

OpenMAX™ Integration Layer  
Conformance Test

Version 1.1.2

Copyright © 2019 The Khronos Group Inc.

Document version 2008.09.01

[Appendix 1. Scope 5](#_Toc192642328)

[1.1. Document Purpose 5](#_Toc192642329)

[1.2. Intended Audience 5](#_Toc192642330)

[1.3. Word Definitions used in this document 5](#_Toc192642331)

[Appendix 2. IL Component States, Events, and Structures 6](#_Toc192642332)

[2.1. IL Component Definition 6](#_Toc192642333)

[2.2. Component States 6](#_Toc192642334)

[2.2.1. State Definitions 6](#_Toc192642335)

[2.2.2. Valid state transitions 7](#_Toc192642336)

[ADD transition to StateInvalid from all states 8](#_Toc192642337)

[2.3. OMX Events & Callbacks 8](#_Toc192642338)

[2.4. IL API contents 9](#_Toc192642339)

[2.4.1. Basic Data Types 9](#_Toc192642340)

[2.4.2. OMX State Enumeration 9](#_Toc192642341)

[2.4.3. OMX Errors 9](#_Toc192642342)

[2.4.4. Data Structures 10](#_Toc192642343)

[Appendix 3. IL API Base Profile Conformance Tests 10](#_Toc192642344)

[3.1. Component Version Test 11](#_Toc192642345)

[3.2. Port Parameter Test 11](#_Toc192642346)

[3.2.1. Failed SetParam Test 11](#_Toc192642347)

[3.2.2. Required Parameter Test 11](#_Toc192642348)

[3.2.3. Supported Parameter Test 12](#_Toc192642349)

[3.2.4. Successful Default Parameters Test 12](#_Toc192642350)

[3.2.5. Unsuccessful Default Parameters Test 12](#_Toc192642351)

[3.2.6. Required Configuration Test 12](#_Toc192642352)

[3.2.7. Supported Configurations Test 12](#_Toc192642353)

[3.2.8. Successful Default Configuration Test 12](#_Toc192642354)

[3.2.9. Unsuccessful Default Configuration Test 13](#_Toc192642355)

[3.3. Resource Exhaustion Test 13](#_Toc192642356)

[3.3.1. Component Initialization Failure(insufficient resources) 13](#_Toc192642357)

[3.3.2. Successful De-allocation(resource constrained) 13](#_Toc192642358)

[3.4. Buffer Test 13](#_Toc192642359)

[3.4.1. Port populated 13](#_Toc192642360)

[3.4.2. Port unpopulated 14](#_Toc192642361)

[3.4.3. Per-Port Enable / Disable 14](#_Toc192642362)

[3.5. Port Communication Test 14](#_Toc192642363)

[3.5.1. Non-Tunneling Setup Test 14](#_Toc192642364)

[3.5.2. Transmit Buffers Between Input and Output Ports and Application 14](#_Toc192642365)

[3.5.3. Port Repack Test 15](#_Toc192642366)

[3.5.4. Port Stop and Restart Test 15](#_Toc192642367)

[3.5.5. Component Pause and Resume Test 15](#_Toc192642368)

[3.6. State Transition Tests 15](#_Toc192642369)

[3.7. Flush Tests 16](#_Toc192642370)

[3.7.1. Successfully Flush Component Buffers 16](#_Toc192642371)

[3.8. Base Multi Thread Test 16](#_Toc192642372)

[3.8.1. Buffer Processing in a separate Thread 16](#_Toc192642373)

[3.8.2. Initialization from 2 threads 17](#_Toc192642374)

[3.8.3. Start and Stop in different Threads 17](#_Toc192642375)

[3.8.4. Unexpected Stop from another Thread Test 17](#_Toc192642376)

[3.8.5. Multiple Init/Deinit case 17](#_Toc192642377)

[3.9. Buffer Flag Tests 17](#_Toc192642378)

[3.9.1. Mark Buffer Propagation 17](#_Toc192642379)

[3.9.2. Demonstrate Mark Buffer Event and Clearing 18](#_Toc192642380)

[3.9.3. Demonstrate Mark Buffer For Previously Marked Buffers 18](#_Toc192642381)

[3.9.4. Demonstrate Mark Buffer For Previously Marked Buffers for EOS condition 18](#_Toc192642382)

[3.9.5. Demonstrate EOS Functionality 18](#_Toc192642383)

[3.10. Component Naming Convention 18](#_Toc192642384)

[Appendix 4. Interop Profile Tests 18](#_Toc192642385)

[4.1. Tunneling Tests 19](#_Toc192642386)

[4.2. Incomplete Stop Failure 19](#_Toc192642387)

[4.3. Demonstrate Port Stop and Restart 19](#_Toc192642388)

[4.4. Minimum Buffer Size for Standard Port 20](#_Toc192642389)

[4.5. Multithreaded components tunneled test 20](#_Toc192642390)

[4.6. Resource Manager Tests 20](#_Toc192642391)

[4.6.1. Resource Preemption 20](#_Toc192642392)

[4.6.2. Wait For Resources 20](#_Toc192642393)

[4.7. Clock Component Test 21](#_Toc192642394)

[4.8. Seeking Test 21](#_Toc192642395)

[Appendix 5. Standard Component Conformance Tests 21](#_Toc192642396)

[5.1. Standard Component Interface Test 24](#_Toc192642397)

[5.2. Standard Component Data Metabolism Test 24](#_Toc192642398)

[5.2.1. Per Role Requirements 25](#_Toc192642399)

[Appendix 6. Leveraging Buffer Variance File for Interop and Base Profile Tests 28](#_Toc192642400)

[Table 1 - Definitions for Commonly Used Words 5](#_Toc171397982)

[Table 2 - OMX States 6](#_Toc171397983)

[Table 3 - State Transitions 7](#_Toc171397984)

[Table 4 - OMX Data Types 9](#_Toc171397985)

[Table 5 – OMX State Enumeration 9](#_Toc171397986)

# Scope

## Document Purpose

This document defines the requirements for a component claiming conformance to the OpenMAX IL 1.1.2 specification and the associated tests for verifying compliance to those requirements. There are three groups of tests included in the OpenMax IL 1.1.2 conformance test suite associated with the three categories of conformance:

*Base profile conformance* entails interoperability between a component and an IL core and application/framework. Every OpenMax IL 1.1.2 component implementation must be base profile conformant. The OpenMax IL 1.0 conformance test document defines the associated base profile tests.

*Interop conformance* entails direct interoperability between two OpenMAX components and required behavior in the context of resource management. Interop conformance is not required of every OpenMax IL 1.1.2 component implementation. The OpenMax IL 1.0 conformance test document defines the associated interop profile tests.

