence extension **must** be enabled prior to using XrSpatialContextPersistenceConfigEXT

- type **must** be XR_TYPE_SPATIAL_CONTEXT_PERSISTENCE_CONFIG_EXT

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

- persistenceContexts **must** be a pointer to an array of persistenceContextCount valid XrSpatialPersistenceContextEXT handles

- The persistenceContextCount parameter **must** be greater than 0

Create discovery snapshot

Discover entities with specific UUIDs

The XrSpatialDiscoveryPersistenceUuidFilterEXT structure is defined as:

```
// Provided by XR_EXT_spatial_persistence
typedef struct XrSpatialDiscoveryPersistenceUuidFilterEXT {
    XrStructureType    type;
    const void*        next;
    uint32_t           persistedUuidCount;
    const XrUuid*      persistedUuids;
} XrSpatialDiscoveryPersistenceUuidFilterEXT;
```

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.

- `persistedUuidCount` is a `uint32_t` describing the count of elements in the `persistedUuids` array

- `persistedUuids` is an array of XrUuid.

The application **can** use XrSpatialDiscoveryPersistenceUuidFilterEXT in the next chain of XrSpatialDiscoverySnapshotCreateInfoEXT to scope the discovery operation to just the entities whose persisted UUIDs are in the set of the UUIDs provided in `persistedUuids`.

If the application adds XrSpatialDiscoveryPersistenceUuidFilterEXT in the next chain of XrSpatialDiscoverySnapshotCreateInfoEXT but the xrCreateSpatialDiscoverySnapshotAsyncEXT ::spatialContext was not configured with any XrSpatialPersistenceContextEXT using XrSpatialContextPersistenceConfigEXT, the runtime **must** return `XR_ERROR_VALIDATION_FAILURE` from xrCreateSpatialDiscoverySnapshotAsyncEXT.

The runtime **must** treat the XrSpatialDiscoveryPersistenceUuidFilterEXT filter as an 'AND' condition with any other filters provided in XrSpatialDiscoverySnapshotCreateInfoEXT or its next chain. The runtime **must** treat the `persistedUuids` array itself as an 'OR' condition i.e. filter for entities that have any of the UUIDs provided in that array.

The runtime **must** include one entry in the created snapshot for each of the UUIDs in `persistedUuids` for which it was able to determine the XrSpatialPersistenceStateEXT state at this time.

- If the runtime has successfully found the UUID in its storage, then -

- The runtime **must** set the XrSpatialPersistenceStateEXT in the XrSpatialPersistenceDataEXT of this entity to `XR_SPATIAL_PERSISTENCE_STATE_LOADED_EXT`.

- The runtime **must** include a valid XrSpatialEntityIdEXT for this entity in the created snapshot.

- The runtime **must** set the XrSpatialEntityTrackingStateEXT of that entity to `XR_SPATIAL_ENTITY_TRACKING_STATE_TRACKING_EXT` if it is actively tracking the entity and has valid data for its components. Otherwise, the runtime **must** set the XrSpatialEntityTrackingStateEXT to `XR_SPATIAL_ENTITY_TRACKING_STATE_PAUSED_EXT`.

- If the runtime has determined that the UUID is not present in its storage (regardless of whether that UUID was never in the storage or has was present once but has since been unpersisted), then -

  - The runtime **must** set the XrSpatialPersistenceStateEXT in the XrSpatialPersistenceDataEXT of this entity to `XR_SPATIAL_PERSISTENCE_STATE_NOT_FOUND_EXT` to indicate to the application that this UUID is no longer present in the storage.

  - The runtime **must** set the XrSpatialEntityIdEXT for this entity in the created snapshot to XR_NULL_SPATIAL_ENTITY_ID_EXT.

  - The runtime **must** set the XrSpatialEntityTrackingStateEXT of that entity to `XR_SPATIAL_ENTITY_TRACKING_STATE_STOPPED_EXT` to indicate to the application that this entity will never be tracked.

- If the runtime was not able to determine if the UUID is present in its storage or not, it **must** not include it in the snapshot.

The application **can** also use XrSpatialDiscoveryPersistenceUuidFilterEXT in the next chain of XrSpatialComponentDataQueryConditionEXT to query for entities of specific UUIDs in existing snapshots. When used with XrSpatialComponentDataQueryConditionEXT, if XrSpatialDiscoveryPersistenceUuidFilterEXT::`persistedUuids` contains any XrUuid that is not in the XrSpatialSnapshotEXT, the runtime **must** not include an entry for that XrUuid in the query result. Also, the order (sequence) of entities in the query result **may** not match the order of UUIDs provided in XrSpatialDiscoveryPersistenceUuidFilterEXT::`persistedUuids`. Application **should** include `XR_SPATIAL_COMPONENT_TYPE_PERSISTENCE_EXT` in XrSpatialComponentDataQueryConditionEXT::`componentTypes` and XrSpatialComponentPersistenceListEXT in the next chain of XrSpatialComponentDataQueryResultEXT and then check the XrSpatialPersistenceDataEXT::`persistUuid` of the each query result index to understand which UUID the current result index corresponds to.

- The `XR_EXT_spatial_persistence` extension **must** be enabled prior to using XrSpatialDiscoveryPersistenceUuidFilterEXT

- `type` **must** be `XR_TYPE_SPATIAL_DISCOVERY_PERSISTENCE_UUID_FILTER_EXT`

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

- `persistedUuids` **must** be a pointer to an array of `persistedUuidCount` XrUuid structures

- The `persistedUuidCount` parameter **must** be greater than `0`

Discover all persisted entities

If the application uses xrCreateSpatialDiscoverySnapshotAsyncEXT without XrSpatialDiscoveryPersistenceUuidFilterEXT and with an XrSpatialContextEXT which has been configured with an XrSpatialPersistenceContextEXT, then the runtime **must** include those entities in the created snapshot that are persisted in the storage represented by XrSpatialPersistenceContextEXT and satisfy the filters provided in XrSpatialDiscoverySnapshotCreateInfoEXT. For those entities -

- The runtime **must** set the XrSpatialPersistenceStateEXT in the XrSpatialPersistenceDataEXT of this entity to `XR_SPATIAL_PERSISTENCE_STATE_LOADED_EXT`.

- The runtime **must** include a valid `XrSpatialEntityIdEXT` for this entity in the created snapshot.

- The runtime **must** set the XrSpatialEntityTrackingStateEXT of that entity to `XR_SPATIAL_ENTITY_TRACKING_STATE_TRACKING_EXT` if it is actively tracking the entity and has valid data for its components. Otherwise, the runtime **must** set the XrSpatialEntityTrackingStateEXT to `XR_SPATIAL_ENTITY_TRACKING_STATE_PAUSED_EXT`.

12.38.4. New Object Types

- XrSpatialPersistenceContextEXT

12.38.5. New Commands

- xrCreateSpatialPersistenceContextAsyncEXT

- xrCreateSpatialPersistenceContextCompleteEXT

- xrDestroySpatialPersistenceContextEXT

- xrEnumerateSpatialPersistenceScopesEXT

12.38.6. New Structures

- XrCreateSpatialPersistenceContextCompletionEXT

- XrSpatialPersistenceContextCreateInfoEXT

- XrSpatialPersistenceDataEXT

- Extending XrSpatialComponentDataQueryResultEXT:

  - XrSpatialComponentPersistenceListEXT

- Extending XrSpatialContextCreateInfoEXT:

  - XrSpatialContextPersistenceConfigEXT

- Extending XrSpatialDiscoverySnapshotCreateInfoEXT, XrSpatialComponentDataQueryConditionEXT:

  - XrSpatialDiscoveryPersistenceUuidFilterEXT

12.38.7. New Enums

- XrSpatialPersistenceContextResultEXT

- XrSpatialPersistenceScopeEXT

- XrSpatialPersistenceStateEXT

12.38.8. New Enum Constants

- XR_EXT_SPATIAL_PERSISTENCE_EXTENSION_NAME

- XR_EXT_spatial_persistence_SPEC_VERSION

- Extending XrObjectType:

  - XR_OBJECT_TYPE_SPATIAL_PERSISTENCE_CONTEXT_EXT

- Extending XrResult:

  - XR_ERROR_SPATIAL_PERSISTENCE_SCOPE_UNSUPPORTED_EXT

- Extending XrSpatialComponentTypeEXT:

  - XR_SPATIAL_COMPONENT_TYPE_PERSISTENCE_EXT

- Extending XrStructureType:

  - XR_TYPE_CREATE_SPATIAL_PERSISTENCE_CONTEXT_COMPLETION_EXT

  - XR_TYPE_SPATIAL_COMPONENT_PERSISTENCE_LIST_EXT

  - XR_TYPE_SPATIAL_CONTEXT_PERSISTENCE_CONFIG_EXT

  - XR_TYPE_SPATIAL_DISCOVERY_PERSISTENCE_UUID_FILTER_EXT

  - XR_TYPE_SPATIAL_PERSISTENCE_CONTEXT_CREATE_INFO_EXT

12.38.9. Example Code

Create Persistence Context

```
// Check if the required persistence scope supported
uint32_t scopeCount;
CHK_XR(xrEnumerateSpatialPersistenceScopesEXT(instance, systemId, 0, &scopeCount,
nullptr));
std::vector<XrSpatialPersistenceScopeEXT> persistenceScopes(scopeCount);
CHK_XR(xrEnumerateSpatialPersistenceScopesEXT(instance, systemId, scopeCount,
&scopeCount, persistenceScopes.data()));

if (std::find(persistenceScopes.begin(), persistenceScopes.end(),
XR_SPATIAL_PERSISTENCE_SCOPE_SYSTEM_MANAGED_EXT) == persistenceScopes.end()) {
  return;
}

XrSpatialPersistenceContextEXT persistenceContext{};

XrSpatialPersistenceContextCreateInfoEXT
persistenceContextCreateInfo{XR_TYPE_SPATIAL_PERSISTENCE_CONTEXT_CREATE_INFO_EXT};
persistenceContextCreateInfo.scope = XR_SPATIAL_PERSISTENCE_SCOPE_SYSTEM_MANAGED_EXT;
XrFutureEXT createContextFuture;
CHK_XR(xrCreateSpatialPersistenceContextAsyncEXT(session, &persistenceContextCreateInfo,
&createContextFuture));

waitUntilReady(createContextFuture);

XrCreateSpatialPersistenceContextCompletionEXT
completion{XR_TYPE_CREATE_SPATIAL_PERSISTENCE_CONTEXT_COMPLETION_EXT};
CHK_XR(xrCreateSpatialPersistenceContextCompleteEXT(session, createContextFuture,
&completion));
if (completion.futureResult != XR_SUCCESS || completion.createResult !=
XR_SPATIAL_PERSISTENCE_CONTEXT_RESULT_SUCCESS_EXT) {
  return;
}

persistenceContext = completion.persistenceContext;

// ...
// Connect persistence context to a spatial context and discover persisted entities.
// ...

CHK_XR(xrDestroySpatialPersistenceContextEXT(persistenceContext));
```

Connect Persistence Context to a Spatial Context

```
// Note: Anchor capability is just used as an example here. Persistence can be
// supported by other capabilities too. xrEnumerateSpatialCapabilityComponentTypesEXT()
can
```

```
// be used to check if a certain capability supports persistence.
if (!isSpatialCapabilitySupported(instance, systemId, XR_SPATIAL_CAPABILITY_ANCHOR_EXT))
{
  return;
}

const bool supportsPersistenceComponent = isSpatialComponentSupported(instance, systemId,
XR_SPATIAL_CAPABILITY_ANCHOR_EXT, XR_SPATIAL_COMPONENT_TYPE_PERSISTENCE_EXT);

// Create a spatial spatial context
XrSpatialContextEXT spatialContext{};
{

  std::vector<XrSpatialComponentTypeEXT> enabledComponents = {
    XR_SPATIAL_COMPONENT_TYPE_ANCHOR_EXT,
  };

  if (supportsPersistenceComponent) {
    enabledComponents.push_back(XR_SPATIAL_COMPONENT_TYPE_PERSISTENCE_EXT);
  }

  XrSpatialCapabilityConfigurationAnchorEXT
anchorConfig{XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_ANCHOR_EXT};
  anchorConfig.capability = XR_SPATIAL_CAPABILITY_ANCHOR_EXT;
  anchorConfig.enabledComponentCount = enabledComponents.size();
  anchorConfig.enabledComponents = enabledComponents.data();

  std::array<XrSpatialCapabilityConfigurationBaseHeaderEXT*, 1> capabilityConfigs = {
    reinterpret_cast<XrSpatialCapabilityConfigurationBaseHeaderEXT*>(&anchorConfig),
  };

  XrSpatialContextCreateInfoEXT
spatialContextCreateInfo{XR_TYPE_SPATIAL_CONTEXT_CREATE_INFO_EXT};
  spatialContextCreateInfo.capabilityConfigCount = capabilityConfigs.size();
  spatialContextCreateInfo.capabilityConfigs = capabilityConfigs.data();

  XrSpatialContextPersistenceConfigEXT
persistenceConfig{XR_TYPE_SPATIAL_CONTEXT_PERSISTENCE_CONFIG_EXT};
  persistenceConfig.persistenceContextCount = 1;
  persistenceConfig.persistenceContexts = &persistenceContext;

  if (supportsPersistenceComponent) {
    spatialContextCreateInfo.next = &persistenceConfig;
  }

  XrFutureEXT createContextFuture;
  CHK_XR(xrCreateSpatialContextAsyncEXT(session, &spatialContextCreateInfo,
&createContextFuture));
```

```
  waitUntilReady(createContextFuture);

  XrCreateSpatialContextCompletionEXT
completion{XR_TYPE_CREATE_SPATIAL_CONTEXT_COMPLETION_EXT};
  CHK_XR(xrCreateSpatialContextCompleteEXT(session, createContextFuture, &completion));
  if (completion.futureResult != XR_SUCCESS) {
    return;
  }

  spatialContext = completion.spatialContext;
}

// ...
// Discover persisted anchors.
// ...

CHK_XR(xrDestroySpatialContextEXT(spatialContext));
```

Discover all persisted entities

```
XrFutureEXT future = XR_NULL_FUTURE_EXT;

// We want to look for entities that have the following components.
std::vector<XrSpatialComponentTypeEXT> snapshotComponents = {
  XR_SPATIAL_COMPONENT_TYPE_PERSISTENCE_EXT,
};

auto discoverSpatialEntities = [&](XrSpatialContextEXT spatialContext, XrTime time) {
  XrSpatialDiscoverySnapshotCreateInfoEXT
snapshotCreateInfo{XR_TYPE_SPATIAL_DISCOVERY_SNAPSHOT_CREATE_INFO_EXT};
  snapshotCreateInfo.componentTypeCount = snapshotComponents.size();
  snapshotCreateInfo.componentTypes = snapshotComponents.data();
  CHK_XR(xrCreateSpatialDiscoverySnapshotAsyncEXT(spatialContext, &snapshotCreateInfo,
&future));

  waitUntilReady(future);

  XrCreateSpatialDiscoverySnapshotCompletionInfoEXT
completionInfo{XR_TYPE_CREATE_SPATIAL_DISCOVERY_SNAPSHOT_COMPLETION_INFO_EXT};
  completionInfo.baseSpace = localSpace;
  completionInfo.time = time;
  completionInfo.future = future;

  XrCreateSpatialDiscoverySnapshotCompletionEXT
completion{XR_TYPE_CREATE_SPATIAL_DISCOVERY_SNAPSHOT_COMPLETION_EXT};
```

```
    CHK_XR(xrCreateSpatialDiscoverySnapshotCompleteEXT(spatialContext, &completionInfo,
&completion));
    if (completion.futureResult == XR_SUCCESS) {

        // Query for the semantic label component data
        XrSpatialComponentDataQueryConditionEXT
queryCond{XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_CONDITION_EXT};
        queryCond.componentTypeCount = snapshotComponents.size();
        queryCond.componentTypes = snapshotComponents.data();

        XrSpatialComponentDataQueryResultEXT
queryResult{XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_RESULT_EXT};
        CHK_XR(xrQuerySpatialComponentDataEXT(completion.snapshot, &queryCond,
&queryResult));

        std::vector<XrSpatialEntityIdEXT> entityIds(queryResult.entityIdCountOutput);
        std::vector<XrSpatialEntityTrackingStateEXT>
entityStates(queryResult.entityIdCountOutput);
        queryResult.entityIdCapacityInput = entityIds.size();
        queryResult.entityIds = entityIds.data();
        queryResult.entityStateCapacityInput = entityStates.size();
        queryResult.entityStates = entityStates.data();

        std::vector<XrSpatialPersistenceDataEXT>
persistenceData(queryResult.entityIdCountOutput);
        XrSpatialComponentPersistenceListEXT
persistenceDataList{XR_TYPE_SPATIAL_COMPONENT_PERSISTENCE_LIST_EXT};
        persistenceDataList.persistDataCount = persistenceData.size();
        persistenceDataList.persistData = persistenceData.data();
        queryResult.next = &persistenceDataList;

        CHK_XR(xrQuerySpatialComponentDataEXT(completion.snapshot, &queryCond,
&queryResult));

        for (int32_t i = 0; i < queryResult.entityIdCountOutput; ++i) {
            // persistenceData[i].persistUuid is the UUID of the persisted entity whose entity
ID is entityIds[i].
            // The persistenceData array is essentially the uuids persisted in the scope that
the current
            // XrSpatialPersistenceContextEXT is configured with.
        }

        CHK_XR(xrDestroySpatialSnapshotEXT(completion.snapshot));
    }
};

while (1) {
    // ...
```

```
  // For every frame in frame loop
  // ...

  XrFrameState frameState;  // previously returned from xrWaitFrame
  const XrTime time = frameState.predictedDisplayTime;

  // Poll for the XR_TYPE_EVENT_DATA_SPATIAL_DISCOVERY_RECOMMENDED_EXT event
  XrEventDataBuffer event = {XR_TYPE_EVENT_DATA_BUFFER};
  XrResult result = xrPollEvent(instance, &event);
  if (result == XR_SUCCESS) {
      switch (event.type) {
          case XR_TYPE_EVENT_DATA_SPATIAL_DISCOVERY_RECOMMENDED_EXT: {
              const XrEventDataSpatialDiscoveryRecommendedEXT& eventdata =
                  *reinterpret_cast<XrEventDataSpatialDiscoveryRecommendedEXT*>(&event);
              // Discover spatial entities for the context that we received the
"discovery
              // recommended" event for.
              discoverSpatialEntities(eventdata.spatialContext, time);
              break;
          }
      }
  }

  // ...
  // Finish frame loop
  // ...
}
```

