OpenXR™ Conformance Test Suite Usage Instructions and Developer Guide

The Khronos® OpenXR Working Group

Version 1.1.50: from git ref cts-1.1.50.0-20250814

Table of Contents

Chapter 1. Introduction

The OpenXR Conformance Test Suite is a collection of tests covering the breadth of the OpenXR API. The primary purpose of the OpenXR Conformance Test Suite (CTS) is to verify that runtimes have correctly implemented the standard as a part of the conformance process for specification adopters. Additional purposes are:

- To promote consistent behavior among all runtimes, even from non-adopters, for the health of the ecosystem.
- To provide an aid to runtime development.
- To provide a quality assurance tool for runtime vendors to integrate into their own processes.

It is **not** broadly intended to show examples of usage for application authors, in most cases, as the CTS tests the runtime's responses to incorrect and invalid behavior as well as intended application flow. However, in the absence of other samples and sufficiently useful specification text, it can be used as an additional reference.

Chapter 2. Configuration

The canonical interface to the OpenXR CTS is through a command line application, `conformance_cli`. This presents a modified version of the Catch2 command line interface that adds parameters. Other frontends, such as the Android APK, internally convert their configuration into the equivalent command line arguments to pass to the CTS shared library.

All shorthand versions of added options use capital (uppercase) letters, to avoid conflicting with the underlying Catch2 options.

```
-G, --graphicsPlugin <Vulkan|Vulkan2      Specify a graphics plugin to use.
|OpenGLES|OpenGL|D3D11|D3D12|Metal>      Required.
--minApiVersion <1.0|1.1>                 Specify the minimum (and default)
                                          OpenXR API version to use.
                                          Default is 1.0.
-S, --randSeed <uint64_t random seed>     Specify a random seed to use
                                          (decimal or hex). Default is a
                                          dynamically chosen value.
-F, --formFactor <HMD|Handheld>           Specify a form factor to use.
                                          Default is HMD.
--hands <left|right|both>                 Choose which hands to test: left,
                                          right, or both. Default is both.
-V, --viewConfiguration <Stereo|Mono>     Specify view configuration.
                                          Default is Stereo.
-B, --environmentBlendMode <Opaque        Specify blend mode. Default is
|Additive|AlphaBlend>                     Opaque.
-L, --enabledAPILayer <API layer name>    Specify API layer. May repeat for
                                          multiple layers. Default is none.
-E, --enabledInstanceExtension            Specify instance extension. May
<extension name>                          repeat for multiple extensions.
                                          Default is none.
-I, --interactionProfiles <interaction    Specify interaction profiles. May
profile>                                  repeat for multiple profiles.
                                          Default is /interaction_profiles/
                                          khr/simple_controller.
-H, --invalidHandleValidation             Enables testing of invalid handle
                                          checking.
-T, --invalidTypeValidation               Enables testing of invalid type
                                          checking.
--nonDisconnectableDevices                Disables tests that requires
                                          disconnectable devices (for
                                          debugging).
-F, --disableFileLineLogging              Disables logging file/line data.
--pollGetSystem                           Retry xrGetSystem until success
                                          or timeout expires before running
                                          tests.
--autoSkipTimeout <uint64_t auto skip     Automatic Skip Timeout (in
timeout milliseconds>                     milliseconds) for tests which
                                          support it
-D, --debugMode                           Sets debug mode as enabled or
                                          disabled.
```

2.1. Instance Configuration

The correct value to use for these options may be dictated by the testing step you are on and the

features and versions of OpenXR you support. See Testing steps.

2.1.1. Minimum API Version

The same basic instance configuration is used across the test suite for a single execution, with additional versions and extensions added as required. By default, all tests are run as OpenXR 1.0 except those that require a higher version. Tests whose behavior is expected to change between versions are invoked from two separate test cases: one run as each version. In this way, a default run will test all API versions reported to be supported by the runtime. In the rare case that you are testing a runtime that is 1.1 conformant but not aiming to be 1.0 conformant (not recommended!), or if you want to verify that even basic functionality works when running at the OpenXR 1.1 level, you can pass:

- `--minApiVersion 1.1` - Create at least OpenXR 1.1.x `XrInstance` handles. (Do not run any tests as OpenXR 1.0.x.)

2.1.2. Graphics Plugin

Specify with the `--graphicsPlugin <plugin>` or `-G <plugin>` option.

- `vulkan` - uses `XR_KHR_vulkan_enable`
- `vulkan2` - uses `XR_KHR_vulkan_enable2`
- `opengl` - uses `XR_KHR_opengl_enable`
- `opengles` - uses `XR_KHR_opengl_es_enable`
- `d3d11` - uses `XR_KHR_D3D11_enable`
- `d3d12` - uses `XR_KHR_D3D12_enable`
- `metal` - uses `XR_KHR_metal_enable`

If a testing step does not require you to run every supported graphics plugin, you may select an arbitrary one to use for that step. However, your runtime must: be able to pass all tests in that step with any supported graphics plugin. The Working Group has simply designated the tests in that step as not heavily depending on the graphics API.

The only case in which a graphics plugin is not specified, is if a "headless" or no-display extension is in use (via the `-E` option) and is being tested.

2.1.3. Interaction Profiles

Some tests use a user-specified interaction profile.

Pass `-I` or `--interactionProfiles` followed by an interaction profile to test, without the leading `/interaction_profile/` (omitted due to technical reasons).