*Standard component conformance* entails adherence to the criteria defined for each standard component a component implementation claims to support. Every OpenMax IL IL 1.1.2 component implementation must pass standard component conformance though it is possible the component implements no standard roles. The standard component tests are defined in the remainder of this document.

The “cc” command exposed by the conformance test harness automatically determines which tests (e.g. for base profile, for interop profile, and for standard components) are applicable and executes those tests on the specified component. The “cc” command operating on a component must execute successful (i.e. all tests invoked must pass) for the component to be considered a OpenMAX IL 1.1.2 compliant.

## Intended Audience

This document is intended for OpenMax IL 1.1.2 test implementers and prospective adopters.

## Word Definitions used in this document

Within this document, the words listed in Table 1 are used in a specific way to convey a requirement or not convey a requirement. The definition of each word and an example of how it is used is provided.

|  |  |
| --- | --- |
| Table 1 - Definitions for Commonly Used Words | |
| WORD | Definition |
| Shall | “Shall” means it is a requirement. If a component fails to meet the “shall” specification, it fails the test. Shall is always used as a requirement as in “The component designers shall produce good documentation.” |
| Will | “Will”, means that is not a requirement. The word “Will” is usually used when referring to a third party as in “the application framework will correctly handle errors.” |
| Should | “Should” is not a requirement, but is the correct thing to do or is a good practice. The word “Should” is usually used in the as follows, “The component should begin processing buffers immediately after it receives a Start Command.” While this is good practice, there may be a valid reason to delay processing buffers such as not having input data available. |
| May | “May” is not a requirement. This is an optional feature as in “The component may have vendor specific extensions” |

# IL Component States, Events, and Structures

This section of the document summarizes the required states, events, and structures for a component to claim conformance to the OpenMAX IL specification. The requirements for the described conformance tests were derived from the “IL Conformance Test” document revision 0.9. Conformance is based entirely on an anticipated manipulation of state transitions, events, and structures.

## IL Component Definition

An IL component shall exist on the host processor and may wrap the “processing functions” that exist in other processors and/or may also wrap the functionality of hardware accelerators.

## Component States

The compliant IL component shall make state information available to the application framework.

### State Definitions

The component shall be in one of the states defined in Table 2 at all times. The GetState function can be used to request the state of the component. Frequently, within this document, the states are referred to with a shortened version of the state name to promote document readability. For example the phrase “verify the component is in the OMX\_STATE\_EXECUTING state” will likely be shortened to “verify the component is in the Executing state”. The reader should understand these two statements are intended by the author of this document to be equivalent.

|  |  |
| --- | --- |
| Table 2 - OMX States | |
| OMX\_StateInvalid | The component has detected an unrecoverable problem and puts the component into this state. The component must subsequently be de-initialized and unloaded. |
| OMX\_StateLoaded | The component has been loaded by the OMX core, but has not yet been initialized by the application. In this state, the component may not hold resources nor process data. |
| OMX\_StateIdle | Upon successful execution of the StateSet command, the component shall enter this state. As a note resources are allocated to the component during execution of the StateSet command. Therefore, during this state, the component holds any resources necessary for processing data, but the actual processing has not started. |
| OMX\_StateExecuting | Upon successful execution of the StateSet command the component will enter this state. In this state the component will process data. |
| OMX\_StatePaused | The component was running, but has been temporarily halted by the StateSet command. |
| OMX\_StateWaitForResources | The component is waiting for resources to become available before transitioning to the IDLE state. |

### Valid state transitions

A conformant component shall transition from one state to another state in accordance with Table 3.

|  |  |  |
| --- | --- | --- |
| Table 3 - State Transitions | | |
| Current State | Event/Command | Next State |
| NULL | GetHandle Event | OMX\_StateLoaded |
| OMX\_StateLoaded | Unsuccessful StateSet | OMX\_StateLoaded |
| OMX\_StateLoaded | Successful StateSet Command | OMX\_StateInvalid |
| OMX\_StateLoaded | Successful StateSet Command | OMX\_StateIdle |
| OMX\_StateLoaded | Successful StateSet Command | OMX\_StateWaitForResources |
| OMX\_StateIdle | Succesful StateSet Command | OMX\_StateLoaded |
| OMX\_StateIdle | Successful StateSet Command | OMX\_StateExecuting |
| OMX\_StateIdle | Successful StateSet Command | OMX\_StatePaused |
| OMX\_StateIdle | Unsuccessful StateSet or any other command | OMX\_StateIdle |
| OMX\_StateExecuting | Unsuccessful StateSet or any other command | OMX\_StateExecuting |
| OMX\_StateExecuting | Successful StateSet Command | OMX\_StatePaused |
| OMX\_StateExecuting | Successful StateSet Command | OMX\_StateIdle |
| OMX\_StatePaused | Unsuccessful StateSet or any other command | OMX\_StatePaused |
| OMX\_StatePaused | Successful StateSet Command | OMX\_StateExecuting |
| OMX\_StatePaused | Successful StateSet Command | OMX\_StateIdle |
| OMX\_StateWaitForResources | Unsuccessful StateSet | OMX\_StateWaitForResources |
| OMX\_StateWaitForResources | Successful StateSet Command | OMX\_StateLoaded |
| OMX\_StateWaitForResources | Successful Allocation of resources by resource manager. | OMX\_StateIdle |

## ADD transition to StateInvalid from all states

Add error code to indicate already in state.

## OMX Events & Callbacks

OMX IL Events are defined in OMX\_Core.h. There are four types of events:

EventCmdComplete, EventError, EventMark, and EventPortSettingsChanged. Beyond these four events there are two buffer handling callbacks: FillBufferDone and EmptyBufferDone.