Discover entities with specific UUIDs

```
XrFutureEXT future = XR_NULL_FUTURE_EXT;

// Load up the uuids that the app has stored on its own i.e. the uuids it is interested
in.
std::vector<XrUuid> uuidsStoredByApp = loadPersistedUuids();

// We want to look for entities that have the following components.
std::vector<XrSpatialComponentTypeEXT> snapshotComponents = {
  XR_SPATIAL_COMPONENT_TYPE_PERSISTENCE_EXT,
};

auto discoverSpatialEntities = [&](XrSpatialContextEXT spatialContext, XrTime time) {
  XrSpatialDiscoveryPersistenceUuidFilterEXT
persistenceFilter{XR_TYPE_SPATIAL_DISCOVERY_PERSISTENCE_UUID_FILTER_EXT};
  persistenceFilter.persistedUuidCount = uuidsStoredByApp.size();
  persistenceFilter.persistedUuids = uuidsStoredByApp.data();
```

```
    XrSpatialDiscoverySnapshotCreateInfoEXT
snapshotCreateInfo{XR_TYPE_SPATIAL_DISCOVERY_SNAPSHOT_CREATE_INFO_EXT};
    snapshotCreateInfo.componentTypeCount = snapshotComponents.size();
    snapshotCreateInfo.componentTypes = snapshotComponents.data();
    snapshotCreateInfo.next = &persistenceFilter;

    CHK_XR(xrCreateSpatialDiscoverySnapshotAsyncEXT(spatialContext, &snapshotCreateInfo,
&future));

    waitUntilReady(future);

    XrCreateSpatialDiscoverySnapshotCompletionInfoEXT
completionInfo{XR_TYPE_CREATE_SPATIAL_DISCOVERY_SNAPSHOT_COMPLETION_INFO_EXT};
    completionInfo.baseSpace = localSpace;
    completionInfo.time = time;
    completionInfo.future = future;

    XrCreateSpatialDiscoverySnapshotCompletionEXT
completion{XR_TYPE_CREATE_SPATIAL_DISCOVERY_SNAPSHOT_COMPLETION_EXT};
    CHK_XR(xrCreateSpatialDiscoverySnapshotCompleteEXT(spatialContext, &completionInfo,
&completion));
    if (completion.futureResult == XR_SUCCESS) {

        // Query for the semantic label component data
        XrSpatialComponentDataQueryConditionEXT
queryCond{XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_CONDITION_EXT};
        queryCond.componentTypeCount = snapshotComponents.size();
        queryCond.componentTypes = snapshotComponents.data();

        XrSpatialComponentDataQueryResultEXT
queryResult{XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_RESULT_EXT};
        CHK_XR(xrQuerySpatialComponentDataEXT(completion.snapshot, &queryCond,
&queryResult));

        std::vector<XrSpatialEntityIdEXT> entityIds(queryResult.entityIdCountOutput);
        std::vector<XrSpatialEntityTrackingStateEXT>
entityStates(queryResult.entityIdCountOutput);
        queryResult.entityIdCapacityInput = entityIds.size();
        queryResult.entityIds = entityIds.data();
        queryResult.entityStateCapacityInput = entityStates.size();
        queryResult.entityStates = entityStates.data();

        std::vector<XrSpatialPersistenceDataEXT>
persistenceData(queryResult.entityIdCountOutput);
        XrSpatialComponentPersistenceListEXT
persistenceDataList{XR_TYPE_SPATIAL_COMPONENT_PERSISTENCE_LIST_EXT};
        persistenceDataList.persistDataCount = persistenceData.size();
```

```
    persistenceDataList.persistData = persistenceData.data();
    queryResult.next = &persistenceDataList;

    CHK_XR(xrQuerySpatialComponentDataEXT(completion.snapshot, &queryCond,
&queryResult));

    for (int32_t i = 0; i < queryResult.entityIdCountOutput; ++i) {
      if (persistenceData[i].persistState == XR_SPATIAL_PERSISTENCE_STATE_LOADED_EXT) {
        // persistenceData[i].persistUuid, requested by the app, is present in the
persistence scope
        // and its corresponding entity ID and state are entityIds[i] & entityStates[i]
respectively.
      } else if (persistenceData[i].persistState ==
XR_SPATIAL_PERSISTENCE_STATE_NOT_FOUND_EXT) {
        // persistenceData[i].persistUuid, requested by the app, is NOT present in the
persistence scope
        // and its corresponding entity ID (entityIds[i]) would be
XR_NULL_SPATIAL_ENTITY_ID_EXT
        // and tracking state (entityStates[i]) would be
XR_SPATIAL_ENTITY_TRACKING_STATE_STOPPED_EXT.
      }
    }

    CHK_XR(xrDestroySpatialSnapshotEXT(completion.snapshot));
  }
};

while (1) {
  // ...
  // For every frame in frame loop
  // ...

  XrFrameState frameState;  // previously returned from xrWaitFrame
  const XrTime time = frameState.predictedDisplayTime;

  // Poll for the XR_TYPE_EVENT_DATA_SPATIAL_DISCOVERY_RECOMMENDED_EXT event
  XrEventDataBuffer event = {XR_TYPE_EVENT_DATA_BUFFER};
  XrResult result = xrPollEvent(instance, &event);
  if (result == XR_SUCCESS) {
      switch (event.type) {
          case XR_TYPE_EVENT_DATA_SPATIAL_DISCOVERY_RECOMMENDED_EXT: {
              const XrEventDataSpatialDiscoveryRecommendedEXT& eventdata =
                  *reinterpret_cast<XrEventDataSpatialDiscoveryRecommendedEXT*>(&event);
              // Discover spatial entities for the context that we received the
"discovery
              // recommended" event for.
              discoverSpatialEntities(eventdata.spatialContext, time);
              break;
```

```
            }
        }
    }

    // ...
    // Finish frame loop
    // ...
}
```

12.38.10. Issues

12.38.11. Version History

- Revision 1, 2024-08-29 (Nihav Jain, Google)
  - Initial extension description

12.39. XR_EXT_spatial_persistence_operations

Name String

XR_EXT_spatial_persistence_operations

Extension Type

Instance extension

Registered Extension Number

782

Revision

1

Ratification Status

Ratified

Extension and Version Dependencies

XR_EXT_spatial_persistence
and
XR_EXT_future

Contributors

Nihav Jain, Google
Jared Finder, Google
Natalie Fleury, Meta
Yuichi Taguchi, Meta
Ron Bessems, Meta

Yin Li, Microsoft

Jimmy Alamparambil, ByteDance

Zhipeng Liu, ByteDance

Jun Yan, ByteDance

12.39.1. Overview

While `XR_EXT_spatial_persistence` allows applications to discover persisted entities, this extension allows applications to persist and unpersist spatial entities.

12.39.2. Persist spatial entities

The xrPersistSpatialEntityAsyncEXT function is defined as:

```
// Provided by XR_EXT_spatial_persistence_operations
XrResult xrPersistSpatialEntityAsyncEXT(
    XrSpatialPersistenceContextEXT              persistenceContext,
    const XrSpatialEntityPersistInfoEXT*        persistInfo,
    XrFutureEXT*                                future);
```

<div>

Parameter Descriptions

- `persistenceContext` is an XrSpatialPersistenceContextEXT previously created using xrCreateSpatialPersistenceContextAsyncEXT.

- `persistInfo` is the XrSpatialEntityPersistInfoEXT used to specify the persist operation parameters.

- `future` is a pointer to an `XrFutureEXT`.

</div>

An application **can** persist a spatial entity using the xrPersistSpatialEntityAsyncEXT function.

The runtime **must** return `XR_ERROR_SPATIAL_ENTITY_ID_INVALID_EXT` if XrSpatialEntityPersistInfoEXT::`spatialEntityId` does not belong to XrSpatialEntityPersistInfoEXT::`spatialContext`.

The runtime **must** return `XR_ERROR_PERMISSION_INSUFFICIENT` if the XrSpatialPersistenceScopeEXT that `persistenceContext` was configured with is a read-only scope and does not allow applications to modify the storage represented by it. An example of this would be if `persistenceContext` was created with `XR_SPATIAL_PERSISTENCE_SCOPE_SYSTEM_MANAGED_EXT` and the application uses xrPersistSpatialEntityAsyncEXT with that `persistenceContext`.

The runtime **must** return `XR_ERROR_SPATIAL_PERSISTENCE_SCOPE_INCOMPATIBLE_EXT` if the XrSpatialPersistenceScopeEXT that `persistenceContext` was configured does allow the application to

persist entities of its choice in the storage but XrSpatialEntityPersistInfoEXT::spatialEntityId is not covered in the configured scope. An example of this would be if the persistence context scope is set to XR_SPATIAL_PERSISTENCE_SCOPE_LOCAL_ANCHORS_EXT, and XrSpatialEntityPersistInfoEXT::spatialEntityId does not represent an anchor.

The runtime **must** not return an error if XrSpatialEntityPersistInfoEXT::spatialContext was not configured with persistenceContext using XrSpatialContextPersistenceConfigEXT. Using xrPersistSpatialEntityAsyncEXT does not require that persistenceContext be connected with the spatial context.

This function starts an asynchronous operation and creates a corresponding XrFutureEXT, usable with xrPollFutureEXT and related functions. The return value of this function only indicates whether the parameters were acceptable to schedule the asynchronous operation. The corresponding completion function is xrPersistSpatialEntityCompleteEXT, usable when a future from this function is in the READY state, with outputs populated by that function in the completion structure XrPersistSpatialEntityCompletionEXT.

If the XrSpatialEntityTrackingStateEXT of XrSpatialEntityPersistInfoEXT::spatialEntityId is not XR_SPATIAL_ENTITY_TRACKING_STATE_TRACKING_EXT when xrPersistSpatialEntityAsyncEXT is called, the runtime **must** not return an error from this function or set XrPersistSpatialEntityCompletionEXT ::futureResult to an error code to indicate this. The runtime **may** either set future to the READY state immediately and set XrPersistSpatialEntityCompletionEXT::persistResult to XR_SPATIAL_PERSISTENCE_CONTEXT_RESULT_ENTITY_NOT_TRACKING_EXT to indicate the lack of tracking state, or wait for the entity to get into tracking state as part of the async operation and set XrPersistSpatialEntityCompletionEXT::persistResult to XR_SPATIAL_PERSISTENCE_CONTEXT_RESULT_ENTITY_NOT_TRACKING_EXT if the entity does not get into tracking state until a runtime determined timeout. A common usage pattern of applications is to create a spatial anchor using xrCreateSpatialAnchorEXT and then immediately request to persist the newly created spatial anchor using xrPersistSpatialEntityAsyncEXT. XR_EXT_spatial_anchor states that the tracking state of an anchor **may** not be XR_SPATIAL_ENTITY_TRACKING_STATE_TRACKING_EXT immediately upon its creation. For such cases, the runtime **should** wait for the anchor to get into tracking state as part of the persist async operation instead of immediately setting future to the READY state and fail the operation with XR_SPATIAL_PERSISTENCE_CONTEXT_RESULT_ENTITY_NOT_TRACKING_EXT only if the anchor does not get into tracking state within a runtime determined timeout.

If the spatial entity represented by XrSpatialEntityPersistInfoEXT::spatialEntityId has already been persisted in the scope associated with persistenceContext, the runtime **must** not treat that as an error but instead complete the async operation successfully and provide the appropriate persist UUID to the application.

Valid Usage (Implicit)

- The `XR_EXT_spatial_persistence_operations` extension **must** be enabled prior to calling xrPersistSpatialEntityAsyncEXT

- `persistenceContext` **must** be a valid XrSpatialPersistenceContextEXT handle

- `persistInfo` **must** be a pointer to a valid XrSpatialEntityPersistInfoEXT structure

- `future` **must** be a pointer to an XrFutureEXT 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`

- `XR_ERROR_SPATIAL_PERSISTENCE_SCOPE_INCOMPATIBLE_EXT`

- `XR_ERROR_SPATIAL_ENTITY_ID_INVALID_EXT`

- `XR_ERROR_PERMISSION_INSUFFICIENT`

The XrSpatialEntityPersistInfoEXT structure is defined as:

```
// Provided by XR_EXT_spatial_persistence_operations
typedef struct XrSpatialEntityPersistInfoEXT {
    XrStructureType        type;
    const void*            next;
    XrSpatialContextEXT    spatialContext;
    XrSpatialEntityIdEXT   spatialEntityId;
} XrSpatialEntityPersistInfoEXT;
```

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.

- `spatialContext` is an XrSpatialContextEXT to which `spatialEntityId` belongs.

- `spatialEntityId` is an XrSpatialEntityIdEXT of the spatial entity that is to be persisted.

The XrSpatialEntityPersistInfoEXT structure describes the information to persist a spatial entity represented by `spatialEntityId` in an XrSpatialPersistenceContextEXT.

Valid Usage (Implicit)

- The `XR_EXT_spatial_persistence_operations` extension **must** be enabled prior to using XrSpatialEntityPersistInfoEXT

- `type` **must** be `XR_TYPE_SPATIAL_ENTITY_PERSIST_INFO_EXT`

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

- `spatialContext` **must** be a valid XrSpatialContextEXT handle

The xrPersistSpatialEntityCompleteEXT function is defined as:

```
// Provided by XR_EXT_spatial_persistence_operations
XrResult xrPersistSpatialEntityCompleteEXT(
    XrSpatialPersistenceContextEXT              persistenceContext,
    XrFutureEXT                                 future,
    XrPersistSpatialEntityCompletionEXT*        completion);
```

<div style="background:#eee">

Parameter Descriptions

- `persistenceContext` is the XrSpatialPersistenceContextEXT previously passed to xrPersistSpatialEntityAsyncEXT::`persistenceContext`.

- `future` is the XrFutureEXT received from xrPersistSpatialEntityAsyncEXT.

- `completion` is a pointer to an XrPersistSpatialEntityCompletionEXT.

</div>

xrPersistSpatialEntityCompleteEXT completes the asynchronous operation started by xrPersistSpatialEntityAsyncEXT. The runtime **must** return `XR_ERROR_FUTURE_PENDING_EXT` if `future` is not in READY state. The runtime **must** return `XR_ERROR_FUTURE_INVALID_EXT` if `future` has already been completed or cancelled.

This is the completion function corresponding to the operation started by xrPersistSpatialEntityAsyncEXT. Do not call until the `future` is READY.

If XrPersistSpatialEntityCompletionEXT::`persistUuid` is a UUID that has already been provided to the application either via a previous successful completion of xrPersistSpatialEntityAsyncEXT or by discovering existing persisted entities, then the XrSpatialEntityPersistInfoEXT::`spatialEntityId` **must** represent the same entity as the one the UUID was originally provided for.