Any extension required to permit the use of a given interaction profile specified this way (but not any additional binding paths in it) is automatically enabled, but only for the tests that use interaction

profile specified in this way.

2.2. Optional Assertions

The Conformance Test Suite contains a variety of checks of optional runtime behavior, in addition to the required behavior. This is primarily checking to see if a runtime validates application input in a way that the specification **permits** but does not require. Performing this optional validation makes a runtime more useful in the software development process and is generally encouraged from a perspective of improving the health of the ecosystem. As these options only add assertions, they **may** be used when running the CTS for submission purposes.

The subsequent documentation assumes the use of `conformance_cli`. If running on Android, these can be appended to the comma-separated `--esa args` string array as shown in the example ADB command lines. If using a different frontend, pass these to `xrcRunConformanceTests` as additional elements of the `argv`/`argc` array in `ConformanceLaunchSettings`.

2.2.1. Handle Validation

To verify that the runtime under test checks handle values for validity and returns `XR_ERROR_INVALID_HANDLE` if a handle provided is invalid, add `--invalidHandleValidation` (or `-H`, its shorthand version) to your `conformance_cli` command line.

2.2.2. Structure Type Enumerant Validation

To verify that the runtime under test checks the `type` member of structures passed and returns `XR_ERROR_VALIDATION_FAILURE` if the given enumerant is invalid, add `--invalidTypeValidation` (or `-T`, its shorthand version) to your `conformance_cli` command line.

2.3. Compatibility Options

- `--pollGetSystem` - Will poll `xrGetSystem` at test suite startup until it succeeds, then proceeds with tests as usual. As not all applications will perform this behavior, runtimes that require it may see decreased compatibility. **Must be called out and justified if used in a submission!**
- `--hands <left|right|both>` - The test by default assumes there is a controller or other interaction method available for both hands. (That is, this option defaults to `both`.) If there is only a single controller/interaction method, pass this option and the hand that it is expected to be associated with. If the controller can be used in both hands, **run the "Interactive action tests" twice** - once with `--hands left` and once with `--hands right`. **TODO** improve this - https://gitlab.khronos.org/openxr/openxr/-/issues/2385
- `--nonDisconnectableDevices` - If you cannot turn off or otherwise disconnect input devices on request, this skips some assertions that requires a person to do that. Applies only to the interactive tests tagged `[actions]` and `[interactive]`. **Must be called out and justified if used in a submission!**

Chapter 3. Conformance Submissions

The following sub-sections are focused primarily on the needs of someone preparing a conformance submission for review.

3.1. Platform, porting, and build considerations

The core of the test suite is build into a `conformance_test` shared library.

`conformance_cli`, a command-line interface application, is provided for running on PCs and other devices and platforms which support this form of application. `conformance_cli` also demonstrates how to build an application which can interop with the `conformance_test` shared library. If the device being tested does not support a command-line interface, a host application **must** be built for the device which the OpenXR runtime will run on. The conformance host **must** invoke `conformance_test`, the test suite shared library.

There is an Android application APK built from largely the same sources that provides similar functionality. The role of command line arguments is served instead by "Intent Extras".

In addition to the main test shared library, use of the "Conformance Layer" is mandatory. This is an OpenXR API layer that automatically checks the conformance of various aspects of runtime behavior. This layer **must** be loaded and active for a test run to be considered valid.

When you plan to submit for conformance, please observe the following considerations to ensure that the build system has accurate source code revision information available to embed in the test suite and output reports. You **must** build from a git repo (forked from either the internal Gitlab or public GitHub) with tags available (a full clone, not shallow). You also **must** either perform a clean build, from an empty binary tree, or at least run `cmake` immediately before building to pick up current source tree status. If your "porting" process (as described by the conformance process documents) involves replacing the build system, you **must** populate the revision data constants in `utilities/git_revision.cpp.in` accurately. The contents of that file affect all "ctsxml" format outputs, as well as an automated "SourceCodeRevision" test that warns if it cannot identify an approved release. (It only checks for the presence of an appropriately-named tag: it does not check for a signature on the tag, so if you have added tags to your repo it **may** not warn if you are not on a release.)

3.2. Testing steps

The tests are marked with tags to accomplish several objectives.

- selecting a subset of tests to run with different arguments (graphics API, preferred interaction profile, API version) to ensure maximum coverage when running the CTS to make a conformance submission.
- selecting a related subset of tests to use as an aid during runtime development.
- selecting those tests that may be run in a scheduled or continuous testing setup for runtime quality

assurance.

The following text lays out the steps to running conformance for submission.

Note: If you do not support OpenXR 1.0, you must supply the `--minApiVersion` argument to specify the lowest OpenXR API version you support, e.g. `--minApiVersion 1.1`. Some tests will automatically test behavior on higher API versions if your runtime supports them, but most tests will run on `minApiVersion`, which defaults to `1.0`.

Please be aware that depending on how you view these instructions, the (very long) sample command lines **may** be wrapped automatically. All the desktop command lines start with `conformance_cli`, while all the Android ones start with `adb shell am`.

3.2.1. Automated tests on each supported graphics API

One run and result file for the each graphics API supported by your runtime.

When using the following examples, omit any graphics API binding extensions your runtime does not support.

