OpenMAX™ Integration Layer Application Programming Interface Specification

Version 1.2.0 Provisional

November 7, 2011
Document version 1.2.0.0

Contents

1 Overview

1.1 Introduction

This document details the Application Programming Interface (API) for the OpenMAX Integration Layer (IL). Developed as an open standard by The Khronos Group, the IL serves as a low-level interface for audio, video, and imaging components used in embedded and/or mobile devices. The principal goal of the IL is to give components a degree of system abstraction for the purpose of portability across operating systems and software stacks.

1.1.1 About the Khronos Group

The Khronos Group is a member-funded industry consortium focused on the creation of open standard APIs to enable the authoring and playback of dynamic media on a wide variety of platforms and devices. All Khronos members may contribute to the development of Khronos API specifications, may vote at various stages before public deployment, and may accelerate the delivery of their multimedia platforms and applications through early access to specification drafts and conformance tests. The Khronos Group is responsible for open APIs such as OpenGL ES, OpenML, and OpenVG.

1.1.2 A Brief History of OpenMAX

The OpenMAX set of APIs was originally conceived as a method of enabling portability of components and media applications throughout the mobile device landscape. Brought into the Khronos Group in mid-2004 by a handful of key mobile hardware companies, OpenMAX has gained the contributions of companies and institutions stretching the breadth of the multimedia field. As such, OpenMAX stands to unify the industry in taking steps toward media component portability. Stepping beyond mobile platforms, the general nature of the OpenMAX IL API makes it applicable to all media platforms.

1.2 The OpenMAX Integration Layer

The OpenMAX IL API strives to give media components portability across an array of platforms. The interface abstracts the hardware and software architecture in the system. Each component and relevant transform is encapsulated in a component interface. The OpenMAX IL API allows the user to load, control, connect, and unload the individual components. This flexible core architecture allows the Integration Layer to easily implement almost any media use case and mesh with existing graph-based media frameworks.

1.2.1 Key Features and Benefits

The OpenMAX IL API gives applications and media frameworks the ability to interface with multimedia codecs and supporting components (i.e., sources and sinks) in a unified

manner. The components themselves may be any combination of hardware or software and are completely transparent to the user. Without a standardized interface of this nature, component vendors have little alternative than to write to proprietary or closed interfaces to integrate into mobile devices. In this case, the portability of the component is minimal at best, costing many development-years of effort in re-tooling these solutions between systems.

Thus, the IL incorporates a specialized arsenal of features, honed to combat the problem of portability among many vastly different media systems. Such features include:

- A flexible component-based API core
- Ability to easily plug in new components
- Coverage of targeted domains (audio, video, and imaging) while remaining easily extensible by both the Khronos Group and individual vendors
- Capable of being implemented as either static or dynamic libraries
- Retention of key features and configuration options needed by parent software (such as media frameworks)
- Ease of communication between the client and the components and between components themselves
- Standardized definition of key components so all implementations of such "standard components" expose the same external interface (i.e. same inputs, outputs, and controls)

1.2.2 Design Philosophy

As previously stated, the key focus of the OpenMAX IL API is portability of media components. The diversity of existing devices and media implementation solutions necessitates that the OpenMAX IL target the higher level of the media software stack as the key initial user. For many operating systems, this means an existing media framework or some form of multimedia middleware.

Another key target is the OpenMAX AL API which standardizes a higher application level interface companion to OpenMAX IL. OpenMAX AL is designed to be amenable to OpenMAX IL implementations.

Thus, much of the OpenMAX IL API accommodates the needs of multimedia middleware allowing that layer to be as lightweight as possible. The result is an interface that is easily pluggable into most software stacks across operating system and multimedia middleware solutions.

The design of the API also strove to accommodate as many system architectures as possible. The resulting design uses highly asynchronous communications, which allows processing to take place in another thread, on multiple processing elements, or on specialized hardware. In addition, the ability of hardware-accelerated components to communicate directly with one another via tunneling affords implementation architectures even greater flexibility and efficiency.

1.2.3 Software Landscape

In some systems, a user-level media framework already exists. In those without such multimedia middleware, OpenMAX AL may serve to fill the gap. The OpenMAX IL API is designed to easily fit below this layer with little to no overhead between the interfaces. In most cases, a native media framework can be replaced with a thin layer that simply translates the API. Likewise, given the co-operative design of the two APIs, OpenMAX IL can even more seamlessly fit into an OpenMAX AL implementation. Figure 1-1 illustrates the software landscape for the OpenMAX IL API.

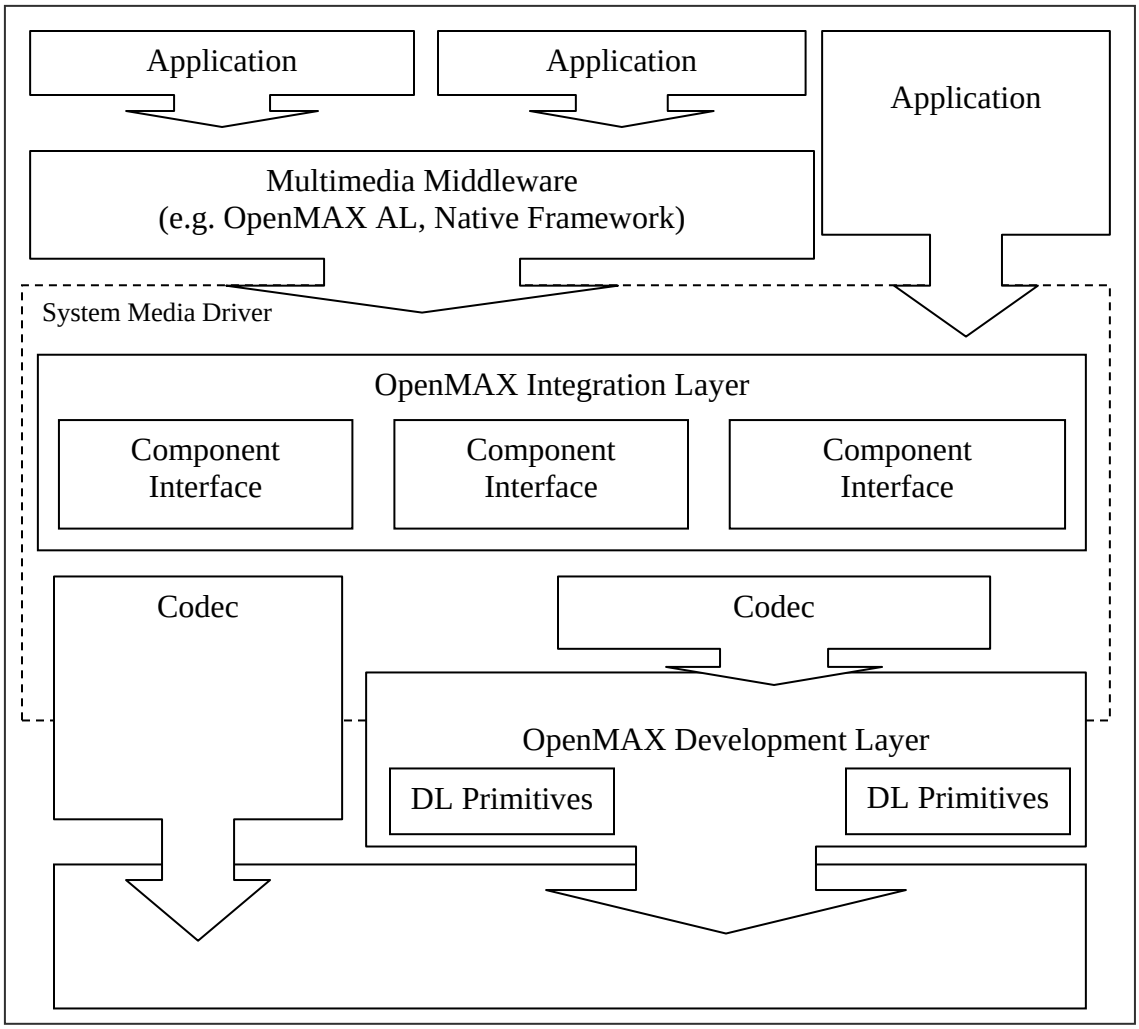


Figure 1-1. OpenMAX IL API Software Landscape

The OpenMAX standard also defines a set of Development Layer (DL) primitives on which components can be built. The DL primitives and their full relationship to the IL are specified in the OpenMAX Development Layer API specification documents.

1.2.4 Stakeholders

A few categories of stakeholders represent the broad array of companies participating in the production of multimedia solutions, each with their own interest in the IL API.

1.2.4.1 Silicon Vendors

Silicon vendors (SV) are responsible for delivering a representative set of OpenMAX IL components that are specific to the vendor's platform. The vendors are anticipated to also supply components that are representative of the capabilities of their platforms.

1.2.4.2 Independent Software Vendors

Independent software vendors (ISV) are anticipated to deliver additional differentiated OpenMAX IL components that may or may not be specific to a given silicon vendor's platform.

1.2.4.3 Operating System Vendors

Operating System Vendors (OSV) are anticipated to deliver software multimedia framework and standard reference OpenMAX IL components that enable integration of the representative silicon vendor's components and ISV components.

1.2.4.4 Original Equipment Manufacturers

Original Equipment Manufacturers (OEM) are anticipated to modify and optimize the integration of OpenMAX IL components provided by SVs, ISVs, and OSVs to their specific product architectures to enable delivery of OpenMAX IL integrated multimedia devices. OEMs may also develop and integrate their own proprietary OpenMAX IL components.

1.2.5 The Interface

The OpenMAX IL API is a component-based media API that consists of two main segments: the core API and the component API.

1.2.5.1 Core

The OpenMAX IL core is used for dynamically loading and unloading components and for facilitating component communication. Once loaded, the API allows the user to communicate directly with the component, which eliminates any overhead for high level commands. Similarly, the core allows a user to establish a communication tunnel between two components. Once established, the core API is no longer used and communications flow directly between components.

1.2.5.2 Components

In the OpenMAX Integration Layer, components represent individual blocks of functionality. Components can be sources, sinks, codecs, filters, splitters, mixers, or any other data operator. Depending on the implementation, a component could possibly represent a piece of hardware, a software codec, another processor, or a combination thereof.

The individual parameters of a component can be set or retrieved through a set of associated data structures, enumerations, and interfaces. The parameters include data relevant to the component's operation (i.e., codec options) or the actual execution state of the component.

Buffer status, errors, and other time-sensitive data are relayed to the application via a set of callback functions. These are set via the normal parameter facilities and allow the API to expose more of the asynchronous nature of system architectures.

Data communication to and from a component is conducted through interfaces called ports. Ports represent both the connection for components to the data stream and the buffers needed to maintain the connection. Users may send data to components through input ports or receive data through output ports. Similarly, a communication tunnel between two components can be established by connecting the output port of one component to a similarly formatted input port of another component.

1.3 Definitions

When this specification discusses requirements and features of the OpenMAX IL API, specific words are used to convey their necessity in an implementation. Table 1-1 shows a list of these words.

Table 1-1: Definitions of Commonly Used Words

Word	Definition
May	The stated functionality is an optional requirement for an implementation of the OpenMAX IL API. Optional features are not required by the specification but may have conformance requirements if they are implemented. This is an optional feature as in "The component may have vendor specific extensions."
Shall	The stated functionality is a requirement for an implementation of the OpenMAX IL API. If a component fails to meet a shall statement, it is not considered to conform to this specification. Shall is always used as a requirement, as in "The component designers shall produce good documentation."
Should	The stated functionality is not a requirement for an implementation of the OpenMAX IL API but is recommended or is a good practice. Should is usually used as follows: "The component should begin processing buffers immediately after it transitions to the `OMX_StateExecuting` state." While this is good practice, there may be a valid reason to delay processing buffers, such as not having input data available.
Will	The stated functionality is not a requirement for an implementation of the OpenMAX IL API. Will is usually used when referring to a third party, as in "the application framework will correctly handle errors."

1.4 Authors

The following individuals, listed alphabetically by company, contributed to this OpenMAX Integration Layer Application Programming Interface Specification.

- Tim Granger (Broadcom)
- Roger Nixon (Broadcom)
- Harald Gustafson (Ericsson)
- Zhengrong Yao (Ericsson)
- Juha Ylinen (Google)
- Glenn Kasten (Google)
- Kevan Ahmadi (Imagination Technologies)
- Neeraj Agrawal (Imagination Technologies)
- Dave Murray (Incoras)
- Adrian Burian (Nokia)
- Brian Evans (Nokia)
- Jarmo Hiipakka (Nokia)
- Juan Rubio (Nokia)
- Maria Pascual-Borrego (Nokia)
- Rajesh Rathinasamy (Nokia)
- Yeshwant Muthusamy (Nokia)
- Isaac Richards (NVIDIA)
- Jim Van Welzen (NVIDIA)
- Dusan Veselinovic (PacketVideo)
- Tom Longo (Qualcomm)
- Alwyn Dos Remedios (Qualcomm)
- Eric Auger (ST-Ericsson)
- Laurent Gerard (ST-Ericsson)
- Sebastien Le Duc (ST-Ericsson)
- Thierry Vuillaume (ST-Ericsson)
- Giulio Urlini (STMicroelectronics)
- Sripal Bagadia (Texas Instruments)
- Nikhil Mande (Texax Instruments)

1.5 Features New to Version 1.2

A summary of new features included into this release of this specification include:

State Machine Related Updates

- Introduction of dynamically allocated buffer support in addition to statically allocating buffers
- Removal of OMX_StateInvalid state
- Introduction of tunneled port status in order to eliminate race conditions during state transitions\port enabled when attempting to accept buffer usage and initiate buffer exchanges
- Ability to cancel pending submitted commands in order to recover from dead-lock situations
- Clarification of flush operation behavior

Buffer Flags Updates

- Ability to signal sub-frame boundaries
- Ability to signal the presence of valid timestamp information
- Ability to signal read-only buffers

Events Updates

- Updated EventHandler parameter usage to clarify event reasons
- New Event Types
    - Ability to detect the need to disable or flush other component ports in cases of buffer sharing
    - Ability to detect parameter or config index settings
- Updated conditions for when Port Settings Changes are to be signaled

Error Codes Updates

- Clean up of error code types: deletion of types, addition of types and updated descriptions for some existing types
- Updated cross references between methods and the errors that may be reported.

Methods Updates

- Introduction of separate OMX IL core methods to handle component ports tunnel setup and tear down
- Ability to introduce OMX IL core related extensions
- Ability to update the component callback method

Audio Features Updates

- Introduction of 3D Audio support

- Introduction of AMR WB+ support
- Extended WMA and AMR formats
- Ability to dynamically update settings such as bitrate, AMR modes and SBC Bitpool sizes

Camera Features Updates

- Enhanced Digital and Optical Zoom support
- Enhanced Exposure, White Balance and Focus Lock support
- Enhanced capture mode settings: pre-capture enablement and per port capturing control
- Enhanced Focus Range, Region and Status support
- Introduction of Field of View controls
- Introduction of Flash status reporting
- Introduction of ND Filter support
- Introduction of Assistant Light Control support
- Information of Flicker Rejection support
- Introduction of Histogram information
- Introduction of Sharpness control
- Ability to synchronize shutter opening and closing events with audio playback

Video Features Updates

- Introduction of Interlace detection of processing support
- Introduction of VC1 support
- Introduction of VP8 support
- Introduction of NAL Format support

Imaging Features Updates

- Introduction of additional Image Filters

Clock Features Updates

- Enhanced reference clock selections
- Enhanced media time notification mechanism

Standard Components Updates

- Introduction of a new header file to encapsulate all the standard component role names

- Updated feature list and processing criteria: e.g. additional ports, expanded list of supportable color formats, port slaving behavior, mandating that audio renders shall support the ability to provide reference clocks
- New standard components: AMR-WB+ Decoder\Encoder, 3D Audio Mixers, VC1 Video Decoder\Encoder

Other Updates

- Ability to group and commit multiple configuration settings atomically
- Clarifications of various parameter usage and processing operations
- Additional color formats

1.6 Backward Compatibility

The OpenMAX IL specification defines components and structures that evolve and improve with subsequent versions of the specification. The version of the specification is indicated with 4 digits Ma.Mi.R.S (Respectively Major, Minor, Revision and Step). Increments of these digits give the following indications:

- An increment of Major indicates a significant number of fundamental non-backward compatible changes.
- An increment of Minor indicates a significant number of functional changes like the addition of new structures and components. Essential corrections may create limited non backward compatible changes. An increment of revision indicates a significant number of corrections and clarifications which should be backward compatible unless stated explicitly. Any component of a later revision should interoperate with components of an earlier revision.
- An increment of step indicates a significant number of editorial corrections.

This specification version continues to support a significant level of functionality available as part of previous releases, however due to the nature of some of the improvements introduced, backwards compatibility with previous versions is not being maintained.

OpenMAX IL core and component providers shall only be required to provide functionality as described in this specification.

While backwards compatibility is not a requirement of the specification, OpenMAX IL solution providers may choose to include support for previous specification releases as part of their offerings.

2 OpenMAX IL Introduction and Architecture

This section of the document describes the OpenMAX IL features and architecture. The OpenMAX IL layer is an API that defines a software interface used to provide an access layer around software components in a system. The intent of the software interface is to take components with disparate initialization and command methodologies and provide a software layer that has a standardized command set and a standardized methodology for construction and destruction of the components.

2.1 Architectural Overview

Consider a system that requires the implementation of four multimedia processing functions denoted as F1, F2, F3, and F4. Each of these functions may be from different vendors or may be developed in house but by different groups within the organization. Each may have different requirements for setup and teardown. Each may have different methods of facilitating configuration and data transfer. The OpenMAX IL API provides a means of encapsulating these functions, singly or in logical groups, into components. The API includes a standard protocol that enables compliant components that are potentially from different vendors/groups to exchange data with one another and be used interchangeably.

The OpenMAX IL API interfaces with a higher-level entity denoted as the IL client, which is typically a functional piece of a filter graph multimedia framework, OpenMAX AL, or an application. The IL client interacts with a centralized IL entity called the core. The IL client uses the OpenMAX IL core for loading and unloading components, setting up direct communication between two OpenMAX IL components, and accessing the component's methods.

An IL client always communicates with a component via the IL core. In most cases, this communication equates to calling one of the IL core's macros, which translates directly to a call on one of the component methods. Exceptions (where the IL client calls an actual core function that works) include component creation and destruction, queries about installed components and the roles they support, and connection via tunneling of two components.

Components embody the media processing function or functions. Although this specification clearly defines the functionality of the OpenMAX IL core, the component provider defines the functionality of a given component. Components operate on four types of data that are defined according to the parameter structures that they export: audio, video, image, and other (e.g., time data for synchronization).

An OpenMAX IL component provides access to a standard set of component functions via its component handle. These functions allow a client to get and set component and port configuration parameters, get and set the state of the component, send commands to the component, receive event notifications, allocate buffers, establish communications with a single component port, and establish communication between two component ports.

Every OpenMAX IL component shall have at least one port to claim OpenMAX IL conformance. Although a vendor may provide an OpenMAX IL-compatible component without ports, the bulk of conformance testing is dependent on at least one conformant port. The four types of ports defined in OpenMAX IL correspond to the types of data a port may transfer: audio, video, and image data ports, and other ports. Each port is defined as either an input or output depending on whether it consumes or produces buffers.

In a system containing four multimedia processing functions F1, F2, F3, and F4, a system implementer might provide a standard OpenMAX IL interface for each of the functions. The implementer might just as easily choose any combination of functions. The delineation for the separation of this functionality is based on ports. Figure 2-1 shows a few possible partitions for an OpenMAX IL implementation that provides these functions.