<div style="background:#eee">

Valid Usage (Implicit)

- The `XR_EXT_spatial_persistence_operations` extension **must** be enabled prior to calling xrPersistSpatialEntityCompleteEXT

- `persistenceContext` **must** be a valid XrSpatialPersistenceContextEXT handle

- `completion` **must** be a pointer to an XrPersistSpatialEntityCompletionEXT structure

</div>

The XrPersistSpatialEntityCompletionEXT structure is defined as:

```
// Provided by XR_EXT_spatial_persistence_operations
typedef struct XrPersistSpatialEntityCompletionEXT {
    XrStructureType                      type;
    void*                                next;
    XrResult                             futureResult;
    XrSpatialPersistenceContextResultEXT persistResult;
    XrUuid                               persistUuid;
} XrPersistSpatialEntityCompletionEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.
- `next` is `NULL` or a pointer to the next structure in a structure chain.
- `futureResult` is the XrResult of the spatial persistence context creation operation.
- `persistResult` is the XrSpatialPersistenceContextResultEXT of the spatial entity persist operation.
- `persistUuid` is an XrUuid that the application **can** use to identify the persisted spatial entity across sessions.

Future Return Codes

`futureResult` values:

Success

- `XR_SUCCESS`
- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_RUNTIME_FAILURE`
- `XR_ERROR_INSTANCE_LOST`
- `XR_ERROR_SESSION_LOST`
- `XR_ERROR_OUT_OF_MEMORY`
- `XR_ERROR_LIMIT_REACHED`

If `futureResult` and `persistResult` are both success codes, `persistUuid` **must** be valid and the application **can** use it to identify the persisted spatial entity across sessions.

The runtime **must** set `persistResult` to `XR_SPATIAL_PERSISTENCE_CONTEXT_RESULT_ENTITY_NOT_TRACKING_EXT` if it lost tracking of the entity represented by XrSpatialEntityPersistInfoEXT::`spatialEntityId` before it could be successfully persisted.

The runtime **must** set `persistResult` only if `futureResult` is a success code.

12.39.3. Unpersist spatial entities

The xrUnpersistSpatialEntityAsyncEXT function is defined as:

```
// Provided by XR_EXT_spatial_persistence_operations
XrResult xrUnpersistSpatialEntityAsyncEXT(
    XrSpatialPersistenceContextEXT              persistenceContext,
    const XrSpatialEntityUnpersistInfoEXT*      unpersistInfo,
    XrFutureEXT*                                future);
```

An application **can** unpersist a spatial entity using the xrUnpersistSpatialEntityAsyncEXT function.

The runtime **must** return `XR_ERROR_PERMISSION_INSUFFICIENT` if the XrSpatialPersistenceScopeEXT that `persistenceContext` was configured with is a read-only scope and does not allow applications to modify the storage represented by it. An example of this would be if `persistenceContext` was created with `XR_SPATIAL_PERSISTENCE_SCOPE_SYSTEM_MANAGED_EXT` and application uses xrUnpersistSpatialEntityAsyncEXT with that `persistenceContext`.

This function starts an asynchronous operation and creates a corresponding `XrFutureEXT`, usable with xrPollFutureEXT and related functions. The return value of this function only indicates whether the parameters were acceptable to schedule the asynchronous operation. The corresponding completion

function is xrUnpersistSpatialEntityCompleteEXT, usable when a future from this function is in the READY state, with outputs populated by that function in the completion structure XrUnpersistSpatialEntityCompletionEXT.

Valid Usage (Implicit)

- The `XR_EXT_spatial_persistence_operations` extension **must** be enabled prior to calling xrUnpersistSpatialEntityAsyncEXT
- `persistenceContext` **must** be a valid XrSpatialPersistenceContextEXT handle
- `unpersistInfo` **must** be a pointer to a valid XrSpatialEntityUnpersistInfoEXT structure
- `future` **must** be a pointer to an XrFutureEXT 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`
- `XR_ERROR_PERMISSION_INSUFFICIENT`

The XrSpatialEntityUnpersistInfoEXT structure is defined as:

```
// Provided by XR_EXT_spatial_persistence_operations
typedef struct XrSpatialEntityUnpersistInfoEXT {
    XrStructureType     type;
    const void*         next;
    XrUuid              persistUuid;
} XrSpatialEntityUnpersistInfoEXT;
```

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.

- `persistUuid` is an XrUuid prepresting the spatial entity that is to be unpersisted.

The XrSpatialEntityUnpersistInfoEXT structure describes the information to unpersist a spatial entity previously persisted using xrPersistSpatialEntityAsyncEXT.

Valid Usage (Implicit)

- The `XR_EXT_spatial_persistence_operations` extension **must** be enabled prior to using XrSpatialEntityUnpersistInfoEXT

- `type` **must** be `XR_TYPE_SPATIAL_ENTITY_UNPERSIST_INFO_EXT`

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

The xrUnpersistSpatialEntityCompleteEXT function is defined as:

```
// Provided by XR_EXT_spatial_persistence_operations
XrResult xrUnpersistSpatialEntityCompleteEXT(
    XrSpatialPersistenceContextEXT              persistenceContext,
    XrFutureEXT                                 future,
    XrUnpersistSpatialEntityCompletionEXT*      completion);
```

- `persistenceContext` is the XrSpatialPersistenceContextEXT previously passed to xrUnpersistSpatialEntityAsyncEXT::`persistenceContext`.

- `future` is the `XrFutureEXT` received from xrUnpersistSpatialEntityAsyncEXT.

- `completion` is a pointer to an XrUnpersistSpatialEntityCompletionEXT.

xrUnpersistSpatialEntityCompleteEXT completes the asynchronous operation started by xrUnpersistSpatialEntityAsyncEXT. The runtime **must** return `XR_ERROR_FUTURE_PENDING_EXT` if `future` is not in READY state. The runtime **must** return `XR_ERROR_FUTURE_INVALID_EXT` if `future` has already been completed or cancelled.

This is the completion function corresponding to xrUnpersistSpatialEntityAsyncEXT. It completes the asynchronous operation and returns the results. Do not call until the `future` is READY.

Valid Usage (Implicit)

- The `XR_EXT_spatial_persistence_operations` extension **must** be enabled prior to calling xrUnpersistSpatialEntityCompleteEXT

- `persistenceContext` **must** be a valid XrSpatialPersistenceContextEXT handle

- `completion` **must** be a pointer to an XrUnpersistSpatialEntityCompletionEXT 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_FUTURE_PENDING_EXT`

- `XR_ERROR_FUTURE_INVALID_EXT`

The XrUnpersistSpatialEntityCompletionEXT structure is defined as:

```
// Provided by XR_EXT_spatial_persistence_operations
typedef struct XrUnpersistSpatialEntityCompletionEXT {
    XrStructureType                         type;
    void*                                   next;
    XrResult                                futureResult;
    XrSpatialPersistenceContextResultEXT    unpersistResult;
} XrUnpersistSpatialEntityCompletionEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `futureResult` is the XrResult of the spatial entity unpersist operation.

- `unpersistResult` is the XrSpatialPersistenceContextResultEXT of the spatial entity unpersist operation.

Future Return Codes

`futureResult` values:

Success

- `XR_SUCCESS`

- `XR_SESSION_LOSS_PENDING`

Failure

- `XR_ERROR_RUNTIME_FAILURE`

- `XR_ERROR_INSTANCE_LOST`

- `XR_ERROR_SESSION_LOST`

- `XR_ERROR_OUT_OF_MEMORY`

- `XR_ERROR_LIMIT_REACHED`

The runtime **must** set `unpersistResult` only if `futureResult` is a success code.

If XrSpatialEntityUnpersistInfoEXT::`persistUuid` is not found in the storage represented by xrUnpersistSpatialEntityCompleteEXT::`persistenceContext`, then the runtime **must** set `unpersistResult`

to `XR_SPATIAL_PERSISTENCE_CONTEXT_RESULT_PERSIST_UUID_NOT_FOUND_EXT`.

<div style="border:1px solid #ccc; padding:1em;">

Valid Usage (Implicit)

- The `XR_EXT_spatial_persistence_operations` extension **must** be enabled prior to using XrUnpersistSpatialEntityCompletionEXT

- `type` **must** be `XR_TYPE_UNPERSIST_SPATIAL_ENTITY_COMPLETION_EXT`

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

- `futureResult` **must** be a valid XrResult value

- `unpersistResult` **must** be a valid XrSpatialPersistenceContextResultEXT value

</div>

12.39.4. Anchor Persistence Local Scope

If the runtime supports persistence for spatial anchors, and stores them on the same device, for the same user and application that originally created it, it **must** indicate this by enumerating `XR_SPATIAL_PERSISTENCE_SCOPE_LOCAL_ANCHORS_EXT` in xrEnumerateSpatialPersistenceScopesEXT.

If a runtime enumerates `XR_SPATIAL_PERSISTENCE_SCOPE_LOCAL_ANCHORS_EXT` in xrEnumerateSpatialPersistenceScopesEXT, the runtime **must** also enumerate `XR_SPATIAL_CAPABILITY_ANCHOR_EXT` in xrEnumerateSpatialCapabilitiesEXT and `XR_SPATIAL_COMPONENT_TYPE_PERSISTENCE_EXT` in xrEnumerateSpatialCapabilityComponentTypesEXT for `XR_SPATIAL_CAPABILITY_ANCHOR_EXT`.

12.39.5. New Commands

- xrPersistSpatialEntityAsyncEXT

- xrPersistSpatialEntityCompleteEXT

- xrUnpersistSpatialEntityAsyncEXT

- xrUnpersistSpatialEntityCompleteEXT

12.39.6. New Structures

- XrPersistSpatialEntityCompletionEXT

- XrSpatialEntityPersistInfoEXT

- XrSpatialEntityUnpersistInfoEXT

- XrUnpersistSpatialEntityCompletionEXT

12.39.7. New Enum Constants

- `XR_EXT_SPATIAL_PERSISTENCE_OPERATIONS_EXTENSION_NAME`

- XR_EXT_spatial_persistence_operations_SPEC_VERSION
- Extending XrResult:
  - XR_ERROR_SPATIAL_PERSISTENCE_SCOPE_INCOMPATIBLE_EXT
- Extending XrSpatialPersistenceContextResultEXT:
  - XR_SPATIAL_PERSISTENCE_CONTEXT_RESULT_ENTITY_NOT_TRACKING_EXT
  - XR_SPATIAL_PERSISTENCE_CONTEXT_RESULT_PERSIST_UUID_NOT_FOUND_EXT
- Extending XrSpatialPersistenceScopeEXT:
  - XR_SPATIAL_PERSISTENCE_SCOPE_LOCAL_ANCHORS_EXT
- Extending XrStructureType:
  - XR_TYPE_PERSIST_SPATIAL_ENTITY_COMPLETION_EXT
  - XR_TYPE_SPATIAL_ENTITY_PERSIST_INFO_EXT
  - XR_TYPE_SPATIAL_ENTITY_UNPERSIST_INFO_EXT
  - XR_TYPE_UNPERSIST_SPATIAL_ENTITY_COMPLETION_EXT

12.39.8. Issues

12.39.9. Version History

- Revision 1, 2024-08-29 (Nihav Jain, Google)
  - Initial extension description

12.40. XR_EXT_spatial_plane_tracking

Name String

XR_EXT_spatial_plane_tracking

Extension Type

Instance extension

Registered Extension Number

742

Revision

1

Ratification Status

Ratified

Extension and Version Dependencies

> XR_EXT_spatial_entity

Contributors

> Nihav Jain, Google
> Natalie Fleury, Meta
> Yuichi Taguchi, Meta
> Ron Bessems, Meta
> Yin Li, Microsoft
> Jimmy Alamparambil, ByteDance
> Zhipeng Liu, ByteDance
> Jun Yan, ByteDance

12.40.1. Overview

This extension builds on XR_EXT_spatial_entity and defines the plane tracking spatial capability for the spatial entity framework.

12.40.2. Runtime Support

If the runtime supports plane tracking, it **must** indicate this by enumerating XR_SPATIAL_CAPABILITY_PLANE_TRACKING_EXT in xrEnumerateSpatialCapabilitiesEXT.

12.40.3. Configuration

The XrSpatialCapabilityConfigurationPlaneTrackingEXT structure is defined as:

```
// Provided by XR_EXT_spatial_plane_tracking
typedef struct XrSpatialCapabilityConfigurationPlaneTrackingEXT {
    XrStructureType                    type;
    const void*                        next;
    XrSpatialCapabilityEXT             capability;
    uint32_t                           enabledComponentCount;
    const XrSpatialComponentTypeEXT*   enabledComponents;
} XrSpatialCapabilityConfigurationPlaneTrackingEXT;
```

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `capability` is an XrSpatialCapabilityEXT and **must** be `XR_SPATIAL_CAPABILITY_PLANE_TRACKING_EXT`.

- `enabledComponentCount` is a `uint32_t` describing the count of elements in the `enabledComponents` array.

- `enabledComponents` is a pointer to an array of XrSpatialComponentTypeEXT.

Applications **can** enable the `XR_SPATIAL_CAPABILITY_PLANE_TRACKING_EXT` spatial capability by including a pointer to an XrSpatialCapabilityConfigurationPlaneTrackingEXT structure in XrSpatialContextCreateInfoEXT::`capabilityConfigs`.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` if `capability` is not `XR_SPATIAL_CAPABILITY_PLANE_TRACKING_EXT`.

Valid Usage (Implicit)

- The `XR_EXT_spatial_plane_tracking` extension **must** be enabled prior to using XrSpatialCapabilityConfigurationPlaneTrackingEXT

- `type` **must** be `XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_PLANE_TRACKING_EXT`

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

- `capability` **must** be a valid XrSpatialCapabilityEXT value

- `enabledComponents` **must** be a pointer to an array of `enabledComponentCount` valid XrSpatialComponentTypeEXT values

- The `enabledComponentCount` parameter **must** be greater than `0`

12.40.4. Guaranteed Components

A runtime that supports `XR_SPATIAL_CAPABILITY_PLANE_TRACKING_EXT` **must** provide the following spatial components as guaranteed components of all entities discovered by this capability and **must** enumerate them in xrEnumerateSpatialCapabilityComponentTypesEXT:

- `XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT`

- `XR_SPATIAL_COMPONENT_TYPE_PLANE_ALIGNMENT_EXT`

Bounded 2D Component

The bounded 2D component provides the center and extents of the plane represented by the entity it is on. See Bounded 2D for more details.

Plane Alignment Component

Component data

```
// Provided by XR_EXT_spatial_plane_tracking
typedef enum XrSpatialPlaneAlignmentEXT {
    XR_SPATIAL_PLANE_ALIGNMENT_HORIZONTAL_UPWARD_EXT = 0,
    XR_SPATIAL_PLANE_ALIGNMENT_HORIZONTAL_DOWNWARD_EXT = 1,
    XR_SPATIAL_PLANE_ALIGNMENT_VERTICAL_EXT = 2,
    XR_SPATIAL_PLANE_ALIGNMENT_ARBITRARY_EXT = 3,
    XR_SPATIAL_PLANE_ALIGNMENT_MAX_ENUM_EXT = 0x7FFFFFFF
} XrSpatialPlaneAlignmentEXT;
```

The XrSpatialPlaneAlignmentEXT enumeration describes the alignment of the plane associated with the spatial entity with an `XR_SPATIAL_COMPONENT_TYPE_PLANE_ALIGNMENT_EXT` component.

The enumeration values have the following meanings:

Enum	Description
XR_SPATIAL_PLANE_ALIGNMENT_HORIZONTAL_UPWARD_EXT	The entity is horizontal and faces upward (e.g. floor).
XR_SPATIAL_PLANE_ALIGNMENT_HORIZONTAL_DOWNWARD_EXT	The entity is horizontal and faces downward (e.g. ceiling).
XR_SPATIAL_PLANE_ALIGNMENT_VERTICAL_EXT	The entity is vertical (e.g. wall).
XR_SPATIAL_PLANE_ALIGNMENT_ARBITRARY_EXT	The entity has an arbitrary, non-vertical and non-horizontal orientation.

Component list structure to query data

The XrSpatialComponentPlaneAlignmentListEXT structure is defined as:

```
// Provided by XR_EXT_spatial_plane_tracking
typedef struct XrSpatialComponentPlaneAlignmentListEXT {
    XrStructureType              type;
    void*                        next;
    uint32_t                     planeAlignmentCount;
    XrSpatialPlaneAlignmentEXT*  planeAlignments;
} XrSpatialComponentPlaneAlignmentListEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `planeAlignmentCount` is a `uint32_t` describing the count of elements in the `planeAlignments` array.

- `planeAlignments` is an array of XrSpatialPlaneAlignmentEXT.

To query the plane alignment component of the spatial entities in an XrSpatialSnapshotEXT, include `XR_SPATIAL_COMPONENT_TYPE_PLANE_ALIGNMENT_EXT` in XrSpatialComponentDataQueryConditionEXT ::`componentTypes` and add XrSpatialComponentPlaneAlignmentListEXT to the XrSpatialComponentDataQueryResultEXT::`next` chain.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` from xrQuerySpatialComponentDataEXT if XrSpatialComponentPlaneAlignmentListEXT is in the XrSpatialComponentDataQueryResultEXT::`next` chain but `XR_SPATIAL_COMPONENT_TYPE_PLANE_ALIGNMENT_EXT` is not included in XrSpatialComponentDataQueryConditionEXT::`componentTypes`.

The runtime **must** return `XR_ERROR_SIZE_INSUFFICIENT` from xrQuerySpatialComponentDataEXT if `planeAlignmentCount` is less than XrSpatialComponentDataQueryResultEXT::`entityIdCountOutput`.

Valid Usage (Implicit)

- The `XR_EXT_spatial_plane_tracking` extension **must** be enabled prior to using XrSpatialComponentPlaneAlignmentListEXT

- `type` **must** be `XR_TYPE_SPATIAL_COMPONENT_PLANE_ALIGNMENT_LIST_EXT`

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

- `planeAlignments` **must** be a pointer to an array of `planeAlignmentCount` XrSpatialPlaneAlignmentEXT values

- The `planeAlignmentCount` parameter **must** be greater than `0`

Configuration

If `XR_SPATIAL_COMPONENT_TYPE_PLANE_ALIGNMENT_EXT` is enumerated in XrSpatialCapabilityComponentTypesEXT::`componentTypes` for some capability, an application **can** enable it by including the enumerant in the XrSpatialCapabilityConfigurationBaseHeaderEXT ::`enabledComponents` list of the XrSpatialCapabilityConfigurationBaseHeaderEXT derived structure of the capability that supports this component.

This component does not require any special configuration to be included in the XrSpatialCapabilityConfigurationBaseHeaderEXT::`next` chain.

12.40.5. Optional Components

A runtime that supports `XR_SPATIAL_CAPABILITY_PLANE_TRACKING_EXT` **may** support other spatial components in addition to the ones listed in the Guaranteed Components section. An application uses xrEnumerateSpatialCapabilityComponentTypesEXT to get the full list of components that a runtime supports, then configures the ones it is interested in when creating the spatial context.

Mesh 2D Component

Component data

`XR_SPATIAL_COMPONENT_TYPE_MESH_2D_EXT` uses the XrSpatialMeshDataEXT structure for its data.

Component list structure to query data

The XrSpatialComponentMesh2DListEXT structure is defined as:

```
// Provided by XR_EXT_spatial_plane_tracking
typedef struct XrSpatialComponentMesh2DListEXT {
    XrStructureType         type;
    void*                   next;
    uint32_t                meshCount;
    XrSpatialMeshDataEXT*    meshes;
} XrSpatialComponentMesh2DListEXT;
```

Member Descriptions

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `meshCount` is a `uint32_t` describing the count of elements in the `meshes` array.

- `meshes` is an array of XrSpatialMeshDataEXT.

To query the mesh 2D component of the spatial entities in an XrSpatialSnapshotEXT, include `XR_SPATIAL_COMPONENT_TYPE_MESH_2D_EXT` in XrSpatialComponentDataQueryConditionEXT::`componentTypes` and add XrSpatialComponentMesh2DListEXT to the XrSpatialComponentDataQueryResultEXT::`next` chain.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` from xrQuerySpatialComponentDataEXT if XrSpatialComponentMesh2DListEXT is in the XrSpatialComponentDataQueryResultEXT::`next` chain but `XR_SPATIAL_COMPONENT_TYPE_MESH_2D_EXT` is not included in XrSpatialComponentDataQueryConditionEXT::`componentTypes`.

The runtime **must** return `XR_ERROR_SIZE_INSUFFICIENT` from xrQuerySpatialComponentDataEXT if `meshCount` is less than XrSpatialComponentDataQueryResultEXT::`entityIdCountOutput`.

For the XrSpatialMeshDataEXT populated by the runtime in the `meshes` array, the XrSpatialBufferEXT::`bufferType` for XrSpatialMeshDataEXT::`vertexBuffer` **must** be `XR_SPATIAL_BUFFER_TYPE_VECTOR2F_EXT` and XrSpatialBufferEXT::`bufferType` for XrSpatialMeshDataEXT::`indexBuffer` **must** be `XR_SPATIAL_BUFFER_TYPE_UINT16_EXT`.

Valid Usage (Implicit)

- The `XR_EXT_spatial_plane_tracking` extension **must** be enabled prior to using XrSpatialComponentMesh2DListEXT

- `type` **must** be `XR_TYPE_SPATIAL_COMPONENT_MESH_2D_LIST_EXT`

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

- `meshes` **must** be a pointer to an array of `meshCount` XrSpatialMeshDataEXT structures

- The `meshCount` parameter **must** be greater than `0`

Configuration

If `XR_SPATIAL_COMPONENT_TYPE_MESH_2D_EXT` is enumerated in XrSpatialCapabilityComponentTypesEXT::`componentTypes` for some capability, an application **can** enable it by including the enumerant in the XrSpatialCapabilityConfigurationBaseHeaderEXT::`enabledComponents` list of the XrSpatialCapabilityConfigurationBaseHeaderEXT derived structure of the capability that supports this component.

This component does not require any special configuration to be included in the XrSpatialCapabilityConfigurationBaseHeaderEXT::`next` chain.

Polygon 2D Component

Component Data

The XrSpatialPolygon2DDataEXT structure is defined as:

```
// Provided by XR_EXT_spatial_plane_tracking
typedef struct XrSpatialPolygon2DDataEXT {
    XrPosef             origin;
    XrSpatialBufferEXT  vertexBuffer;
} XrSpatialPolygon2DDataEXT;
```

Member Descriptions

- `origin` is an XrPosef defining the origin of the polygon. All vertices of the polygon are relative to this origin in the X-Y plane.

- `vertexBuffer` is an XrSpatialBufferEXT that provides the ID for a buffer of type `XR_SPATIAL_BUFFER_TYPE_VECTOR2F_EXT` and represents the vertex buffer of the entity this component is on. The vertices **must** be returned in counter-clockwise order. The polygon represented by these vertices **must** not be self-intersecting and **may** be concave.

XrSpatialBufferEXT::`bufferType` for `vertexBuffer` **must** be `XR_SPATIAL_BUFFER_TYPE_VECTOR2F_EXT`.

Valid Usage (Implicit)

- The `XR_EXT_spatial_plane_tracking` extension **must** be enabled prior to using XrSpatialPolygon2DDataEXT

- `vertexBuffer` **must** be a valid XrSpatialBufferEXT structure

Component list structure to query data

The XrSpatialComponentPolygon2DListEXT structure is defined as:

```
// Provided by XR_EXT_spatial_plane_tracking
typedef struct XrSpatialComponentPolygon2DListEXT {
    XrStructureType            type;
    void*                      next;
    uint32_t                   polygonCount;
    XrSpatialPolygon2DDataEXT* polygons;
} XrSpatialComponentPolygon2DListEXT;
```

To query the polygon 2D component of the spatial entities in an XrSpatialSnapshotEXT, include `XR_SPATIAL_COMPONENT_TYPE_POLYGON_2D_EXT` in XrSpatialComponentDataQueryConditionEXT::`componentTypes` and add XrSpatialComponentPolygon2DListEXT to the XrSpatialComponentDataQueryResultEXT::`next` chain.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` from xrQuerySpatialComponentDataEXT if XrSpatialComponentPolygon2DListEXT is in the XrSpatialComponentDataQueryResultEXT::`next` chain but `XR_SPATIAL_COMPONENT_TYPE_POLYGON_2D_EXT` is not included in XrSpatialComponentDataQueryConditionEXT::`componentTypes`.

The runtime **must** return `XR_ERROR_SIZE_INSUFFICIENT` from xrQuerySpatialComponentDataEXT if `polygonCount` is less than XrSpatialComponentDataQueryResultEXT::`entityIdCountOutput`.

Configuration

If `XR_SPATIAL_COMPONENT_TYPE_POLYGON_2D_EXT` is enumerated in XrSpatialCapabilityComponentTypesEXT::`componentTypes` for some capability, an application **can** enable it by including the enumerant in the XrSpatialCapabilityConfigurationBaseHeaderEXT ::`enabledComponents` list of the XrSpatialCapabilityConfigurationBaseHeaderEXT derived structure of the capability that supports this component.

This component does not require any special configuration to be included in the XrSpatialCapabilityConfigurationBaseHeaderEXT::`next` chain.

Plane Semantic Label

Component Data

```
// Provided by XR_EXT_spatial_plane_tracking
typedef enum XrSpatialPlaneSemanticLabelEXT {
    XR_SPATIAL_PLANE_SEMANTIC_LABEL_UNCATEGORIZED_EXT = 1,
    XR_SPATIAL_PLANE_SEMANTIC_LABEL_FLOOR_EXT = 2,
    XR_SPATIAL_PLANE_SEMANTIC_LABEL_WALL_EXT = 3,
    XR_SPATIAL_PLANE_SEMANTIC_LABEL_CEILING_EXT = 4,
    XR_SPATIAL_PLANE_SEMANTIC_LABEL_TABLE_EXT = 5,
    XR_SPATIAL_PLANE_SEMANTIC_LABEL_MAX_ENUM_EXT = 0x7FFFFFFF
} XrSpatialPlaneSemanticLabelEXT;
```

The XrSpatialPlaneSemanticLabelEXT enumeration describes a set of semantic labels for planes.

Enum	Description
XR_SPATIAL_PLANE_SEMANTIC_LABEL_UNCATEGORIZED_EXT	The runtime was unable to classify this entity.
XR_SPATIAL_PLANE_SEMANTIC_LABEL_FLOOR_EXT	The entity is a floor.
XR_SPATIAL_PLANE_SEMANTIC_LABEL_WALL_EXT	The entity is a wall.
XR_SPATIAL_PLANE_SEMANTIC_LABEL_CEILING_EXT	The entity is a ceiling.
XR_SPATIAL_PLANE_SEMANTIC_LABEL_TABLE_EXT	The entity is a table.

Component List Structure to Query Data

The XrSpatialComponentPlaneSemanticLabelListEXT structure is defined as:

```
// Provided by XR_EXT_spatial_plane_tracking
typedef struct XrSpatialComponentPlaneSemanticLabelListEXT {
    XrStructureType                    type;
    void*                              next;
    uint32_t                           semanticLabelCount;
    XrSpatialPlaneSemanticLabelEXT*    semanticLabels;
} XrSpatialComponentPlaneSemanticLabelListEXT;
```

To query the plane semantic label component of the spatial entities in an XrSpatialSnapshotEXT, include `XR_SPATIAL_COMPONENT_TYPE_PLANE_SEMANTIC_LABEL_EXT` in XrSpatialComponentDataQueryConditionEXT::`componentTypes` and add XrSpatialComponentPlaneSemanticLabelListEXT to the XrSpatialComponentDataQueryResultEXT::`next` chain.

The runtime **must** return `XR_ERROR_VALIDATION_FAILURE` from xrQuerySpatialComponentDataEXT if XrSpatialComponentPlaneSemanticLabelListEXT is in the next chain of XrSpatialComponentDataQueryResultEXT::`next` but `XR_SPATIAL_COMPONENT_TYPE_PLANE_SEMANTIC_LABEL_EXT` is not included in XrSpatialComponentDataQueryConditionEXT::`componentTypes`.

The runtime **must** return `XR_ERROR_SIZE_INSUFFICIENT` from xrQuerySpatialComponentDataEXT if `semanticLabelCount` is less than XrSpatialComponentDataQueryResultEXT::`entityIdCountOutput`.

Configuration

If `XR_SPATIAL_COMPONENT_TYPE_PLANE_SEMANTIC_LABEL_EXT` is enumerated in XrSpatialCapabilityComponentTypesEXT::`componentTypes` for some capability, an application **can** enable it by including the enumerant in the XrSpatialCapabilityConfigurationBaseHeaderEXT::`enabledComponents` list of the XrSpatialCapabilityConfigurationBaseHeaderEXT derived structure of the capability that supports this component.

This component does not require any special configuration to be included in the

XrSpatialCapabilityConfigurationBaseHeaderEXT::next chain.

12.40.6. Example Code

Configure Plane Tracking Capability

The following example code demonstrates how to configure plane tracking capability when creating a spatial context.

```
// Check if plane tracking capability is supported
uint32_t capabilityCount;
CHK_XR(xrEnumerateSpatialCapabilitiesEXT(instance, systemId, 0, &capabilityCount,
nullptr));
std::vector<XrSpatialCapabilityEXT> capabilities(capabilityCount);
CHK_XR(xrEnumerateSpatialCapabilitiesEXT(instance, systemId, capabilityCount,
&capabilityCount, capabilities.data()));

if (std::find(capabilities.begin(), capabilities.end(),
XR_SPATIAL_CAPABILITY_PLANE_TRACKING_EXT) == capabilities.end()) {
  return;
}

// Enumerate supported components for plane tracking capability
XrSpatialCapabilityComponentTypesEXT
planeComponents{XR_TYPE_SPATIAL_CAPABILITY_COMPONENT_TYPES_EXT};
CHK_XR(xrEnumerateSpatialCapabilityComponentTypesEXT(instance, systemId,
XR_SPATIAL_CAPABILITY_PLANE_TRACKING_EXT, &planeComponents));
std::vector<XrSpatialComponentTypeEXT>
planeCapabilityComponents(planeComponents.componentTypeCountOutput);
planeComponents.componentTypeCapacityInput = planeCapabilityComponents.size();
planeComponents.componentTypes = planeCapabilityComponents.data();
CHK_XR(xrEnumerateSpatialCapabilityComponentTypesEXT(instance, systemId,
XR_SPATIAL_CAPABILITY_PLANE_TRACKING_EXT, &planeComponents));

// Check if polygon 2D and plane semantic labels optional components are supported
const auto supportsComponent = [&planeCapabilityComponents](XrSpatialComponentTypeEXT
component) {
  return std::find(planeCapabilityComponents.begin(), planeCapabilityComponents.end(),
component) != planeCapabilityComponents.end();
};

const bool supportsPolygon2DComponent =
supportsComponent(XR_SPATIAL_COMPONENT_TYPE_POLYGON_2D_EXT);
const bool supportsSemanticLabelComponent =
supportsComponent(XR_SPATIAL_COMPONENT_TYPE_PLANE_SEMANTIC_LABEL_EXT);

// Create a spatial context
```

```cpp
XrSpatialContextEXT spatialContext{};

// Enable the 2 guaranteed components of the plane tracking capability
std::vector<XrSpatialComponentTypeEXT> enabledComponents = {
  XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT,
  XR_SPATIAL_COMPONENT_TYPE_PLANE_ALIGNMENT_EXT,
};
// Optionally enable polygon2D if it is supported
if (supportsPolygon2DComponent) {
  enabledComponents.push_back(XR_SPATIAL_COMPONENT_TYPE_POLYGON_2D_EXT);
}
// Optionally enable semantic labels if it is supported
if (supportsSemanticLabelComponent) {
  enabledComponents.push_back(XR_SPATIAL_COMPONENT_TYPE_PLANE_SEMANTIC_LABEL_EXT);
}

XrSpatialCapabilityConfigurationPlaneTrackingEXT
planeConfig{XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_PLANE_TRACKING_EXT};
planeConfig.capability = XR_SPATIAL_CAPABILITY_PLANE_TRACKING_EXT;
planeConfig.enabledComponentCount = enabledComponents.size();
planeConfig.enabledComponents = enabledComponents.data();

std::array<XrSpatialCapabilityConfigurationBaseHeaderEXT*, 1> capabilityConfigs = {
  reinterpret_cast<XrSpatialCapabilityConfigurationBaseHeaderEXT*>(&planeConfig),
};

XrSpatialContextCreateInfoEXT
spatialContextCreateInfo{XR_TYPE_SPATIAL_CONTEXT_CREATE_INFO_EXT};
spatialContextCreateInfo.capabilityConfigCount = capabilityConfigs.size();
spatialContextCreateInfo.capabilityConfigs = capabilityConfigs.data();
XrFutureEXT createContextFuture;
CHK_XR(xrCreateSpatialContextAsyncEXT(session, &spatialContextCreateInfo,
&createContextFuture));

waitUntilReady(createContextFuture);

XrCreateSpatialContextCompletionEXT
completion{XR_TYPE_CREATE_SPATIAL_CONTEXT_COMPLETION_EXT};
CHK_XR(xrCreateSpatialContextCompleteEXT(session, createContextFuture, &completion));
if (completion.futureResult != XR_SUCCESS) {
  return;
}

spatialContext = completion.spatialContext;

// ...
// Discover entities with the spatial context
// ...
```

```
CHK_XR(xrDestroySpatialContextEXT(spatialContext));
```

Discover Spatial Entities & Query Component Data

The following example code demonstrates how to discover spatial entities for a context configured
with XR_SPATIAL_CAPABILITY_PLANE_TRACKING_EXT and query its component data.