Example command lines

```
conformance_cli "exclude:[interactive]" -G d3d11 --reporter ctsxml::out
=automated_d3d11.xml --reporter console

conformance_cli "exclude:[interactive]" -G d3d12 --reporter ctsxml::out
=automated_d3d12.xml --reporter console

conformance_cli "exclude:[interactive]" -G vulkan --reporter ctsxml::out
=automated_vulkan.xml --reporter console

conformance_cli "exclude:[interactive]" -G vulkan2 --reporter ctsxml::out
=automated_vulkan2.xml --reporter console

conformance_cli "exclude:[interactive]" -G opengl --reporter ctsxml::out
=automated_opengl.xml --reporter console

conformance_cli "exclude:[interactive]" -G metal --reporter ctsxml::out
=automated_metal.xml --reporter console
```

Corresponding ADB commands to launch on Android

Omit any graphics API binding extensions your runtime does not support. These commands do not match one-to-one with the desktop examples due to different graphics API availability on Android.

```
adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "exclude:[interactive]" -e graphicsPlugin vulkan -e xmlFilename
automated_vulkan.xml

# Wait until tests complete, then retrieve results with
adb pull /sdcard/Android/data/org.khronos.openxr.cts/files/automated_vulkan.xml


adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "exclude:[interactive]" -e graphicsPlugin vulkan2 -e xmlFilename
automated_vulkan2.xml

# Wait until tests complete, then retrieve results with
adb pull /sdcard/Android/data/org.khronos.openxr.cts/files/automated_vulkan2.xml


adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "exclude:[interactive]" -e graphicsPlugin opengles -e xmlFilename
automated_opengles.xml

# Wait until tests complete, then retrieve results with
adb pull /sdcard/Android/data/org.khronos.openxr.cts/files/automated_opengles.xml
```

3.2.2. Interactive composition tests on each supported graphics API

These are separate as a tester **must** evaluate the results and determine if each test is passed or failed.

These tests use the `khr/simple_controller` interaction profile to review instructions and sample output, as well as to indicate pass or fail.

One run and result file for the each graphics API supported by your runtime.

As previously, when using these examples, omit any graphics API binding extensions your runtime does not support.

Example command lines

```
conformance_cli "[composition][interactive]" -G d3d11 --reporter ctsxml::out
=interactive_composition_d3d11.xml --reporter console

conformance_cli "[composition][interactive]" -G d3d12 --reporter ctsxml::out
=interactive_composition_d3d12.xml --reporter console

conformance_cli "[composition][interactive]" -G vulkan --reporter ctsxml::out
=interactive_composition_vulkan.xml --reporter console

conformance_cli "[composition][interactive]" -G vulkan2 --reporter ctsxml::out
=interactive_composition_vulkan2.xml --reporter console

conformance_cli "[composition][interactive]" -G opengl --reporter ctsxml::out
=interactive_composition_opengl.xml --reporter console

conformance_cli "[composition][interactive]" -G metal --reporter ctsxml::out
=interactive_composition_metal.xml --reporter console
```

Corresponding ADB commands to launch on Android

Omit any graphics API binding extensions your runtime does not support. These commands do not match one-to-one with the desktop examples due to different graphics API availability on Android.

```
adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "[composition][interactive]" -e graphicsPlugin vulkan -e xmlFilename
interactive_composition_vulkan.xml

# Wait until tests complete, then retrieve results with
adb pull
/sdcard/Android/data/org.khronos.openxr.cts/files/interactive_composition_vulkan.xml


adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "[composition][interactive]" -e graphicsPlugin vulkan2 -e xmlFilename
interactive_composition_vulkan2.xml

# Wait until tests complete, then retrieve results with
adb pull
/sdcard/Android/data/org.khronos.openxr.cts/files/interactive_composition_vulkan2.xml


adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "[composition][interactive]" -e graphicsPlugin opengles -e xmlFilename
interactive_composition_opengles.xml

# Wait until tests complete, then retrieve results with
adb pull
/sdcard/Android/data/org.khronos.openxr.cts/files/interactive_composition_opengles.xm
l
```

3.2.3. Interactive scenario tests, for at least one graphics API

These tests focus on interactive behavior of the runtime and are generally undemanding on the graphics API. While they should be possible to pass on all graphics APIs, a conformance submission only requires a report showing an overall pass on a single API.

These tests use the `khr/simple_controller` interaction profile to perform actions and indicate pass or fail.

- One graphics API

Example command lines

```
conformance_cli "[scenario][interactive]" -G opengl --reporter ctsxml::out
=interactive_scenarios.xml --reporter console
```

Corresponding ADB commands to launch on Android

```
adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "[scenario][interactive]" -e xmlFilename interactive_scenarios.xml

# Wait until tests complete, then retrieve results with
adb pull /sdcard/Android/data/org.khronos.openxr.cts/files/interactive_scenarios.xml
```

3.2.4. Interactive action tests, for each generic interaction profile you can bind completely, and each interaction profile whose concrete device/method is supported by your runtime.

These tests are selected with `[actions][interactive]`. Pass the interaction profile to test, without the leading `/interaction_profile/`, with the option `-I`

- One graphics API
- Each interaction profile that is either:
  - A generic interaction profile (such as `khr/simple_controller`) for which all inputs can be activated.
  - The interaction profile with which the device/input method is most closely associated.
    - If more than one interaction profile is closely associated with the device/method, e.g in the case of evolutionary upgrades to controllers, it is recommended to run specifying each of those interaction profiles.