## IL API contents

### Basic Data Types

A compliant component will export functions and structures that use the data types shown in Table 4 or structures and type definitions derived from these data types.

|  |  |  |  |
| --- | --- | --- | --- |
| Table 4 - OMX Data Types | | | |
| Type Name | Data Size | Data Type | Data Alignment |
| OMX\_U8 | 8 bit | unsigned | Byte aligned |
| OMX\_S8 | 8 bit | signed | Byte aligned |
| OMX\_U16 | 16 bit | unsigned | 16 bit word aligned |
| OMX\_S16 | 16 bit | signed | 16 bit word aligned |
| OMX\_U32 | 32 bit | unsigned | 32 bit word aligned |
| OMX\_S32 | 32 bit | signed | 32 bit word aligned |
| OMX\_BOOL | 32 bit | Boolean (FALSE=0 and TRUE=1) | 32 bit word aligned |
| OMX\_Handle | 32 bit | pointer (e.g. void\*) | 32 bit word aligned |
|  |  |  |  |

### OMX State Enumeration

A compliant component shall use the OMX\_Core.h header file which provides the OMX State Enumeration shown in Table 5 to communicate state information.

|  |
| --- |
| Table 5 – OMX State Enumeration |
| typedef enum OMX\_STATETYPE {  OMX\_StateInvalid,  OMX\_StateLoaded,  OMX\_StateIdle,  OMX\_StateExecuting,  OMX\_StatePaused,  OMX\_StateWaitForResources  } OMX\_STATETYPE; |

### OMX Errors

OMX IL Errors are defined in the file OMX\_Core.h. The value assigned to OMX\_NoError is zero, and the other errors are in the range of 0x8001XXXX. Vendors are permitted to add vendor specific errors for their component in the range of 0x9001XXXX, so any error handling routines must take into account errors outside of the minimum error range defined in the OMX\_Errors.h file.

### Data Structures

The compliant component shall use the OMX\_Handle defined in OMX\_Core.h as the basis of all operations. The GetParam, SetParam, SetConfig, and GetConfig methods will access all instance specific structures via an index. Standard configuration structures are specified in the corresponding audio.h, video.h, image.h, other.h and ivcommon.h header files. Vendor specific structures must be tested according to provided documentation.

# IL API Base Profile Conformance Tests

This section of the document defines the testing to be done for a candidate IL component to determine if the component is base profile conformant with the OpenMAX IL specification. All applicable tests must pass for the component to be considered base profile conformant. For example, for an audio sink component, there are no outputs and tests related to the outputs are not required to be run. The tester may combine or rearrange tests as required to efficiently complete the testing. Many of the tests are very similar and one actual test may satisfy more than one of the requirements listed below.

Change instantiate and initialize to load and idle.

The diagram below shows the conformance test architecture. Each of the three test components(Reference core, IL client application, and tunnel test component) may be implemented within a single process to provide a single time stamped log of activity across each interface. The source code for these test components must be published.



## Component Version Test

The component shall be loaded. The GetComponentVersion method shall be called and the results recorded in a log file. The returned results should be consistent with the specification.

## Port Parameter Test

Add test for SetParam when in Executing, Paused or Idle to fail unless port is stopped and is port specific.

### Failed SetParam Test

The component shall be loaded and idled. A Setparam call shall be made with the appropriate error logged.

### Required Parameter Test

The component shall be loaded. The component will be tested against the four parameter index’s OMX\_IndexParamAudioInit, OMX\_IndexParamImageInit, OMX\_IndexParamVideoInit, and OMX\_IndexParamOtherInit. At least one of these 4 parameter index’s SHALL be supported.

The parameter index OMX\_PRIORITYMGMTTYPE is not tested against on base profile, but is required for interop profile.

Each defined parameter index that is supported with the corresponding structure will be passed to OMX\_GetParameter. If any failures occur they will be logged.

### Supported Parameter Test

The component shall be loaded. Each defined parameter index with the corresponding structure will be passed to OMX\_GetParameter to determine which parameters are supported by the component. A log of all parameters will be reported indicating if the parameter is supported by the component or not.

### Successful Default Parameters Test

The component shall be loaded. Each parameter supported by the component will be queried using OMX\_GetParameter to populate default values into the parameter structure. The test will validate that the component accepts it’s own default populated values via OMX\_SetParameter. Any failures will be logged indicating the parameter index that failed, with the failure return code.

### Unsuccessful Default Parameters Test

The component shall be loaded. Each parameter supported by the component will be queried using OMX\_GetParameter. The test will explicitly program the nVersion field incorrectly calling OMX\_GetParameter and OMX\_SetParameter expecting the component to detect the failure returning OMX\_ErrorVersionMismatch.

The test will also validate that the component can detect invalid port index values for parameter index’s that apply to ports. The test will explicitly program the nPortIndex field of the incorrectly calling OMX\_GetParameter, and OMX\_SetParameter, expecting the component to detect the failure returning OMX\_ErrorBadPortIndex.

### Required Configuration Test

The component shall be loaded. Each defined configuration index that \*MUST\* be supported with the corresponding structure will be passed to OMX\_GetConfig. If any of the required configurations are not supported the failure will be logged.

### Supported Configurations Test

The component shall be loaded. Each defined configuration index with the corresponding structure will be passed to OMX\_GetConfig to determine which configurations are supported by the component. A log of all configurations will be reported indicating if the configuration was supported or not.

### Successful Default Configuration Test

The component shall be loaded. Each configuration supported by the component will be queried using OMX\_GetConfig to populate default values into the configuration structure. The test will validate that the component accepts it’s own default populated values via OMX\_SetConfig. Any failures will be logged indicating the configuration index that failed, with the failure return code.

### Unsuccessful Default Configuration Test

The component shall be loaded. Each configuration supported by the component will be queried using OMX\_GetConfig. The test will explicitly program the nVersion field incorrectly calling OMX\_GetConfig and OMX\_SetConfig expecting the component to detect the failure returning OMX\_ErrorVersionMismatch.

The test will also validate that the component can detect invalid port index values for configuration index’s that apply to ports. The test will explicitly program the nPortIndex field of the incorrectly calling OMX\_GetConfig, and OMX\_SetConfig, expecting the component to detect the failure returning OMX\_ErrorBadPortIndex.

## Resource Exhaustion Test

### Component Initialization Failure(insufficient resources)

Multiple instances of the component will be loaded and idled until the component exhausts resources and the final attempted instance reports insufficient resources when attempting transition to idle. (a hard coded limit may need to be set to enable this test).

### Successful De-allocation(resource constrained)

The component shall be repeatedly loaded and idled until resources are exhausted. The last instance successfully idled will then be de-allocated using the StateSet(LOADED) command. The instance previously failing to idle should now successfully idle. The failing and successfully idled components should be alternately idled and de-idled(loaded) to ensure memory leaks are not occurring at the resource exhaustion boundary 100 times with results logged.

.

## Buffer Test

### Port populated

The component shall be commanded to the idle state, with no buffers allocated on any ports. The test will validate that the component does not transition to the idle state within a certain timeout. The test will then allocate all buffers but one, on each port, and validate again that the component does not transition to idle within a certain timeout. One more buffer on each port shall be allocated. The test will expect the component to successfully transition to the idle state. The test will also validate that each port reports it is populated via the port definition structure. Any error will be logged.