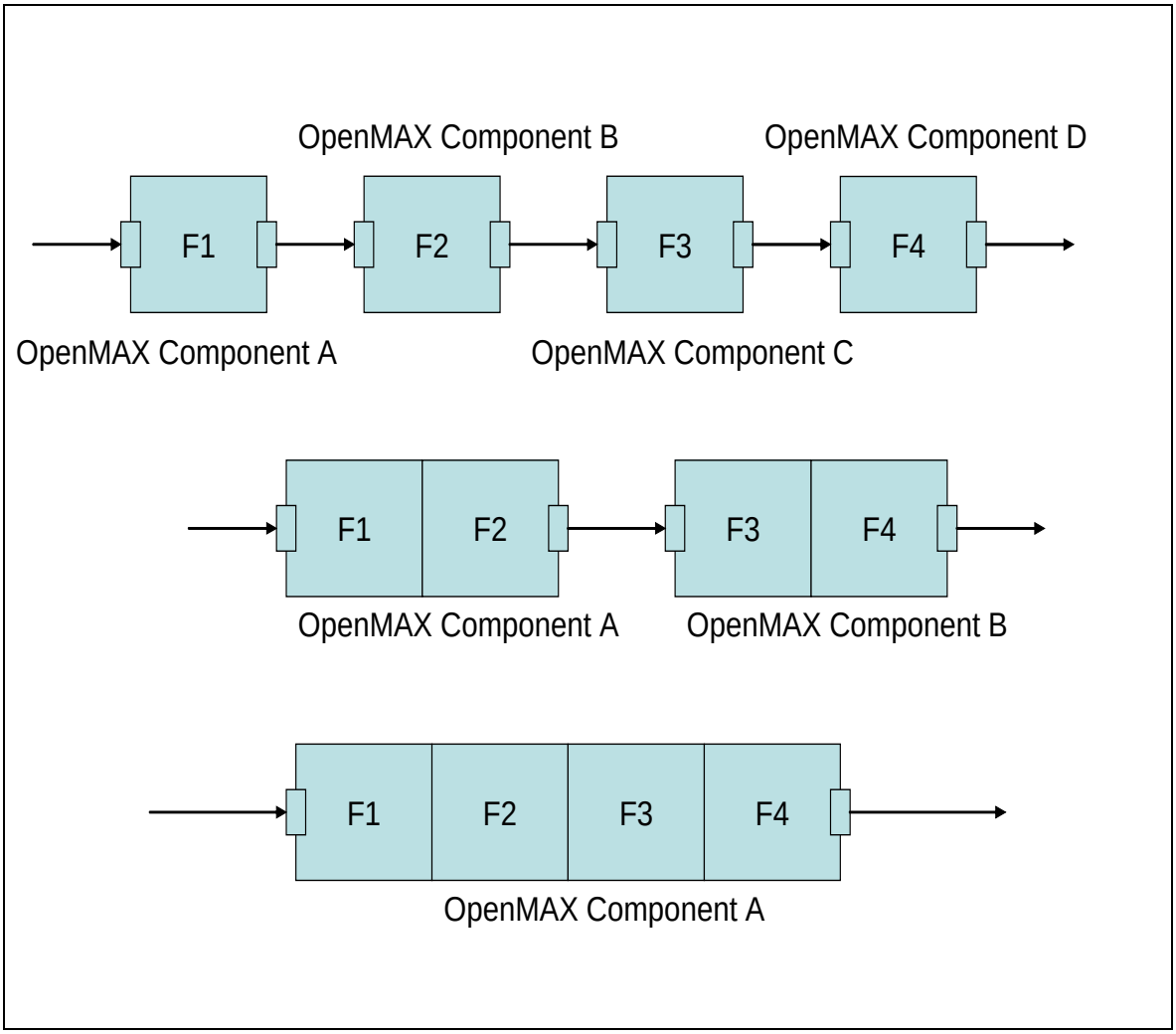


Figure 2-1. Possible Partitions for an OpenMAX IL Implementation

2.2 Key Vocabulary

This section describes acronyms and definitions commonly used in describing the OpenMAX IL API.

2.2.1 Key Definitions

Table 2-1 lists key definitions used in describing the OpenMAX IL API.

Table 2-1: Key Definitions

Key word	Meaning
Accelerated component	OpenMAX IL components that wrap a function with a portion running on an accelerator.
Accelerator	Hardware designed to speed up processing of some functions. This hardware may also be referred to as accelerated hardware. Note that the accelerator may actually be software running in a different processor and not be hardware at all.
Allocator port	An allocator port is a port that allocates its own buffers.
Buffer Supplier	The entity that requests the buffer headers to be allocated, i.e. the entity (port or IL client) invoking either UseBuffer or AllocateBuffer calls.
Container	A format for encapsulating elementary streams of data and associated metadata (e.g. the 3gp file format).
Content Pipe	The abstraction of a means to access (read or write) some content external to OpenMAX IL. Content may manifest itself as a file and a pipe may leverage system file i/o functions, but the abstraction is not limited to these particular types of content or content access.
Component Group	A group of components that is functionally dependent upon one another. If one component of a group is inoperable then all components in a group are inoperable.
Component Suspension	A component is suspended when it lacks a critical resource but holds all other resources so that, if and when the required resource is again available, that component may resume from the point of suspension.
Dynamic resources	Any component resources that are allocated after the initial transition to the idle state. Dynamic resource allocation is discouraged and should only occur when the parameters of the allocation (e.g. the size or number of internal memory buffers) is not known at the preferred times to allocate resources.
Host processor	The processor in a multi-core system that controls media acceleration and typically runs a high-level operating system.
IL client	The layer of software that invokes the methods of the core or component. The IL client may be a layer below the GUI application, such as GStreamer, or may be several layers below the GUI layer. In this document, the application refers to any software that invokes the OpenMAX IL methods.
Main memory	Typically external memory that the host processor and the accelerator share.
Non-supplier port	A port that is not a supplier port, i.e. the port requested to allocate the buffer headers by receiving either UseBuffer or AllocateBuffer calls.

Key word	Meaning
OpenMAX IL component	A component that is intended to wrap functionality that is required in the target system. The OpenMAX IL wrapper provides a standard interface for the function being wrapped.
OpenMAX IL core	Platform-specific code that has the functionality necessary to locate and then load an OpenMAX IL component into main memory. The core also is responsible for unloading the component from memory when the application indicates that the component is no longer needed. In general, after the OpenMAX IL core loads a component into memory, the core will not participate in communication between the application and the component.
Resource manager	A software entity that manages hardware resources in the system.
Sharing port	A supplier port that re-uses buffers allocated by another port.' or 'A supplier port that re-uses buffers provided by another port within the same component.
Static resources	Component resources that are allocated as a prerequisite to entering the idle state. Most component resources fall into this category.
Supplier Port	A port with a role of Buffer Supplier (as defined above).
Synchronization	A mechanism for gating the operation of one component with another.
Tunnels/Tunneling	The establishment and use of a standard data path that is managed directly between two OpenMAX IL components.
Tunnelling port	A tunneling port is one of a pair of ports that has established a standard data path, or *tunnel.*

2.3 System Components

Figure 2-2 depicts the various types of communication enabled with OpenMAX IL. Each component can have an arbitrary number of ports for data communication. Components with a single output port are referred to as source components. Components with a single input port are referred to as sink components. Components running entirely on the host processor are referred to as host components. Components running on a loosely coupled accelerator are referred to as accelerator components. OpenMAX IL may be integrated directly with an application or may be integrated with multimedia framework components enabling heterogeneous implementations.

Three types of communication are described. Non-tunneled communications defines a mechanism for exchanging data buffers between the IL client and a component. Tunneling defines a standard mechanism for components to exchange data buffers directly with each other in a standard way. Proprietary communication describes a proprietary mechanism for direct data communications between two components and may be used as an alternative when a tunneling request is made, provided both components are capable of doing so.

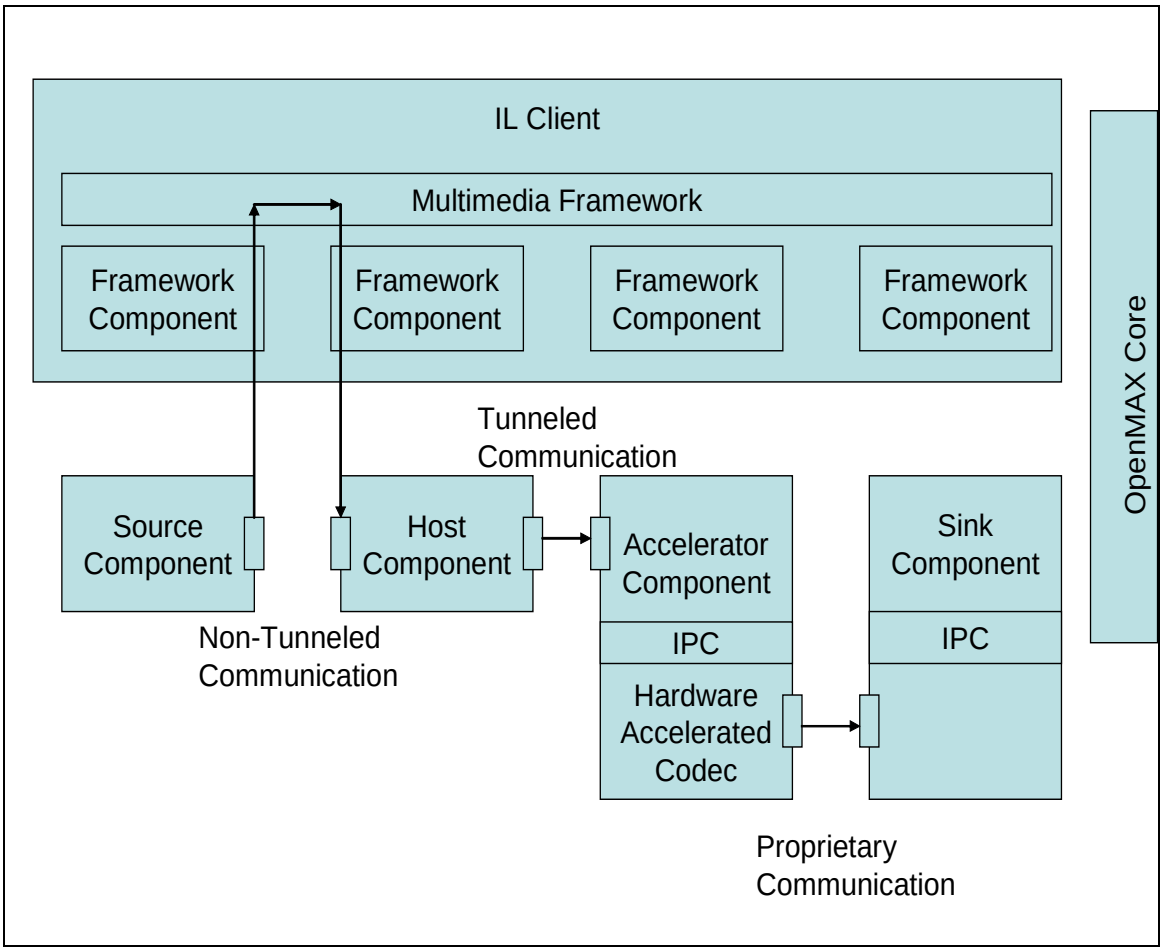


Figure 2-2. OpenMAX IL API System Components

2.3.1 Component Profiles

OpenMAX IL component functionality is grouped into two profiles: base profile and interop profile.

The base profile shall support non-tunneled communication. Base profile components may support proprietary communication. Base profile components do not support tunneled communication.

The interop profile is a superset of the base profile. An interop profile component shall support non-tunneled communication and tunneled communication. An interop profile component may support proprietary communication.

The primary difference between the interop profile and the base profile is that the component supports tunneled communication. The base profile exists to reduce the adoption barrier for OpenMAX IL implementers by simplifying the implementation.

Table 2-2: Types of Communication Supported Per Component Profile

Type of Communication	Base Profile Support	Interop Profile Support
Non-Tunneled Communication	Yes	Yes
Tunneled Communication	No	Yes
Proprietary Communication	Yes	Yes

2.4 Component States

Each OpenMAX IL component can undergo a series of state transitions, as depicted in Figure 2-3. Every component is first considered to be unloaded. The component shall be loaded through a call to the OpenMAX IL core. All other state transitions may then be achieved by communicating directly with the component.

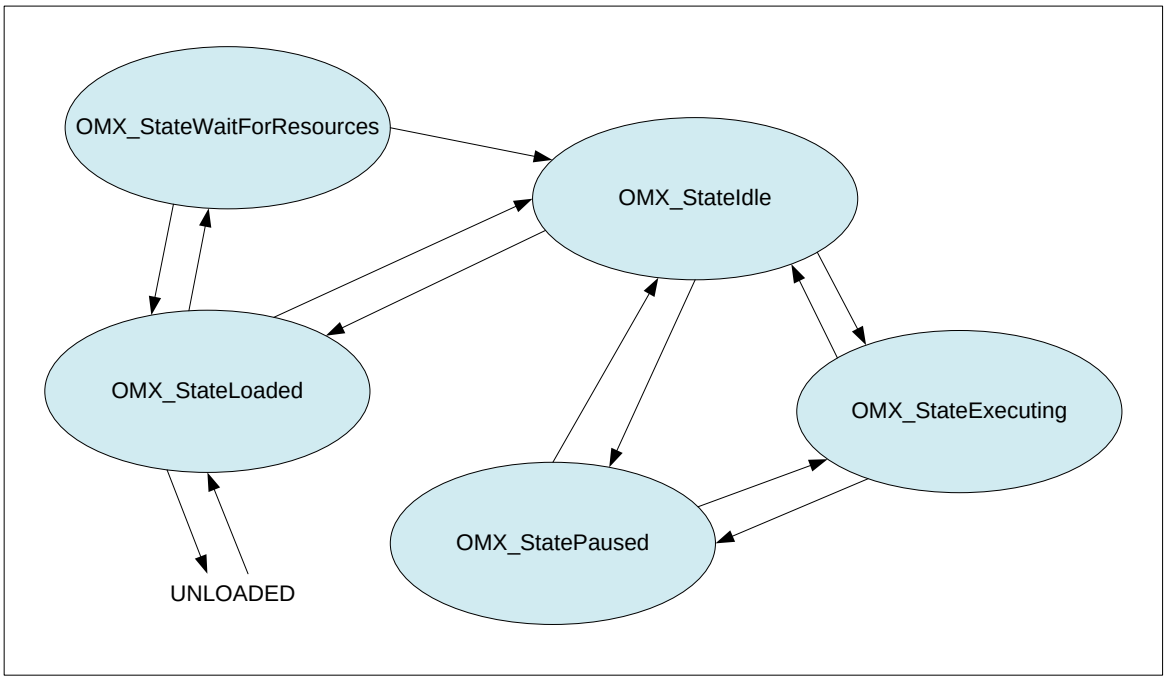


Figure 2-3. Component States

In general, the component should have all its operational resources when in the `OMX_StateIdle`, `OMX_StatePause` or `OMX_StateExecuting` states. There are, however, exceptions when the parameters for the resource allocation are not known at the time of the transition to `OMX_StateIdle`. For example, a component that decodes video does not know how many reference frames are required until the data stream is examined yet the component cannot examine the stream prior to transition to `OMX_StateIdle`. In these cases the component may defer the allocation of resources until such time as it knows the parameters of allocation. If dynamic allocation fails the component may suspend itself, as described in Section 3.1.3.6. Thus we often distinguish between those resources allocated "up front" (e.g. on a transition to `OMX_StateIdle`)

and those allocated later by calling the former static resources and the latter dynamic resources.

Transitioning into the `OMX_StateIdle` state may fail since this state requires allocation of all operational static resources. When the transition from `OMX_StateLoaded` to `OMX_StateIdle` fails, the IL client may try again or may choose to put the component into `OMX_StateWaitForResources`. Upon entering `OMX_StateWaitForResources`, the component registers with a vendor-specific resource manager to alert it when resources have become available. The component will subsequently transition into the `OMX_StateIdle` state.

The `OMX_StateIdle` state indicates that the component has all of its needed static resources but is not processing data. The `OMX_StateExecuting` state indicates that the component is processing data when possible. The `OMX_StatePause` state maintains a context of buffer execution with the component without processing data or exchanging buffers. Transitioning from `OMX_StatePause` to `OMX_StateExecuting` enables buffer processing to resume where the component left off. Transitioning from `OMX_StateExecuting` or `OMX_StatePause` to `OMX_StateIdle` will cause the context in which buffers were processed to be lost, which requires the start of a stream to be reintroduced. Transitioning from `OMX_StateIdle` to `OMX_StateLoaded` will cause operational resources such as communication buffers to be lost.

2.5 Component Architecture

Figure 2-4 depicts the component architecture. Note that there is only one entry point for the component (through its handle to an array of standard functions) but there are multiple possible outgoing calls that depend on how many ports the component has. Each component will make calls to a specified IL client event handler. Each port will also make calls (or callbacks) to a specified external function. A queue for pointers to buffer headers is also associated with each port. These buffer headers point to the actual buffers. The command function also has a queue for commands. All parameter or configuration calls are performed on a particular index and include a structure associated with that parameter or configuration, as depicted in Figure 2-4.

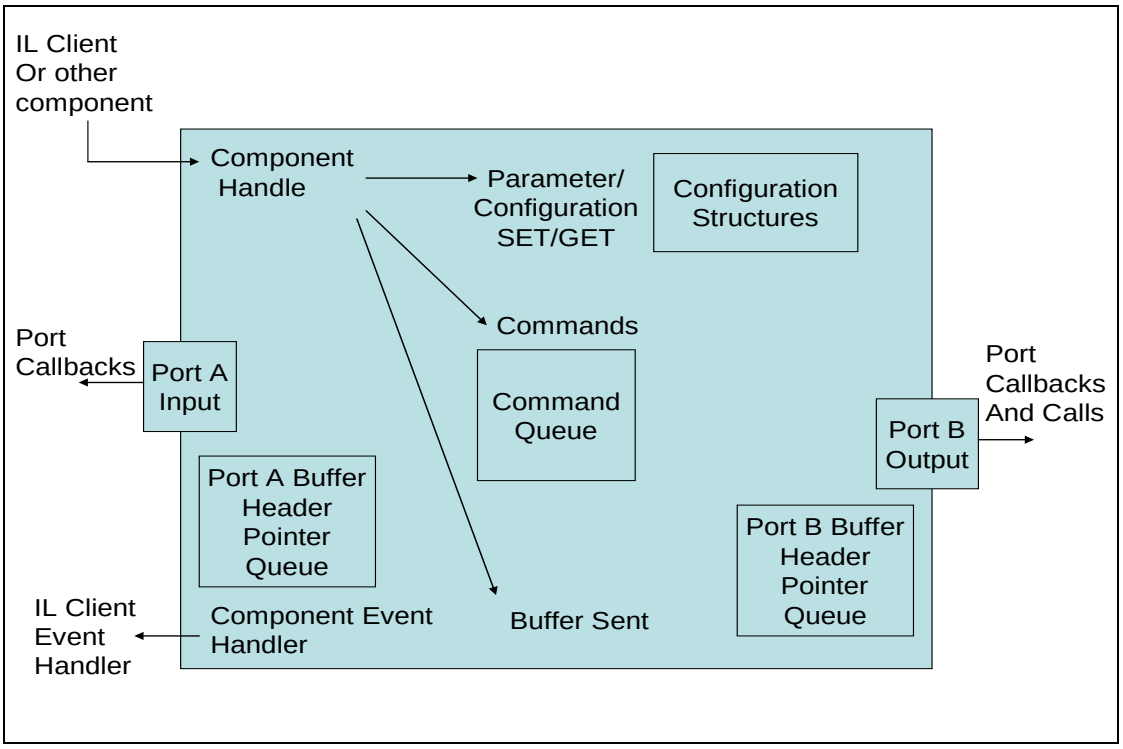


Figure 2-4. OpenMAX IL API Component Architecture

A port shall support callbacks to the IL client and, when part of an interop profile component, shall support communication with ports on other components.

2.6 Communication Behavior

Configuration of a component may be accomplished once the handle to the component has been received from the OpenMAX IL core. Data communication calls with a component are non-blocking and are enabled once the number of ports has been configured, each port has been configured for a specific data format, and the component has been put in the appropriate state. Data communication is specific to a port of the component. Input ports receive buffers via the `OMX_EmptyThisBuffer` call (for more information, see section 3.2.2.17). Output ports receive buffers via the `OMX_FillThisBuffer` call (for more information, see section 3.2.2.18).