Note that if any extension or extensions are required for a specified interaction profile to be available, the CTS will automatically enable them for the appropriate tests.

One of these tests requires being able to activate all inputs specified (and not marked as "system") for an interaction profile.

For any supported interaction profiles that are valid for `/user/gamepad` rather than `/user/hand/left` and `/user/hand/right`, further filter the tests by specifying the tag `[gamepad]`.

To clarify, if you support providing input for all components of one of the following interaction profiles, specify them as well.

- /interaction_profiles/khr/simple_controller
- /interaction_profiles/ext/hand_interaction_ext
- /interaction_profiles/ext/eye_gaze_interaction

Example command lines

Select the interaction profiles to test based on the preceding description.

```
## Generic: Simple controller
conformance_cli "[actions][interactive]" -G d3d11 -I "khr/simple_controller"
--reporter ctsxml::out=interactive_action_simple_controller.xml --reporter console

## Generic: Hand interaction (whether via hand tracking or controller)
conformance_cli "[actions][interactive]" -G d3d11 -I "ext/hand_interaction_ext"
--reporter ctsxml::out=interactive_action_ext_hand_interaction_interaction.xml
--reporter console

## Generic: Eye gaze
conformance_cli "[actions][interactive]" -G d3d11 -I "ext/eye_gaze_interaction"
--reporter ctsxml::out=interactive_action_ext_eye_gaze_interaction.xml --reporter
console

# Sample device-associated profiles
conformance_cli "[actions][interactive]" -G d3d11 -I "microsoft/motion_controller"
--reporter ctsxml::out=interactive_action_microsoft_motion_controller.xml --reporter
console

conformance_cli "[actions][interactive]" -G d3d11 -I "oculus/touch_controller"
--reporter ctsxml::out=interactive_action_oculus_touch_controller.xml --reporter
console

conformance_cli "[actions][interactive]" -G d3d11 -I "htc/vive_controller" --reporter
ctsxml::out=interactive_action_htc_vive_controller.xml --reporter console
```

Example command lines for a gamepad

```
conformance_cli "[gamepad]" -G d3d11 -I "microsoft/xbox_controller" --reporter
ctsxml::out=interactive_action_microsoft_xbox_controller.xml --reporter console
```

Corresponding ADB commands to launch on Android

Select the interaction profiles to test based on the preceding description.

```
## Generic: Simple controller
adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "[actions][interactive],-I,khr/simple_controller" -e xmlFilename
interactive_action_simple_controller.xml

# Wait until tests complete, then retrieve results with
adb pull
/sdcard/Android/data/org.khronos.openxr.cts/files/interactive_action_simple_controlle
r.xml


## Generic: Hand interaction (whether via hand tracking or controller)
adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "[actions][interactive],-I,ext/hand_interaction_ext" -e xmlFilename
interactive_action_ext_hand_interaction.xml

# Wait until tests complete, then retrieve results with
adb pull
/sdcard/Android/data/org.khronos.openxr.cts/files/interactive_action_ext_hand_interac
tion.xml


## Generic: Eye gaze interaction
adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "[actions][interactive],-I,ext/eye_gaze_interaction" -e xmlFilename
interactive_action_ext_eye_gaze_interaction.xml

# Wait until tests complete, then retrieve results with
adb pull
/sdcard/Android/data/org.khronos.openxr.cts/files/interactive_action_ext_eye_gaze_int
eraction.xml


## Sample device-associated profile
adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "[actions][interactive],-I,oculus/touch_controller" -e xmlFilename
interactive_action_oculus_touch_controller.xml

# Wait until tests complete, then retrieve results with
adb pull
/sdcard/Android/data/org.khronos.openxr.cts/files/interactive_action_oculus_touch_con
troller.xml
```

```
## Sample gamepad profile
adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "[gamepad],-I,microsoft/xbox_controller" -e xmlFilename
interactive_action_microsoft_xbox_controller.xml

# Wait until tests complete, then retrieve results with
adb pull
/sdcard/Android/data/org.khronos.openxr.cts/files/interactive_action_microsoft_xbox_c
ontroller.xml
```

3.3. Conformance Submission Package Requirements

The submission package **must** include each of the following:

1. XML files produced from test runs
2. The console output produced by the CTS runs above.
3. Information on the build of conformance used in generating the results
4. Conformance Statement

3.3.1. XML files produced from test runs

One or more automated test result XML files, 1 per graphics API supported, therefore one or more of the following generated output files:

- `automated_d3d11.xml`
- `automated_d3d12.xml`
- `automated_opengl.xml`
- `automated_gles.xml`
- `automated_vulkan.xml`
- `automated_vulkan2.xml`

The output XML file(s) from running the interactive tests, 1 per supported graphics API, therefore one or more of the following generated output files:

- `interactive_composition_d3d11.xml`
- `interactive_composition_d3d12.xml`
- `interactive_composition_opengl.xml`
- `interactive_composition_gles.xml`
- `interactive_composition_vulkan.xml`

- `interactive_composition_vulkan2.xml`

At least one output file from running the interactive scenario tests on a single graphics API (more is better):

- `interactive_scenarios.xml`

The output XML file(s) from running the interactive action tests, 1 per supported interaction profile, therefore one or more of the following generated output files. This list below are example files, each platform **may** have their own controllers though `simple_controller` is expected to be supported at a minimum.