### Port unpopulated

The component shall be commanded to the idle state, with buffers allocated for all ports. The test will command the component to the loaded state, and validate the component does not transition to the idle state within the timeout. All buffers but one will be freed on each port, and the test will validate that the component does not transition to the loaded state within a certain timeout. On each port, the remaining allocated buffer will be freed. Once all buffers are freed, the test will validate the component successfully transitioned to the loaded state, and that each port reports it is not populated via the port definition structure.

### Per-Port Enable / Disable

All ports on the component shall be disabled. The component shall be commanded to the idle state. The test sequence described below is applied to each port, one at a time.

The test shall validate that the port reports it is not enabled, and not populated via the port definition structure. The port shall be commanded to enabled. The test will validate that the port does not report the command has completed, as no buffers have been allocated on the port. The test will further validate that the port reports it is enabled, but not populated via the port definition structure. The required buffers are allocated on the port. The test will validate that the component reports the command to enable the port has completed. The test will further validate that the port reports it is enabled, and populated via the port definition structure.

The test shall then free all buffers on the port. The test will validate that the component reports error OMX\_ErrorPortUnpopulated to the EventHandler. The test will then command the port to disable, and validate that the port reports the command completed. The test will further validate that the port reports it is not enabled, and not populated via the port definition structure.

.

## Port Communication Test

### Non-Tunneling Setup Test

The component shall be loaded. OMX\_SetupTunnel will be called for each input port, with a NULL component handle for the output port, and visa-versa to confirm that the component supports setup of non-tunneling. Any failures reported by OMX\_SetuipTunnel are logged.

### Transmit Buffers Between Input and Output Ports and Application

The component shall be loaded and then the component shall be idled and started in a non-tunneling environment and the test application should perform normal buffer processing for the component for several seconds. The component vendor shall publish a test file to be used as input to the component which shall be fed to the component in varying payloads of full, half and quarter buffer. The component will have each of its ports exercised with client and component buffer allocation. Repeat this test for a representative set of supported standard input data formats documenting results.

### Port Repack Test

The component shall be loaded, idled, and executed. The IL client shall send a series of buffers with a single byte payload until all buffers have been sent to the component. The component must return the first buffer in the pool to be sent(requiring the data in this buffer to be moved to an internal buffer for repacking) – the IL client shall then send full payloads of data until the entire buffer pool is circulated. The experiment will then be repeated one more time with results logged.

### Port Stop and Restart Test

The component shall be loaded and then the component shall be idled and started and the test application should perform normal buffer processing for the component for several seconds. The component port shall be stopped by sending the PortStop command. The test application should continue processing buffers and making calls to FillThisBuffer or EmptyThisBuffer until all buffers are returned to the IL client and ensure that no new EmptyBufferDone or FillBufferDone callbacks are issued. Subsequent EmptyThisBuffer and FillThisBuffer calls to the port return errors. The port should subsequently be restarted to ensure normal operation with results logged.

### Component Pause and Resume Test

The component shall be loaded and then the component shall be idled and started and the test application should perform normal buffer processing for the component for several seconds. The component shall be paused by sending the StateSet(PAUSE) command. The test application should continue processing buffers and making FillThisBuffer and EmptyThisBuffer until all buffers are returned to the component and it should verify that no new FillBufferDone or EmptyBufferDone callbacks are issued. Next verify that the OMX\_EventCmdComplete event is sent by the component and verify that the component has entered the Paused state. The component should then be resumed by entering the executing state with anticipated normal behavior logged.

## State Transition Tests

The component shall be loaded to execute a series of state transition tests. The table below describes the state transitions, and expected behavior for the component on each state transition.

There are 6 defined states in OMX\_STATETYPE, and 1 implicit unloaded state, for a total of 49 state transition tests reflected in the table.

The left column of the table represents the state of the component, and the top row of the table lists the state set via OMX\_SendCommand (for convenience of the document the state names have been abbreviated). The expected result from the test is entered in the cell for the table. Any tests which do not result in the expected result are logged.

In transitioning from loaded to idle, the component must ensure that buffer allocation calls to each port are first made.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Unloaded | Loaded | WaitFor  Resources | Idle | Executing | Paused | Invalid |
| Unloaded | N/A | OK | N/A | N/A | N/A | N/A | N/A |
| Loaded | OK | SameState | OK | OK | Invalid | Invalid | OK |
| WaitForResources | OK | OK | SameState | Ok | Invalid | Invalid | OK |
| Idle | OK | OK | Invalid | SameState | OK | OK | OK |
| Executing | OK | Invalid | Invalid | OK | SameState | OK | OK |
| Paused | OK | Invalid | Invalid | OK | OK | SameState | OK |
| Invalid | OK | Invalid | Invalid | Invalid | Invalid | Invalid | SameState |

## Flush Tests

### Successfully Flush Component Buffers

The component shall be loaded and then the component shall be idled and paused with the maximum supported number of buffers of minimum specified size for each input port. These buffers will all be queued and the flush command for each port will be sent – the results will be logged and all of the buffers should be returned without processing in the order in which they were initially sent to the component. The test will also validate that the component successful returned OMX\_EventCmdComplete via the EventHandler()callback when all buffers have been returned to the IL Client.

The test will be repeated with a single flush command to flush all ports.

## Base Multi Thread Test

These tests are not applicable to single threaded environments.

### Buffer Processing in a separate Thread

The component shall be loaded and then the component shall be idled and started in a non-tunneling environment and the test application should perform normal buffer processing for the component for several seconds. The Conformance Test App for this test shall handle the buffers in a thread separate from the initialization/setup processing. Verify that the buffers are handled correctly and the correct data is output by the component. (multithreaded environment only)

### Initialization from 3 threads

The component shall be loaded from one thread, then idled from a second thread and finally started (transitioned to OMX\_StateExecuting) in a non-tunneling environment from a third thread. The test application should perform normal buffer processing for the component for several seconds. The Application Framework for this test shall handle the buffers in a thread separate from the initialization/setup processing. Verify that the buffers are handled correctly and the correct data is output by the component. . (multithreaded environment only)

### Start and Stop in different Threads

The component shall be loaded and then the component shall be idled and started in a non-tunneling environment and the test application should perform normal buffer processing for the component for several seconds. The Application should start a new thread and use that to stop the component using the StateSet(IDLE) command. Verify that the component stops correctly.

### Unexpected Stop from another Thread Test

The component shall be loaded and started in a non-tunneling environment and the test application should perform normal buffer processing. The Application Framework for this test shall handle the buffers normally. While buffers are being processed, a second thread shall issue the StateSet(IDLE) command to the component. The component shall respond to the FillThisBuffer command for output ports and to the EmptyThisBuffer command for input ports with an appropriate error message and shall not segfault or gpf.