Data communications with components is always directed to a specific component port. Each port has a component-defined minimum number of buffers it shall allocate or use. A port associates a buffer header with each buffer. A buffer header references data in the buffer and provides metadata associated with the contents of the buffer. Every component port shall be capable of allocating its own buffers or using pre-allocated buffers; one of these choices will usually be more efficient than the other.

Buffers are consumed by the component when it is in `OMX_StateExecuting`, unless the corresponding port is disabled. A component may return partially consumed or unconsumed buffers to the IL client or tunneled component only if:

- The IL client commands a transition from `OMX_StateExecuting` or `OMX_StatePause` to `OMX_StateIdle`.
- Corresponding port is flushed or disabled

2.7 Thread Safety

OpenMAX IL core and Component implementations shall be thread-safe. Any of the IL functions may be called from any thread, or multiple threads simultaneously, running on either single or multi-core processors. The only restriction is that the IL client shall not call IL core or component functions from within an IL callback context. IL implementations shall pose no special thread safety or synchronizing requirements to the calling thread, such as expecting the client to be using or setting up a specific thread synchronization mechanism.

IL callbacks may originate from any thread or multiple threads in the process that created the IL component. IL implementations shall not call client-provided callback functions from a non-thread context, such as an interrupt service routine (ISR). There are no guarantees on how many threads the IL component is running. The IL client shall accept callbacks from any thread context, including its own. Multiple callbacks can be in progress at the same time, including callbacks to the same function. Therefore, the IL client callback implementations shall also be thread-safe.

2.8 Tunneled Buffer Allocation

A component that is tunneling shall:

- provide buffers on all of its supplier ports;
- accurately communicate buffer requirements on all its ports;
- pass a buffer from an output port to an input port with an `OMX_EmptyThisBuffer` call;
- pass a buffer from an input port to an output port with an `OMX_FillThisBuffer` call.

For any given tunnel, exactly one port supplies the buffers and passes those buffers to the non-supplier port.

A port that is not tunnelled and receives the AllocateBuffer call from the IL client is also considered a non-supplier port.

The set of buffer requirements for a tunneling port includes the number of buffers required, as well as the required size and alignment of each buffer.

Figure 2-5 illustrates the concepts relevant to tunneled buffer allocation.

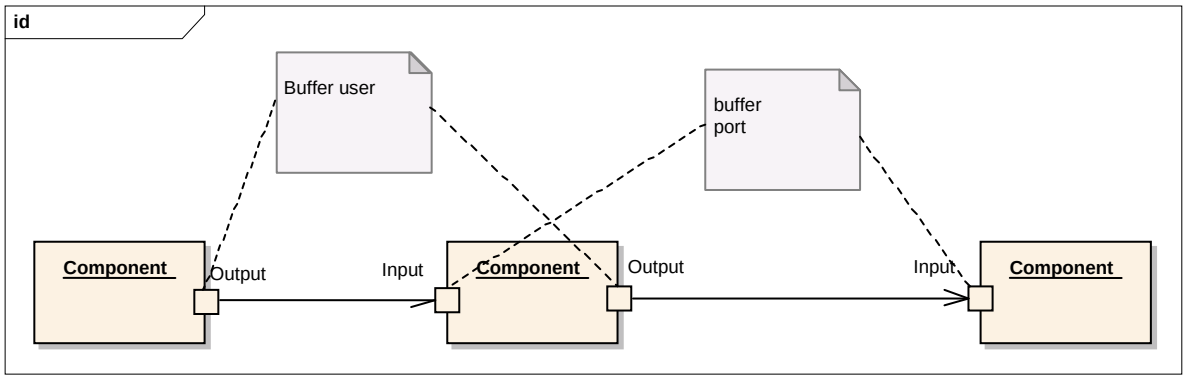


Figure 2-5. Port Relationships

At any time after `OMX_SetupTunnel` completes successfully but before allocating its buffers the supplier port retrieves the tunnel's buffer requirements from its tunneled port in an `OMX_PARAM_PORTDEFINITIONTYPE` structure via an `OMX_GetParameter` call on the tunneled port's component.

It then computes the minimum buffering requirements applicable to the tunnel from its own minimum buffering requirements and those of its tunneled port.

The minimum of buffer requirements for a tunnel is the set defined as follows:

- The largest number of buffers mandated in any set. i.e. The maximum value of `nBufferCountMin` from either port
- The largest size mandated in any set. The maximum value of `nBufferSize` from either port
- The largest alignment mandated in any set. The maximum value of `nBufferAlignment` from either port

Note that, whilst normally the supplier port of a tunnel also allocates its own buffers, under the right circumstances, a component may choose to re-use buffers from one port on another upon the same component to avoid memory copies and optimize memory usage. This optional practice, known as buffer sharing is described in detail in Section 10—Implementing Buffer Sharing.

Buffer sharing may require that a tunneled port needs to include its shared port's buffer requirements in the minimum requirements calculation above determine the correct tunnel buffer requirements. However it may determine the buffer requirements from a port that shares its buffers without resorting to an OMX_GetParameter call since they are both contained in the same component.

Section 3.4.1.2 describes tunneling setup and initialization in further detail.

2.8.1 IL Client Component Setup

To set up tunneling components, the IL client should perform the following setup operations in this order:

1. Load all tunneling components and set up the tunnels on these components.
2. Command all tunneling components to transition from `OMX_StateLoaded` to `OMX_StateIdle`.

Note that if an IL client does not operate in this manner when some components are sharing buffers, a tunneling component might never transition to `OMX_StateIdle` because of the possible dependencies between components.

2.8.2 Component Transition from OMX_StateLoaded to OMX_StateIdle

When commanded to transition from `OMX_StateLoaded` to `OMX_StateIdle`, each allocator port shall:

1. Determine the buffer requirements of its tunneled port via `OMX_GetParameter`.
2. Allocate buffers according to the maximum of its own requirements and the requirements of the tunneled port.
3. Communicate the actual number of buffers that shall be used for the tunnel and the contiguity of those buffers via a call to `OMX_SetParameter(OMX_IndexParamPortDefinition)`, if the `nBufferCountActual` or `bBuffersContiguous` fields are different from that advertised by the non-supplier port.
4. Call `OMX_UseBuffer` on its tunneling port after it receives the `OMX_SetConfig(OMX_IndexConfigTunneledPortStatus)` call as specified in section 3.1.3.13.

2.9 Port Reconnection

Port reconnection enables a tunneled component to be replaced with another tunneled component without having to tear down surrounding components. In Figure 2-6, component B1 is to be replaced with component B2. To do this, the component A output port and the component B input port shall first be disabled with the port disable command. Once all allocated buffers have returned to their rightful owner and freed, the tunnel between A and B1 can be torn down, and the component A output port may be connected to component B2. The component B1 output port and the component C input port should similarly be given the port disable command. After all allocated buffers have returned to their owners and freed, the tunnel between B1 and C can be torn down, and the component C input port may be connected to the component B2 output port. Then all ports may be given the enable command. Refer to Section 3.4.4 Port Disablement and Enablement for additional information regarding port disabling and enabling.

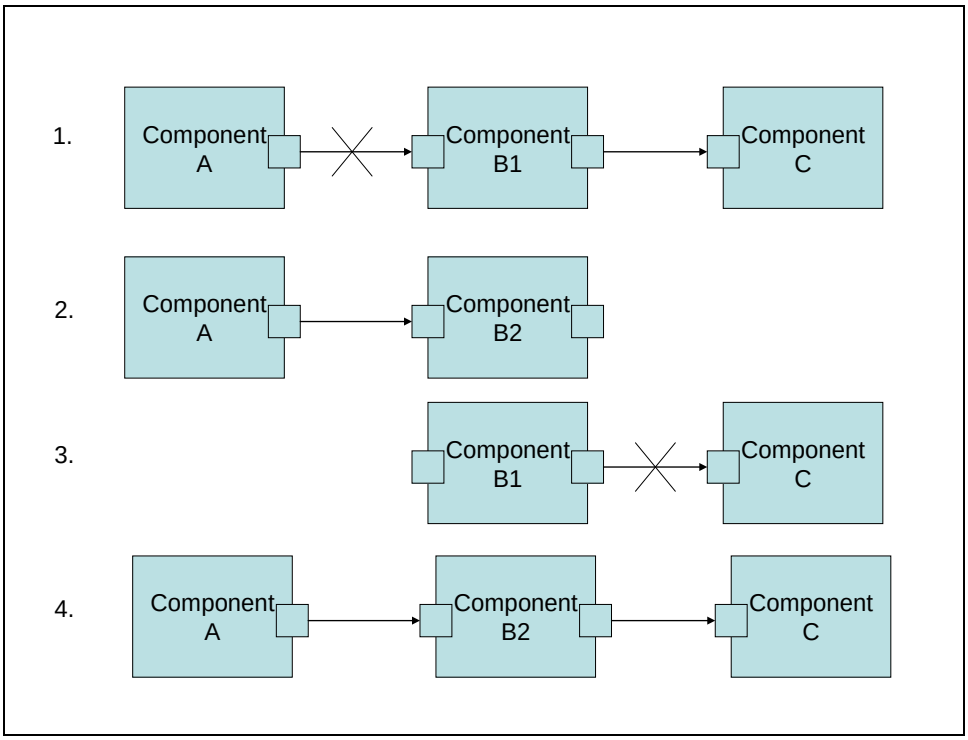


Figure 2-6. Port Recommention

In some cases such as audio, reconnecting one component to another and then fading in data for one component while fading out data for the original component may be desirable. Figure 2-7 illustrates how this would work. In step 1, component A sends data to component B1, which then sends the data on to component C. Components A and C both have an extra port that is disabled. In step 2, the IL client first establishes a tunnel between component A and B2, then establishes a tunnel between B2 and C, and then enables all ports in the two tunnels. Component C may be able to mix data from components B1 and B2 at various gains, assuming that these are audio components. In step 3, the ports connected to component B1 from components A and C are disabled, the tunnels involving B1 are torn down, and component B1 resources may be de-allocated.

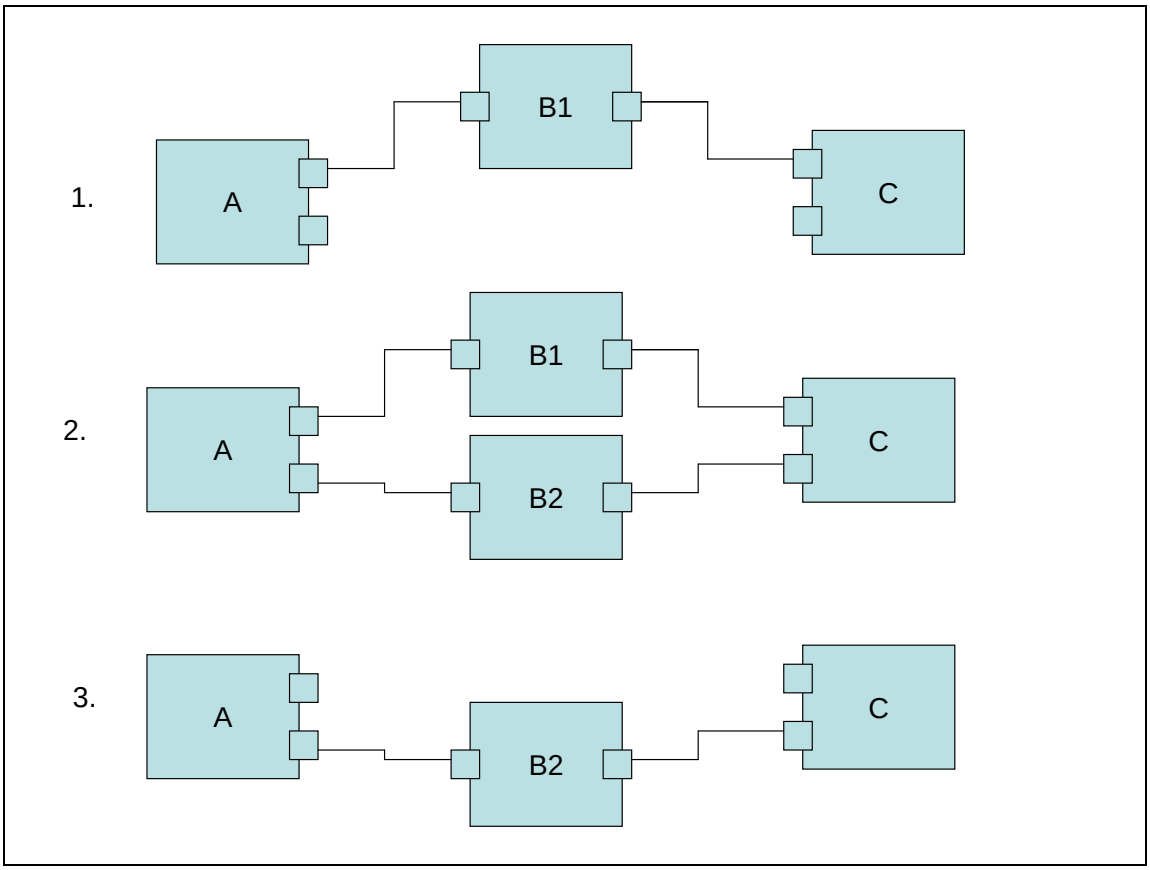


Figure 2-7. Reconnecting Components

2.10 Queues and Flush

A flush command causes the component to flush buffers that have not been processed and return these buffers to the IL client when using non-tunneled communication, or to the supplier port when using tunneled communication. For example, assume that a component has an output port that is using buffers allocated by the IL client. In this example, the client sends a series of five buffers to the component before sending the flush command. Upon processing the flush command, the component returns each unprocessed buffer and triggers its event handler to notify the IL client. Two buffers were already processed before the flush command got processed. The component returns the remaining three buffers unfilled and generates an event. The IL client should wait for the event before attempting to de-initialize the component.

2.11 Marking Buffers

An IL client can also trigger an event to be generated when a marked buffer is encountered. A buffer can be marked in its buffer header. The mark is internally transmitted from an input buffer to an output buffer in a chain of OpenMAX IL components. The mark enables a component to send an event to the IL client when the marked buffer is encountered. Figure 2-8 depicts how this works.

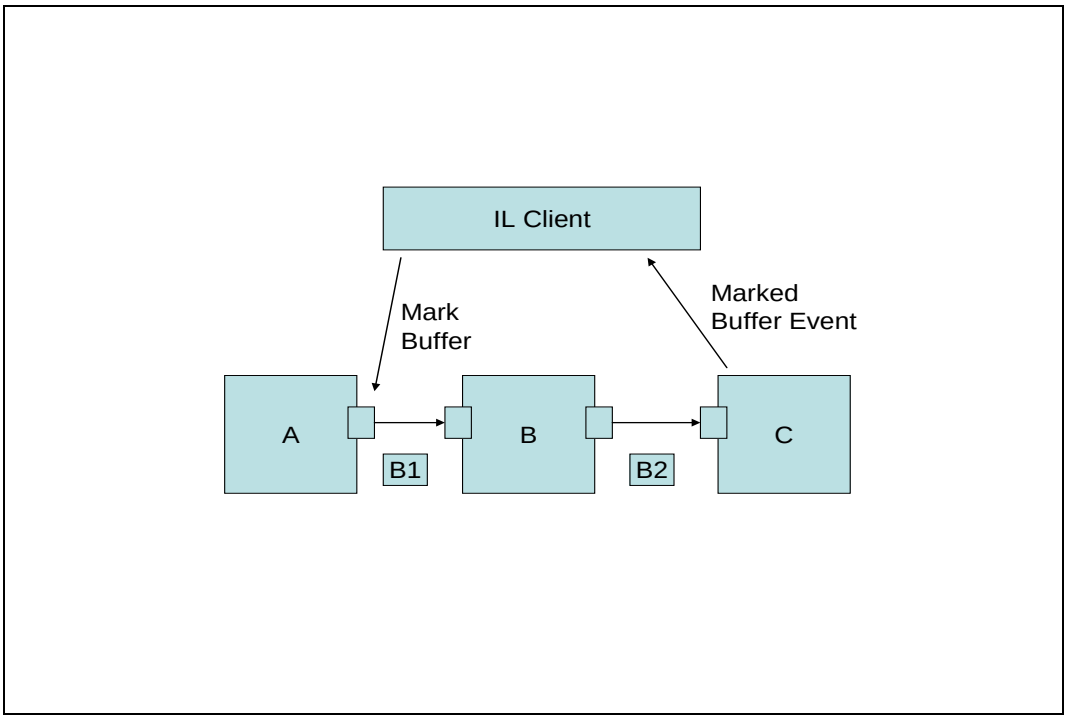


Figure 2-8. Marking Buffers

The IL client sends a command to mark a buffer from the output port of Component A, with Component C as the target. The next buffer, B1, sent from this port is marked. Component B processes B1 and provides the results in buffer B2, along with the mark. After component C has processed B2 it calls its event handler with the buffer mark.

2.12 Events and Callbacks

Several kinds of events are sent by a component to the IL client:

- *Error events* are enumerated and can occur at any time.
- *Command complete notification events* are triggered upon successful or unsuccessful execution of a command.
- *Marked buffer events* are triggered upon detection of a marked buffer by a component.
- *Settings changed notification events* are generated when the component changes its settings.
- A *buffer flag event* is triggered when an end of stream is encountered.
- *Resource events* are generated when a component has a change in its resource status.
- *Port command events* are generated when further IL client action is needed to complete an ongoing command.

Ports make buffer handling callbacks upon availability of a buffer or to indicate that a buffer is needed.

2.13 Buffer Payload

The port configuration is used to determine and define the format of the data to be transferred on a component port, but the configuration does not define how that data exists in the buffer.

There are generally three cases that describe how a buffer can be filled with data. Each case presents its own benefits.

In all cases, the range and location of valid data in a buffer is defined by the `pBuffer`, `nOffset`, and `nFilledLen` parameters of the buffer header. The `pBuffer` parameter points to the start of the buffer. The `nOffset` parameter is set by the entity that places data into the buffer (or forwards data in a shared buffer) to indicate the number of bytes between the start of the buffer and the start of valid data. The `nFilledLen` parameter specifies the number of contiguous bytes of valid data in the buffer. The valid payload data in the buffer is therefore located in the range `pBuffer + nOffset` to `pBuffer + nOffset + nFilledLen`. Metadata may be present in the buffer after this region.

The following cases are representative of compressed data in a buffer that is transferred into or out of a component when decoding or encoding. In all cases, the buffer just provides a transport mechanism for the data with no particular requirement on the content. The requirement for the content is defined by the port configuration parameters.

The shaded portion of the buffer represents data and the white portion denotes no data.

Case 1: Each buffer is filled in whole or in part. In the case of buffers containing compressed data frames, the frames are denoted by f1 to fn.

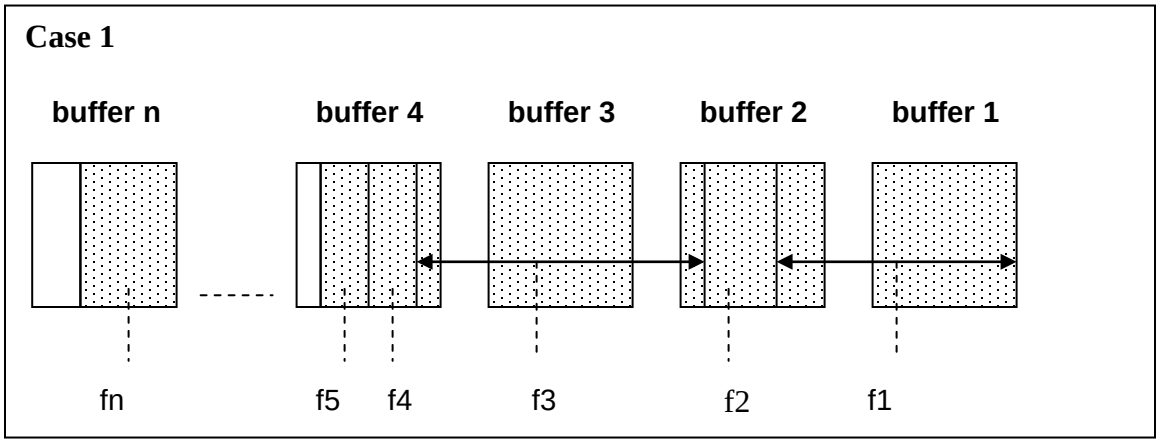


Figure 2-9: Case 1—Each Buffer Filled In Whole or In Part

Case 1 provides a benefit when decoding for playback. The buffer can accommodate multiple frames and reduce the number of transactions required to buffer an amount of data for decoding. However, this case may require the decoder to parse the data when decoding the frames. It also may require the decoder component to have a frame-building buffer in which to put the parsed data or maintain partial frames that would be completed with the next buffer.