- `interactive_action_simple_controller.xml`
- `interactive_action_microsoft_xbox_controller.xml`
- `interactive_action_microsoft_motion_controller.xml`
- `interactive_action_oculus_touch_controller.xml`
- `interactive_action_valve_index_controller.xml`
- `interactive_action_htc_vive_controller.xml`

3.3.2. The console output produced by the CTS runs

Note

Can we drop this requirement? https://gitlab.khronos.org/openxr/openxr/-/issues/2386

Each test suite run starts by printing test configuration data, and ends by printing a "Report" showing details of the runtime and environment (extensions, etc) used in that run. A few tests produce console output in-between that does not show up in the result XML. It is important to have this data for the interpretation of the results.

3.3.3. Information on the build of conformance used in generating the results

Files containing the result of the commands `git status` and `git log` from the CTS directory:

- `git_status.txt`
- `git_log.txt`

If there were changes required to pass the conformance test suite, a diff of the changes from a tagged approved release of the suite **must** be included as well:

- `git_diff.txt`

Alternately, the diff **may** be supplied in `git format-patch` format.

Note that **only the tagged releases on the OpenXR-CTS repo are accepted** without a diff: the latest

such release will always be on the `approved` branch. The default `devel` branch is useful during development, but has not yet been voted on by the working group and is thus ineligible for submissions without a full diff. If the `devel` branch works better for you, you may consider encouraging the working group to tag a new release of conformance.

3.3.4. Conformance Statement

A file containing information regarding the submission called `statement-<adopter>.txt`

```
CONFORM_VERSION:         <git tag of CTS release>
PRODUCT:                 <string-value>
CPU:                     <string-value>
OS:                      <string-value>

WARNING_EXPLANATIONS:    <optional> <paragraph describing why the warnings present in the
conformance logs are not indications of conformance failure>
WAIVERS:                 <optional> <paragraph describing waiver requests for non-
conformant test results>
```

The actual submission package consists of the above set of files which **must** be bundled into a gzipped tar file named `XR<API major><API minor>_<adopter><_info>.tgz`. `<API major>` is the major version of the OpenXR API specification, `<API minor>` is the minor version of the OpenXR API specification. `<adopter>` is the name of the Adopting member company, or some recognizable abbreviation. The `<_info>` field is optional. It **may** be used to uniquely identify a submission by OS, platform, date, or other criteria when making multiple submissions. For example, a company XYZ **may** make a submission for an OpenXR 1.1 implementation named `XR11_XYZ_PRODUCTA_Windows10.tgz`.

3.3.5. Waivers

Any test failures due to presumed bugs in the conformance tests not matching specification behavior should be submitted as issues with potential fixes against the conformance suite. Waivers are requested for test failures where the underlying platform fails to meet the expected specification behavior. These are requested in the statement file as described above. Enough detail should be provided such that submission reviewers can judge the potential impact and risk to the ecosystem of approving the submission.

3.4. Passing Criteria

A conformance run is considered passing if **all tests** required by the testing steps finish with allowed result codes (pass or skip as appropriate), and all warnings are acceptably explained to describe why they are not a conformance failure. `XR_EXT_conformance_automation` **must** not be used for conformance submission.

Any error or failure when testing functionality means your runtime is **not conformant**.

Any warnings **may** indicate non-conformance and **must** be explained in the submission package.

Test results are contained in the output XML files, which are an extension of the common "*Unit" schema with some custom elements. Each test case leaf section is reached by a run of its own, and is recorded with a `testcase` tag, e.g.:

```
<testcase classname="global" name="Swapchains/Swapchain creation test parameters"
time="1.207" status="run">
```

If all assertions in that case passed, there are no child elements to the `testcase` tag. However, `testcase` tags can contain a warning, failure, or error:

```
<cts:warning type="WARN">
```

or

```
<failure message="..." type="...">
```

or

```
<error message="..." type="...">
```

With the results of the entire run summarized in `testsuite` tag (listing the number of assertions):

```
<testsuite errors="0" failures="0" tests="<number of successful assertions>" time=
"1.407">
```

as well as in the `cts:results` tag in `cts:ctsConformanceReport` (listing the number of top level test cases):

```
<cts:results testSuccessCount="1" testFailureCount="0"/>
```

Chapter 4. Requirements/Assumptions for Testing

While the specification defines behavior over a wide range of possible scenarios, the CTS requires some assumptions to be satisfied before running in various modes. They are broadly meant to be representative of nominal usage of an XR system operating as expected without special configuration by the user or per-application. If you are unable to satisfy these requirements, you **may** modify the CTS according to the Conformance Process Document and the "Porting" process detailed therein. Early and frequent discussion with the Working Group is encouraged in this case.

Some of these assumptions can be bypassed or automated away by implementing `XR_EXT_conformance_automation` and passing `-E XR_EXT_conformance_automation` to the test suite. However, this is intended only for automated testing, and not for conformance submission. A conformant runtime **must** be able to pass the tests without this extension.

4.1. Requirements for non-interactive and interactive tests

All test cases depend on these requirements. Some have exceptions or workarounds available as options.

4.1.1. System immediately available

Command line argument `--pollGetSystem` is available to retry `xrGetSystem` at startup of the CTS. Subsequent `XrInstance` creation is assumed to immediately have access to a system after this initial poll.