De-initialization Tests

### Multiple Init/Deinit case

The component shall be loaded and then the component shall be idled and started in a non-tunneling environment and the test application should perform normal buffer processing for the component for several seconds. Stop the component using the StateSet(IDLE) command and de-initialize the component using the StateSet(LOADED) command, but do not unload the component. Now, reinitialize and start the component and begin buffer processing for several seconds. Repeat the sequence 10 times. Verify the component properly processes buffers and returns correctly processed data each time it is started and does not return any error messages.

## Buffer Flag Tests

### Mark Buffer Propagation

A marked buffer is provided to a component that is not designated to generate an event. The component passes the mark from input to output ports.

### Demonstrate Mark Buffer Event and Clearing

A marked buffer is provided to a component that is designated to generate an event. The component generates the event and clears the mark.

### Demonstrate Mark Buffer For Previously Marked Buffers

A marked buffer is provided to a component that is not designated to generate the associated event. The component is commanded to mark its buffer – but chooses to mark the subsequent buffer so as not to overwrite the incoming marked buffer.

### Demonstrate Mark Buffer For Previously Marked Buffers for EOS condition

For this case an additional empty output port buffer will be sent with a mark after the EOS buffer.

### Demonstrate EOS Functionality

The IL client will first load and idle the component in a non-tunneled environment to the tunnel test component. The component will be sent an EOS marked buffer and the expected event should be received. Should the component contain an output port, the EOS is propagated from at least one output port of the component.

## Component Naming Convention

Each publicly accessible component name shall be prefaced in accordance with the specified rules and logged.

# Interop Profile Tests

The tests in this section are to be conducted in addition to the base profile tests for a component to claim interop profile conformance. Components implemented on an environment without threads are exempt from the Multi-threaded test. The Clock Component test is only applicable to components claiming to be a clock component implementation; all other components are exempt from this test. The Seeking test is only applicable to components that claim to be a seeking component; all other components are exempt from this test.

The conformance candidate shall be denoted hereafter as the component under test or simply by the acronym CUT. The portion of the test harness that connects to the CUT shall be denoted hereafter as the tunnel test component or simply by the acronym TTC.

## Tunneling Tests

#### Valid Input/Output Port Tunnel Setup

This test loads the component under test and the tunnel test component then sets up tunnels between each CUT port (for all domains, both input and output ports) to a corresponding TTC port. The test then transitions both the CUT and the TTC to the OMX\_StateIdle state. It also logs any errors. This test ensures that each CUT port can correctly setup a tunnel with a correctly behaving compatible port on another component.

#### Invalid Input/Output Port Tunnel Setup

This test loads the component under test and the tunnel test component. It then commands the TTC to misbehave in various ways and, for each invalid behavior, attempts to set up tunnels between each CUT port (for all domains, both input and output ports) to a corresponding TTC port. Invalid behaviors include:

* The TTC port defines an invalid port format
* The TTC port doesn’t support the OMX\_IndexParmCompBufferSuppler index
* The TTC port doesn’t support the UseBuffer method

This test ensures that each CUT port will fail to establish a tunnel port with an incorrectly behaving compatible port on another component.

#### Port Buffer Supplier Test

This test loads the component under test and the tunnel test component then verifies correct buffer supplier functionality against each input or output port reported by the component for all domains. The test component provides a compatible TTC port for tunneling with each CUT port. The test component confirms that the required buffers are allocated when the component transitions from loaded to idle then confirms that all allocated buffers are freed when the component transitions from idle to loaded.

## Incomplete Stop Failure

This test loads the component under test and the tunnel test components, tunnels all CUT ports to compatible TTC ports, and transitions both components into the OMX\_StateIdle state and then into the OMX\_StateExecuting state. After waiting for a short interval of normal buffer processing, the test commands both components to “stop” via a transition to the OMX\_StateIdle state but instructs the TTC not to return buffers to the CUT. The test then verifies that, in the absence of returned buffers, the CUT does not complete the “stop” (i.e. the transition to the OMX\_StateIdle state). After confirming this, the test next instructs the TTC to return the buffers and then verifies that, given all its buffers, the CUT does the “stop”, specifically the CUT completes the transition to the OMX\_StateIdle state.

## Demonstrate Port Stop and Restart

This test loads the component under test and the tunnel test components, tunnels all CUT ports to compatible TTC ports, and transitions both components into the OMX\_StateIdle state and then into the OMX\_StateExecuting state. After waiting for a short interval of normal buffer processing, the test then disables each CUT port. The test then destroys the TTC instantiation, creates a new TTC instantion, and connects this instantiation to original instantiation of the CUT. Finally the test re-enables each CUT port and verifies normal buffer processing. This process is repeated for two buffer supplier cases. In the first case the test forces all CUT ports to be buffer suppliers. In the second case the test forces all TTC ports to be buffer suppliers.

## Minimum Buffer Size for Standard Port

This test loads the component under test and the tunnel test components, tunnels all CUT ports to compatible TTC ports, and transitions both components into the OMX\_StateIdle state and then into the OMX\_StateExecuting state. On any tunnel where minimum buffer payload rules are applicable, the TTC port either emits the minimum payload (if it is the output port) or verifies the minimum size on incoming payloads (if it is the input port).

## Multithreaded components tunneled test

This test loads the component under test from one thread and the tunnel test component from another. It then tunnels all CUT ports to compatible TTC ports and transitions both components into the OMX\_StateIdle state. The test verifies and logs anticipated communication.

## Resource Manager Tests

### Resource Preemption

This test repeatedly loads instances of the component under test (using a low priority) and transitions them to the OMX\_StateIdle state until an insufficient resources error is returned. The test then increases the priority of the last CUT instance and retries the transition to the OMX\_StateIdle state. This must result in the deinitialization of a previously initiated low priority component (i.e. its transition from the OMX\_StateIdle state to the OMX\_StateLoaded) and successful initialization of the higher priority component (i.e. its transition from the OMX\_StateLoaded state to the OMX\_StateIdle). All transitions must be accompanied by the corresponding state transition and error events.

### Wait For Resources

This test repeatedly loads instances of the component under test (using a low priority) and transitions them to the OMX\_StateIdle state an insufficient resources error is returned. The test then transitions the failing instance into the OMX\_StateWaitForResources state. The test transitions a CUT instance that successfully transitioned to OMX\_StateIdle to OMX\_StateLoaded. The resource manager must subsequently transition the component waiting for resources to the OMX\_StateIdle state and this component must emit the OMX\_EventCmdComplete event to indicate such a transition.