Case 2: Each buffer is filled with only complete frames of compressed data.

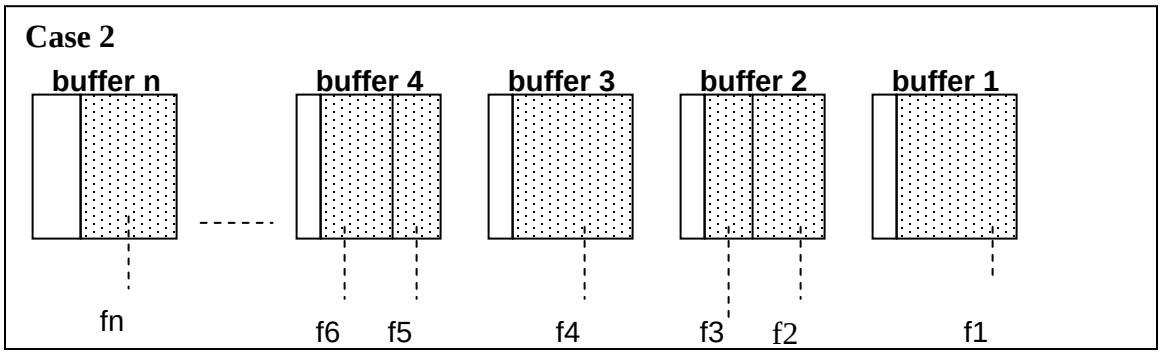


Figure 2-10: Case 2—Each Buffer Filled with Only Complete Frames of Data

Case 2 differs from case 1 because it requires the compressed data to be parsed first so that only complete frames are put in the buffers. Case 2 may also require the decoder component to parse the data for decoding. This case may not require the extra working buffer for parsing frames required in case 1.

Case 3: Each buffer is filled with only one frame of compressed data.

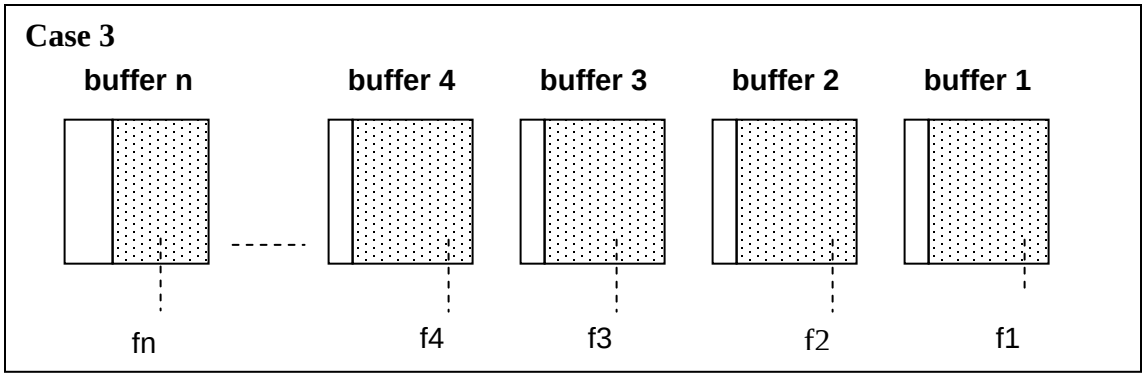


Figure 2-11: Case 3—Each Buffer Filled with Only One Frame of Compressed Data

The benefit in case 3 is that a decoding component does not have to parse the data. Parsing would be required at the source component. However, this method creates a bottleneck in data transfer. Data transfer would be limited to one frame per transfer. Depending on the implementation, one transaction per frame could have a greater impact on performance than parsing frames from a buffer.

At a minimum, a decoder or encoder component would be required to support case 1. By definition, if a codec component can support case 1, then it can support cases 2 and 3, but only if the compression format allows for byte-aligned frame boundaries. Operating in case 2 or 3 may not make sense when, for example, configuring an Adaptive Multi-Rate (AMR) codec for RTP-payload format, bandwidth-efficient mode. The non-byte aligned frames defined by this format would not fit the byte-aligned frame boundaries defined by these cases.

When filling a buffer with compressed data for input to a decoder or output from an encoder, a problem with limiting the filling to complete frames only might arise when

frames are not byte aligned. Padding would have to be added outside of any padding defined in the format specification. The padding would then need to be removed, since the data could not be appended as is. This would require knowledge of the padding bits outside of any standard specification. Likewise, if this padding were not in place to maintain compliance with the standards specification for the port configuration, complete frames could not always be placed in the buffers. In either case, specific knowledge of how this situation is handled would be required, and may be different between components.

For interoperability, the content delivered in a buffer should not be assumed or required to be any number of complete frames, although at least one complete unit of data will be delivered in a buffer for uncompressed data formats. Compressed data formats do not place restrictions on the amount of content delivered in each buffer.

There are a number of codecs that do not include frame boundary signaling in the elementary bitstream, instead relying on container signaling. Examples of this are WMV Simple and Main Profiles, VP8 and Real Video. For such cases it is expected that the component/source entity, up stream of the decoder (like demuxer/parser) supplies the data with frame boundaries signaled (for example through use of the `OMX_BUFFERFLAG_ENDOFFRAME` and `OMX_BUFFERFLAG_SKIPFRAME` flag).

2.14 Signalling frames and subframes

To aid processing of data the producer of data can explicitly signal frames or subframes that appear in a buffer payload. These flags can be applied to a buffer, and signal that the end of the valid payload in that buffer ends a frame or a subframe.

2.14.1 Signalling frames

In the use-cases where frames are fragmented over multiple buffer payloads, as is often the case in streaming, the usage of the optional `OMX_BUFFERFLAG_ENDOFFRAME` flag (defined in 3.1.3.7.1) provides a mechanism to easily identify the end of a frame boundary and the beginning of the next frame without the need for additional processing and bit-stream parsing. This is especially useful if the frames are fragmented over multiple buffer payloads and bit-stream format does not contain sync-words, start-codes or other means of identifying frame boundaries (e.g. WMV simple/main profile frames fragmented over multiple buffer payloads).
Even if the bit-stream contains other means for identifying frame boundaries (e.g. through presence of start-codes or sync-words), the decoder component must still wait for the arrival of the next buffer to clearly identify the end of the previous frame and the beginning of the next frame.

The usage of the optional `OMX_BUFFERFLAG_ENDOFFRAME` flag eliminates the need for waiting for the next buffer and allows the decoder component to start decoding the bit-stream immediately after receiving the buffer marked with `OMX_BUFFERFLAG_ENDOFFRAME` flag.

Referring to Figure 2-12 as an example, with the presence of OMX_BUFFERFLAG_ENDOFFRAME flag, the decoder component can detect the end of Frame 1 immediately after receiving Buffer 3. On the other hand, in the absence of OMX_BUFFERFLAG_ENDOFFRAME flags, the decoder component would have to wait to receive at least Buffer 4 as well before being able to detect the end of Frame 1 (even if start-codes or sync-words are present in the bitstream).

2.14.2 Signalling subframes

Several video compression formats define usage of sub-frames (independently decodable units smaller than a frame, or decodable units that represent side information). For example, in the ITU H.264\AVC specification, video content is divided into NALUs, a set of NALUs representing a coded picture or other side information. Similarly, in the case of MPEG4 [MPEG-4 Visual v2] or H263 bit-streams [H.263, RFC4629] – a decoder may use a picture slice / GOB as an independently decodable unit. In the case of VC-1 Advanced Profile [VC1], a picture slice is independently decodable, as well as a field in case of interlaced content of VC-1 Advanced Profile.

The benefits of using OMX_BUFFERFLAG_ENDOFSUBFRAME (defined in 3.1.3.7.1) are in some aspects similar to the benefits of using OMX_BUFFERFLAG_ENDOFFRAME, except that they apply to sub-frames rather than frames. In some aspects, however, the OMX_BUFFERFLAG_ENDOFSUBFRAME flag has its own unique benefits.

2.14.2.1

Firstly, in the case where NALUs are provided without Start Codes, as is the case of RTP streaming of H264 [Section 4.3.32] (i.e. when NALU format is "OMX_NaluFormatOneNaluPerBuffer"), OMX_BUFFERFLAG_ENDOFSUBFRAME flag provides a mechanism to identify the end of a NALU and the beginning of the next NALU, specifically for the case where a NALU is fragmented over multiple RTP payloads i.e. buffers. Without this mechanism, it is impossible for the decoder component to identify the boundaries of the NALU, since the bit-stream itself contains no sync-words, start-codes or other clues that would allow the decoder component to reconstruct the NALU boundary.
Secondly, if the output port of the source component implements and sets the OMX_BUFFERFLAG_ENDOFSUBFRAME flag, the decoder components that support decoding of sub-frames can start decoding the bit-stream immediately after receiving a buffer marked with OMX_BUFFERFLAG_ENDOFSUBFRAME flag, rather than waiting for the entire frame to buffer up. Waiting to receive the entire frame may take several buffer payloads and may incur a delay penalty. Referring to Figure 2-12 as an example, the decoder component that supports sub-frame decoding can start decoding immediately after receiving Buffer 1, rather than buffering Buffers 1 through 3 (which all belong to Frame 1).

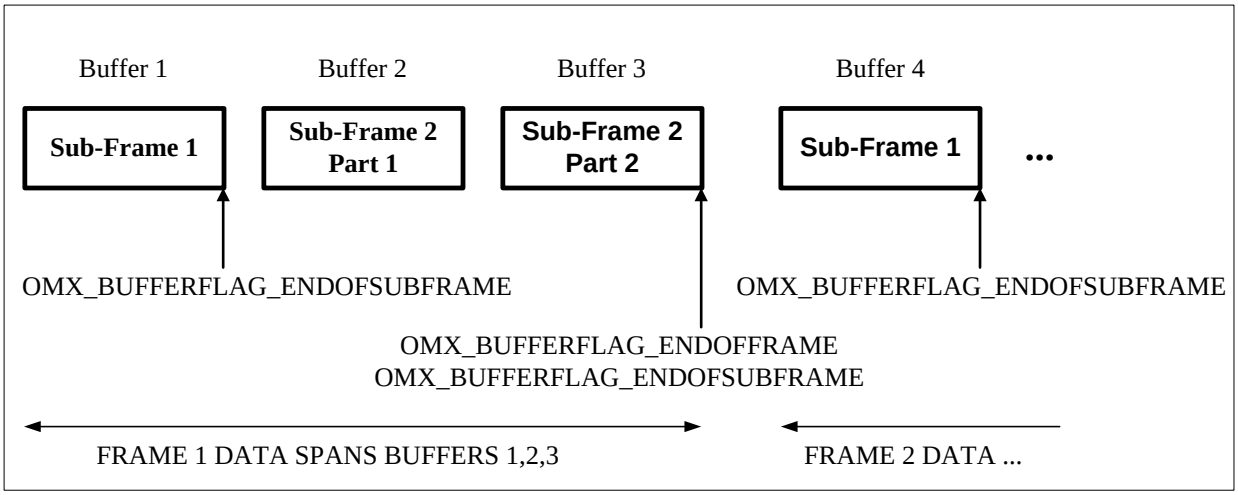


Figure 2-12: Example usage of OMX_BUFFERFLAG_ENDOFFRAME and OMX_BUFFERFLAG_ENDOFSUBFRAME flags

Finally, even if the bit-stream contains other means for identifying sub-frame boundaries (e.g. through presence of start codes), the decoder component must still wait for the arrival of the next buffer to clearly identify the end of the previous sub-frame and the beginning of the next sub-frame.

The usage of the optional `OMX_BUFFERFLAG_ENDOFSUBFRAME` flag eliminates the need for waiting for the next buffer and allows the decoder component that supports sub-frame decoding to start decoding the bit-stream immediately after receiving the buffer marked with `OMX_BUFFERFLAG_ENDOFSUBFRAME` flag.

Again, referring to Figure 2-12 as an example, with the presence of `OMX_BUFFERFLAG_ENDOFSUBFRAME`, the decoder component can detect the end of Sub-frame 1 immediately after receiving Buffer 1. On the other hand, in the absence of `OMX_BUFFERFLAG_ENDOFSUBFRAME` flags, the decoder component would have to wait to receive at least Buffer 1 and Buffer 2 before being able to detect the end of Sub-frame 1, even if start-codes are present in the bit-stream.

This benefit is similar to the benefit of using the `OMX_BUFFERFLAG_ENDOFFRAME` flag, except that it applies to sub-frames, rather than frames.

2.15 Buffer Flags and Timestamps

Buffer flags associate certain properties (e.g., the end of a data stream) with the data contained in a buffer. A buffer timestamp associates a presentation time in microseconds with the data in the buffer used to time the rendering of that data. Once a timestamp is associated with a buffer, no component should alter the timestamp for rate control or synchronization, which are implemented in the clock component.

Buffer metadata (i.e., flags and timestamps) applies to the first new logical unit in the buffer. Thus, given the presence of multiple logical units in a buffer, the metadata applies to the logical unit whose starting boundary occurs first in the buffer. Subsequent logical units in a buffer don't have explicit flags or timestamps. If explicit flag and

timestamps are required on every logical unit, at most one logical unit should be included in each buffer. Unless otherwise stated (e.g., in a flag definition), a component that receives a logical input unit marked with a flag or timestamp shall copy that metadata to all logical output units that the input contributes to.

2.16 Synchronization

Synchronization is enabled by the use of synchronization (sync) ports on a clock component. These ports and the clock component are defined within the "other" domain and operate with the same protocols and calls that regulate data ports. The clock component maintains a media clock that tracks the position in the media stream based on audio and video reference clocks. The clock component calls `OMX_SetConfig` or transmits buffers containing time information (denoted by a media time update and containing the media clock's current position, scale, and state) to client components via sync ports. A client component may time the execution of an operation (e.g., the presentation of a video frame) to a timestamp by requesting that the clock component send that timestamp when it matches the media clock. In this case, the client component executes the operation when it receives the fulfillment of the request over its sync port.

Figure 2-13 illustrates the flow of time and information in an example configuration of components.

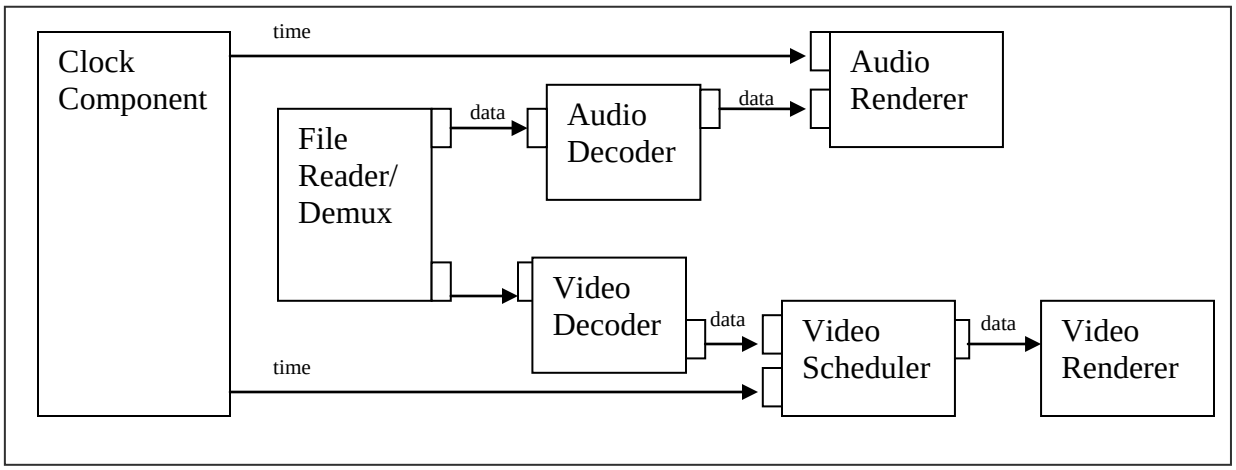


Figure 2-13. Flow of Time and Data Buffers

2.17 Rate Control

The clock component also implements all rate control by exposing a set of configurations for controlling its media clock. The IL client may change the scale factor of the media clock (effectively changing the rate and direction that the media clock advances) to implement play, fast forward, rewind, pause, and slow motion trick modes. The IL client may also start and stop the clock by using these configurations to change the state of the media clock. The clock component makes all of its client components aware of a change to the media clock scale and state by sending a media time update with the new scale or state on all sync ports. Although a component may not alter a buffer timestamp in

reaction to a scale change, a component may alter its processing accordingly. For instance, an audio component might scale and pitch correct audio during trick modes or cease transmitting output entirely.

2.18 Component Registration

How components are registered with a core is generally core specific.

However, if the core supports static linking with components, then it will support a standard compile-time component registration scheme as described in Section 3.1.3.1. Vendors can therefore supply components that are suitable for static linking with all cores that support it; this is achieved by placing component information into a data structure that is linked with the component and the core.

A component can be registered statically using this mechanism but have the bulk of its code dynamically loaded.

A component supplies an interface for retrieving the standard component roles it supports. The core may leverage this interface for exposing role-related information to the IL client.

2.19 Resource Management

This section discusses the role of resource management in the OpenMAX IL API.

2.19.1 Need for Resource Management

When a component fails to go to `OMX_StateIdle` due to lack of resources, the IL client may not know what the limited resource is or which components are using that resource. Therefore, the IL client cannot resolve the resource conflict. These situations necessitate IL resource management.

One of the goals of OpenMAX IL is hardware independence provided by the IL layer to the layers above it. The goal of hardware independence can be achieved by specifying the following requirements regarding resource management:

- An IL client (e.g., a multimedia plug-in that is typically part of a software platform) should not need to know the details of an IL implementation or which resource an IL component is using.
- In case of resource conflicts, an IL client should be able to rely on consistent component behavior across IL implementations and hardware platforms.
- An IL client should not have to interface directly with a hardware vendor-specific resource manager for two reasons.
    - This method violates the goal of hardware independence.
    - This method adds considerable re-work to the IL client, which has an impact on the re-usability of the IL client on multiple hardware platforms.

Although resource management is not fully addressed in OpenMAX IL API version 1.2, “hooks” for resource management have been put in place in the form of behavioral rules,

component priorities, and a resource management-related component state. These "hooks" lay the groundwork for full-fledged resource management in later versions of the OpenMAX IL API.

Before proceeding further, the terms resource management and policy are defined for the benefit of the discussion that follows:

- *A Resource manager* is responsible for controlling the access of components to a resource. It is aware of how much of each resource any component is using, and how much is available. It will grant, and pre-empt, access to resources based on availability, and the component priorities.
- *Policy* is the part of the IL client that sets component group identifiers and priorities based on the current use case.

2.19.2 Example Architecture

Figure 2-15 shows a high-level architecture diagram of an exemplar OpenMAX IL-based system. In this example, a multimedia framework with a policy manager exists between the applications and the IL layer. This exemplar system also has multiple hardware platforms that are used by different OpenMAX IL components and that are managed by multiple hardware vendor-specific resource managers. But this system would work just as well with a single centralized resource manager.

This example architecture is used as a background for the following discussion on component priorities, behavioral rules and hardware-specific resource managers. It is to be noted, however, that this discussion applies to any OpenMAX IL-based architecture.