4.1.2. Session state progress

Some tests require that a begun session progresses through the lifecycle to `XR_SESSION_STATE_FOCUSED` without user interaction. These tests will synchronize their frame loop and start submitting frames and expect to proceed to `FOCUSED`. This **may** require donning a device or otherwise triggering a user presence sensor, for these tests to run fully automatedly.

4.1.3. Valid view pose in local space

Some tests require valid view pose (`XR_VIEW_STATE_ORIENTATION_VALID_BIT` & `XR_VIEW_STATE_POSITION_VALID_BIT`), though not necessarily tracked view position, relative to LOCAL reference space (`XR_REFERENCE_SPACE_TYPE_LOCAL`).

4.1.4. No customized rebinding applied

Unless otherwise specified, the behavior of the runtime **must** match the "when following suggested

bindings" case. Any manual rebindings necessary to pass the tests **must** be discussed with the Working Group.

4.2. Requirements for interactive tests

4.2.1. At least one controller/interaction method is available, by default two

Assumes both a left and right hand input device (controller, hand interaction) is available unless arguments are specified to switch to a single hand mode.

Tests without the `[interactive]` flag do not depend on the presence of a controller.

4.2.2. Available interaction method supports `/interaction_profile/khr/simple_controller`

The runtime **must** support `khr/simple_controller` to manually pass or fail each test through the `⋯/input/menu/click` and `⋯/input/select/click` input paths. The tester **must** evaluate the composed output and pass or fail the tests by comparing it to the provided expected result image. While `/interaction_profile/khr/simple_controller` is not formally required for conformance as of OpenXR 1.1, it is strongly encouraged. If it cannot be included for some reason, use of another interaction profile **may** be performed through the porting process.

4.2.3. View reference space is tracked in local space

Some interactive tests require tracked pose (`XR_SPACE_LOCATION_ORIENTATION_TRACKED_BIT` & `XR_SPACE_LOCATION_POSITION_TRACKED_BIT`) of VIEW reference space relative to LOCAL reference space (`XR_REFERENCE_SPACE_TYPE_LOCAL`). They **must** remain tracked throughout the test: if tracking is lost during the test, a spurious failure may be recorded, requiring a re-run of the tests.

4.2.4. The "Menu" click input on the controller can be held without interrupting the application

Interactive tests typically show their instructions and sample output only while a boolean action, with suggested bindings for left and right `⋯/input/menu/click`, is "true". Additionally, a boolean action with suggested bindings for left and right `⋯/input/select/click` is typically used to report pass or fail, with both of those actions being "true" indicating failure. If pressing or holding the "menu" button brings up system UI, reviewing the instructions and expected behavior for interactive tests may be difficult. (It would be possible to remove this assumption through the porting process or other CTS development.) If pressing "select" while holding "menu" is not possible, it will be impossible to record an interactive test failure.

(Tests using `InteractiveLayerManager` have this behavior.)

4.2.5. The controller or input method must be capable of being turned off/disconnected in some way

A command line flag is available, `--nonDisconnectableDevices`, although it is primarily for debugging. Submissions that require it **must** explain their use of this flag in the conformance statement.

4.2.6. A person must perform actions to evaluate the runtime behavior for several tests

For some interactive tests, a person operating the test suite **must** use the in-test instructions to evaluate whether the behavior of the runtime matches the expected behavior according to the specification. If there is any doubt about whether the behavior in a test is conformant, consult with the Working Group.

- `[scenario]` tests - the tester **must** perform a certain set of actions to pass the test. The runtime **must** additionally behave as described in the instructions, which involves some subjective judgement, even if semi-automated requirements are met.
- `[composition][interactive]` tests - the visible rendering result **must** be compared against a reference image.
- `[actions]` tests - in at least one interactive test, you have to wait for haptic feedback and confirm it, in addition to performing input actions requested.

Chapter 5. Interactive Self Tests

Some interactive tests are primarily a test of mechanisms within the CTS, rather than runtime functionality. These are labeled with the tag `[self_test]` rather than `[scenario]`, `[actions]`, or `[composition]`. While it is good to run these, and doing so may help troubleshoot failures with tests that build on them, submission of a CTS results package does not require them. Currently, the only self-tests tagged in this way are for the PBR/glTF rendering subsystem. They (asynchronously) load very large, artificial test assets, originally from the "glTF-Sample-Models" repository, to test specific details of the renderer.

To run the self-tests, commands similar to the following can be used:

Example command lines

Omit any graphics API binding extensions your runtime does not support.

```
conformance_cli "[self_test][interactive]" -G d3d11 --reporter ctsxml::out
=interactive_self_test_d3d11.xml --reporter console
conformance_cli "[self_test][interactive]" -G d3d12 --reporter ctsxml::out
=interactive_self_test_d3d12.xml --reporter console
conformance_cli "[self_test][interactive]" -G vulkan --reporter ctsxml::out
=interactive_self_test_vulkan.xml --reporter console
conformance_cli "[self_test][interactive]" -G vulkan2 --reporter ctsxml::out
=interactive_self_test_vulkan2.xml --reporter console
conformance_cli "[self_test][interactive]" -G opengl --reporter ctsxml::out
=interactive_self_test_opengl.xml --reporter console
```