## Clock Component Test

This test applies only to a clock component implementation; all other components are exempt. Because it tests for internal semantics (i.e. that the component under test performs certain work) not simply interface semantics (i.e. that the CUT obeys the generalized interfaces for configuration, state management, and data exchange), the clock component test is unlike most other tests.

This test loads the component under test and the tunnel test component, established tunnels between the CUT and the TTC ports, then verifies behavior expected of a clock component test. This includes:

* Verification that the CUT supports the configs required of a clock component. The test accomplishes this by attempting to set and get the relevant configs.
* Verification that the CUT obeys the behavior mandated for clock states. The test accomplishes this by commanding transitions between pairs of clock states under various circumstances and observing resulting behavior.
* Verification of support for rate control. The test accomplishes this by modifying the media clock scale and verifying media time requests are fulfilled or not fulfilled appropriately.

## Seeking Test

This test applies only to a component that claims to implement seeking; all other components are exempt. Because it tests for internal semantics (i.e. that the component under test performs certain work) not simply interface semantics (i.e. that the CUT obeys the generalized interfaces for configuration, state management, and data exchange), the seeking tests is unlike most other tests.

This test loads the component under test and the tunnel test component, established tunnels between the CUT and the TTC ports, then verifies behavior expected of a clock component test. This includes:

* Verification that the CUT supports the configs required of a component that supports seeking. The test accomplishes this by attempting to set and get the relevant configs.
* Verification that the CUT obeys the behavior mandated for a component that supports seeking. The test accomplishes this by performing a “seek” (i.e. setting the CUT’s media position) then watching for OMX\_BUFFERFLAG\_STARTTIME flag on one of the buffers emit by the CUT.

# Standard Component Conformance Tests

This section of the document defines the tests performed against an IL component to determine if the component is a standard component as defined in OpenMAX IL 1.1.2 specification. All applicable tests must pass for the component to be considered a standard IL component.

The diagram below shows the conformance test architecture. Each of the three test entities (IL core, IL client application, and tunnel test component) may be implemented within a single process to provide a single time stamped log of activity across each interface. The source code for these test objects WILL be published.

When testing a component which consumes input, the adopter provides a one input file and one “buffer variance” file per input port. We hardcode the input filenames and buffer variance filenames used to test the standard roles. This frees the test harness and the user of the burden of specifying them (they are extremely numerous). The table below defines the particular filenames per standard component but they are formatted in general in the following way.

* input data file for default setting:

<role\_name>\_<port\_specifier>def.bin

* buffer variance file for default setting:

<role\_name>\_<port\_specifier>def.txt

* input data file for minimum setting:

<role\_name>\_<port\_specifier>min.bin

* buffer variance file for minimum setting:

<role\_name>\_<port\_specifier>min.txt

* input data file for maximum setting:

<role\_name>\_<port\_specifier>max.bin

* buffer variance file for maximum setting:

<role\_name>\_<port\_specifier>max.txt

Where:

* <role\_name> corresponds to the name of the role the file is associated with.
* <port\_specifier> consists of a single character prefix to indicate domain (a=audio, v=video, i=image, o=other) followed by the index (relative to nStartIndex for the particular domain) of the port. Example: “a0” denotes the port at audio port base + 0.
* The existence of a pair of “def” files implies the default setting will be tested.
* The existence of a pair of “min” files implies the minimum setting will be tested.
* The existence of a pair of “max” files implies maximum value setting will be tested

Thus the submitter supplies input/variance file pairs for every port of every role tested. The “buffer variance” is an ASCII file specifying:

1. the port associated with the file
2. the number of parameters to be varied
3. the parameter containing the field to be varied
4. the field to be varied
5. the value of said field
6. the payload sizes of the buffers the test architecture will feed to the *n*th input port, e.g. one integer per line such that the integer on line *m* is the payload size of the *m*th buffer the test architecture will feed to the corresponding input port. These files are subject to peer review along with test logs.

The number of times that items 3-5 will appear in the buffer variance file is dependent on the number of paramters to be varied (item 2).

In the case of a video decoder component’s buffer variance file, 0 parameters need to be modified. An example of a buffer variance file is:

v0

0

5369

1914

1926

…

In the case of a Real Audio audio component’s buffer various file, 5 parameters need to be modified. An example of a buffer variance file is:

a0

5

OMX\_AUDIO\_PARAM\_RATYPE

nSamplingRate

8000

OMX\_AUDIO\_PARAM\_RATYPE

nChannels

1

OMX\_AUDIO\_PARAM\_RATYPE

nBitsPerFrame

256

OMX\_AUDIO\_PARAM\_RATYPE

nSamplePerFrame

256

OMX\_AUDIO\_PARAM\_RATYPE

nNumRegions

12

128

110

256

112

112

...

The standard component conformance test suite consists of two tests: “the standard component interface test” and the “standard component data metabolism test”. Both tests are run on every candidate component implementation.



## 

## Standard Component Interface Test

The component shall be loaded. The component SHALL be queried to determine which roles it supports. For each role a component implementation claims support for this test SHALL verify:

* The component implementation supports all the mandated input and outputs ports of the specified domains.
* For each said port, the component implementation supports all the mandated configs and parameters.
* For each said config or parameter, the component implementation supports all the mandated default and optional settings.

The OpenMax IL 1.1.2 spec explicitly defines the above criteria in the context of each standard component. Results will be logged.

## Standard Component Data Metabolism Test

This test shall validate a standard component implementation’s ability to metabolize a representative set of input and/or produce a representative set of output. Sources must demonstrate an ability to produce prescribe output, sinks to consume prescribed input, and transform components to do both.

We define a set of representative inputs or outputs for each standard component by identifying a distinguishing setting (e.g. sampling rate, bitrate, etc) and mandating that an implementation handle either the maximum and minimum values or the default of that distinguishing setting (as defined by the OpenMax IL 1.1.2 specification).

The test harness uses hardcoded filenames (see description above) to map one input file and one buffer variance file per input port (e.g. “APB+0”) per role (e.g. “audio\_decoder.mp3”) per min/max/default value of the distinguishing setting (e.g. a sampling rate of 44.1KHz). Furthermore the adopter may optionally use the test harness to map output files to output ports. If output files are mapped then the metabolism test will write corresponding output to the specified files.