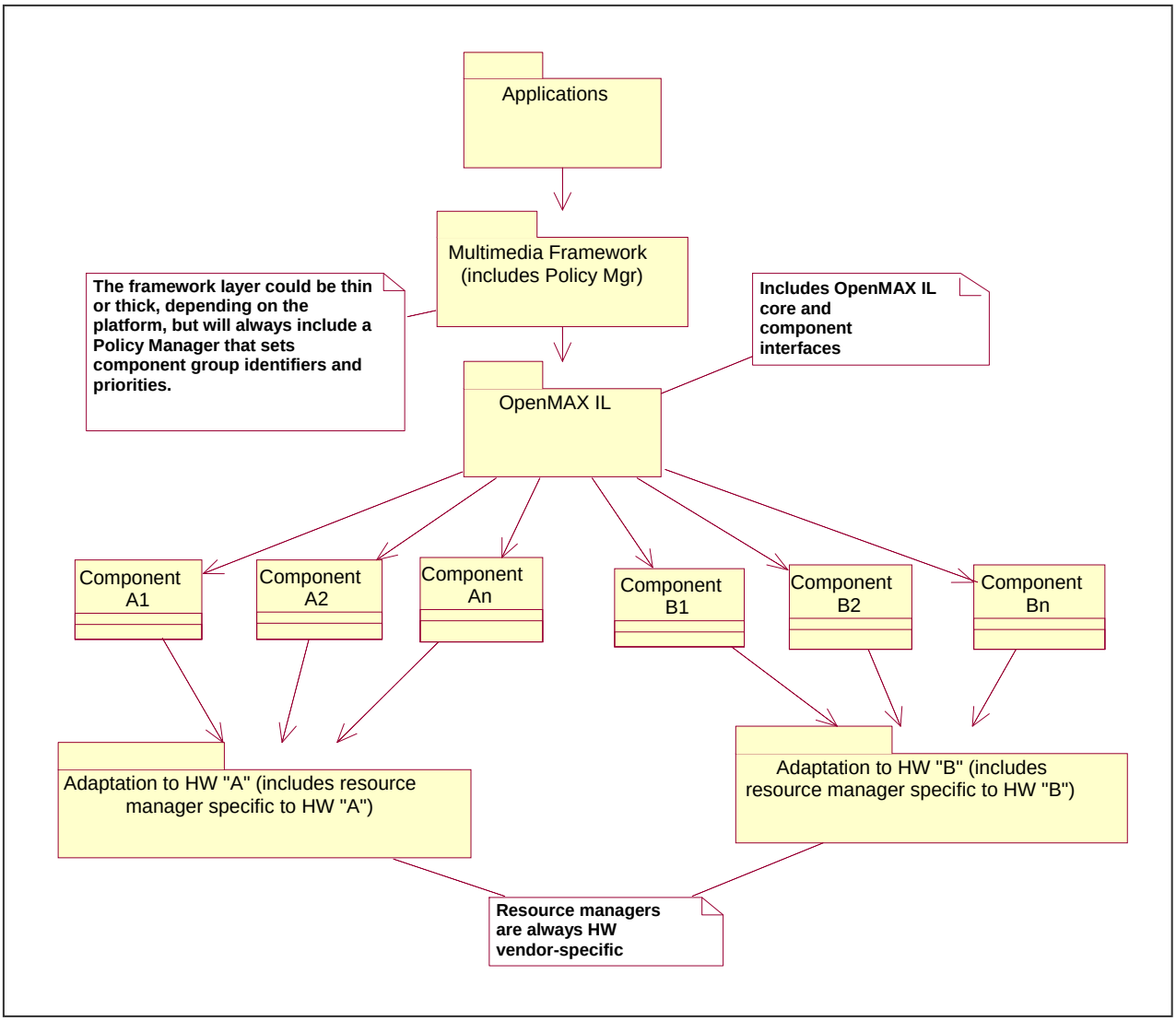


Figure 2-14. Example Architecture

To ensure consistent component behavior in case of resource conflicts, a common definition of component priority and a set of behavioral rules are needed.

2.19.3 Component Priorities

Each IL component has a priority value (an `OMX_U32` integer) that the IL client sets.

A descending order of priority is chosen with 0 denoting the highest priority. The following tie-breaking rule also applies: *When comparing components with the same priority, components that have acquired the resource most recently should be deemed to be of higher priority than components that have had the resource longer.*

IL components may also be assigned a group priority by the IL client. Any component sharing the same group ID maintains the same group priority.

2.19.4 Behavioral Rules

The following behavior is defined on the IL layer:

- The `OMX_ErrorInsufficientResources` error is sent by a component that attempts to go to `OMX_StateIdle` when there are insufficient resources and sufficient resources cannot be freed by preempting lower priority components.
- A component is not aware that preemption is occurring when it tries to go to `OMX_StateIdle`, and the resources it requires need to be freed by preempting lower priority components.
- When a component has resources which need to be preempted, it will send the `OMX_ErrorResourcesPreempted` error to the IL client as it moves from `OMX_StateExecuting` or `OMX_StatePause` to `OMX_StateIdle`. The component will send the `OMX_ErrorResourcesLost` error to the IL client as it moves from `OMX_StateIdle` to `OMX_StateLoaded` once the resources are released.
- In cases where the IL client wants to know when the stream associated with the component can be resumed or started, the IL client shall request to be notified when resources are available. This occurs by putting the component into the `OMX_StateWaitForResources` state. When the resources become available, the component automatically goes to `OMX_StateIdle`. When the client receives the notification that the component is in `OMX_StateIdle`, it can try to move the rest of the components in that chain to `OMX_StateIdle` as well. This automatic movement to `OMX_StateIdle` ensures that in cases where multiple IL clients are waiting for the same resource, the IL client can resume or start the stream as soon as the resource is available. If the component were to automatically move just to `OMX_StateLoaded`, then another IL client could grab that resource first.

These behavioral rules are intended to cover only the interactions between the IL client(s) and the IL components.

2.19.5 Hardware Vendor-Specific Resource Manager

To implement the behavioral rules, a hardware vendor-specific resource manager may exist and perform the following functions:

- Implement and manage the wait queue(s).
- Keep track of available resources.
- Keep track of each component that has resources and which resources they are using.
- Notify a component or multiple components that they need to give up their resources when a higher priority component requests the resource.

- Notify the highest priority component waiting for a resource when the resource is available.

The actual interactions between the components and the hardware vendor-specific resource manager(s) are vendor-specific and outside the scope of this document. Section 3 provides more details of the parameter structures and use cases related to priority and resource management.

2.19.6 Component Suspension

When a component lacks sufficient resources to process data it may elect to suspend itself as a means to enable more optimal dynamic resource management. Component suspension addresses two use cases:

1. Component has lost an essential resource and the resource loss is potentially temporary in nature.
2. Dynamic allocation of essential resources has failed

In the absence of the ability to suspend, the component's only possible reaction to the preemption and loss of a resource is deinitialization via a transition to `OMX_StateIdle` and then `OMX_StateLoaded`. Such deinitialization causes the state of the data stream to be lost because the buffers have to be returned to their allocator. Suspension allows a component to retain its state so that it may be resumed at the point of suspension after some delay.

Suspension is a property of a component when it is in `OMX_StateIdle` or `OMX_StatePause`. Specifically a component is "suspended" when it has lost one or more resources that prevent it from processing data. This means that a component cannot be suspended and be in `OMX_StateExecuting` at the same time (since `OMX_StateExecuting` implies the component will process or output data whenever that data is available). Therefore, a component may be suspended anytime it is normally holding some resources but not seeking to process data, namely when in `OMX_StateIdle` or `OMX_StatePause`.

Component suspension requires no new component states but adds one new component-initiated state transition, namely a transition from `OMX_StateExecuting` to `OMX_StatePause` which an executing component performs on itself upon suspension. IL client may perform any of the normal state transitions on a suspended component with the following exception: a client may not transition a suspended component to `OMX_StateExecuting`. Any attempt to do so will fail and return the `OMX_ErrorComponentSuspended` error.

2.20 Content Pipes

IL components may leverage content piping to synchronously pull in or push out content (e.g. a filestream) from a source or destination abstracting the platform implementation specifics of the source or destination (e.g. local file, remote file, broadcast, etc). A

content pipe is an object that provides content access by implementing the data access abstraction interface defined in the content pipe structure.

The content pipe interface includes functions for conventional content manipulation including:

- opening, closing, and creating content
- seeking to a particular position in the content
- getting the position in the content
- reading data from the current position
- writing data to the current position

2.21 File Parsing

OpenMAX IL defines both standard container format demuxers and the mechanisms to facilitate file parsing functionality in such components. These include means:

- For a component to indicate whether or not it successfully detected and supports the datastream format it was given.
- For a component to inspect and select the streams available on each of the components output ports (when there are multiple alternative streams).
- For the IL client to traverse, extract, and filter the metadata a component captures from a data stream.

2.22 Video Decoder Error Mapping

A video decoder component has the ability to inform the IL client of any macroblock (MB) errors it encounters while decoding the stream. The client may query the component for a map of the MB errors it has encountered at any time via a dedicated parameter.

One potential use for this functionality is the Video Telephony use case where the video terminal at one end of the connection generates an encoded bitstream for a remote video terminal. The encoded bitstream might get corrupted during transmission resulting in MB errors when the remote terminal receives and decodes it. An application that can communicate with both may extract the MB error map at the decoding terminal and transmit it to the encoding terminal allowing it to refresh the macroblocks in error with intra macroblocks in a subsequent encoded frame.

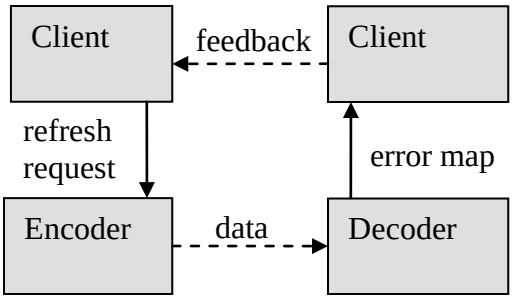


Figure 2-15. Example Use Case for Error Mapping

2.23 Buffer Payload Additional Information

Depending on buffer payload types and component requirements, a need may arise where additional supporting information will need to be appended to the end of the buffer to further process the buffer payload content within the next component.

For instance, video deblocking algorithms require macroblock level quantization information in order to perform the deblocking process on the video content.
The existence of additional buffer payload information shall be identified via the “extra data” buffer flag within the buffer header structure, which is described in section 3.1.3.7 — OMX_BUFFERHEADERTYPE.

This additional buffer payload information applies to the first new logical unit in the buffer. Thus, given the presence of multiple logical units in a buffer, the “extra data” flag applies to the logical unit whose starting boundary occurs first in the buffer. Subsequent logical units in a buffer don’t have explicit “extra data”. If explicit “extra data” are required on every logical unit, one or less logical unit should be included in each buffer.

2.23.1 Buffer Data Formatting

When extra data is present, the data attributes like type and size are identified by a corresponding data structure, immediately following the buffer payload and preceding the actual data. Multiple types of extra data may be appended to the end of the normal payload as series of block pairs (supporting data structure and actual data). To terminate this list of extra data sections, a further data structure should be included in the buffer which indicates that this is the terminating item. For more details see Section 4.2.32.

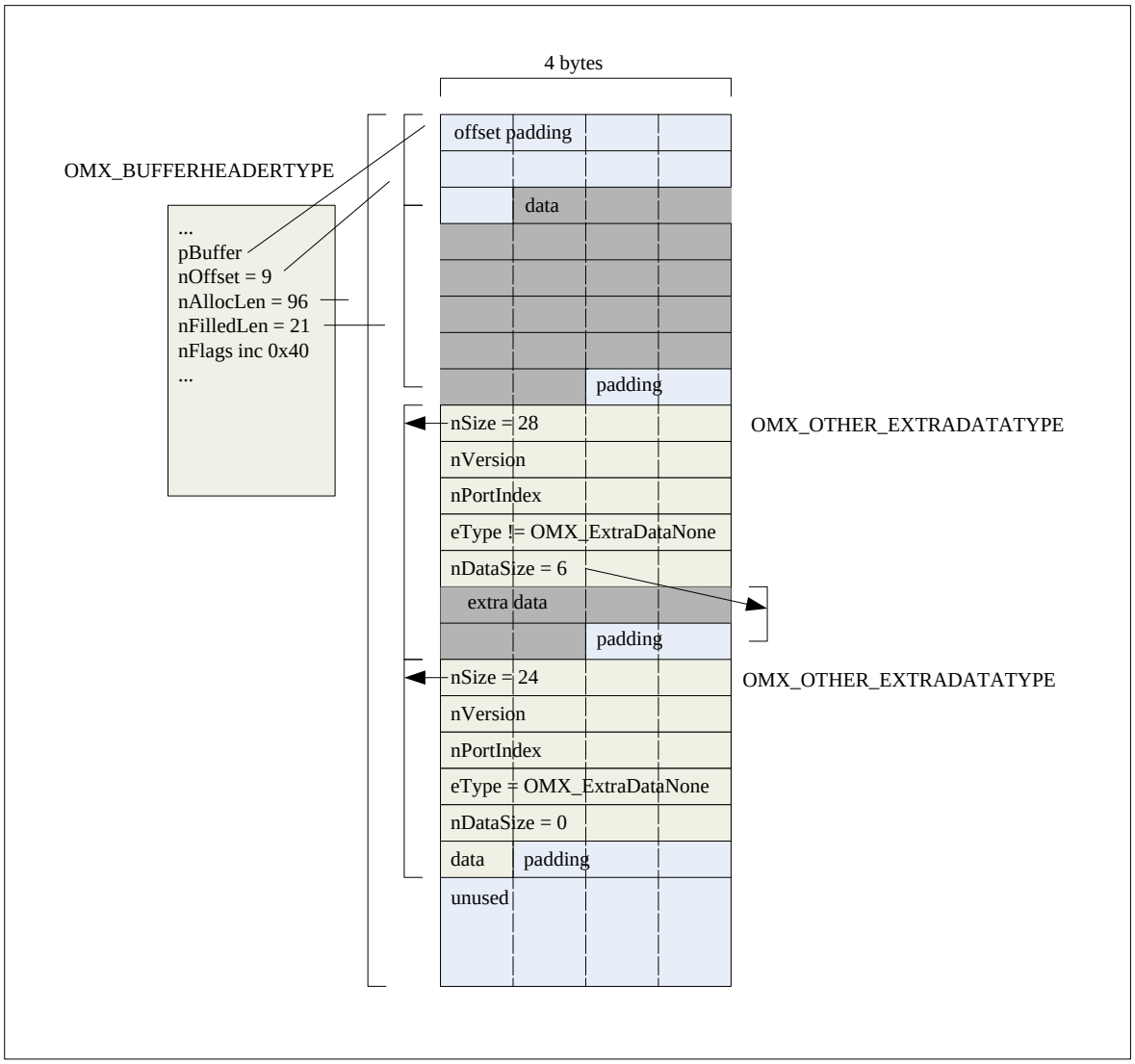


Figure 2-16. Formatting of Extra Buffer Data

2.24 Endianness

The endianness used in the implementation of OpenMAX IL API data structures shall obey the endianness of the platform on which the IL client is running. This requirement includes interfaces used by the IL client and interfaces between components (e.g. functions executed exclusively between two tunneling components). The OpenMAX IL implementation is responsible for any endianness conversions inherent in supporting this requirement; any such conversions are transparent to the IL client and to components using the same endianness as the IL client.

3 OpenMAX Integration Layer Control API

The OpenMAX Integration Layer API allows integration layer clients to control multimedia components in the audio, video and image domains. An "other" domain is also included to provide for extra functionality, such as audio-video (A/V) synchronization. The user of the OpenMAX Integration Layer API is usually a multimedia framework. In the rest of this document, the user of the OpenMAX Integration Layer API will be referred to as the IL client.

The OpenMAX Integration Layer API is defined in a set of header files, namely:

- `OMX_Types.h`: Data types used in the OpenMAX IL API
- `OMX_Core.h`: OpenMAX IL core API
- `OMX_Component.h`: OpenMAX IL component API
- `OMX_Audio.h`: OpenMAX IL audio domain data structures
- `OMX_IVCommon.h`: OpenMAX IL structures common to image and video domains
- `OMX_Video.h`: OpenMAX IL video domain data structures
- `OMX_Image.h`: OpenMAX IL image domain data structures
- `OMX_Other.h`: OpenMAX IL other domain data structures (includes A/V synchronization)
- `OMX_Index.h`: Index of all OpenMAX IL-defined data structures
- `OMX_RoleNames.h`: OpenMAX IL standard role names as defined macros

This section describes how the OpenMAX IL core and OpenMAX IL components are configured for operation.

First, the OpenMAX IL data types are introduced. Next, the methods of the OpenMAX IL core are described. The methods that components implement are discussed in section 3.2.3. Finally, section 3.4 shows calling sequences for a few meaningful operations, including component initialization, normal data flow, data tunnel setup, and data flow in the presence of data tunneling. Such sequence diagrams aim at describing the dynamic interactions between the IL client, the IL core, and the OpenMAX IL components.

When documenting functions, the following convention is used for function parameters:

- <param_name> [in] specifies an input parameter, which is set by the function caller and read by the function implementation.
- <param_name> [out] specifies an output parameter, which is set by the function implementation and passed back to the caller. When the function returns, the caller can read the new value of the parameter, which is passed as a reference.

- <param_name> [inout] specifies an input/output parameter, which the function caller can set. The function implementation can modify the parameter before returning it back to the function caller.

This parameter classification can also be found in the OpenMAX IL header files, where the null macros `OMX_IN`, `OMX_OUT` and `OMX_INOUT` are defined. `OMX_IN` corresponds to the function parameter <param_name> [in]. `OMX_OUT` corresponds to the function parameter <param_name> [out], and `OMX_INOUT` corresponds to the function parameter <param_name> [inout].

3.1 OpenMAX IL Types

3.1.1 Enumerations

Six enumerations are defined in OMX_Core.h:

- `OMX_ERRORTYPE` is returned by each function defined in the OpenMAX Integration Layer API (see section 3.1.1.3).
- `OMX_COMMANDTYPE` includes the possible commands that an IL client can send to an OpenMAX IL component (see section 3.1.1.1).
- `OMX_EVENTTYPE` includes events that can be generated inside an OpenMAX IL component and that are passed to the IL client through a callback function (see section 3.1.1.4).
- `OMX_BUFFERSUPPLIERTYPE`, which is described section 3.1.1.5.
- `OMX_STATETYPE`, which is described in section 3.1.1.2.
- `OMX_EXTRADATATYPE`, which describes the format of extra data carried in data buffers (see section 4.2.32).

3.1.1.1 OMX_COMMANDTYPE

Table 3-1 represents the possible commands that an IL client can send to an OpenMAX IL component. Since commands are non-blocking, the OpenMAX IL component generates a command completion event via a callback function when the command has completed. Callbacks are defined in a dedicated structure; see section 3.1.3.8.

Table 3-1: OpenMAX IL Commands

Field Name	Description
`OMX_CommandStateSet`	Change the component state
`OMX_CommandFlush`	Flush the queue(s) of buffers on a port of a component
`OMX_CommandPortDisable`	Disable a port on a component
`OMX_CommandPortEnable`	Enable a port on a component
`OMX_CommandMarkBuffer`	Mark a buffer and specify which other component will raise the event mark received

Table 3-2 describes the parameters to be used for each command.

Table 3-2: Command Syntax

Command code	nParam	pCmdData
OMX_CommandStateSet	OMX_STATETYPE – state to transition to	NULL
OMX_CommandFlush	OMX_U32 – target port ID	NULL
OMX_CommandPortDisable	OMX_U32 – target port ID	NULL
OMX_CommandPortEnable	OMX_U32 – target port ID	NULL
OMX_CommandMarkBuffer	OMX_U32 – target port ID	OMX_MARKTYPE* - mark data and target component

3.1.1.2 OMX_STATETYPE

Figure 3-1 illustrates the transitions among states. These can occur as a consequence of the IL client calling OMX_SendCommand(OMX_CommandStateSet, <*state*>), where the new state for the component is passed as a parameter. A transition name surrounded by "<" and ">" brackets indicates that the transition is not triggered by a command sent by the IL client but is a consequence of internal component events.