```
XrFutureEXT future = XR_NULL_FUTURE_EXT;

// We want to look for entities that have the following components.
std::vector<XrSpatialComponentTypeEXT> snapshotComponents = {
  XR_SPATIAL_COMPONENT_TYPE_BOUNDED_2D_EXT,
  XR_SPATIAL_COMPONENT_TYPE_PLANE_ALIGNMENT_EXT,
};
if (supportsPolygon2DComponent) {
  snapshotComponents.push_back(XR_SPATIAL_COMPONENT_TYPE_POLYGON_2D_EXT);
}
if (supportsSemanticLabelComponent) {
  snapshotComponents.push_back(XR_SPATIAL_COMPONENT_TYPE_PLANE_SEMANTIC_LABEL_EXT);
}

auto discoverSpatialEntities = [&](XrSpatialContextEXT spatialContext, XrTime time) {
  XrSpatialDiscoverySnapshotCreateInfoEXT
snapshotCreateInfo{XR_TYPE_SPATIAL_DISCOVERY_SNAPSHOT_CREATE_INFO_EXT};
  snapshotCreateInfo.componentTypeCount = snapshotComponents.size();
  snapshotCreateInfo.componentTypes = snapshotComponents.data();
  CHK_XR(xrCreateSpatialDiscoverySnapshotAsyncEXT(spatialContext, &snapshotCreateInfo,
&future));

  waitUntilReady(future);

  XrCreateSpatialDiscoverySnapshotCompletionInfoEXT
completionInfo{XR_TYPE_CREATE_SPATIAL_DISCOVERY_SNAPSHOT_COMPLETION_INFO_EXT};
  completionInfo.baseSpace = localSpace;
  completionInfo.time = time;
  completionInfo.future = future;

  XrCreateSpatialDiscoverySnapshotCompletionEXT
completion{XR_TYPE_CREATE_SPATIAL_DISCOVERY_SNAPSHOT_COMPLETION_EXT};
  CHK_XR(xrCreateSpatialDiscoverySnapshotCompleteEXT(spatialContext, &completionInfo,
&completion));
  if (completion.futureResult == XR_SUCCESS) {

    // Query for the semantic label component data
    XrSpatialComponentDataQueryConditionEXT
```

```
queryCond{XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_CONDITION_EXT};
    queryCond.componentTypeCount = snapshotComponents.size();
    queryCond.componentTypes = snapshotComponents.data();

    XrSpatialComponentDataQueryResultEXT
queryResult{XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_RESULT_EXT};
    CHK_XR(xrQuerySpatialComponentDataEXT(completion.snapshot, &queryCond,
&queryResult));

    std::vector<XrSpatialEntityIdEXT> entityIds(queryResult.entityIdCountOutput);
    std::vector<XrSpatialEntityTrackingStateEXT>
entityStates(queryResult.entityIdCountOutput);
    queryResult.entityIdCapacityInput = entityIds.size();
    queryResult.entityIds = entityIds.data();
    queryResult.entityStateCapacityInput = entityStates.size();
    queryResult.entityStates = entityStates.data();

    std::vector<XrSpatialBounded2DDataEXT> bounded2D(queryResult.entityIdCountOutput);
    XrSpatialComponentBounded2DListEXT
bounded2DList{XR_TYPE_SPATIAL_COMPONENT_BOUNDED_2D_LIST_EXT};
    bounded2DList.boundCount = bounded2D.size();
    bounded2DList.bounds = bounded2D.data();
    queryResult.next = &bounded2DList;

    std::vector<XrSpatialPolygon2DDataEXT> polygons;
    XrSpatialComponentPolygon2DListEXT
polygonList{XR_TYPE_SPATIAL_COMPONENT_POLYGON_2D_LIST_EXT};
    if (supportsPolygon2DComponent) {
      polygons.resize(queryResult.entityIdCountOutput);
      polygonList.polygonCount = polygons.size();
      polygonList.polygons = polygons.data();
      polygonList.next = queryResult.next;
      queryResult.next = &polygonList;
    }

    std::vector<XrSpatialPlaneSemanticLabelEXT> semanticLabels;
    XrSpatialComponentPlaneSemanticLabelListEXT
semanticLabelsList{XR_TYPE_SPATIAL_COMPONENT_PLANE_SEMANTIC_LABEL_LIST_EXT};
    if (supportsSemanticLabelComponent) {
      semanticLabels.resize(queryResult.entityIdCountOutput);
      semanticLabelsList.semanticLabelCount = semanticLabels.size();
      semanticLabelsList.semanticLabels = semanticLabels.data();
      semanticLabelsList.next = queryResult.next;
      queryResult.next = &semanticLabelsList;
    }

    CHK_XR(xrQuerySpatialComponentDataEXT(completion.snapshot, &queryCond,
&queryResult));
```

```
    for (int32_t i = 0; i < queryResult.entityIdCountOutput; ++i) {
      if (entityStates[i] != XR_SPATIAL_ENTITY_TRACKING_STATE_TRACKING_EXT) {
        continue;
      }

      // 2D bounds for entity entityIds[i] is bounded2D[i].extents centered on
bounded2D[i].center.

      if (supportsPolygon2DComponent) {
        // 2D polygon for entity entityIds[i] is the buffer represented by
polygons[i].bufferId.
        // Application uses flink:xrGetSpatialBufferVector2fEXT to get the buffer data.
      }

      if (supportsSemanticLabelComponent) {
        // semantic label for entity entityIds[i] is semanticLabels[i].
      }
    }

    CHK_XR(xrDestroySpatialSnapshotEXT(completion.snapshot));
  }
};

while (1) {
  // ...
  // For every frame in frame loop
  // ...

  XrFrameState frameState;  // previously returned from xrWaitFrame
  const XrTime time = frameState.predictedDisplayTime;

  // Poll for the XR_TYPE_EVENT_DATA_SPATIAL_DISCOVERY_RECOMMENDED_EXT event
  XrEventDataBuffer event = {XR_TYPE_EVENT_DATA_BUFFER};
  XrResult result = xrPollEvent(instance, &event);
  if (result == XR_SUCCESS) {
    switch (event.type) {
      case XR_TYPE_EVENT_DATA_SPATIAL_DISCOVERY_RECOMMENDED_EXT: {
        const XrEventDataSpatialDiscoveryRecommendedEXT& eventdata =
          *reinterpret_cast<XrEventDataSpatialDiscoveryRecommendedEXT*>(&event);
        // Discover spatial entities for the context that we received the
"discovery
        // recommended" event for.
        discoverSpatialEntities(eventdata.spatialContext, time);
        break;
      }
    }
  }
```

```
  // ...
  // Finish frame loop
  // ...
}
```

12.40.7. New Structures

- XrSpatialCapabilityConfigurationPlaneTrackingEXT

- XrSpatialPolygon2DDataEXT

- Extending XrSpatialComponentDataQueryResultEXT:

  - XrSpatialComponentMesh2DListEXT

  - XrSpatialComponentPlaneAlignmentListEXT

  - XrSpatialComponentPlaneSemanticLabelListEXT

  - XrSpatialComponentPolygon2DListEXT

12.40.8. New Enums

- XrSpatialPlaneAlignmentEXT

- XrSpatialPlaneSemanticLabelEXT

12.40.9. New Enum Constants

- `XR_EXT_SPATIAL_PLANE_TRACKING_EXTENSION_NAME`

- `XR_EXT_spatial_plane_tracking_SPEC_VERSION`

- Extending XrSpatialCapabilityEXT:

  - `XR_SPATIAL_CAPABILITY_PLANE_TRACKING_EXT`

- Extending XrSpatialComponentTypeEXT:

  - `XR_SPATIAL_COMPONENT_TYPE_MESH_2D_EXT`

  - `XR_SPATIAL_COMPONENT_TYPE_PLANE_ALIGNMENT_EXT`

  - `XR_SPATIAL_COMPONENT_TYPE_PLANE_SEMANTIC_LABEL_EXT`

  - `XR_SPATIAL_COMPONENT_TYPE_POLYGON_2D_EXT`

- Extending XrStructureType:

  - `XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_PLANE_TRACKING_EXT`

  - `XR_TYPE_SPATIAL_COMPONENT_MESH_2D_LIST_EXT`

  - `XR_TYPE_SPATIAL_COMPONENT_PLANE_ALIGNMENT_LIST_EXT`

  - `XR_TYPE_SPATIAL_COMPONENT_PLANE_SEMANTIC_LABEL_LIST_EXT`

- XR_TYPE_SPATIAL_COMPONENT_POLYGON_2D_LIST_EXT

12.40.10. Issues

12.40.11. Version History

- Revision 1, 2024-07-02 (Nihav Jain, Google)

  - Initial extension description

12.41. XR_EXT_view_configuration_depth_range

Name String

XR_EXT_view_configuration_depth_range

Extension Type

Instance extension

Registered Extension Number

47

Revision

1

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0

Last Modified Date

2019-08-16

IP Status

No known IP claims.

Contributors

Blake Taylor, Magic Leap
Gilles Cadet, Magic Leap
Michael Liebenow, Magic Leap
Supreet Suresh, Magic Leap
Alex Turner, Microsoft
Bryce Hutchings, Microsoft
Yin Li, Microsoft

Overview

For XR systems there may exist a per view recommended min/max depth range at which content should be rendered into the virtual world. The depth range may be driven by several factors, including user comfort, or fundamental capabilities of the system.

Displaying rendered content outside the recommended min/max depth range would violate the system requirements for a properly integrated application, and can result in a poor user experience due to observed visual artifacts, visual discomfort, or fatigue. The near/far depth values will fall in the range of (0..+infinity] where max(`recommendedNearZ`, `minNearZ`) < min(`recommendedFarZ`, `maxFarZ`). Infinity is defined matching the standard library definition such that std::isinf will return true for a returned infinite value.

In order to provide the application with the appropriate depth range at which to render content for each XrViewConfigurationView, this extension provides additional view configuration information, as defined by XrViewConfigurationDepthRangeEXT, to inform the application of the min/max recommended and absolute distances at which content should be rendered for that view.

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_VIEW_CONFIGURATION_DEPTH_RANGE_EXT`

New Enums

New Structures

The XrViewConfigurationDepthRangeEXT structure is defined as:

```
// Provided by XR_EXT_view_configuration_depth_range
typedef struct XrViewConfigurationDepthRangeEXT {
    XrStructureType    type;
    void*              next;
    float              recommendedNearZ;
    float              minNearZ;
    float              recommendedFarZ;
    float              maxFarZ;
} XrViewConfigurationDepthRangeEXT;
```

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.
- `recommendedNearZ` is the recommended minimum positive distance in meters that content should be rendered for the view to achieve the best user experience.
- `minNearZ` is the absolute minimum positive distance in meters that content should be rendered for the view.
- `recommendedFarZ` is the recommended maximum positive distance in meters that content should be rendered for the view to achieve the best user experience.
- `maxFarZ` is the absolute maximum positive distance in meters that content should be rendered for the view.

When enumerating the view configurations with [xrEnumerateViewConfigurationViews](#), the application **can** provide a pointer to an [XrViewConfigurationDepthRangeEXT](#) in the `next` chain of [XrViewConfigurationView](#).

Valid Usage (Implicit)

- The `XR_EXT_view_configuration_depth_range` extension **must** be enabled prior to using [XrViewConfigurationDepthRangeEXT](#)
- `type` **must** be `XR_TYPE_VIEW_CONFIGURATION_DEPTH_RANGE_EXT`
- `next` **must** be `NULL` or a valid pointer to the [next structure in a structure chain](#)

New Functions

Issues

Version History

- Revision 1, 2019-10-01 (Blake Taylor)
  - Initial proposal.

Chapter 13. List of Deprecated Extensions

- `XR_KHR_locate_spaces` (promoted to core)

- `XR_KHR_maintenance1` (promoted to core)

13.1. XR_KHR_locate_spaces

Name String

XR_KHR_locate_spaces

Extension Type

Instance extension

Registered Extension Number

472

Revision

1

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0

Deprecation State

- *Promoted* to OpenXR 1.1

Last Modified Date

2024-01-19

IP Status

No known IP claims.

Contributors

Yin Li, Microsoft
Bryce Hutchings, Microsoft
Andreas Loeve Selvik, Meta Platforms
John Kearney, Meta Platforms
Robert Blenkinsopp, Ultraleap
Rylie Pavlik, Collabora
Ron Bessems, Magic Leap
Jakob Bornecrantz, NVIDIA

13.1.1. Overview

This extension introduces the xrLocateSpacesKHR function, which enables applications to locate an array of spaces in a single function call. Runtimes **may** provide performance benefits for applications that use many spaces.

Compared to the xrLocateSpace function, the new xrLocateSpacesKHR function also provides extensible input parameters for future extensions to extend using additional chained structures.

13.1.2. Locate spaces

Applications **can** use xrLocateSpacesKHR function to locate an array of spaces.

The xrLocateSpacesKHR function is defined as:

```
// Provided by XR_KHR_locate_spaces
XrResult xrLocateSpacesKHR(
    XrSession                               session,
    const XrSpacesLocateInfo*               locateInfo,
    XrSpaceLocations*                       spaceLocations);
```

Parameter Descriptions

- `session` is an XrSession handle previously created with xrCreateSession.
- `locateInfo` is a pointer to an XrSpacesLocateInfoKHR that provides the input information to locate spaces.
- `spaceLocations` is a pointer to an XrSpaceLocationsKHR for the runtime to return the locations of the specified spaces in the base space.

xrLocateSpacesKHR provides the physical location of one or more spaces in a base space at a specified time, if currently known by the runtime.

The XrSpacesLocateInfoKHR::`time`, the XrSpacesLocateInfoKHR::`baseSpace`, and each space in XrSpacesLocateInfoKHR::`spaces`, in the `locateInfo` parameter, all follow the same specifics as the corresponding inputs to the xrLocateSpace function.

Valid Usage (Implicit)

- The XR_KHR_locate_spaces extension **must** be enabled prior to calling xrLocateSpacesKHR
- `session` **must** be a valid XrSession handle
- `locateInfo` **must** be a pointer to a valid XrSpacesLocateInfo structure
- `spaceLocations` **must** be a pointer to an XrSpaceLocations 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_TIME_INVALID

The XrSpacesLocateInfoKHR structure is defined as:

```
// Provided by XR_KHR_locate_spaces
// XrSpacesLocateInfoKHR is an alias for XrSpacesLocateInfo
typedef struct XrSpacesLocateInfo {
    XrStructureType    type;
    const void*        next;
    XrSpace            baseSpace;
    XrTime             time;
    uint32_t           spaceCount;
    const XrSpace*     spaces;
} XrSpacesLocateInfo;

typedef XrSpacesLocateInfo XrSpacesLocateInfoKHR;
```

- `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.

- `baseSpace` identifies the underlying space in which to locate `spaces`.

- `time` is the time for which the location is requested.

- `spaceCount` is a `uint32_t` specifying the count of elements in the `spaces` array.

- `spaces` is an array of valid XrSpace handles to be located.

The `time`, the `baseSpace`, and each space in `spaces` all follow the same specifics as the corresponding inputs to the xrLocateSpace function.

The `baseSpace` and all of the XrSpace handles in the `spaces` array **must** be valid and share the same parent XrSession.

If the `time` is invalid, the xrLocateSpacesKHR **must** return `XR_ERROR_TIME_INVALID`.

The `spaceCount` **must** be a positive number, i.e. the array `spaces` **must** not be empty. Otherwise, the runtime **must** return `XR_ERROR_VALIDATION_FAILURE`.

Valid Usage (Implicit)

- The `XR_KHR_locate_spaces` extension **must** be enabled prior to using XrSpacesLocateInfoKHR

- **Note:** XrSpacesLocateInfoKHR is an alias for XrSpacesLocateInfo, so the following items replicate the implicit valid usage for XrSpacesLocateInfo

- `type` **must** be `XR_TYPE_SPACES_LOCATE_INFO`

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

- `baseSpace` **must** be a valid XrSpace handle

- `spaces` **must** be a pointer to an array of `spaceCount` valid XrSpace handles

- The `spaceCount` parameter **must** be greater than `0`

- Both of `baseSpace` and the elements of `spaces` **must** have been created, allocated, or retrieved from the same XrSession

The XrSpaceLocationsKHR structure is defined as:

```
// Provided by XR_KHR_locate_spaces
// XrSpaceLocationsKHR is an alias for XrSpaceLocations
typedef struct XrSpaceLocations {
    XrStructureType         type;
    void*                   next;
    uint32_t                locationCount;
    XrSpaceLocationData*    locations;
} XrSpaceLocations;

typedef XrSpaceLocations XrSpaceLocationsKHR;
```

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 XrSpaceVelocitiesKHR.

- `locationCount` is a `uint32_t` specifying the count of elements in the `locations` array.

- `locations` is an array of XrSpaceLocationsKHR for the runtime to populate with the locations of the specified spaces in the XrSpacesLocateInfoKHR::`baseSpace` at the specified XrSpacesLocateInfoKHR::`time`.

The XrSpaceLocationsKHR structure contains an array of space locations in the member `locations`, to be used as output for xrLocateSpacesKHR. The application **must** allocate this array to be populated with the function output. The `locationCount` value **must** be the same as XrSpacesLocateInfoKHR::`spaceCount`, otherwise, the xrLocateSpacesKHR function **must** return `XR_ERROR_VALIDATION_FAILURE`.

Valid Usage (Implicit)

- The `XR_KHR_locate_spaces` extension **must** be enabled prior to using XrSpaceLocationsKHR

- **Note:** XrSpaceLocationsKHR is an alias for XrSpaceLocations, so the following items replicate the implicit valid usage for XrSpaceLocations

- `type` **must** be `XR_TYPE_SPACE_LOCATIONS`

- `next` **must** be `NULL` or a valid pointer to the next structure in a structure chain. See also: XrSpaceVelocities

- `locations` **must** be a pointer to an array of `locationCount` XrSpaceLocationData structures

- The `locationCount` parameter **must** be greater than `0`

The XrSpaceLocationDataKHR structure is defined as:

```
// Provided by XR_KHR_locate_spaces
// XrSpaceLocationDataKHR is an alias for XrSpaceLocationData
typedef struct XrSpaceLocationData {
    XrSpaceLocationFlags    locationFlags;
    XrPosef                 pose;
} XrSpaceLocationData;

typedef XrSpaceLocationData XrSpaceLocationDataKHR;
```

Member Descriptions

- `locationFlags` is a bitfield, with bit masks defined in XrSpaceLocationFlagBits. It behaves the same as XrSpaceLocation::`locationFlags`.

- `pose` is an XrPosef that behaves the same as XrSpaceLocation::`pose`.

This is a single element of the array in XrSpaceLocationsKHR::`locations`, and is used to return the pose and location flags for a single space with respect to the specified base space from a call to xrLocateSpacesKHR. It does not accept chained structures to allow for easier use in dynamically allocated container datatypes. Chained structures are possible with the XrSpaceLocationsKHR that describes an array of these elements.

Valid Usage (Implicit)

- The `XR_KHR_locate_spaces` extension **must** be enabled prior to using XrSpaceLocationDataKHR

13.1.3. Locate space velocities

Applications **can** request the velocities of spaces by chaining the XrSpaceVelocitiesKHR structure to the next pointer of XrSpaceLocationsKHR when calling xrLocateSpacesKHR.

The XrSpaceVelocitiesKHR structure is defined as:

```
// Provided by XR_KHR_locate_spaces
// XrSpaceVelocitiesKHR is an alias for XrSpaceVelocities
typedef struct XrSpaceVelocities {
    XrStructureType         type;
    void*                   next;
    uint32_t                velocityCount;
    XrSpaceVelocityData*    velocities;
} XrSpaceVelocities;

typedef XrSpaceVelocities XrSpaceVelocitiesKHR;
```

Member Descriptions

- `type` is the XrStructureType of this structure.

- `next` is `NULL` or a pointer to the next structure in a structure chain.

- `velocityCount` is a `uint32_t` specifying the count of elements in the `velocities` array.

- `velocities` is an array of XrSpaceVelocityDataKHR for the runtime to populate with the velocities of the specified spaces in the XrSpacesLocateInfoKHR::`baseSpace` at the specified XrSpacesLocateInfoKHR::`time`.

The `velocities` member contains an array of space velocities in the member `velocities`, to be used as output for xrLocateSpacesKHR. The application **must** allocate this array to be populated with the function output. The `velocityCount` value **must** be the same as XrSpacesLocateInfoKHR::`spaceCount`, otherwise, the xrLocateSpacesKHR function **must** return `XR_ERROR_VALIDATION_FAILURE`.

Valid Usage (Implicit)

- The XR_KHR_locate_spaces extension **must** be enabled prior to using XrSpaceVelocitiesKHR

- **Note:** XrSpaceVelocitiesKHR is an alias for XrSpaceVelocities, so the following items replicate the implicit valid usage for XrSpaceVelocities

- `type` **must** be `XR_TYPE_SPACE_VELOCITIES`

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

- `velocities` **must** be a pointer to an array of `velocityCount` XrSpaceVelocityData structures

- The `velocityCount` parameter **must** be greater than `0`

The XrSpaceVelocityDataKHR structure is defined as:

```
// Provided by XR_KHR_locate_spaces
// XrSpaceVelocityDataKHR is an alias for XrSpaceVelocityData
typedef struct XrSpaceVelocityData {
    XrSpaceVelocityFlags    velocityFlags;
    XrVector3f              linearVelocity;
    XrVector3f              angularVelocity;
} XrSpaceVelocityData;

typedef XrSpaceVelocityData XrSpaceVelocityDataKHR;
```

Member Descriptions

- `velocityFlags` is a bitfield, with bit values defined in XrSpaceVelocityFlagBits. It behaves the same as XrSpaceVelocity::`velocityFlags`.

- `linearVelocity` is an XrVector3f. It behaves the same as XrSpaceVelocity::`linearVelocity`.

- `angularVelocity` is an XrVector3f. It behaves the same as XrSpaceVelocity::`angularVelocity`.

This is a single element of the array in XrSpaceVelocitiesKHR::`velocities`, and is used to return the linear and angular velocity and velocity flags for a single space with respect to the specified base space from a call to xrLocateSpacesKHR. It does not accept chained structures to allow for easier use in dynamically allocated container datatypes.

Valid Usage (Implicit)

- The `XR_KHR_locate_spaces` extension **must** be enabled prior to using XrSpaceVelocityDataKHR

13.1.4. Example code for xrLocateSpacesKHR

The following example code shows how an application retrieves both the location and velocity of one or more spaces in a base space at a given time using the xrLocateSpacesKHR function.

```
XrInstance instance; // previously initialized
XrSession session; // previously initialized
XrSpace baseSpace; // previously initialized
std::vector<XrSpace> spacesToLocate;  // previously initialized

// Prepare output buffers to receive data and get reused in frame loop.
std::vector<XrSpaceLocationDataKHR> locationBuffer(spacesToLocate.size());
std::vector<XrSpaceVelocityDataKHR> velocityBuffer(spacesToLocate.size());

// Get function pointer for xrLocateSpacesKHR.
```

```
PFN_xrLocateSpacesKHR xrLocateSpacesKHR;
CHK_XR(xrGetInstanceProcAddr(instance, "xrLocateSpacesKHR",
                             reinterpret_cast<PFN_xrVoidFunction*>(
                             &xrLocateSpacesKHR)));

// application frame loop
while (1) {
    // Typically the time is the predicted display time returned from xrWaitFrame.
    XrTime displayTime; // previously initialized.

    XrSpacesLocateInfoKHR locateInfo{XR_TYPE_SPACES_LOCATE_INFO_KHR};
    locateInfo.baseSpace = baseSpace;
    locateInfo.time = displayTime;
    locateInfo.spaceCount = (uint32_t)spacesToLocate.size();
    locateInfo.spaces = spacesToLocate.data();

    XrSpaceLocationsKHR locations{XR_TYPE_SPACES_LOCATE_INFO_KHR};
    locations.locationCount = (uint32_t)locationBuffer.size();
    locations.locations = locationBuffer.data();

    XrSpaceVelocitiesKHR velocities{XR_TYPE_SPACE_VELOCITIES_KHR};
    velocities.velocityCount = (uint32_t)velocityBuffer.size();
    velocities.velocities = velocityBuffer.data();

    locations.next = &velocities;
    CHK_XR(xrLocateSpacesKHR(session, &locateInfo, &locations));

    for (uint32_t i = 0; i < spacesToLocate.size(); i++) {
        const auto positionAndOrientationTracked =
          XR_SPACE_LOCATION_POSITION_TRACKED_BIT |
XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT;
        const auto orientationOnlyTracked = XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT;

        if ((locationBuffer[i].locationFlags & positionAndOrientationTracked) ==
positionAndOrientationTracked) {
            // if the location is 6dof tracked
            do_something(locationBuffer[i].pose.position);
            do_something(locationBuffer[i].pose.orientation);

            const auto velocityValidBits =
              XR_SPACE_VELOCITY_LINEAR_VALID_BIT | XR_SPACE_VELOCITY_ANGULAR_VALID_BIT;
            if ((velocityBuffer[i].velocityFlags & velocityValidBits) ==
velocityValidBits) {
                do_something(velocityBuffer[i].linearVelocity);
                do_something(velocityBuffer[i].angularVelocity);
            }
        }
        else if ((locationBuffer[i].locationFlags & orientationOnlyTracked) ==
```

```
orientationOnlyTracked) {
            // if the location is 3dof tracked
            do_something(locationBuffer[i].pose.orientation);

            if ((velocityBuffer[i].velocityFlags & XR_SPACE_VELOCITY_ANGULAR_VALID_BIT)
== XR_SPACE_VELOCITY_ANGULAR_VALID_BIT) {
                do_something(velocityBuffer[i].angularVelocity);
            }
        }
    }
}
```

New Object Types

New Flag Types

New Enum Constants

XrStructureType enumeration is extended with:

- `XR_TYPE_SPACES_LOCATE_INFO_KHR`

- `XR_TYPE_SPACE_LOCATIONS_KHR`

- `XR_TYPE_SPACE_VELOCITIES_KHR`

New Enums

New Structures

- XrSpacesLocateInfoKHR

- XrSpaceLocationsKHR

- XrSpaceLocationDataKHR

- XrSpaceVelocitiesKHR

- XrSpaceVelocityDataKHR

New Functions

- xrLocateSpacesKHR

Issues

Version History

- Revision 1, 2023-04-22 (Yin LI)

    ◦ Initial extension description

13.2. XR_KHR_maintenance1

Name String

XR_KHR_maintenance1

Extension Type

Instance extension

Registered Extension Number

711

Revision

1

Ratification Status

Ratified

Extension and Version Dependencies

OpenXR 1.0

API Interactions

- Interacts with XR_BD_controller_interaction
- Interacts with XR_EXT_hp_mixed_reality_controller
- Interacts with XR_EXT_samsung_odyssey_controller
- Interacts with XR_FB_touch_controller_pro
- Interacts with XR_HTCX_vive_tracker_interaction
- Interacts with XR_HTC_hand_interaction
- Interacts with XR_HTC_vive_cosmos_controller_interaction
- Interacts with XR_HTC_vive_focus3_controller_interaction
- Interacts with XR_HUAWEI_controller_interaction
- Interacts with XR_META_touch_controller_plus
- Interacts with XR_ML_ml2_controller_interaction
- Interacts with XR_MSFT_hand_interaction
- Interacts with XR_OPPO_controller_interaction
- Interacts with XR_VARJO_xr4_controller_interaction
- Interacts with XR_YVR_controller_interaction

Deprecation State

- *Promoted* to OpenXR 1.1

Last Modified Date

  2023-10-25

IP Status

  No known IP claims.

Contributors

  Ron Bessems, Magic Leap

  Karthik Kadappan, Magic Leap

  Rylie Pavlik, Collabora

  Nihav Jain, Google

  Lachlan Ford, Google

  John Kearney, Meta

  Yin Li, Microsoft

  Robert Blenkinsopp, Ultraleap

13.2.1. Overview

XR_KHR_maintenance1 adds a collection of minor features that were intentionally left out or overlooked from the original OpenXR 1.0 release. All are promoted to the OpenXR 1.1 release.

```
// Provided by XR_KHR_maintenance1
// XrColor3fKHR is an alias for XrColor3f
typedef struct XrColor3f {
    float     r;
    float     g;
    float     b;
} XrColor3f;

typedef XrColor3f XrColor3fKHR;
```

```
// Provided by XR_KHR_maintenance1
// XrExtent3DfKHR is an alias for XrExtent3Df
typedef struct XrExtent3Df {
    float     width;
    float     height;
    float     depth;
} XrExtent3Df;

typedef XrExtent3Df XrExtent3DfKHR;
```

```
// Provided by XR_KHR_maintenance1
// XrSpherefKHR is an alias for XrSpheref
typedef struct XrSpheref {
    XrPosef    center;
    float      radius;
} XrSpheref;

typedef XrSpheref XrSpherefKHR;
```

```
// Provided by XR_KHR_maintenance1
// XrBoxfKHR is an alias for XrBoxf
typedef struct XrBoxf {
    XrPosef        center;
    XrExtent3Df    extents;
} XrBoxf;

typedef XrBoxf XrBoxfKHR;
```

```
// Provided by XR_KHR_maintenance1
// XrFrustumfKHR is an alias for XrFrustumf
typedef struct XrFrustumf {
    XrPosef    pose;
    XrFovf     fov;
    float      nearZ;
    float      farZ;
} XrFrustumf;

typedef XrFrustumf XrFrustumfKHR;
```

13.2.2. New Structures

- XrBoxfKHR

- XrColor3fKHR

- XrExtent3DfKHR

- XrFrustumfKHR

- XrSpherefKHR

13.2.3. New Enum Constants

- `XR_KHR_MAINTENANCE1_EXTENSION_NAME`

- `XR_KHR_maintenance1_SPEC_VERSION`

- Extending XrResult:

    ◦ `XR_ERROR_EXTENSION_DEPENDENCY_NOT_ENABLED_KHR`

    ◦ `XR_ERROR_PERMISSION_INSUFFICIENT_KHR`

13.2.4. Version History

- Revision 1, 2023-10-25 (Ron Bessems)

    ◦ Initial extension description

Chapter 14. Core Revisions (Informative)

New minor versions of the OpenXR API are defined periodically by the Khronos OpenXR Working Group. These consist of some amount of additional functionality added to the core API, potentially including both new functionality and functionality promoted from extensions.

14.1. Version 1.1

14.1.1. OpenXR 1.1 Promotions

OpenXR version 1.1 promoted a number of key extensions into the core API:

- XR_KHR_locate_spaces
- XR_KHR_maintenance1
- XR_EXT_hp_mixed_reality_controller
- XR_EXT_local_floor
- XR_EXT_palm_pose
- XR_EXT_samsung_odyssey_controller
- XR_EXT_uuid
- XR_BD_controller_interaction
- XR_FB_touch_controller_pro
- XR_HTC_vive_cosmos_controller_interaction
- XR_HTC_vive_focus3_controller_interaction
- XR_META_touch_controller_plus
- XR_ML_ml2_controller_interaction
- XR_VARJO_quad_views

All differences in behavior between these extensions and the corresponding OpenXR 1.1 functionality are summarized below.

Differences Relative to XR_EXT_local_floor

The definition of this space was made more precise, and it was clarified that the mandatory support of this space does **not** dictate any particular quality of floor level estimation. Applications that can provide a head-relative interaction experience in the absence of a defined stage continue to use LOCAL space, while those that need higher quality assertions about floor level continue to use STAGE space or scene understanding extensions to detect floor level. The (mandatory) presence of this space when

enumerating reference spaces is a convenience for portability rather than an assertion that e.g. floor detection scene understanding has taken place or that the floor is inherently walkable.

Differences Relative to `XR_EXT_palm_pose`

The input identifier `palm_ext` defined in the extension has been renamed to `grip_surface` to more clearly describe its intended use and distinguish it from hand tracking.

Differences Relative to `XR_VARJO_quad_views`

The view configuration type enumerant `XR_VIEW_CONFIGURATION_TYPE_PRIMARY_QUAD_VARJO` was renamed to `XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO_WITH_FOVEATED_INSET`, to clarify that it is not vendor-specific nor the only way four views are possible. In OpenXR 1.1, a runtime **may** support `XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO_WITH_FOVEATED_INSET`, but this is **optional** like the other view configuration types. Use xrEnumerateViewConfigurations to determine if it is provided, rather than using the presence or absence of the extension.

Differences Relative to `XR_FB_touch_controller_pro`

The interaction profile path was changed from */interaction_profiles/facebook/touch_controller_pro* to */interaction_profiles/meta/touch_pro_controller*. Note the updated company name and different word order in the device name level.

The following input/output subpaths were renamed when changing to this new interaction profile path:

- *.../input/stylus_fb/force* → *.../input/stylus/force*
- *.../input/trigger/proximity_fb* → *.../input/trigger/proximity*
- *.../output/haptic_trigger_fb* → *.../output/haptic_trigger*
- *.../output/haptic_thumb_fb* → *.../output/haptic_thumb*
- *.../input/thumb_fb/proximity_fb* → *.../input/thumb_resting_surfaces/proximity*
- *.../input/trigger/curl_fb* → *.../input/trigger_curl/value*
- *.../input/trigger/slide_fb* → *.../input/trigger_slide/value*

The last two changes listed moved from being components on the trigger identifier to being independent identifiers in order to clarify how they relate to actions bound to other trigger components with regards to action priority.

Differences Relative to `XR_META_touch_controller_plus`

The interaction profile path was changed from */interaction_profiles/meta/touch_controller_plus* to */interaction_profiles/meta/touch_plus_controller*. Note the different word order in the device name level.

The following input subpaths were renamed when changing to this new interaction profile path: *

*…/input/trigger/proximity_meta → …/input/trigger/proximity * …/input/thumb_meta/proximity_meta → …/input/thumb_resting_surfaces/proximity * …/input/trigger/curl_meta → …/input/trigger_curl/value * …/input/trigger/slide_meta → …/input/trigger_slide/value*

14.1.2. Additional OpenXR 1.1 Changes

In addition to the promoted extensions described above, OpenXR 1.1 changed the following:

- Substantial clarifications in the input and fundamentals chapters, intended to be non-substantive.
- Added the following legacy interaction profiles to represent specific controllers shipped under the Oculus/Meta Touch name and previously grouped into a single Oculus Touch interaction profile:
  - */interaction_profiles/meta/touch_controller_rift_cv1* - Meta Touch Controller (Rift CV1) Profile
  - */interaction_profiles/meta/touch_controller_quest_1_rift_s* - Meta Touch Controller (Rift S / Quest 1) Profile
  - */interaction_profiles/meta/touch_controller_quest_2* - Meta Touch Controller (Quest 2) Profile

14.1.3. New Macros

- XR_API_VERSION_1_1

14.1.4. New Commands

- xrLocateSpaces

14.1.5. New Structures

- XrBoxf
- XrColor3f
- XrExtent3Df
- XrFrustumf
- XrSpaceLocationData
- XrSpaceLocations
- XrSpaceVelocityData
- XrSpacesLocateInfo
- XrSpheref
- XrUuid
- Extending XrSpaceLocations:
  - XrSpaceVelocities

14.1.6. New Enum Constants

- `XR_UUID_SIZE`

- Extending XrReferenceSpaceType:

  - `XR_REFERENCE_SPACE_TYPE_LOCAL_FLOOR`

- Extending XrResult:

  - `XR_ERROR_EXTENSION_DEPENDENCY_NOT_ENABLED`

  - `XR_ERROR_PERMISSION_INSUFFICIENT`

- Extending XrStructureType:

  - `XR_TYPE_SPACES_LOCATE_INFO`

  - `XR_TYPE_SPACE_LOCATIONS`

  - `XR_TYPE_SPACE_VELOCITIES`

- Extending XrViewConfigurationType:

  - `XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO_WITH_FOVEATED_INSET`

14.2. Loader Runtime and API Layer Negotiation Version 1.0

The OpenXR version 1.0.33 patch release included ratification of the runtime and API layer negotiation API, associated with the identifier `XR_LOADER_VERSION_1_0`, substantially unchanged from the unratified form previously described in the loader design document. This interface is intended for use only between the loader, runtimes, and API layers, and is not typically directly used by an application.

14.2.1. New Macros

- XR_API_LAYER_CREATE_INFO_STRUCT_VERSION

- XR_API_LAYER_INFO_STRUCT_VERSION

- XR_API_LAYER_MAX_SETTINGS_PATH_SIZE

- XR_API_LAYER_NEXT_INFO_STRUCT_VERSION

- XR_CURRENT_LOADER_API_LAYER_VERSION

- XR_CURRENT_LOADER_RUNTIME_VERSION

- XR_LOADER_INFO_STRUCT_VERSION

- XR_RUNTIME_INFO_STRUCT_VERSION

14.2.2. New Commands

- xrCreateApiLayerInstance

- xrNegotiateLoaderApiLayerInterface

- xrNegotiateLoaderRuntimeInterface

14.3. Version 1.0

OpenXR version 1.0 defined the initial core API.

14.3.1. New Macros

- XR_API_VERSION_1_0

- XR_CURRENT_API_VERSION

- XR_DEFINE_HANDLE

- XR_DEFINE_OPAQUE_64

- XR_EXTENSION_ENUM_BASE

- XR_EXTENSION_ENUM_STRIDE

- XR_FAILED

- XR_FREQUENCY_UNSPECIFIED

- XR_INFINITE_DURATION

- XR_MAX_EVENT_DATA_SIZE

- XR_MAY_ALIAS

- XR_MIN_COMPOSITION_LAYERS_SUPPORTED

- XR_MIN_HAPTIC_DURATION

- XR_NO_DURATION

- XR_NULL_HANDLE

- XR_NULL_PATH

- XR_NULL_SYSTEM_ID

- XR_SUCCEEDED

- XR_UNQUALIFIED_SUCCESS

- XR_VERSION_MAJOR

- XR_VERSION_MINOR

- XR_VERSION_PATCH

14.3.2. New Base Types

- XrVersion

14.3.3. New Commands

- xrAcquireSwapchainImage
- xrApplyHapticFeedback
- xrAttachSessionActionSets
- xrBeginFrame
- xrBeginSession
- xrCreateAction
- xrCreateActionSet
- xrCreateActionSpace
- xrCreateInstance
- xrCreateReferenceSpace
- xrCreateSession
- xrCreateSwapchain
- xrDestroyAction
- xrDestroyActionSet
- xrDestroyInstance
- xrDestroySession
- xrDestroySpace
- xrDestroySwapchain
- xrEndFrame
- xrEndSession
- xrEnumerateApiLayerProperties
- xrEnumerateBoundSourcesForAction
- xrEnumerateEnvironmentBlendModes
- xrEnumerateInstanceExtensionProperties
- xrEnumerateReferenceSpaces
- xrEnumerateSwapchainFormats
- xrEnumerateSwapchainImages
- xrEnumerateViewConfigurationViews
- xrEnumerateViewConfigurations
- xrGetActionStateBoolean
- xrGetActionStateFloat

- xrGetActionStatePose

- xrGetActionStateVector2f

- xrGetCurrentInteractionProfile

- xrGetInputSourceLocalizedName

- xrGetInstanceProcAddr

- xrGetInstanceProperties

- xrGetReferenceSpaceBoundsRect

- xrGetSystem

- xrGetSystemProperties

- xrGetViewConfigurationProperties

- xrLocateSpace

- xrLocateViews

- xrPathToString

- xrPollEvent

- xrReleaseSwapchainImage

- xrRequestExitSession

- xrResultToString

- xrStopHapticFeedback

- xrStringToPath

- xrStructureTypeToString

- xrSuggestInteractionProfileBindings

- xrSyncActions

- xrWaitFrame

- xrWaitSwapchainImage

14.3.4. New Structures

- XrBaseInStructure

- XrBaseOutStructure

- XrColor4f

- XrCompositionLayerProjection

- XrCompositionLayerQuad

- XrEventDataBaseHeader

- XrEventDataEventsLost

- XrEventDataInstanceLossPending

- XrEventDataInteractionProfileChanged

- XrEventDataReferenceSpaceChangePending

- XrEventDataSessionStateChanged

- XrExtent2Df

- XrHapticVibration

- XrOffset2Df

- XrRect2Df

- XrVector4f

- Extending XrSpaceLocation:

  - XrSpaceVelocity

14.3.5. New Enums

- XrObjectType

14.3.6. New Headers

- `openxr_platform_defines`

14.3.7. New Enum Constants

- `XR_FALSE`

- `XR_MAX_API_LAYER_DESCRIPTION_SIZE`

- `XR_MAX_API_LAYER_NAME_SIZE`

- `XR_MAX_APPLICATION_NAME_SIZE`

- `XR_MAX_ENGINE_NAME_SIZE`

- `XR_MAX_EXTENSION_NAME_SIZE`

- `XR_MAX_PATH_LENGTH`

- `XR_MAX_RESULT_STRING_SIZE`

- `XR_MAX_RUNTIME_NAME_SIZE`

- `XR_MAX_STRUCTURE_NAME_SIZE`

- `XR_MAX_SYSTEM_NAME_SIZE`

- `XR_TRUE`

Appendix

Code Style Conventions

These are the code style conventions used in this specification to define the API.

> ### Conventions
>
> - Enumerants and defines are all upper case with words separated by an underscore.
>
> - Neither type, function or member names contain underscores.
>
> - Structure members start with a lower case character and each consecutive word starts with a capital.
>
> - A structure that has a pointer to an array includes a structure member named `fooCount` of type `uint32_t` to denote the number of elements in the array of `foo`.
>
> - A structure that has a pointer to an array lists the `fooCount` member first and then the array pointer.
>
> - Unless a negative value has a clearly defined meaning all `fooCount` variables are unsigned.
>
> - Function parameters that are modified are always listed last.

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:

Prefix	Description
`XR_`	Enumerants and defines are prefixed with these characters.
`Xr`	Non-function-pointer types are prefixed with these characters.
`xr`	Functions are prefixed with these characters.
`PFN_xr`	Function pointer types are prefixed with these characters.

Application Binary Interface

This section describes additional definitions and conventions that define the application binary interface.

Structure Types