The sequence of steps is as follows:

The component WILL be loaded. The component WILL be queried to determine which roles it supports. For each appropriate distinguishing setting value (min/max/default) for each role reported the test WILL:

1. Set the component to said role
2. Populate the buffers of each port
3. Transition the component to idle, then executing
4. Feed each input port (if any) data using the appropriate input file and buffer variance file. For the *n*th buffer of input the test WILL:
   1. Read the *n*th number in the buffer variance file and interpret this as the payload size *s*
   2. Read the next *s* bytes from the input file into the payload on the *n*th buffer
   3. Send the *n*th buffer to the input port

The test WILL repeat this for all integers in the buffer variance file in the order those integers occur and reading from the input file sequentially.

1. Transition the component to idle then loaded in preparation for any subsequent role testing.

Each standard component will have a number of input test stream files and corresponding buffer variance files as specified in this document. The submitter supplies only those files required to perform tests on the submitted components. These input test stream files and corresponding buffer variance files must be posted along with test logs to the OpenMAX IL working group for review.

### Per Role Requirements

The table below defines the exact requirements for the metabolism test per role tested. The table lists:

* the name of the role (as specified in the API documentation)
* the names of the required input and variance files
  + by implication this also defines what values of the varying setting are tested (min/max/default)
  + “n.a.” denotes not applicable. In these cases no metabolism test will be run on the given role.
* the varied setting including:
  + Which port the setting applies to
  + The parameter of the setting
  + The field of the setting within the parameter
  + “n.a” denotes not applicable. In these cases either the metabolism test will not be run (i.e. filenames
* Actual min/max/default values are defined by the OpenMax IL 1.1.2 specification

|  |  |  |
| --- | --- | --- |
| **Role** | **Input file names** | **Varied setting** |
| audio\_decoder.acc | audio\_decoder\_aac\_a0min.bin  audio\_decoder\_aac\_a0max.bin  audio\_decoder\_acc\_a0min.txt  audio\_decoder\_acc\_a0max.txt | A0:OMX\_AUDIO\_PARAM\_AACPROFILETYPE.nSampleRate |
| audio\_decoder.amrnb | audio\_decoder\_amrnb \_a0min.bin  audio\_decoder\_amrnb\_a0max.bin  audio\_decoder\_amrnb\_a0min.txt  audio\_decoder\_amrnb\_a0max.txt | A0:OMX\_AUDIO\_PARAM\_AMRTYPE.nBitRate |
| audio\_decoder.amrwb | audio\_decoder\_amrwb\_a0min.bin  audio\_decoder\_amrwb\_a0max.bin  audio\_decoder\_amrwb\_a0min.txt audio\_decoder\_amrwb\_a0max.txt | A0:OMX\_AUDIO\_PARAM\_AMRTYPE.nBitRate |
| audio\_decoder.mp3 | audio\_decoder\_mp3\_a0min.bin  audio\_decoder\_mp3\_a0max.bin  audio\_decoder\_mp3\_a0min.txt audio\_decoder\_mp3\_a0max.txt | A0:OMX\_AUDIO\_PARAM\_M3TYPE.nSampleRate |
| audio\_decoder.ra | audio\_decoder\_ra\_a0min.bin  audio\_decoder\_ra\_a0max.bin  audio\_decoder\_ra\_a0min.txt audio\_decoder\_ra\_a0max.txt | A0:OMX\_AUDIO\_PARAM\_RATYPE.nSampleRate  Note: In order to vary the Sample Rate, Real Audio requires that additional parameters be varied also based on the value of the sample rate.  The parameters are:  OMX\_AUDIO\_PARAM.RATYPE.nChannels  OMX\_AUDIO\_PARAM\_RATYPE.nBitsPerFrame  OMX\_AUDIO\_PARAM\_RATYPE.nSamplesPerFrame  OMX\_AUDIO\_PARAM\_RATYPE.nNumRegions |
| audio\_decoder.wma | audio\_decoder\_wma\_a0min.bin  audio\_decoder\_wma\_a0max.bin  audio\_decoder\_wma\_a0min.txt  audio\_decoder\_wma\_a0max.txt | A0:OMX\_AUDIO\_PARAM\_WMATYPE.nSampleRate |
| audio\_encoder.aac | audio\_encoder\_aac\_a0min.bin  audio\_encoder\_aac\_a0max.bin  audio\_decoder\_acc\_a0min.txt  audio\_encoder\_acc\_a0max.txt | A0:OMX\_AUDIO\_PARAM\_AACTYPE.nSampleRate |
| audio\_endoder.amrnb | audio\_encoder\_amrnb\_a0min.bin  audio\_encoder\_amrnb\_a0max.bin  audio\_decoder\_amrnb\_a0min.txt  audio\_encoder\_amrnb\_a0max.txt | A0:OMX\_AUDIO\_PARAM\_AMRTYPE.nBitRate |
| audio\_encoder.amrwb | audio\_encoder\_amrwb\_a0min.bin  audio\_encoder\_amrwb\_a0max.bin  audio\_decoder\_amrwb\_a0min.txt audio\_encoder\_amrwb\_a0max.txt | A0:OMX\_AUDIO\_PARAM\_AMRTYPE.nBitRate |
| audio\_mixer*N*.pcm | audio\_mixer\_pcm\_a1min.bin  audio\_mixer\_pcm\_a1max.bin  audio\_mixer\_pcm\_a1min.txt  audio\_mixer\_pcm\_a1max.txt  …  audio\_mixer\_pcm\_a*N*min.bin  audio\_mixer\_pcm\_a*N*max.bin  audio\_mixer\_pcm\_a*N*min.txt  audio\_mixer\_pcm\_a*N*max.txt | A1-AN: OMX\_AUDIO\_PARAM\_PCMTYPE.nSampleRate |
| audio\_reader.binary | n.a. | n.a. |
| audio\_renderer.pcm | audio\_renderer\_pcm\_a0min.bin  audio\_renderer\_pcm\_a0max.bin  audio\_renderer\_pcm\_a0min.txt  audio\_renderer\_pcm\_a0max.txt | A0: OMX\_AUDIO\_PCMTYPE.nSampleRate |
| audio\_writer.bin | n.a. | n.a. |
| audio\_capturer.pcm | n.a. | n.a. |
| audio\_processor.pcm.  stereo\_widening\_loudspeakers | audio\_processor\_pcm\_sw\_ls\_a0min.bin  audio\_processor\_pcm\_sw\_lw\_a0max.bin  audio\_processor\_pcm\_sw\_ls\_a0min.txt  audio\_processor\_pcm\_sw\_ls\_a0max.txt | A0: OMX\_AUDIO\_PCMTYPE.nSampleRate |
| audio\_processor.pcm.  stereo\_widening\_headphones | audio\_processor\_pcm\_sw\_hp\_a0min.bin  audio\_processor\_pcm\_sw\_hp\_a0max.bin  audio\_processor\_pcm\_sw\_hp\_a0min.txt  audio\_processor\_pcm\_sw\_hp\_a0max.txt | A0: OMX\_AUDIO\_PCMTYPE.nSampleRate |
| audio\_processor.pcm.  reverberation | audio\_processor\_pcm\_reverb\_a0min.bin  audio\_processor\_pcm\_reverb\_ a0max.bin  audio\_processor\_pcm\_reverb\_a0min.txt  audio\_processor\_pcm\_reverb\_a0max.txt | A0: OMX\_AUDIO\_PCMTYPE.nSampleRate |
| audio\_processor.pcm.  chorus | audio\_processor\_pcm\_chorus \_a0min.bin  audio\_processor\_pcm\_chorus\_a0max.bin  audio\_processor\_pcm\_chorus\_a0min.txt  audio\_processor\_pcm\_chorus \_a0max.txt | A0: OMX\_AUDIO\_PCMTYPE.nSampleRate |
| audio\_processor.pcm.  equalizerN | audio\_processor\_pcm\_eq\_a0min.bin  audio\_processor\_pcm\_eq\_a0max.bin  audio\_processor\_pcm\_eq\_a0min.txt  audio\_processor\_pcm\_eq \_a0max.txt | A0: OMX\_AUDIO\_PCMTYPE.nSampleRate |
| image\_decoder.jpeg | image\_decoder\_jpeg\_i0def.bin  image\_decoder\_jpeg\_i0def.txt | n.a. |
| image\_encoder.jpeg | image\_encoder\_jpeg\_i0def.bin  image\_encoder\_jpeg\_i0def.txt | n.a. |
| image\_reader.binary | n.a. | n.a. |
| image\_writer.binary | image\_writer\_binary\_i0def.bin  image\_writer\_binary\_i0def.txt | n.a. |
| video\_decoder.h263 | video\_decoder\_h263\_v0def.bin  video\_decoder\_h263\_v0def.txt | n.a. |
| video\_decoder.avc | video\_decoder\_avc\_v0def.bin  video\_decoder\_avc\_v0def.txt | n.a. |
| video\_decoder.mpeg4 | video\_decoder\_mpeg4\_v0def.bin  video\_decoder\_mpeg4\_v0def.txt | n.a. |
| video\_decoder.rv | video\_decoder\_rv\_v0def.bin  video\_decoder\_rv\_v0def.txt | n.a. |
| video\_decoder.wmv | video\_decoder\_wmv\_v0min.bin  video\_decoder\_wmv\_v0min.txt | n.a. |
| video\_encoder.h263 | video\_encoder\_h263\_v0min.bin  video\_encoder\_h263\_v0min.txt | n.a. |
| video\_encoder.avc | video\_encoder\_avc\_v0min.bin  video\_encoder\_avc\_v0min.txt | n.a. |
| video\_encoder.mpeg4 | video\_encoder\_mpeg4\_v0min.bin  video\_encoder\_mpeg4\_v0min.txt | n.a. |
| video\_reader.binary | not applicable (source component) | n.a. |
| video\_scheduler.binary | video\_scheduler\_binary\_v0.bin  video\_scheduler\_binary\_v0.txt | n.a. |
| video\_writer.binary | video\_writer\_binary\_v0def.bin  video\_writer\_binary\_v0def.txt | n.a. |
| camera.yuv | not applicable (source component) | n.a. |
| clockN.binary | n.a. | n.a. |
| container\_demux.3gp | n.a. | n.a. |
| container\_demux.asf | n.a. | n.a. |
| container\_demux.real | n.a. | n.a. |
| container\_mux.3gp | n.a. | n.a. |
| iv\_procesor.yuv | iv\_processor\_yuv\_v0def.bin  iv\_processor\_yuv\_v0def.txt | n.a. |
| iv\_renderer.yuv.overlay | iv\_renderer\_yuv\_overlay\_v0def.bin  iv\_renderer\_yuv\_overlay\_v0def.txt | n.a. |
| iv\_renderer.yuv.blter | iv\_renderer\_yuv\_blter\_v0def.bin  iv\_renderer\_yuv\_blter \_v0def.txt | n.a. |
| iv\_renderer.rgb.overlay | iv\_renderer\_rgb\_overlay\_v0def.bin  iv\_renderer\_rgb\_overlay\_v0def.txt | n.a. |
| iv\_renderer.rgb.blter | iv\_renderer\_rgb\_blter\_v0def.bin  iv\_renderer\_rgb\_blter\_v0def.txt | n.a. |