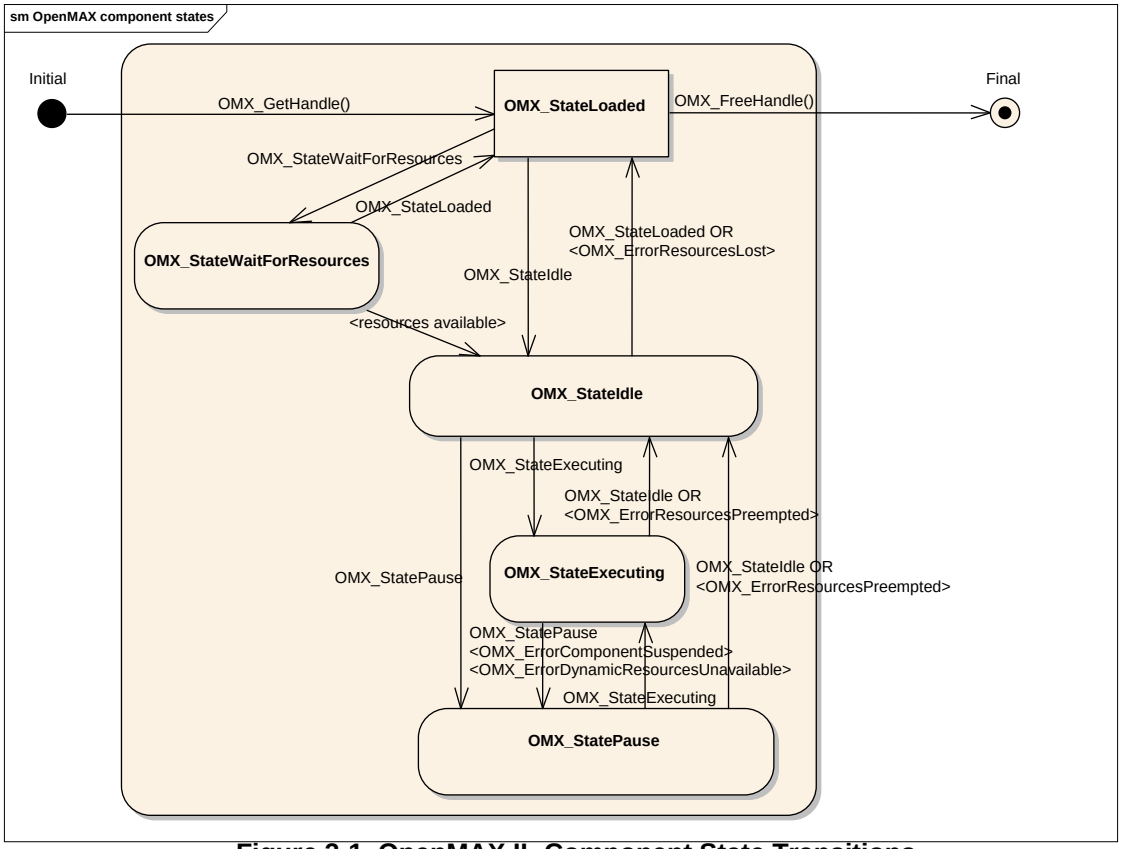


Figure 3-1. OpenMAX IL Component State Transitions

This section describes component states. An IL client commands a component to change states via the `OMX_SendCommand` function using the `OMX_CommandStateSet` command.

Table 3-3 represents the states of an OpenMAX IL component.

Table 3-3: OpenMAX IL Component States

Field Name	Description	Static Resources Allocated	Location of buffer
`OMX_StateLoaded`	Component has been loaded but has no resources allocated.	No	Not available
`OMX_StateIdle`	Component has all static resources but has not transferred any buffers or begun processing data.	Yes	Supplier only
`OMX_StateExecuting`	Component is transferring buffers and is processing data (if data is available).	Yes	Supplier or non-supplier
`OMX_StatePause`	Component data processing has been paused but may be resumed from the point it was paused.	Yes	Supplier or non-supplier
`OMX_StateWaitForResources`	Component is waiting for a resource to become available.	No	Not available

3.1.1.2.1 OMX_StateLoaded

A component is in the `OMX_StateLoaded` state after it has been created via an `OMX_GetHandle` call and before allocation of its resources. In this state, the IL client may modify the component's parameters via `OMX_SetParameter`, set up data tunnels on the component's ports with `OMX_SetupTunnel`, tear down tunnels on the component's port with `OMX_TeardownTunnel`, or transition the component to either the `OMX_StateIdle` state or the `OMX_StateWaitForResources` state.

The IL client may elect to transition a component that is currently in the `OMX_StateLoaded` state into the `OMX_StateWaitForResources` state if, for example, the component failed to acquire all of its static resources on an attempted transition to the `OMX_StateIdle` state.

3.1.1.2.1.1 OMX_StateLoaded to OMX_StateIdle

If the IL client requests a state transition from OMX_StateLoaded to OMX_StateIdle, the component shall acquire all of its static resources, including buffer headers for all enabled ports, before completing the transition. The component does not acquire buffer headers for any disabled ports. Buffer headers are allocated through calls to `OMX_UseBuffer` or `OMX_AllocateBuffer`.

The number of buffers and buffer headers used on a port is specified in its port definition (see `OMX_IndexParamPortDefinition`), which defaults to the minimum (specified in the same structure) but which may be modified by the supplier before the sequence of `OMX_UseBuffer` and `OMX_AllocateBuffer` calls via a call to `OMX_SetParameter`.

IL allows the actual buffers to be pre-announced during the transition from OMX_StateLoaded to OMX_StateIdle. This is achieved by calling `OMX_UseBuffer` with a non-NULL `pBuffer` pointer. This call establishes a one-to-one relationship between the allocated buffer header and the buffer pointer value. No changes to the pointer value or buffer size are allowed until `OMX_FreeBuffer` is called on the buffer header. Pre-announcing the buffers may result in components being able to utilize platform specific optimizations for accessing the buffer memory area. Note that pre-announcing buffers matches the behavior mandated in previous versions of the standard (OpenMAX IL 1.1.2 and earlier).

When `OMX_UseBuffer` is called with a NULL `pBuffer` pointer, the actual buffer memory area is available to the non-supplier only during execution, (`OMX_StateExecuting` and `OMX_StatePause` states) when `OMX_EmptyThisBuffer` or `OMX_FillThisBuffer` is called. In this case, a buffer allocator port is allowed to change the value of the `pBuffer` pointer and the `nAllocLen` field in the buffer header each time the buffer header is passed to the non-supplier. This mode allows flexibility in selecting the buffering scheme.

The caller of the `OMX_UseBuffer` macro shall pre-announce either all the buffers or none of the buffers on one port; mixing the two behaviors is prohibited.

For a port connected to the IL client, the IL client may alternatively direct the port to perform the buffer allocations via `OMX_AllocateBuffer` calls on the port. For each port, the IL client shall exclusively use `OMX_UseBuffer` or `OMX_AllocateBuffer`; mixing the two behaviors is prohibited. When `OMX_AllocateBuffer` is used, the buffer allocator (i.e., the component) is allowed to change the value of the `pBuffer` pointer and `nAllocLen` field in the buffer header each time the buffer header is passed to the IL client.

3.1.1.2.2 OMX_StateIdle

In the `OMX_StateIdle` state, the component is ready to be used, meaning that all necessary static resources have been properly allocated. However, the suppliers retain all their buffers, and no buffer exchange or processing is taking place. Thus, if this state is entered from an `OMX_StateExecuting` or `OMX_StatePause` state, the component shall have returned all buffers it was processing to their respective suppliers. The IL client may transition the component to any states other than the `OMX_StateWaitForResources` state.

3.1.1.2.2.1 OMX_StateIdle to OMX_StateLoaded

On a transition from `OMX_StateIdle` to `OMX_StateLoaded`, each buffer supplier shall call `OMX_FreeBuffer` on the non-supplier port for each buffer. If the supplier allocated the buffer, it shall free the buffer after calling `OMX_FreeBuffer`. If the non-supplier port allocated the buffer, it shall free the buffer upon receipt of an `OMX_FreeBuffer` call. Furthermore, a non-supplier port shall always free the buffer header upon receipt of an `OMX_FreeBuffer` call. When all of the buffers have been removed from the component, the state transition is complete. If the transition was initiated by `OMX_SendCommand`, the component indicates completion via an `OMX_EventCmdComplete` callback event. Alternatively, the component raises an `OMX_ErrorResourcesLost` error callback.

3.1.1.2.2.2 OMX_StateIdle to OMX_StateExecuting

This transition is disallowed when the component is suspended. If the IL client requests a state transition from `OMX_StateIdle` to `OMX_StateExecuting` and the component is not suspended, the component shall begin transferring and processing data. If the client requests this transition when the component is suspended the component shall raise the `OMX_ErrorComponentSuspended` error via the event callback. For ports that communicate with the IL client, the IL client will initiate buffer transfers via `OMX_EmptyThisBuffer` and `OMX_FillThisBuffer`. Among tunneling ports, any input port that is also a supplier shall transfer its empty buffers to the tunneled output port via `OMX_FillThisBuffer`.

3.1.1.2.3 OMX_StateExecuting

In this state, an OpenMAX IL component is transferring and processing data buffers; the component can therefore not be suspended and in this state. The component shall accept calls to `OMX_EmptyThisBuffer` on its input ports and `OMX_FillThisBuffer` on its output ports. Any port that communicates with the IL client shall call the `EmptyBufferDone` and `FillBufferDone` callbacks to return an empty or full buffer, respectively, back to the IL client. Any tunneling port shall call `OMX_FillThisBuffer` or `OMX_EmptyThisBuffer` on its corresponding tunneled port to return an empty or full buffer, respectively, back to its tunneled port.

In case the buffers were not pre-announced on a port during transition from `OMX_StateLoaded` to `OMX_StateIdle` or when enabling a port, the buffer pointers and size are allowed to change during execution as described in section 3.1.1.2.1.1. The combined size of buffers placed into a non-supplier port's buffer queue shall never exceed the sum of the `nSizeBytes` values used on `OMX_AllocateBuffer` or `OMX_UseBuffer` calls on the port. This is also the maximum allowable size for an individual buffer.

An IL client may transition a component in the `OMX_StateExecuting` state to either the `OMX_StateIdle` state or the `OMX_StatePause` state.

3.1.1.2.3.1 OMX_StateExecuting to OMX_StateIdle

If the IL client requests a state transition from `OMX_StateExecuting` to `OMX_StateIdle`, the component shall return all buffers to their respective suppliers and receive all buffers belonging to its supplier ports before completing the transition. Any port communicating with the IL client shall return any buffers it is holding via `EmptyBufferDone` and `FillBufferDone` callbacks, which are used by input and output ports, respectively. Any non-supplier port shall return all buffers it is holding to the input port or output port it is tunneling with using `OMX_EmptyThisBuffer` or `OMX_FillThisBuffer`, respectively. Likewise, any supplier tunneling port shall wait for all of its buffers to be returned from its tunneled port.

3.1.1.2.3.2 OMX_StateExecuting to OMX_StatePause

A transition from the `OMX_StateExecuting` state to the `OMX_StatePause` state occurs under one of circumstances:

- When the client explicitly requests the transition
- When the component loses a resource required for execution but may be resumed from the point of resource loss if the resource is reacquired later. In this case the component shall execute the transition automatically and issue an error event with the `OMX_ErrorComponentSuspended` error.
- When the component is unsuccessful in an attempt to allocate dynamic resources. In this case the component shall execute the transition automatically and issue an error event with the `OMX_ErrorDynamicResourcesUnavailable` error.
- When a camera component completes a capture and the auto-pause after capture feature has been enabled by the IL client.

3.1.1.2.4 OMX_StatePause

In this state, an OpenMAX IL component is not transferring or processing data but buffers are not necessarily returned to their suppliers. From the `OMX_StatePause` state, execution may be resumed via a transition to `OMX_StateExecuting`, preferably without dropping data. However, if the client requests this transition when the component is suspended the component shall fail the call returning the `OMX_ErrorResourcesSuspended` error. The component shall accept data buffers on all enabled ports, but such buffers will be queued only and not processed. The IL client may transition a component in the `OMX_StatePause` state to `OMX_StateIdle` or `OMX_StateExecuting`. On a transition from `OMX_StatePause` to `OMX_StateIdle`, the component shall return all buffers to their respective suppliers in a manner identical to the `OMX_StateExecuting`-to-`OMX_StateIdle` transition described in section 3.1.1.2.3.1.

3.1.1.2.5 OMX_StateWaitForResources

In this state, the component is waiting for one or more of its required resources to become available. This state is related to resource management. The assumption is that one or more hardware-specific resource managers exist on the platform to handle available resources. The interaction among OpenMAX IL components and resource managers is outside the scope of this specification.

If a component in the `OMX_StateLoaded` state fails to enter the `OMX_StateIdle` state because resources other than buffers are insufficient, the IL client may put the component in the `OMX_StateWaitForResources` state if the IL client wants to be notified when the needed resources become available. The IL client may command the component to discontinue waiting for resources by transitioning it from the `OMX_StateWaitForResources` state to the `OMX_StateLoaded` state. If a component in the `OMX_StateWaitForResources` state acquires all the resources upon which it is waiting, it shall initiate a transition to the `OMX_StateIdle` state.

3.1.1.2.5.1 OMX_StateWaitForResources to OMX_StateIdle

When a component initiates a transition from the `OMX_StateWaitForResources` state to the `OMX_StateIdle` state, it shall communicate the initiation of this transition to the IL client via an `OMX_EventResourcesAcquired` event. When the IL client receives the `OMX_EventResourcesAcquired` event, it shall call `OMX_UseBuffer` and `OMX_AllocateBuffer` in the manner of a transition from `OMX_StateLoaded` to `OMX_StateIdle`. Likewise, the component cannot complete its transition to `OMX_StateIdle` until it acquires all of its static resources, including buffers.

3.1.1.3 OMX_ERRORTYPE

The `OMX_ERRORTYPE` enumeration shown in Table 3-4 defines the standard OpenMAX IL errors. These errors should cover most of the common failure cases. However, vendors are free to add additional error messages of their own as long as they follow these rules:

- Vendor error messages shall be in the range of 0x90000000 to 0x9000FFFF.
- Vendor error messages shall be defined in a header file provided with the component. No error messages are allowed that are not defined.

Table 3-4: OpenMAX IL Error Codes

Field Name	Description
`OMX_ErrorNone`	The function returned successfully.
`OMX_ErrorInsufficientResources`	There were insufficient resources to perform the requested operation.
`OMX_ErrorUndefined`	There was an error but the cause of the error could not be determined. When received, the IL client shall treat this error as critical.

Field Name	Description
OMX_ErrorInvalidComponentName	The component name string was invalid.
OMX_ErrorComponentNotFound	No component with the specified name string was found.
OMX_ErrorBadParameter	One or more parameters were invalid.
OMX_ErrorNotImplemented	The requested function is not implemented.
OMX_ErrorUnderflow	A buffer with new data was not available when it was needed : e.g. component processing realtime data is not being supplied data in time, the component is underflowing.
OMX_ErrorOverflow	An empty buffer was not available when it was needed, data loss may be occuring : e.g. component processing realtime data did not have an empty buffer to fill with new incoming data, the component is overflowing with data.
OMX_ErrorHardware	The hardware failed to respond as expected. When received, the IL client shall treat this error as critical.
OMX_ErrorStreamCorrupt	The stream is found to be corrupt. OMX IL components processing coded data (typically decoders) may have the ability to detect corruption in the data stream. Also they may have the ability to detect missing frames and perform error concealment. Such components should report these errors to the client using this error code on a frame basis. Note that the components shall continue normal operation and continue to output data.
OMX_ErrorStreamCorruptStalled	The stream is found to be corrupt enough to temporarily stop data output. OMX IL components processing coded data (typically decoders) may have the ability to detect corruption in the data stream. Such components should report these errors to the client using this error code on a frame or buffer basis. The component will stop outputting data but will remain in OMX_StateExecuting and continue to (try to) process the corrupted data. It is entirely possible for the component to resume data output at some point in the future, if the corruption in the stream were to dissipate. This error provides the IL client with the option of either waiting for the stream corruption to dissipate or to take some other action it deems appropriate for that use case.

Field Name	Description
OMX_ErrorStreamCorruptFatal	The corruption is such that the component is unable to continue processing the stream even if the corruption were to cease. IL client intervention is required, as the use-case cannot continue. The component shall output no more data on the affected port, but will remain in OMX_StateExecuting. The component shall only report this error once on each port, although the error may be sent again once the port has been disabled or the component transitioned back to loaded state.
OMX_ErrorPortsNotCompatible	Ports being set up for tunneled communication are incompatible.
OMX_ErrorResourcesLost	Resources allocated to a component in the OMX_StateIdle state have been lost, which has resulted in the component returning to the OMX_StateLoaded state.
OMX_ErrorNoMore	No more indices can be enumerated.
OMX_ErrorVersionMismatch	The component detected a version mismatch.
OMX_ErrorNotReady	The component is not ready to process the call at this time.
OMX_ErrorTimeout	A timeout occurred where the component was unable to process the call in a reasonable amount of time.
OMX_ErrorSameState	The component tried to transition into the state that it is currently in.
OMX_ErrorResourcesPreempted	Resources allocated to a component in the OMX_StateExecuting or OMX_StatePause states have been pre-empted, causing the component to return to the OMX_StateIdle state.
OMX_ErrorIncorrectStateTransition	A state transition was attempted that is not allowed.
OMX_ErrorIncorrectStateOperation	A command or method was attempted that is not allowed during the present state.
OMX_ErrorUnsupportedSetting	One or more values encapsulated in the parameter or configuration structure are unsupported.
OMX_ErrorUnsupportedIndex	The parameter or configuration indicated by the given index is unsupported.
OMX_ErrorBadPortIndex	The port index that was supplied is incorrect.
OMX_ErrorPortUnpopulated	The port has lost one or more of its buffers and is thus unpopulated.

Field Name	Description
OMX_ErrorComponentSuspended	Component suspended due to temporary loss of resources.
OMX_ErrorDynamicResourcesUnavailable	Component suspended due to inability to acquire dynamic resources.
OMX_ErrorMbErrorsInFrame	Errors detected in frame.
OMX_ErrorFormatNotDetected	OMX IL components performing parsing when reading input buffers or content pipes have the ability to check correct formatting of input data. Such components should report this error to the client (in the form of an OMX_EventError event passed via the EventHandler callback) when it cannot parse or determine the format of the given datastream. This reporting is performed only once in case of file parsing error. In other cases, it is performed on every data unit (e.g. frame) formatting error.
OMX_ErrorSeperateTablesUsed	Attempting to query for single Chroma table when separate quantization tables are used for the Chroma (Cb and Cr) coefficients
OMX_ErrorTunnelingUnsupported	Tunneling is not supported by the component
OMX_ErrorInvalidMode	Attempting to apply a setting while in an invalid mode.
OMX_ErrorPortsNotConnected	OMX_TeardownTunnel has been called with two ports that are not currently connected together.
OMX_ErrorContentURINotSpecified	A content URI has not been specified by the IL client. This error is signaled by the component via EventHandler.
OMX_ErrorContentURIError	SetParameter or SetConfig APIs may return this error when the content URI provided is invalid. Alternatively, if an error occurred while accessing the resource identified by a valid content URI, this error shall be signaled to the IL client using the EventHandler callback.
OMX_ErrorCommandCanceled	A command has been canceled.

3.1.1.4 OMX_EVENTTYPE

The OMX_EVENTTYPE enumeration shown in Table 3-5 includes the event types that an OpenMAX IL component can generate. Section 3.1.3.9 describes events that the OpenMAX IL component generates and passes to the IL client by means of the callback

mechanism. Events have associated parameters that are also passed in the callback. These are described in detail in Section 3.1.3.9.1.

Table 3-5: OpenMAX IL Event Types

Field Name	Description
`OMX_EventCmdComplete`	Component has completed the execution of a command.
`OMX_EventError`	Component has detected an error condition.
`OMX_EventMark`	A buffer mark has reached the target component, and the IL client has received this event with the private data pointer of the mark.
`OMX_EventPortSettingsChanged`	Component has changed port settings. For example, the component has changed port settings resulting from bit stream parsing.
`OMX_EventBufferFlag`	The event that a component sends when it detects the end of a stream.
`OMX_EventResourcesAcquired`	The component has been granted resources and is transitioning from `OMX_StateWaitForResources` to `OMX_StateIdle`.
`OMX_EventComponentResumed`	The component has been resumed (i.e. no longer suspended) due to reacquisition of resources.
`OMX_EventDynamicResourcesAvailable`	The component has acquired previously unavailable dynamic resources.
`OMX_EventPortFormatDetected`	The component has detected a supported media container format.
`OMX_EventIndexSettingChanged`	The component has reported a settings changed associated with an index on its port.
`OMX_EventPortNeedsDisable`	Component requests the IL client to disable one of its ports.
`OMX_EventPortNeedsFlush`	Component requests the IL client to flush one of its ports.