```
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,
// Provided by XR_VERSION_1_1
    XR_TYPE_SPACES_LOCATE_INFO = 1000471000,
// Provided by XR_VERSION_1_1
    XR_TYPE_SPACE_LOCATIONS = 1000471001,
// Provided by XR_VERSION_1_1
    XR_TYPE_SPACE_VELOCITIES = 1000471002,
// Provided by XR_KHR_composition_layer_cube
    XR_TYPE_COMPOSITION_LAYER_CUBE_KHR = 1000006000,
// Provided by XR_KHR_android_create_instance
    XR_TYPE_INSTANCE_CREATE_INFO_ANDROID_KHR = 1000008000,
// Provided by XR_KHR_composition_layer_depth
    XR_TYPE_COMPOSITION_LAYER_DEPTH_INFO_KHR = 1000010000,
// Provided by XR_KHR_vulkan_swapchain_format_list
    XR_TYPE_VULKAN_SWAPCHAIN_FORMAT_LIST_CREATE_INFO_KHR = 1000014000,
// Provided by XR_EXT_performance_settings
    XR_TYPE_EVENT_DATA_PERF_SETTINGS_EXT = 1000015000,
// Provided by XR_KHR_composition_layer_cylinder
    XR_TYPE_COMPOSITION_LAYER_CYLINDER_KHR = 1000017000,
// Provided by XR_KHR_composition_layer_equirect
    XR_TYPE_COMPOSITION_LAYER_EQUIRECT_KHR = 1000018000,
// Provided by XR_KHR_opengl_enable
    XR_TYPE_GRAPHICS_BINDING_OPENGL_WIN32_KHR = 1000023000,
// Provided by XR_KHR_opengl_enable
    XR_TYPE_GRAPHICS_BINDING_OPENGL_XLIB_KHR = 1000023001,
// Provided by XR_KHR_opengl_enable
    XR_TYPE_GRAPHICS_BINDING_OPENGL_XCB_KHR = 1000023002,
// Provided by XR_KHR_opengl_enable
    XR_TYPE_GRAPHICS_BINDING_OPENGL_WAYLAND_KHR = 1000023003,
// Provided by XR_KHR_opengl_enable
    XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_KHR = 1000023004,
// Provided by XR_KHR_opengl_enable
    XR_TYPE_GRAPHICS_REQUIREMENTS_OPENGL_KHR = 1000023005,
// Provided by XR_KHR_opengl_es_enable
    XR_TYPE_GRAPHICS_BINDING_OPENGL_ES_ANDROID_KHR = 1000024001,
// Provided by XR_KHR_opengl_es_enable
    XR_TYPE_SWAPCHAIN_IMAGE_OPENGL_ES_KHR = 1000024002,
// Provided by XR_KHR_opengl_es_enable
    XR_TYPE_GRAPHICS_REQUIREMENTS_OPENGL_ES_KHR = 1000024003,
// Provided by XR_KHR_vulkan_enable
    XR_TYPE_GRAPHICS_BINDING_VULKAN_KHR = 1000025000,
```

```
    // Provided by XR_KHR_vulkan_enable
      XR_TYPE_SWAPCHAIN_IMAGE_VULKAN_KHR = 1000025001,
    // Provided by XR_KHR_vulkan_enable
      XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN_KHR = 1000025002,
    // Provided by XR_KHR_D3D11_enable
      XR_TYPE_GRAPHICS_BINDING_D3D11_KHR = 1000027000,
    // Provided by XR_KHR_D3D11_enable
      XR_TYPE_SWAPCHAIN_IMAGE_D3D11_KHR = 1000027001,
    // Provided by XR_KHR_D3D11_enable
      XR_TYPE_GRAPHICS_REQUIREMENTS_D3D11_KHR = 1000027002,
    // Provided by XR_KHR_D3D12_enable
      XR_TYPE_GRAPHICS_BINDING_D3D12_KHR = 1000028000,
    // Provided by XR_KHR_D3D12_enable
      XR_TYPE_SWAPCHAIN_IMAGE_D3D12_KHR = 1000028001,
    // Provided by XR_KHR_D3D12_enable
      XR_TYPE_GRAPHICS_REQUIREMENTS_D3D12_KHR = 1000028002,
    // Provided by XR_KHR_metal_enable
      XR_TYPE_GRAPHICS_BINDING_METAL_KHR = 1000029000,
    // Provided by XR_KHR_metal_enable
      XR_TYPE_SWAPCHAIN_IMAGE_METAL_KHR = 1000029001,
    // Provided by XR_KHR_metal_enable
      XR_TYPE_GRAPHICS_REQUIREMENTS_METAL_KHR = 1000029002,
    // Provided by XR_EXT_eye_gaze_interaction
      XR_TYPE_SYSTEM_EYE_GAZE_INTERACTION_PROPERTIES_EXT = 1000030000,
    // Provided by XR_EXT_eye_gaze_interaction
      XR_TYPE_EYE_GAZE_SAMPLE_TIME_EXT = 1000030001,
    // Provided by XR_KHR_visibility_mask
      XR_TYPE_VISIBILITY_MASK_KHR = 1000031000,
    // Provided by XR_KHR_visibility_mask
      XR_TYPE_EVENT_DATA_VISIBILITY_MASK_CHANGED_KHR = 1000031001,
    // Provided by XR_KHR_composition_layer_color_scale_bias
      XR_TYPE_COMPOSITION_LAYER_COLOR_SCALE_BIAS_KHR = 1000034000,
    // Provided by XR_EXT_view_configuration_depth_range
      XR_TYPE_VIEW_CONFIGURATION_DEPTH_RANGE_EXT = 1000046000,
    // Provided by XR_EXT_hand_tracking
      XR_TYPE_SYSTEM_HAND_TRACKING_PROPERTIES_EXT = 1000051000,
    // Provided by XR_EXT_hand_tracking
      XR_TYPE_HAND_TRACKER_CREATE_INFO_EXT = 1000051001,
    // Provided by XR_EXT_hand_tracking
      XR_TYPE_HAND_JOINTS_LOCATE_INFO_EXT = 1000051002,
    // Provided by XR_EXT_hand_tracking
      XR_TYPE_HAND_JOINT_LOCATIONS_EXT = 1000051003,
    // Provided by XR_EXT_hand_tracking
      XR_TYPE_HAND_JOINT_VELOCITIES_EXT = 1000051004,
    // Provided by XR_EXT_dpad_binding
      XR_TYPE_INTERACTION_PROFILE_DPAD_BINDING_EXT = 1000078000,
    // Provided by XR_EXT_hand_joints_motion_range
      XR_TYPE_HAND_JOINTS_MOTION_RANGE_INFO_EXT = 1000080000,
```

```
    // Provided by XR_KHR_loader_init_android
    XR_TYPE_LOADER_INIT_INFO_ANDROID_KHR = 1000089000,
  // Provided by XR_KHR_vulkan_enable2
    XR_TYPE_VULKAN_INSTANCE_CREATE_INFO_KHR = 1000090000,
  // Provided by XR_KHR_vulkan_enable2
    XR_TYPE_VULKAN_DEVICE_CREATE_INFO_KHR = 1000090001,
  // Provided by XR_KHR_vulkan_enable2
    XR_TYPE_VULKAN_GRAPHICS_DEVICE_GET_INFO_KHR = 1000090003,
  // Provided by XR_KHR_composition_layer_equirect2
    XR_TYPE_COMPOSITION_LAYER_EQUIRECT2_KHR = 1000091000,
  // Provided by XR_KHR_binding_modification
    XR_TYPE_BINDING_MODIFICATIONS_KHR = 1000120000,
  // Provided by XR_EXT_hand_tracking_data_source
    XR_TYPE_HAND_TRACKING_DATA_SOURCE_INFO_EXT = 1000428000,
  // Provided by XR_EXT_hand_tracking_data_source
    XR_TYPE_HAND_TRACKING_DATA_SOURCE_STATE_EXT = 1000428001,
  // Provided by XR_EXT_future
    XR_TYPE_FUTURE_CANCEL_INFO_EXT = 1000469000,
  // Provided by XR_EXT_future
    XR_TYPE_FUTURE_POLL_INFO_EXT = 1000469001,
  // Provided by XR_EXT_future
    XR_TYPE_FUTURE_COMPLETION_EXT = 1000469002,
  // Provided by XR_EXT_future
    XR_TYPE_FUTURE_POLL_RESULT_EXT = 1000469003,
  // Provided by XR_EXT_spatial_entity
    XR_TYPE_SPATIAL_CAPABILITY_COMPONENT_TYPES_EXT = 1000740000,
  // Provided by XR_EXT_spatial_entity
    XR_TYPE_SPATIAL_CONTEXT_CREATE_INFO_EXT = 1000740001,
  // Provided by XR_EXT_spatial_entity
    XR_TYPE_CREATE_SPATIAL_CONTEXT_COMPLETION_EXT = 1000740002,
  // Provided by XR_EXT_spatial_entity
    XR_TYPE_SPATIAL_DISCOVERY_SNAPSHOT_CREATE_INFO_EXT = 1000740003,
  // Provided by XR_EXT_spatial_entity
    XR_TYPE_CREATE_SPATIAL_DISCOVERY_SNAPSHOT_COMPLETION_INFO_EXT = 1000740004,
  // Provided by XR_EXT_spatial_entity
    XR_TYPE_CREATE_SPATIAL_DISCOVERY_SNAPSHOT_COMPLETION_EXT = 1000740005,
  // Provided by XR_EXT_spatial_entity
    XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_CONDITION_EXT = 1000740006,
  // Provided by XR_EXT_spatial_entity
    XR_TYPE_SPATIAL_COMPONENT_DATA_QUERY_RESULT_EXT = 1000740007,
  // Provided by XR_EXT_spatial_entity
    XR_TYPE_SPATIAL_BUFFER_GET_INFO_EXT = 1000740008,
  // Provided by XR_EXT_spatial_entity
    XR_TYPE_SPATIAL_COMPONENT_BOUNDED_2D_LIST_EXT = 1000740009,
  // Provided by XR_EXT_spatial_entity
    XR_TYPE_SPATIAL_COMPONENT_BOUNDED_3D_LIST_EXT = 1000740010,
  // Provided by XR_EXT_spatial_entity
    XR_TYPE_SPATIAL_COMPONENT_PARENT_LIST_EXT = 1000740011,
```