Corresponding ADB commands to launch on Android

Omit any graphics API binding extensions your runtime does not support. These commands do not match one-to-one with the desktop examples due to different graphics API availability on Android.

```
adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "[self_test][interactive]" -e graphicsPlugin vulkan -e xmlFilename
interactive_self_test_vulkan.xml

# Wait until tests complete, then retrieve results with
adb pull
/sdcard/Android/data/org.khronos.openxr.cts/files/interactive_self_test_vulkan.xml


adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "[self_test][interactive]" -e graphicsPlugin vulkan2 -e xmlFilename
interactive_self_test_vulkan2.xml

# Wait until tests complete, then retrieve results with
adb pull
/sdcard/Android/data/org.khronos.openxr.cts/files/interactive_self_test_vulkan2.xml


adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "[self_test][interactive]" -e graphicsPlugin opengles -e xmlFilename
interactive_self_test_opengles.xml

# Wait until tests complete, then retrieve results with
adb pull
/sdcard/Android/data/org.khronos.openxr.cts/files/interactive_self_test_opengles.xml
```

Chapter 6. Automating CTS for Internal Testing and Development

As the CTS can be a useful tool when developing and maintaining a runtime, there are ways to run it that are amenable to automation

6.1. With Conformance Automation

If your runtime implements `XR_EXT_conformance_automation`, you can run a large number of tests without actually interacting.

The key parts to the command line here are:

- `-E XR_EXT_conformance_automation` - Force enable the conformance automation extension. This is detected and used to replace user input in many tests.
- `--autoSkipTimeout 3000` - Proceeds without interaction (via a test "skip") through tests awaiting user judgement, after 3 seconds (3000ms). This mainly affects the interactive composition tests. This is primarily to exercise runtime behavior incidentally triggered by these tests.
- `exclude:[no_auto]` - the `[no_auto]` tag is applied to test cases that support neither conformance automation nor the auto-skip timeout.

Example command lines

```
conformance_cli "exclude:[no_auto]" -E XR_EXT_conformance_automation
--autoSkipTimeout 3000 -G d3d11 --reporter ctsxml::out
=conformance_automation_d3d11.xml --reporter console

conformance_cli "exclude:[no_auto]" -E XR_EXT_conformance_automation
--autoSkipTimeout 3000 -G d3d12 --reporter ctsxml::out
=conformance_automation_d3d12.xml --reporter console

conformance_cli "exclude:[no_auto]" -E XR_EXT_conformance_automation
--autoSkipTimeout 3000 -G vulkan --reporter ctsxml::out
=conformance_automation_vulkan.xml --reporter console

conformance_cli "exclude:[no_auto]" -E XR_EXT_conformance_automation
--autoSkipTimeout 3000 -G vulkan2 --reporter ctsxml::out
=conformance_automation_vulkan2.xml --reporter console

conformance_cli "exclude:[no_auto]" -E XR_EXT_conformance_automation
--autoSkipTimeout 3000 -G opengl --reporter ctsxml::out
=conformance_automation_opengl.xml --reporter console

conformance_cli "exclude:[no_auto]" -E XR_EXT_conformance_automation
--autoSkipTimeout 3000 -G metal --reporter ctsxml::out
=conformance_automation_metal.xml --reporter console
```

Corresponding ADB commands to launch on Android

Omit any graphics API binding extensions your runtime does not support. These commands do not match one-to-one with the desktop examples due to different graphics API availability on Android.

```
adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "exclude:[no_auto],-E,XR_EXT_conformance_automation,--
autoSkipTimeout,3000" -e graphicsPlugin vulkan -e xmlFilename
conformance_automation_vulkan.xml

# Wait until tests complete, then retrieve results with
adb pull
/sdcard/Android/data/org.khronos.openxr.cts/files/conformance_automation_vulkan.xml


adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "exclude:[no_auto],-E,XR_EXT_conformance_automation,--
autoSkipTimeout,3000" -e graphicsPlugin vulkan2 -e xmlFilename
conformance_automation_vulkan2.xml

# Wait until tests complete, then retrieve results with
adb pull
/sdcard/Android/data/org.khronos.openxr.cts/files/conformance_automation_vulkan2.xml


adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "exclude:[no_auto],-E,XR_EXT_conformance_automation,--
autoSkipTimeout,3000" -e graphicsPlugin opengles -e xmlFilename
conformance_automation_opengles.xml

# Wait until tests complete, then retrieve results with
adb pull
/sdcard/Android/data/org.khronos.openxr.cts/files/conformance_automation_opengles.xml
```

6.2. Without Conformance Automation

If your runtime does not implement `XR_EXT_conformance_automation`, fewer tests are available for automatic use.

The first collection of such tests are the non-interactive (automated) tests: see Testing steps.

The second collection are the interactive composition tests, configured in auto-skip mode as previously discussed. The key parts to the command line here are:

- `"[composition][interactive]"` - Select only the interactive composition tests, which are all compatible with auto-skip timeouts.
- `--autoSkipTimeout 3000` - Proceeds without interaction (via a test "skip") through tests awaiting user judgement, after 3 seconds (3000ms). This is primarily to exercise runtime behavior incidentally triggered by these tests.