3.1.1.4.1 OMX_EventCmdComplete

A component generates an `OMX_EventCmdComplete` event as soon as a command sent by the IL client has completed its execution, or a component-initiated state transition has occurred. In case of a component state change (whether initiated by the IL client or the component), the new state that the component has entered (or unsuccessfully

attempted to enter) is returned as an event parameter. `OMX_EventCmdComplete` event carries an error code as a parameter to indicate whether the command has completed successfully or not. In case of a successfully completed command, the component returns `OMX_ErrorNone` as a parameter, while `i`n the case of a failure to complete the command, the component returns an appropriate error code value as a parameter.

3.1.1.4.2 OMX_EventError

A component generates the `OMX_EventError` event when the component detects an error condition; the type of error detected is returned as an event parameter and will use values defined in `OMX_ERRORTYPE`. A component shall send the following errors via `OMX_EventError`:

- A component sends the `OMX_ErrorResourcesPreempted` error if the component transitions from `OMX_StateExecuting` or `OMX_StatePause` to `OMX_StateIdle` due to the loss of a resource.
- A component sends the `OMX_ErrorResourcesLost` error if the component transitions from `OMX_StateIdle` to `OMX_StateLoaded` due to the loss of a resource.

3.1.1.4.3 OMX_EventMark

A component may generate the `OMX_EventMark` event when it receives a marked buffer. When a component receives a buffer, it shall compare its own pointer to the `pMarkTargetComponent` field contained in the buffer. If the pointers match, then the component shall send a mark event including `pMarkData` as a parameter, immediately after the component has finished processing the buffer. The IL client can use the mark event to measure the propagation delay of a data buffer through a chain of components, or to notify a component that a particular buffer has reached the given destination.

3.1.1.4.4 OMX_EventPortSettingsChanged

A component generates the `OMX_EventPortSettingsChanged` event after it changes the values of the configuration structures of its port, the event is sent to the IL client. Upon receiving the event notification, the IL client may use `OMX_GetParameter`() or `OMX_GetConfig()` – whichever is appropriate – to retrieve the updated port settings.

This event may be issued within any component state and from ports that are disabled.

A component may decide to change its configuration structures associated with its port for a variety of reasons:

- During the course of processing a stream, it may discover new embedded stream properties within in.

  For example, a video decoder may discover new frame size and frame rate embedded within the bit stream it is processing.

- Some components require port settings that are common between their input and output ports. These components are not able to perform any conversions between these common settings, an attempt by an IL client to apply a configuration setting on the input port that differs from the current setting on the output port may trigger an immediate update of the output port settings to ensure the common settings are always maintained between the ports.

  For example, attempting to apply a new sample rate setting on an audio decoder input port that is not capable of performing sample rate conversion between its output port shall trigger an output port settings change.

When an `OMX_EventPortSettingsChanged` event is emitted from a component's output port, the component shall cease transferring data on that port until the IL client takes an action to commence it again. Ceasing the transfer of buffers is required in order to coordinate the updated port settings with any downstream component and possibly facilitate the re-allocation of new port resources (e.g. port buffers). In order to commence the emission of data again on the output port, the IL client shall disable and re-enable the port – this action will also give the component port the opportunity to reallocate any new buffer requirements associated with the port settings change between the IL client or its tunneled port.

In cases when an input port emits the event and the port settings changed is associated with `OMX_IndexParamPortDefinition`, the component may need to have the port disabled and re-enabled in order to reallocate any new buffer requirements associated with the settings change. To prevent the loss of any input data, the component issuing the `OMX_EventPortSettingsChanged` event on its input port should buffer all input port data that arrives between the emission of the `OMX_EventPortSettingsChanged` event and the arrival of the command to disable the input port. For all other parameter indexes reported via the `OMX_EventPortSettingsChanged` event for an input port, the IL client is not required to disable and re-enable the port(s).

3.1.1.4.5 OMX_EventBufferFlag

A component generates the `OMX_EventBufferFlag` event when an output port emits a buffer with the `OMX_BUFFERFLAG_EOS` flag set in the `nFlags` field. The `nData1` field of `EventHandler` specifies the value of the output port's `portindex` field. The

nData2 field of EventHandler specifies the unaltered nFlags field containing the end-of-stream (EOS) flag.

If a component does not propagate a stream further (e.g., the component is an audio or video sink), then the component shall send an OMX_EventBufferFlag event for that stream when it has finished processing a buffer with OMX_BUFFERFLAG_EOS set. The nData1 field of EventHandler specifies the input port that received the buffer. The nData2 field of EventHandler specifies the unaltered nFlags field containing the EOS flag.

3.1.1.4.6 OMX_EventResourcesAcquired

A component generates the OMX_EventResourcesAcquired event when it is in the OMX_StateWaitForResources state, and the resource manager detects that the needed resources are available. When the component generates this event, it is ready to change state into the OMX_StateIdle, once buffers are allocated and assigned to its ports.

3.1.1.4.7 OMX_EventComponentResumed

A suspended component generates the OMX_EventComponentResumed event when the resources it had lost have been reacquired. Upon receipt of this event the component is no longer suspended and the client may attempt to transition a suspended component into OMX_StateExecuting.

3.1.1.4.8 OMX_EventDynamicResourcesAvailable

A suspended component generates the OMX_EventDynamicResourcesAvailable event when some dynamic resource it was formerly unable to allocate has become available. Upon receipt of this event the component is no longer suspended and the client may attempt to transition it into OMX_StateExecuting.

3.1.1.4.9 OMX_EventPortFormatDetected

A component, such as a media container demuxer, generates the OMX_EventPortFormatDetected event when it detects the content format. By issuing this event the component informs that IL client that it is able to process the content. An example of this event usage is available in section 3.4.6.

3.1.1.4.10 OMX_EventIndexSettingChanged

A component generates the OMX_EventIndexSettingChanged event when it detects parameter or config setting change, if the client has previously requested notifications for this index using the controls defined in Section 4.6.4. Upon receipt of this event the IL client may use OMX_GetParameter() or OMX_GetConfig() as appropriate to retrieve the new setting value.

3.1.1.4.11 OMX_EventPortNeedsDisable

A component generates the `OMX_EventPortNeedsDisable` event when it is processing a disable port command on a port doing buffer sharing and it needs the IL client to disable the sharing port in order to be able to complete the current disable port command.

3.1.1.4.12 OMX_EventPortNeedsFlush

A component generates the `OMX_EventPortNeedsFlush` event when is processing a flush command on a port doing buffer sharing and it needs the IL client to flush the sharing port in order to be able to complete the current flush command.

3.1.1.5 OMX_BUFFERSUPPLIERTYPE

The `OMX_BUFFERSUPPLIERTYPE` enumeration shown in Table 3-6 specifies the port in the tunnel that is the supplier port. A buffer supplier port either may allocate its buffers or reuse buffers provided by another port within the same component.

Table 3-6: OpenMAX IL Buffer Supplier Type For Tunnel Setup

Field Name	Description
`OMX_BufferSupplyUnspecified`	The port supplying the buffers is unspecified, or no supplier is preferred.
`OMX_BufferSupplyInput`	The input port supplies the buffers.
`OMX_BufferSupplyOutput`	The output port supplies the buffer.

3.1.2 Data Types

Table 3-7 identifies the data types available in the specification.

Table 3-7: OpenMAX IL Data Types

Data Type	Description
`OMX_U8`	An 8 bit unsigned quantity that is byte aligned.
`OMX_S8`	An 8 bit signed quantity that is byte aligned.
`OMX_U16`	A 16 bit unsigned quantity that is 16 bit word aligned.
`OMX_S16`	A 16 bit signed quantity that is 16 bit word aligned.
`OMX_U32`	A 32 bit unsigned quantity that is 32 bit word aligned.
`OMX_S32`	A 32 bit signed quantity that is 32 bit word aligned.
`OMX_U64`	A 64 bit unsigned quantity that is 64 bit word aligned.
`OMX_S64`	A 64 bit signed quantity that is 64 bit word aligned.

Data Type	Description

OMX_BU32	A bounded 32 bit unsigned quantity.
	
	`typedef struct OMX_BU32 {`
	`    OMX_U32 nValue;`
	`    OMX_U32 nMin;`
	`    OMX_U32 nMax;`
	`} OMX_BU32;`
	
	`nValue` represents the actual value;
	`nMin` represents the minimum for the value (i.e. `nValue >= nMin`)
	`nMax` represents the maximum for the value (i.e. `nValue <= nMax`)

OMX_BS32	A bounded 32 bit signed quantity.
	
	`typedef struct OMX_BS32 {`
	`    OMX_S32 nValue;`
	`    OMX_S32 nMin;`
	`    OMX_S32 nMax;`
	`} OMX_BS32;`
	
	`nValue` represents the actual value;
	`nMin` represents the minimum for the value (i.e. `nValue >= nMin`)
	`nMax` represents the maximum for the value (i.e. `nValue <= nMax`)

OMX_BOOL	Represents a true (OMX_TRUE) or false (OMX_FALSE) value.
OMX_PTR	Used to pass pointers between the OMX applications and the OMX Core and components.
OMX_STRING	Used to pass "C" type strings between the application and the core and component. This is a pointer to a zero terminated string.
OMX_UUIDTYPE	A very long unique identifier to uniquely identify at runtime. This identifier should be generated by a component in a way that guarantees that every instance of the identifier running on the system is unique.
OMX_ALL	Used to as a wildcard to select all entities of the same type when specifying the index, or referring to an object by an index. (I.e. use OMX_ALL to indicate all N channels). When used as a port index for a config, parameter or command OMX_ALL denotes that the config, parameter or command applies to all ports on the component.
OMX_TICKS	Used to represent time or duration in microseconds. Refer to 6.2.1 for more information.

Data Type	Description
OMX_HANDLETYPE	Defines the component handle as seen by the IL client. The component handle is used by the IL core to access all of the public methods of the component. The component handle also contains pointers to the private data area of the component. The OpenMAX IL core allocates and initializes the component handle with help from the component during the process of loading the component. After the component is successfully loaded, the IL client can safely access any of the public functions of the component via the IL core macros, although some may return an error because the state of the component is inappropriate for the access. The component handle shall not be used directly by the IL client or other IL components to access the public methods of an implementation. The IL core macros shall be used instead.

3.1.3 Structures

This section discusses data structures defined by OpenMAX IL. Data-specific structures are discussed in Section 4. The first two fields of each OpenMAX IL data structure denote the size, `nSize`, of the structure and the version of type `OMX_VERSIONTYPE`, `nVersion`, which is defined in section 3.1.3.3. The entity that allocates an OpenMAX IL structure is responsible for filling in these two values. Hereinafter, definitions for these two common fields are omitted in individual structure definitions.

Table 3-8 lists structures that are associated with an index.

Table 3-8: Indices with their associated structures

OpenMAX IL Indices (OMX_Index.h)	Corresponding OpenMAX IL Structures
OMX_IndexParamPriorityMgmt	OMX_PRIORITYMGMTTYPE
OMX_IndexParamDisableResourceConcealment	OMX_RESOURCECONCEALMENTTYPE
OMX_IndexParamSuspensionPolicy	OMX_PARAM_SUSPENSIONPOLICYTYPE
OMX_IndexParamComponentSuspended	OMX_PARAM_SUSPENSIONTYPE
OMX_IndexParamAudioInit	OMX_PORT_PARAM_TYPE
OMX_IndexParamImageInit	OMX_PORT_PARAM_TYPE
OMX_IndexParamVideoInit	OMX_PORT_PARAM_TYPE
OMX_IndexParamOtherInit	OMX_PORT_PARAM_TYPE
OMX_IndexParamCompBufferSupplier	OMX_PARAM_BUFFERSUPPLIERTYPE
OMX_IndexParamPortDefinition	OMX_PARAM_PORTDEFINITIONTYPE
OMX_IndexConfigTunneledPortStatus	OMX_CONFIG_TUNNELEDPORTSTATUSTYPE

3.1.3.1 OMX_COMPONENTREGISTERTYPE

The OMX_COMPONENTREGISTERTYPE structure is used in the case of static linking of components to the core. The core optionally uses it to load the component and run the specific component initialization functions.

OMX_COMPONENTREGISTERTYPE is defined as follows.

```
typedef struct OMX_COMPONENTREGISTERTYPE
{
  const char           * pName;
  OMX_COMPONENTINITTYPE pInitialize;
} OMX_COMPONENTREGISTERTYPE;
```

3.1.3.1.1 Parameter Definitions

- pName contains the string name of the component and has limit of 128 bytes (including '\0').
- pInitialize contains the pointer to the initialization function of the component. The OMX_COMPONENTINITTYPE type definition is the type of function pointer for the component initialization entry point. The definition is as follows:

```
typedef OMX_ERRORTYPE (* OMX_COMPONENTINITTYPE)(OMX_IN  OMX_HANDLETYPE
hComponent);
```

3.1.3.2 OMX_ComponentRegistered[ ]

Any core that statically links its components shall define this global array containing the list of all registered components in the form of OMX_COMPONENTREGISTERTYPE fields.

3.1.3.3 OMX_VERSIONTYPE

The OMX_VERSIONTYPE type indicates the version of a component or structure. Each structure uses an OMX_VERSIONTYPE field to indicate the OpenMAX IL specification version under which the structure is defined. For OpenMAX IL version 1.0, the specification version is 1.0.R.S with any Revision R and Step S values. For OpenMAX IL version 1.2, the specification version is 1.2.R.S with any Revision R and Step S values. The component structure also includes an OMX_VERSIONTYPE field to indicate a vendor-specific component version.

OMX_VERSIONTYPE is defined as follows.

```
typedef union OMX_VERSIONTYPE
{
    struct
    {
        OMX_U8 nVersionMajor;
        OMX_U8 nVersionMinor;
```

```
        OMX_U8 nRevision;
        OMX_U8 nStep;
    } s;
    OMX_U32 nVersion;


} OMX_VERSIONTYPE;
```

3.1.3.3.1 Parameter Definitions

- `nVersionMajor` identifies the major version number. This byte of the version occurs first.
- `nVersionMinor` identifies the minor version number.
- `nRevision` identifies the revision number.
- `nStep` identifies the step number. This byte of the version occurs last.
- `nVersion` is an alternative way of accessing the version information.

`OMX_VERSION_MAJOR`, `OMX_VERSION_MINOR`, `OMX_VERSION_REVISION`, `OMX_VERSION_STEP` and `OMX_VERSION` defines are available to identify the specification version information.

3.1.3.4 OMX_PRIORITYMGMTTYPE

The IL client may use the `OMX_IndexConfigPriorityMgmt` and `OMX_IndexParamPriorityMgmt` parameters with the `OMX_PRIORITYMGMTTYPE` structure. This structure describes the priority assigned to a set of components. A component group identifies a set of co-dependent components associated with the same feature. All components in the same group share the same group ID and priority. If one component in a group loses resources and stops running, the entire feature they collectively contribute to is lost. In this case, the IL client should transition all of the other components in the same group to `OMX_StateLoaded`. A component that is the only one with a certain `nGroupID` acts atomically.

`OMX_PRIORITYMGMTTYPE` is defined as follows.

```
typedef struct OMX_PRIORITYMGMTTYPE {
      OMX_U32 nSize;
      OMX_VERSIONTYPE nVersion;
      OMX_U32 nGroupPriority;
      OMX_U32 nGroupID;
} OMX_PRIORITYMGMTTYPE;
```

3.1.3.4.1 Parameter Definitions

- `nGroupPriority` is the priority value associated with a group of components. If a parameter of this type is assigned to a component, that component belongs to the group identified with `nGroupID` and has a priority equal to

nGroupPriority. By definition, the value 0 represents the highest priority for a group of components.

The exact mechanism to assign priorities to groups of components is outside the scope of this document.

The group is treated as having the same priority. When the priority of one component in the group is changed, that change effectively applies to all components in the group The IL client shall update each component's priority within the group with the same priority. The suspension of one component in a group does not imply the suspension of all components in that group.

- nGroupID is a unique ID for all components in the same component group.

3.1.3.5 OMX_RESOURCECONCEALMENTTYPE

The IL client may use the OMX_IndexParamDisableResourceConcealment parameter with the OMX_RESOURCECONCEALMENTTYPE structure to enable or disable resource concealment in a component.

The definition of OMX_RESOURCECONCEALMENTTYPE is shown as follows:

```
typedef struct OMX_RESOURCECONCEALMENTTYPE
{
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_BOOL bResourceConcealmentForbidden;
} OMX_RESOURCECONCEALMENTTYPE;
```

3.1.3.5.1 Parameter Definitions

- bResourceConcealmentForbidden is a Boolean that shall disallow the use of resource concealment methods by a component to resolve resource conflicts.

3.1.3.6 Component Suspension Policy

A component lacking sufficient resources to process data may elect to suspend itself to resolve a temporary resource conflict. Component suspension is ideal when the resource loss is temporary in nature or driven by a requirement for additional runtime dynamic resources.
The IL client specifies the suspension policy of a component via a parameter, OMX_IndexParamSuspensionPolicy, where possible suspension policies include:

- Suspension Disabled: The component shall not suspend itself. If a component in OMX_StateExecuting loses resource it shall transition through OMX_StateIdle, into OMX_StateLoaded as part of its resource loss. This shall be the **default** component behavior.
- Suspension Enabled: Upon detection of a temporary loss of resources a component may suspend processing. No state transitions are triggered if

suspension occurs in OMX_StatePause or OMX_StateIdle. If the component is in OMX_StateExecuting when it suspends, it shall transition to OMX_StatePause.

The OMX_PARAM_SUSPENSIONPOLICYTYPE is defined as follows:

```
typedef struct OMX_PARAM_SUSPENSIONPOLICYTYPE {
      OMX_U32 nSize;
      OMX_VERSIONTYPE nVersion;
      OMX_SUSPENSIONPOLICYTYPE ePolicy;
} OMX_PARAM_SUSPENSIONPOLICYTYPE;
```

The parameters for OMX_PARAM_SUSPENSIONPOLICYTYPE are defined as follows.

- ePolicy specifies to the component whether suspension is enabled or disabled, the default value shall be OMX_SuspensionDisabled.

Table 3-9: OMX_SUSPENSIONPOLICYTYPE enumeration

Field Name	Description
OMX_SuspensionDisabled	Suspension disabled
OMX_SuspensionEnabled	Suspension enabled

An IL client may query if the component is suspended using the OMX_IndexParamComponentSuspended parameter. The client can use this suspension status of the component to make decisions on how to proceed when a component is suspended. The IL client may opt to leave the component as-is expecting the suspension to be temporary. The IL client may opt to transition the component to the loaded state, or perform some alternative processing.
The OMX_PARAM_SUSPENSIONTYPE is defined as follows:

```
typedef struct OMX_PARAM_SUSPENSIONTYPE {
      OMX_U32 nSize;
      OMX_VERSIONTYPE nVersion;
      OMX_SUSPENSIONTYPE eType;
} OMX_PARAM_SUSPENSIONTYPE;
```

The parameters for OMX_PARAM_SUSPENSIONTYPE are defined as follows.

- eType specifies the suspension state of the component.

Table 3-10: OMX_SUSPENSIONTYPE enumeration

Field Name	Description
OMX_NotSuspended	Component is not suspended
OMX_Suspended	Component is suspended

3.1.3.6.1 Suspension Due to Pre-emption of Resources

The effect of "suspension" on component implementations is minimal, specifically:

- Upon the loss of one or more resources, a component shall decide between either suspending itself (if it is capable of resumption later and its suspension policy

allows it) or de-initializing itself via `OMX_ErrorResourcesPreempted/Lost` (if it is incapable of resumption later or if its suspension policy disallows suspension).