```
  // Provided by XR_EXT_spatial_entity
  XR_TYPE_SPATIAL_COMPONENT_MESH_3D_LIST_EXT = 1000740012,
// Provided by XR_EXT_spatial_entity
  XR_TYPE_SPATIAL_ENTITY_FROM_ID_CREATE_INFO_EXT = 1000740013,
// Provided by XR_EXT_spatial_entity
  XR_TYPE_SPATIAL_UPDATE_SNAPSHOT_CREATE_INFO_EXT = 1000740014,
// Provided by XR_EXT_spatial_entity
  XR_TYPE_EVENT_DATA_SPATIAL_DISCOVERY_RECOMMENDED_EXT = 1000740015,
// Provided by XR_EXT_spatial_entity
  XR_TYPE_SPATIAL_FILTER_TRACKING_STATE_EXT = 1000740016,
// Provided by XR_EXT_spatial_plane_tracking
  XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_PLANE_TRACKING_EXT = 1000741000,
// Provided by XR_EXT_spatial_plane_tracking
  XR_TYPE_SPATIAL_COMPONENT_PLANE_ALIGNMENT_LIST_EXT = 1000741001,
// Provided by XR_EXT_spatial_plane_tracking
  XR_TYPE_SPATIAL_COMPONENT_MESH_2D_LIST_EXT = 1000741002,
// Provided by XR_EXT_spatial_plane_tracking
  XR_TYPE_SPATIAL_COMPONENT_POLYGON_2D_LIST_EXT = 1000741003,
// Provided by XR_EXT_spatial_plane_tracking
  XR_TYPE_SPATIAL_COMPONENT_PLANE_SEMANTIC_LABEL_LIST_EXT = 1000741004,
// Provided by XR_EXT_spatial_marker_tracking
  XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_QR_CODE_EXT = 1000743000,
// Provided by XR_EXT_spatial_marker_tracking
  XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_MICRO_QR_CODE_EXT = 1000743001,
// Provided by XR_EXT_spatial_marker_tracking
  XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_ARUCO_MARKER_EXT = 1000743002,
// Provided by XR_EXT_spatial_marker_tracking
  XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_APRIL_TAG_EXT = 1000743003,
// Provided by XR_EXT_spatial_marker_tracking
  XR_TYPE_SPATIAL_MARKER_SIZE_EXT = 1000743004,
// Provided by XR_EXT_spatial_marker_tracking
  XR_TYPE_SPATIAL_MARKER_STATIC_OPTIMIZATION_EXT = 1000743005,
// Provided by XR_EXT_spatial_marker_tracking
  XR_TYPE_SPATIAL_COMPONENT_MARKER_LIST_EXT = 1000743006,
// Provided by XR_EXT_spatial_anchor
  XR_TYPE_SPATIAL_CAPABILITY_CONFIGURATION_ANCHOR_EXT = 1000762000,
// Provided by XR_EXT_spatial_anchor
  XR_TYPE_SPATIAL_COMPONENT_ANCHOR_LIST_EXT = 1000762001,
// Provided by XR_EXT_spatial_anchor
  XR_TYPE_SPATIAL_ANCHOR_CREATE_INFO_EXT = 1000762002,
// Provided by XR_EXT_spatial_persistence
  XR_TYPE_SPATIAL_PERSISTENCE_CONTEXT_CREATE_INFO_EXT = 1000763000,
// Provided by XR_EXT_spatial_persistence
  XR_TYPE_CREATE_SPATIAL_PERSISTENCE_CONTEXT_COMPLETION_EXT = 1000763001,
// Provided by XR_EXT_spatial_persistence
  XR_TYPE_SPATIAL_CONTEXT_PERSISTENCE_CONFIG_EXT = 1000763002,
// Provided by XR_EXT_spatial_persistence
  XR_TYPE_SPATIAL_DISCOVERY_PERSISTENCE_UUID_FILTER_EXT = 1000763003,
```

```
    // Provided by XR_EXT_spatial_persistence
    XR_TYPE_SPATIAL_COMPONENT_PERSISTENCE_LIST_EXT = 1000763004,
  // Provided by XR_EXT_spatial_persistence_operations
    XR_TYPE_SPATIAL_ENTITY_PERSIST_INFO_EXT = 1000781000,
  // Provided by XR_EXT_spatial_persistence_operations
    XR_TYPE_PERSIST_SPATIAL_ENTITY_COMPLETION_EXT = 1000781001,
  // Provided by XR_EXT_spatial_persistence_operations
    XR_TYPE_SPATIAL_ENTITY_UNPERSIST_INFO_EXT = 1000781002,
  // Provided by XR_EXT_spatial_persistence_operations
    XR_TYPE_UNPERSIST_SPATIAL_ENTITY_COMPLETION_EXT = 1000781003,
  // Provided by XR_KHR_vulkan_enable2
    XR_TYPE_GRAPHICS_BINDING_VULKAN2_KHR = XR_TYPE_GRAPHICS_BINDING_VULKAN_KHR,
  // Provided by XR_KHR_vulkan_enable2
    XR_TYPE_SWAPCHAIN_IMAGE_VULKAN2_KHR = XR_TYPE_SWAPCHAIN_IMAGE_VULKAN_KHR,
  // Provided by XR_KHR_vulkan_enable2
    XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN2_KHR = XR_TYPE_GRAPHICS_REQUIREMENTS_VULKAN_KHR,
  // Provided by XR_KHR_locate_spaces
    XR_TYPE_SPACES_LOCATE_INFO_KHR = XR_TYPE_SPACES_LOCATE_INFO,
  // Provided by XR_KHR_locate_spaces
    XR_TYPE_SPACE_LOCATIONS_KHR = XR_TYPE_SPACE_LOCATIONS,
  // Provided by XR_KHR_locate_spaces
    XR_TYPE_SPACE_VELOCITIES_KHR = XR_TYPE_SPACE_VELOCITIES,
    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 10^9 (base 10). Each extension is assigned a block of 1000 enums, starting at the enum base and arranged by the extension's number.

```
// Provided by XR_VERSION_1_0
#define XR_EXTENSION_ENUM_BASE 1000000000
```

```
// Provided by XR_VERSION_1_0
#define XR_EXTENSION_ENUM_STRIDE 1000
```

For example, if extension number 5 wants to use an enum value of 3, the final enum is computed by:

enum = XR_EXTENSION_ENUM_BASE + (extension_number - 1) * XR_EXTENSION_ENUM_STRIDE + enum_value

1000004003 = 1000000000 + 4 * 1000 + 3

The maximum allowed enum value in an extension is 2,147,482,999, which belongs to extension number 1147483.

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 defined in the core specification were originally listed/defined here, but have been moved to be adjacent to their associated `FlagBits` type. See the Index for a list.

General Macro Definitions

This API is defined in C and uses "C" linkage. The `openxr.h` header file is opened with:

```
1 #ifdef __cplusplus
2 extern "C" {
3 #endif
```

and closed with:

```
1 #ifdef __cplusplus
2 }
3 #endif
```

The supplied `openxr.h` header defines a small number of C preprocessor macros that are described below.

Version Number Macros

Three version numbers are defined in `openxr.h`. Each is packed into a 64-bit integer as described in API Version Number Function-like Macros.

```
// Provided by XR_VERSION_1_0
// OpenXR current version number.
#define XR_CURRENT_API_VERSION XR_MAKE_VERSION(1, 1, 50)
```

XR_CURRENT_API_VERSION is the current version of the OpenXR API.

In many cases, XR_API_VERSION_1_0 or XR_API_VERSION_1_1 are preferred for source forward-

compatibility.

```
// Provided by XR_VERSION_1_0
// OpenXR 1.0 version number
#define XR_API_VERSION_1_0 XR_MAKE_VERSION(1, 0,
XR_VERSION_PATCH(XR_CURRENT_API_VERSION))
```

XR_API_VERSION_1_0 is the version of the OpenXR 1.0 API. The "major" and "minor" components are always 1.0, while the "patch" component matches XR_CURRENT_API_VERSION.

```
// Provided by XR_VERSION_1_1
// OpenXR 1.1 version number
#define XR_API_VERSION_1_1 XR_MAKE_VERSION(1, 1,
XR_VERSION_PATCH(XR_CURRENT_API_VERSION))
```

XR_API_VERSION_1_1 is the version of the OpenXR 1.1 API. The "major" and "minor" components are always 1.1, while the "patch" component matches XR_CURRENT_API_VERSION.

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.

```
// Provided by XR_VERSION_1_0
#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.

```
// Provided by XR_VERSION_1_0
#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.

```
// Provided by XR_VERSION_1_0
#define XR_VERSION_PATCH(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

```
// Provided by XR_VERSION_1_0
#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
#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.

```
// Provided by XR_VERSION_1_0
#if !defined(XR_NULL_HANDLE)
#if (XR_PTR_SIZE == 8) && XR_CPP_NULLPTR_SUPPORTED
    #define XR_NULL_HANDLE nullptr
#else
    #define XR_NULL_HANDLE 0
#endif
#endif
```

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.

```
#if !defined(XR_DEFINE_ATOM)
    #define XR_DEFINE_ATOM(object) typedef uint64_t object;
#endif
```

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.

```
// Provided by XR_VERSION_1_0
#if !defined(XR_DEFINE_OPAQUE_64)
    #if (XR_PTR_SIZE == 8)
        #define XR_DEFINE_OPAQUE_64(object) typedef struct object##_T* object;
    #else
        #define XR_DEFINE_OPAQUE_64(object) typedef uint64_t object;
    #endif
#endif
```

Parameter Descriptions

- object is the name of the resulting C type.

XR_DEFINE_OPAQUE_64 defines an opaque 64 bit value, which **may** be implemented as an opaque, distinct pointer type on platforms with 64 bit pointers.

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

Compile Time Symbol	Graphics API Name
XR_USE_GRAPHICS_API_OPENGL	OpenGL
XR_USE_GRAPHICS_API_OPENGL_ES	OpenGL ES
XR_USE_GRAPHICS_API_VULKAN	Vulkan
XR_USE_GRAPHICS_API_D3D11	Direct3D 11
XR_USE_GRAPHICS_API_D3D12	Direct3D 12

Window System Header Control

Compile Time Symbol	Window System Name
XR_USE_PLATFORM_WIN32	Microsoft Windows
XR_USE_PLATFORM_XLIB	X Window System Xlib
XR_USE_PLATFORM_XCB	X Window System XCB
XR_USE_PLATFORM_EGL	EGL (for OpenGL/OpenGL ES usage on any platform with EGL support)
XR_USE_PLATFORM_WAYLAND	Wayland
XR_USE_PLATFORM_ANDROID	Android Native

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:

```
<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.

Term	Description
Application	The XR application which calls the OpenXR API to communicate with an OpenXR runtime.
Deprecated	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.
Handle	An opaque integer or pointer value used to refer to an object. Each object type has a unique handle type.

Term	Description
Haptic	Haptic or kinesthetic communication recreates the sense of touch by applying forces, vibrations, or motions to the user.
In-Process	Something that executes in the application's process.
Instance	The top-level object, which represents the application's connection to the runtime. Represented by an XrInstance object.
Normalized	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.
Out-Of-Process	Something that executes outside the application's process.
Promoted	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: • It may be renamed • A small number of non-intrusive parameters may have been added • The feature may be advertised differently by device features • The author ID suffixes will be changed or removed as appropriate
Provisional	A feature is released provisionally in order to get wider feedback on the functionality before it is finalized. Provisional features may change in ways that break backwards compatibility, and thus are not recommended for use in production applications.
Required Extensions	Extensions that must be enabled alongside extensions dependent on them, or that must be enabled to use given hardware.

Term	Description
Runtime	The software which implements the OpenXR API and allows applications to interact with XR hardware.
Swapchain	A resource that represents a chain of images in device memory. Represented by an XrSwapchain object.
Swapchain Image	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.

Abbreviations

Abbreviations and acronyms are sometimes used in the API where they are considered clear and commonplace, and are defined here:

Abbreviation	Description
API	Application Programming Interface
AR	Augmented Reality
ER	Eye Relief
IAD	Inter Axial Distance
IPD	Inter Pupillary Distance
MR	Mixed Reality
OS	Operating System
TSG	Technical Sub-Group. A specialized sub-group within a Khronos Working Group (WG).
VR	Virtual Reality
WG	Working Group. An organized group of people working to define/augment an API.
XR	VR + AR + MR

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.

Working Group Contributors to OpenXR

- Adam Gousetis, Google (version 1.0)

- Alain Zanchetta, Microsoft (version 1.1)

- Alex Turner, Microsoft (versions 1.0, 1.1)

- Alex Sink, HTC (version 1.1)

- Alfredo Muniz, XEED (version 1.1) (Working Group Chair)

- Andreas Loeve Selvik, Meta Platforms (versions 1.0, 1.1)

- Andres Rodriguez, Valve Software (version 1.0)

- Armelle Laine, Qualcomm Technologies (version 1.0)

- Attila Maczak, CTRL-labs (version 1.0)

- David Fields, Microsoft (version 1.1)

- Baolin Fu, ByteDance (version 1.1)

- Blake Taylor, Magic Leap (version 1.0)

- Brad Grantham, Google (version 1.0)

- Brandon Jones, Google (version 1.0)

- Brent E. Insko, Intel (version 1.0) (former Working Group Chair)

- Brent Wilson, Microsoft (version 1.0)

- Bryce Hutchings, Microsoft (versions 1.0, 1.1)

- Cass Everitt, Meta Platforms (versions 1.0, 1.1)

- Charles Egenbacher, Epic Games (version 1.0)

- Charlton Rodda, Collabora (version 1.1)

- Chris Kuo, HTC (version 1.1)

- Chris Osborn, CTRL-labs (version 1.0)

- Christine Perey, Perey Research & Consulting (version 1.0)

- Christoph Haag, Collabora (version 1.0, 1.1)

- Christopher Fiala, Epic Games (version 1.1)

- Craig Donner, Google (version 1.0)

- Dan Ginsburg, Valve Software (version 1.0)

- Dave Houlton, LunarG (version 1.0)

- Dave Shreiner, Unity Technologies (version 1.0)

- Darryl Gough, Microsoft (version 1.1)

- Denny Rönngren, Varjo (versions 1.0, 1.1)

- Dmitriy Vasilev, Samsung Electronics (version 1.0)

- Doug Twileager, ZSpace (version 1.0)

- Ed Hutchins, Meta Platforms (version 1.0)

- Eryk Pecyna, Meta Platforms (version 1.1)

- Frederic Plourde, Collabora (version 1.1)

- Gloria Kennickell, Meta Platforms (version 1.0)

- Gregory Greeby, AMD (version 1.0)

- Guodong Chen, Huawei (version 1.0)

- Jack Pritz, Unity Technologies (versions 1.0, 1.1)

- Jakob Bornecrantz, Collabora (versions 1.0, 1.1)

- Jared Cheshier, PlutoVR (versions 1.0, 1.1)

- Jared Finder, Google (version 1.1)

- Javier Martinez, Intel (version 1.0)

- Jeff Bellinghausen, Valve Software (version 1.0)

- Jiehua Guo, Huawei (version 1.0)

- Joe Ludwig, Valve Software (versions 1.0, 1.1)

- John Kearney, Meta Platforms (version 1.1)

- Johannes van Waveren, Meta Platforms (version 1.0)

- Jon Leech, Khronos (version 1.0)

- Jonas Pegerfalk, Tobii (version 1.1)

- Jonathan Wright, Meta Platforms (versions 1.0, 1.1)

- Juan Wee, Samsung Electronics (version 1.0)

- Jules Blok, Epic Games (version 1.0)

- Jun Yan, ByteDance (version 1.1)

- Karl Schultz, LunarG (version 1.0)

- Karthik Kadappan, Magic Leap (version 1.1)

- Karthik Nagarajan, Qualcomm Technologies (version 1.1)

- Kaye Mason, Google (version 1.0)

- Krzysztof Kosiński, Google (version 1.0)

- Kyle Chen, HTC (version 1.1)

- Lachlan Ford, Google (versions 1.0, 1.1)

- Lubosz Sarnecki, Collabora (version 1.0)

- Mark Young, LunarG (version 1.0)

- Martin Renschler, Qualcomm Technologies (version 1.0)

- Matias Koskela, Tampere University of Technology (version 1.0)

- Matt Wash, Arm (version 1.0)

- Mattias Brand, Tobii (version 1.0)

- Mattias O. Karlsson, Tobii (version 1.0)

- Michael Gatson, Dell (version 1.0)

- Minmin Gong, Microsoft (version 1.0)

- Mitch Singer, AMD (version 1.0)

- Nathan Nuber, Valve (version 1.1)

- Nell Waliczek, Microsoft (version 1.0)

- Nick Whiting, Epic Games (version 1.0) (former Working Group Chair)

- Nigel Williams, Sony (version 1.0)

- Nihav Jain, Google, Inc (version 1.1)

- Paul Pedriana, Meta Platforms (version 1.0)

- Paulo Gomes, Samsung Electronics (version 1.0)

- Peter Kuhn, Unity Technologies (versions 1.0, 1.1)

- Peter Peterson, HP Inc (version 1.0)

- Philippe Harscoet, Samsung Electronics (versions 1.0, 1.1)

- Pierre-Loup Griffais, Valve Software (version 1.0)

- Rafael Wiltz, Magic Leap (version 1.1)

- Rajeev Gupta, Sony (version 1.0)

- Remi Arnaud, Starbreeze (version 1.0)

- Remy Zimmerman, Logitech (version 1.0)

- Ria Hsu, HTC (version 1.1)

- River Gillis, Google (version 1.0)

- Robert Blenkinsopp, Ultraleap (version 1.1)

- Robert Memmott, Meta Platforms (version 1.0)

- Robert Menzel, NVIDIA (version 1.0)

- Robert Simpson, Qualcomm Technologies (version 1.0)

- Robin Bourianes, Starbreeze (version 1.0)

- Ron Bessems, Magic Leap (version 1.1) (Working Group Vice-Chair)

- Rune Berg, independent (version 1.1)

- Rylie Pavlik, Collabora (versions 1.0, 1.1) (Spec Editor)

- Ryan Vance, Epic Games (version 1.0)

- Sam Martin, Arm (version 1.0)

- Satish Salian, NVIDIA (version 1.0)

- Scott Flynn, Unity Technologies (version 1.0)

- Shanliang Xu, ByteDance (version 1.1)

- Sean Payne, CTRL-labs (version 1.0)

- Sophia Baldonado, PlutoVR (version 1.0)

- Steve Smith, Epic Games (version 1.0)

- Sungye Kim, Intel (version 1.0)

- Tom Flynn, Samsung Electronics (version 1.0)

- Trevor F. Smith, Mozilla (version 1.0)

- Victor Brodin, Epic Games (version 1.1)

- Vivek Viswanathan, Dell (version 1.0)

- Wenlin Mao, Meta Platforms (version 1.1)

- Xiang Wei, Meta Platforms (version 1.1)

- Yin Li, Microsoft (versions 1.0, 1.1)

- Yuval Boger, Sensics (version 1.0)

- Zhanrui Jia, ByteDance (version 1.1)

- Zheng Qin, Microsoft (version 1.0)

Index