# Leveraging Buffer Variance File for Interop and Base Profile Tests

OpenMax IL 1.1.2 increments the Interop and Base Profile tests (as defined for 1.0) to allow the tester to optionally specify the length of the payload of input data for tests that consume input data. To accomplish this the tests leverage the buffer variance file employed by the Data Metabolism test. This appendix describes specifically how this is accomplished.

Test input reading has been modified to accommodate a .length file along with the input data file. The .length file is in the same format as the buffer variance file. If the .length file exists, then the buffer lengths from this file will be used for subsequent calls to OMX\_OSAL\_ReadFromInputFileWithSize(). If the .length file does not exist, then OMX\_OSAL\_ReadFromInputFileWithSize() behaves as before, with the parameter nMaxBytes indicating how many bytes to read.

For example:

We create an aac input stream called aac.stream. Along with this file, we create a new file called aac.stream.length, which is in the buffer variance file format. When OMX\_OSAL\_OpenInputFile() is called to open aac.stream, it will see if aac.stream.length exists. If aac.stream.length does exist, the .length file will be opened by calling OMX\_OSAL\_ReadBufferVarianceFile(), and the buffer lengths will be read in. All future calls to OMX\_OSAL\_ReadFromInputFileWithSize() will use the buffer lengths read in by OMX\_OSAL\_OpenInputFile().

And again, if the .length file does not exist, then OMX\_OSAL\_ReadBufferVarianceFile() is not called and there are no buffer lengths associated wit the input file.

These buffer variance files are formatting similar to the buffer various files for the standard component test’s buffer variance files.

For these buffer variance files, no parameters need to be modified so the files will indicate that there are 0 parameter entries.

Note: The first parameter needs typically would specify the Port Index, in the case of these tests this information is not used, it can be set to any value such as zero

An example of a buffer variance file is:

0

0

2

7

328

…

For this file, the first call to OMX\_OSAL\_ReadFromInputFileWithSize will return 2 bytes, the second 7 bytes, the third 328 bytes, and so on.