Example command lines

```
conformance_cli "[composition][interactive]" --autoSkipTimeout 3000 -G d3d11
--reporter ctsxml::out=interactive_composition_autoskip_d3d11.xml --reporter console

conformance_cli "[composition][interactive]" --autoSkipTimeout 3000 -G d3d12
--reporter ctsxml::out=interactive_composition_autoskip_d3d12.xml --reporter console

conformance_cli "[composition][interactive]" --autoSkipTimeout 3000 -G vulkan
--reporter ctsxml::out=interactive_composition_autoskip_vulkan.xml --reporter console

conformance_cli "[composition][interactive]" --autoSkipTimeout 3000 -G vulkan2
--reporter ctsxml::out=interactive_composition_autoskip_vulkan2.xml --reporter
console

conformance_cli "[composition][interactive]" --autoSkipTimeout 3000 -G opengl
--reporter ctsxml::out=interactive_composition_autoskip_opengl.xml --reporter console

conformance_cli "[composition][interactive]" --autoSkipTimeout 3000 -G metal
--reporter ctsxml::out=interactive_composition_autoskip_metal.xml --reporter console
```

Corresponding ADB commands to launch on Android

Omit any graphics API binding extensions your runtime does not support. These commands do not match one-to-one with the desktop examples due to different graphics API availability on Android.

```
adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "[composition][interactive],--autoSkipTimeout,3000" -e graphicsPlugin
vulkan -e xmlFilename interactive_composition_autoskip_vulkan.xml

# Wait until tests complete, then retrieve results with
adb pull
/sdcard/Android/data/org.khronos.openxr.cts/files/interactive_composition_autoskip_vu
lkan.xml


adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "[composition][interactive],--autoSkipTimeout,3000" -e graphicsPlugin
vulkan2 -e xmlFilename interactive_composition_autoskip_vulkan2.xml

# Wait until tests complete, then retrieve results with
adb pull
/sdcard/Android/data/org.khronos.openxr.cts/files/interactive_composition_autoskip_vu
lkan2.xml


adb shell am start-activity -S -n org.khronos.openxr.cts/android.app.NativeActivity
--esa args "[composition][interactive],--autoSkipTimeout,3000" -e graphicsPlugin
opengles -e xmlFilename interactive_composition_autoskip_opengles.xml

# Wait until tests complete, then retrieve results with
adb pull
/sdcard/Android/data/org.khronos.openxr.cts/files/interactive_composition_autoskip_op
engles.xml
```

Chapter 7. Test Source Code

The CTS uses Catch2 as the testing framework. In addition to mechanisms provided by Catch2, the conformance framework provides substantial features both for testing as well as for setting up preconditions to be able to test OpenXR functionality.

7.1. Assigning Tags

Tests should be categorized using tags. The following common tags are used:

- `[interactive]`: indicates that a larger set of requirements apply, requiring interaction at some level.
  - Each test labeled interactive **must** have one of the following tags in addition.
    - `[actions]`: indicates that this test is an "Action" test. The additional tag `[gamepad]` selects only the tests relevant to testing a gamepad (rather than a handed motion controller).
    - `[composition]`: indicates that this test requires the tester to do a visual comparison.
    - `[scenario]`: indicates that the tester **must** perform a certain set of actions to pass the test.
    - `[self_test]`: indicates that the test is primarily a test of mechanisms within the CTS, rather than runtime functionality, and are not required in a conformance submission. See Interactive Self Tests
  - A test labeled "interactive" **may** have the following tag to assist running tests in unattended environments:
    - `[no_auto]`: indicates that this interactive test cannot be automated by the `XR_EXT_conformance_automation` extension nor by the `--autoSkipTimeout` argument..
- The name of an extension whose behavior is tested, e.g. `[XR_KHR_visibility_mask]`. This **may** be a required extension for the test, or simply an extension that modifies the behavior and thus interacts with the test. These tags are not used in conformance submissions but are useful in development and testing.
- `[XR_VERSION_1_1]`: indicates a test evaluates functionality specific to OpenXR 1.1. This tag is not used in conformance submissions but is useful in development and testing.

7.2. Choosing an assertion

It provides `CHECK`, `REQUIRE`, and `WARN` families of macros, among others. In addition, the CTS framework provides a collection of `XRC_THROW_…` macros. When should each be used?

- All of these assertions except for `WARN` are considered a conformance failure.
- The difference between `CHECK` and `REQUIRE` is whether the current test case run (that is, a run through a test case up to a given leaf in the section tree) continues after a failed assertion.
  - If it makes no sense for the test to continue (e.g. because preconditions have failed), use `REQUIRE`.

    - Otherwise, use `CHECK`: this still fails the test but allows more than one failure to be logged from a single pass, making it easier in many cases to diagnose and fix the failure.

- Both `CHECK` and `REQUIRE` are considered testing assertions, so they should only be used when the check is relevant to the subject of a given test. That is, they both increment the logged assertion count, and both will log the assertion even if passed if the appropriate command line arguments are passed.
    - For other preconditions, or shared helper code, use an `XRC_THROW` macro: throwing an exception works just like `REQUIRE` (halting that test case run) but checking with them or throwing does not increment the assertion count or log all the data associated with an assertion.

So, to summarize:

`WARN`
: For a check relevant to the current test that requires an explanation by the runtime vendor, as it might be non-conformant.

`CHECK`...
: For most assertions relevant to the current test that should not immediately halt execution of a test case.

`REQUIRE`...
: For assertions relevant to the current test that **do** require immediately halting execution of a test case.

`XRC_THROW_`...
: For things that require stopping the test if they fail, but are not directly relevant to the current test, such as setup code or shared helpers.

See also:

- Catch2 assertion documentation