- In the case of suspension the component shall send the `OMX_ErrorComponentSuspended` error to the IL client. If the component is in `OMX_StateExecuting` the component shall transition itself to `OMX_StatePause`.
- If the component supports suspension, the component shall support the `OMX_IndexParamComponentSuspended` parameter.
- Upon a request to transition to Executing the component shall validate that it is not suspended. If it is suspended, the component shall fail the transition with an `OMX_ErrorComponentSuspended` error.
- Upon reacquisition of resources the component signals the IL client via the `OMX_EventComponentResumed` event. The component remains in `OMX_StatePause` until the IL client resumes the component by transitioning it back to `OMX_StateExecuting`.

Upon the de-allocation of resources, the component shall be aware of which resources have already been de-allocated from a suspension.

3.1.3.6.2 Suspension Due to Unavailable Dynamic Resources

Under certain conditions the size and type of component resources vary within the lifetime of the component. As an example, resource requirements are dependent upon properties of the data stream itself, which are known only after inspection of the stream. This implies a component is in the executing state by which point all static resources shall be allocated.

A component in the executing state may attempt to allocate additional dynamic resources as a result of increased requirements during processing. This dynamic resource allocation is completely transparent to the client except in the case where the component fails to allocate resources while in `OMX_StateExecuting`. Upon failure to allocate resources the component issues an error, `OMX_ErrorDynamicResourcesUnavailable`, and transitions to `OMX_StatePause` if the component suspension policy has been previously enabled by the IL client.

The component upon receiving the dynamic resources issues the event `OMX_EventDynamicResourcesAvailable` to the IL client and remains in `OMX_StatePause`. The component remains in `OMX_StatePause` until the IL client resumes the component by transitioning it back to `OMX_StateExecuting`.

The suspension mechanism follows the case where suspension occurs as a result of preemption with the exception of the errors and events presented to the IL client.

3.1.3.7 OMX_BUFFERHEADERTYPE

Each data buffer has a header associated with it that contains meta-information about the buffer. The IL client shares buffer headers with each port with which it is communicating. Likewise, a connected pair of tunneled ports share buffer headers. If a data buffer is shared across multiple tunnels, each tunnel uses a distinct set of buffer headers.

The definition of the buffer header is shown as follows.

```
typedef struct OMX_BUFFERHEADERTYPE
{
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U8* pBuffer;
    OMX_U32 nAllocLen;
    OMX_U32 nFilledLen;
    OMX_U32 nOffset;
    OMX_PTR pAppPrivate;
    OMX_PTR pPlatformPrivate;
    OMX_PTR pInputPortPrivate;
    OMX_PTR pOutputPortPrivate;
    OMX_HANDLETYPE hMarkTargetComponent;
    OMX_PTR pMarkData;
    OMX_U32 nTickCount;
    OMX_TICKS nTimeStamp;
    OMX_U32 nFlags;
    OMX_U32 nOutputPortIndex;
    OMX_U32 nInputPortIndex;
} OMX_BUFFERHEADERTYPE;
```

3.1.3.7.1 Parameter Definitions

- `pBuffer` is a pointer to the actual buffer where data is stored but not necessarily the start of valid data; for more information, see the description of `nOffset` below.
- `nAllocLen` is the total size of the allocated buffer in bytes, including valid and unused bytes. If `pBuffer` is updated, this field shall be updated accordingly.
- `nFilledLen` is the total size of valid bytes currently in the buffer starting from the location specified by `pBuffer` and `nOffset`. This includes any padding, e.g. the unused bytes at the end of a line of video when stride in bytes is larger than width in bytes.
- `nOffset` is the start offset of valid data in bytes from the start of the buffer. The value of `nOffset` is set by the entity that places data into the buffer (or forwards data in a shared buffer). A pointer to the valid data may be obtained by adding `nOffset` to `pBuffer`.
- `pAppPrivate` is a pointer to an IL client private structure.

- `pPlatformPrivate` is a pointer to a private platform-specific structure. For instance, in the case where the IL client allocates the buffer through the platform's memory manager, this structure may contain information the platform's memory manager associates with the buffer.
- `pOutputPortPrivate` is a private pointer of the output port that uses the buffer. If a buffer header is used on an input port communicating with the IL client, the value of the buffer's `pOutputPortPrivate` is undefined.
- `pInputPortPrivate` is a private pointer of the input port that uses the buffer. If a buffer header is used on an output port communicating with the IL client, the value of the buffer's `pInputPortPrivate` is undefined.
- `hMarkTargetComponent` is the handle of the component that should emit an `OMX_EventMark` event upon processing this buffer. A NULL handle indicates that the buffer carries no mark. The `OMX_CommandMarkBuffer` command provides this handle to the marking component. The marking component, in turn, copies this handle to the marked buffer. Each component that is processing a buffer shall compare its own handle to this handle and if they match it shall emit the mark using the `OMX_EventMark` event, and remove the mark from the buffer.. If the handle of the processing component does not match this field the component shall proagate `hMarkTargetComponent` from an input buffer to its associated output buffer.
- The `pMarkData` pointer refers to IL client-specific data associated with the mark that is sent on `OMX_EventMark` when emitted. Upon receipt of a mark, the IL client may use this data to disambiguate this mark from others. The `OMX_CommandMarkBuffer` command provides this pointer to the marking component. The marking component, in turn, copies this pointer to the marked buffer. A component shall propagate this field from an input buffer to its associated output buffer, unless the mark has matched the component handle and the `pMarkData` has been returned in the `OMX_EventMark` event.
- `nTickCount` is an optional entry that the component and IL client can update with a tick count when they access the component; not all components will update it. The value of `nTickCount` is in microseconds. Since this is a value relative to an arbitrary starting point, `nTickCount` cannot be used to determine absolute time.
- `nTimeStamp` is the presentation timestamp corresponding to the sample starting at the first logical sample boundary in the buffer. Timestamps of successive samples within the buffer may be inferred by adding the duration of the preceding buffer to the timestamp of the preceding buffer. A component should propagate this field from an input buffer to its associated output buffer.
- `nFlags` field contains buffer specific flags, such as the EOS flag. A component should propagate this field from an input buffer to its associated output buffer. The list of flags is as follows:

```
#define OMX_BUFFERFLAG_EOS 0x00000001
#define OMX_BUFFERFLAG_STARTTIME 0x00000002
#define OMX_BUFFERFLAG_DECODEONLY 0x00000004
#define OMX_BUFFERFLAG_DATACORRUPT 0x00000008
#define OMX_BUFFERFLAG_ENDOFFRAME 0x00000010
#define OMX_BUFFERFLAG_SYNCFRAME 0x00000020
#define OMX_BUFFERFLAG_EXTRADATA 0x00000040
#define OMX_BUFFERFLAG_CODECCONFIG 0x00000080
#define OMX_BUFFERFLAG_TIMESTAMPINVALID 0x00000100
#define OMX_BUFFERFLAG_READONLY 0x00000200
#define OMX_BUFFERFLAG_ENDOFSUBFRAME 0x00000400
#define OMX_BUFFERFLAG_SKIPFRAME 0x00000800
```

- OMX_BUFFERFLAG_EOS is set by a source component (e.g. a demuxer) when it has reached the end of the stream content on a particular output port. Some examples of this are:
    - End of a stream within a 3GP file,
    - Camera Component stopping the emission of stream data on its capture port. i.e. OMX_IndexAutoPauseAfterCapture support

  The emission of the OMX_BUFFERFLAG_EOS does not preclude the possibility of subsequent stream content being emitted on the port in response to an IL client command. In the examples above, a port may emit additional stream content when:
    - It receives a seek request to an earlier position earlier in the 3GP file,
    - The Camera Component is requested to start emitting additional content via the capture port.

  Other components forward the OMX_BUFFERFLAG_EOS in a way that is appropriate for their processing.

  OMX_BUFFERFLAG_EOS shall not be emitted in response to a state change command.

- OMX_BUFFERFLAG_STARTTIME is set by the source of a stream (e.g., a de-multiplexing component) on the buffer that contains the starting timestamp for the stream. The starting timestamp corresponds to the first data that should be displayed at startup or after a seek operation.

  The first timestamp of the stream is not necessarily the start time. For instance, in the case of a seek to a particular video frame, the target frame may be an interframe. Thus the first buffer of the stream will be the intraframe preceding the target frame, and the start time will occur with the target frame along with any other required frames required to reconstruct the target intervening.

  The OMX_BUFFERFLAG_STARTTIME flag is directly associated with the buffer timestamp. Thus, the association of the

OMX_BUFFERFLAG_STARTTIME flag to buffer data and its propagation is identical to that of the timestamp.

A clock component client that receives a buffer with the OMX_BUFFERFLAG_STARTTIME flag shall perform an OMX_SetConfig call on its sync port using OMX_IndexConfigTimeClientStartTime and pass the timestamp for the buffer.

- OMX_BUFFERFLAG_DECODEONLY is set by the source of a stream (e.g., a de-multiplexing component) on any buffer that should be decoded but not rendered. This flag is used, for instance, when a source seeks to a target interframe that requires decoding of frames preceding the target to facilitate reconstruction of the target. In this case, the source would emit the frames preceding the target downstream but mark them as decode only.

  The OMX_BUFFERFLAG_DECODEONLY flag is associated with buffer data and propagated in a manner identical to that of the buffer timestamp.

  A component that renders data should ignore all buffers with the OMX_BUFFERFLAG_DECODEONLY flag set.

- OMX_BUFFERFLAG_DATACORRUPT flag is set when the data in the associated buffer is identified as corrupt.

- OMX_BUFFERFLAG_ENDOFFRAME is an optional flag that is set by an output port when the last byte that a buffer payload contains is an end-of-frame. Any component that implements setting the OMX_BUFFERFLAG_ENDOFFRAME flag on an output port shall set this flag for every buffer sent from the output port containing an end-of-frame. No buffer payload can contain data from two separate frames.

  These restrictions enable input ports that receive data from the output port to detect an end-of-frame without requiring additional processing. These restrictions also enable an input port to easily detect if an output port supports this flag by its presence or absence on completion of the first frame.

- OMX_BUFFERFLAG_SYNCFRAME should be set by an output port to indicate that the buffer content contains a coded synchronization frame. A coded synchronization frame is a frame that can be reconstructed without reference to any other frame information. An example of a video synchronization frame is an MPEG4 I-VOP.

  If the OMX_BUFFERFLAG_SYNCFRAME flag is set then the buffer may only contain one frame.

- The OMX_BUFFERFLAG_EXTRADATA is used to identify the availability of additional information after the buffer payload. Each extra block of data is preceded by an OMX_OTHER_EXTRADATATYPE structure, which provides specific information about the extra data.

Extra data shall only be present when the OMX_BUFFERFLAG_EXTRADATA is signaled.

- OMX_BUFFERFLAG_CODECCONFIG is an optional flag that is set by an output port when all bytes in the buffer form part or all of a set of codec specific configuration data. Examples include SPS/PPS NAL units for OMX_VIDEO_CodingAVC or AudioSpecificConfig data for OMX_AUDIO_CodingAAC. Any component that for a given stream sets OMX_BUFFERFLAG_CODECCONFIG shall not mix codec configuration bytes with frame data in the same buffer, and shall send all buffers containing codec configuration bytes before any buffers containing frame data that those configurations bytes describe. If the stream format for a particular codec has a frame specific header at the start of each frame, for example OMX_AUDIO_CodingMP3 or OMX_AUDIO_CodingAAC in ADTS mode, then these shall be presented as normal without setting OMX_BUFFERFLAG_CODECCONFIG.

- OMX_BUFFERFLAG_TIMESTAMPINVALID is set to indicate that the nTimeStamp parameter does not contain valid timestamp information.

  A component emitting stream content with invalid or no timestamp information shall set this flag to indicate presence of no timestamp information.

  A component that updates the nTimeStamp parameter with valid timestamp information shall clear this flag.

- OMX_BUFFERFLAG_READONLY is set when a component emitting the buffer on an output port or the IL client wishes to identify the buffer payload contents to be read-only. An IL client or an input port shall not alter the contents of the buffer.

  This flag shall only be cleared by the originator of the buffer when the buffer is returned.

  For tunneled ports, the usage of this flag shall be allowed only if the components negotiated a read-only tunnel.

- OMX_BUFFERFLAG_ENDOFSUBFRAME is an optional flag set by an output port when the last byte of a buffer payload aligns with an end-of-sub-frame, where a sub-frame represents an independently decodable unit that may be a subset of a frame (e.g., a NAL unit in case of AVC, a slice in case of MPEG4 video, a field in case of VC-1 etc.). Any component that implements setting the OMX_BUFFERFLAG_ENDOFSUBFRAME flag on an output port shall set this flag for every buffer sent from the output port if the last byte of the buffer payload aligns with an end-of-sub-frame. A component that implements setting the OMX_BUFFERFLAG_ENDOFSUBFRAME flag on an output port may apply this flag on buffers whose payloads contain data from more than one sub-frame.

If a component implements and sets both OMX_BUFFERFLAG_ENDOFFRAME flag and OMX_BUFFERFLAG_ENDOFSUBFRAME flag on an output port, it shall apply both of these flags on a buffer whose last byte of buffer payload aligns with an end-of-frame.
These restrictions enable input ports that receive data from the output port to detect end-of-sub-frames without requiring additional processing.

- OMX_BUFFERFLAG_SKIPFRAME is set to indicate the presence of a skipped (not coded) frame within the stream that needs to be signaled out-of-band from the stream content. Each Skipped frame has to be signaled separately. These restrictions enable input ports to detect the presence of skipped frames without requiring additional processing.
  For example, codecs such as VC1 Simple and Main profile (without B-frames) do not support the ability to signal skipped frames in-band (within the video stream syntax), skipped frames are signaled externally (i.e. out-of-band) within the media containers.

- nOutputPortIndex contains the port index of the output port that uses the buffer. If a buffer header is used on an input port that is communicating with the IL client, the value of nOutputPortIndex is undefined.
- nInputPortIndex contains the port index of the input port that uses the buffer. If a buffer header is used on an output port that is communicating with the IL client, the value of nInputPortIndex is undefined.

3.1.3.8 OMX_PORT_PARAM_TYPE

A component uses the OMX_PORT_PARAM_TYPE structure to identify the number and starting index of ports of a particular domain.

OMX_PORT_PARAM_TYPE is defined as follows.

```
typedef struct OMX_PORT_PARAM_TYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPorts;
    OMX_U32 nStartPortNumber;
} OMX_PORT_PARAM_TYPE;
```

3.1.3.8.1 Parameter Definitions

- nPorts is the number of ports of a given port domain (audio, video, image, or other) for the component.
- nStartPortNumber is the index of the first port of a given port domain (audio, video, image, or other) for the component . Subsequent ports of the given domain are numbered sequentially from nStartPortNumber.

3.1.3.9 OMX_CALLBACKTYPE

The OpenMAX IL includes a callback mechanism that allows a component to communicate with the IL client.

To accomplish a callback, the OpenMAX IL has three callback functions defined: a generic event handler and two callbacks related to the dataflow (`EmptyBufferDone` and `FillBufferDone`).

The IL client is responsible for filling in an `OMX_CALLBACKTYPE` structure with its callback entry points and passing the structure to the OpenMAX IL core at initialization (init) time, usually in the `OMX_GetHandle` function.

`OMX_CALLBACKTYPE` is defined as follows.

```
typedef struct OMX_CALLBACKTYPE
{
    OMX_ERRORTYPE (*EventHandler)(
        OMX_IN OMX_HANDLETYPE hComponent,
        OMX_IN OMX_PTR pAppData,
        OMX_IN OMX_EVENTTYPE eEvent,
        OMX_IN OMX_U32 nData1,
        OMX_IN OMX_U32 nData2,
        OMX_IN OMX_PTR pEventData);
    OMX_ERRORTYPE (*EmptyBufferDone)(
        OMX_IN OMX_HANDLETYPE hComponent,
        OMX_IN OMX_PTR pAppData,
        OMX_IN OMX_BUFFERHEADERTYPE* pBuffer);
    OMX_ERRORTYPE (*FillBufferDone)(
        OMX_IN OMX_HANDLETYPE hComponent,
        OMX_IN OMX_PTR pAppData,
        OMX_IN OMX_BUFFERHEADERTYPE* pBuffer);
} OMX_CALLBACKTYPE;
```

3.1.3.9.1 EventHandler

A component uses the `EventHandler` method to notify the IL client when an event of interest occurs within the component. The `OMX_EVENTTYPE` enumeration defines the set of OpenMAX IL events; refer to the definition of this enumeration for the meaning of each event. The information carried within `nData1`, `nData2` and `pEventData` varies depending on `OMX_EVENTTYPE`, refer to Table 3-11 for specific details.

Note that in the case of `OMX_EventCmdComplete`, the component shall cast the error code into `OMX_PTR` for storing it into `pEventData` parameter. Upon receiving the `OMX_EventCmdComplete` event, the IL client shall cast the `pEventData` parameter back into `OMX_ERRORTYPE`. A call to `EventHandler` is a blocking call, so the IL client should return within five milliseconds to avoid blocking the component for an excessively long time period.

The `EventHandler` method is defined as follows.

```
OMX_ERRORTYPE(* OMX_CALLBACKTYPE::EventHandler)(
OMX_IN OMX_HANDLETYPE hComponent,
OMX_IN OMX_PTR pAppData,
OMX_IN OMX_EVENTTYPE eEvent,
OMX_IN OMX_U32 nData1,
OMX_IN OMX_U32 nData2,
OMX_IN OMX_PTR pEventData)
```

The parameters are as follows.

Parameter	Description
hComponent *[in]*	The handle of the component that calls this function.
pAppData *[in]*	A pointer to IL client-defined data.
eEvent *[in]*	The event that the component is communicating to the IL client.
nData1 *[in]*	The first integer event-specific parameter. See Table 3-11 for the meaning in the context of each event.
nData2 *[in]*	The second integer event-specific parameter. See Table 3-11 for the meaning in the context of each event. The default value is 0 if not used.
pEventData *[in]*	A pointer to additional event-specific data or an error code. See Table 3-11 for the meaning in the context of each event.

Table 3-11 lists the parameters used in each event.

Table 3-11: Event Parameter Usage

eEvent	nData1	nData2	pEventData
OMX_EventCmdComplete	OMX_CommandStateSet	State reached	Error code
	OMX_CommandFlush	Port index	Error code
	OMX_CommandPort Disable	Port index	Error code
	OMX_CommandPort Enable	Port index	Error code
	OMX_CommandMark Buffer	Port index	Error code
OMX_EventError	Error code	In some cases, this field is used to convey additional information linked to the error event (see Table 3-12). Zero when no additional information is needed.	Null

eEvent	nData1	nData2	pEventData
OMX_EventMark	0	0	Data linked to the mark, if any
OMX_EventPortSettingsChanged	Port index	Param or config index	Null
OMX_EventBufferFlag	Port index	nFlags unaltered	Null
OMX_EventResourcesAcquired	0	0	Null
OMX_EventComponentResumed	0	0	Null
OMX_EventDynamicResourcesAvailable	0	0	Null
OMX_EventPortFormatDetected	0	0	Null
OMX_EventIndexSettingChanged	Port index (may also be OMX_ALL)	Param or config index	Null
OMX_EventPortNeedsDisable	Port Index	Requesting Port Index	Null
OMX_EventPortNeedsFlush	Port Index	Requesting Port Index	Null

The following table lists the error codes that use the nData2 field to carry additional information related to the error.

Table 3-12: OMX_EventError Event nData2 Parameter Usage

nData1 (error code)	nData2
OMX_ErrorUnderflow, OMX_ErrorOverflow, OMX_ErrorStreamCorrupt, OMX_ErrorStreamCorruptStalled, OMX_ErrorStreamCorruptFatal, OMX_ErrorPortUnpopulated, OMX_ErrorFormatNotDetected	The index of the affected port.

3.1.3.9.2 EmptyBufferDone

A component uses the EmptyBufferDone callback to pass a buffer from an input port back to the IL client. A component updates the nOffset and nFilledLen values of the buffer header to reflect the portion of the buffer it consumed; for example, nFilledLen is set equal to 0 if completely consumed.

In addition to facilitating normal data flow between an executing component and the IL client, a component uses the EmptyBufferDone function to return input buffers to the IL client in the following cases:

- The IL client commands a transition from `OMX_StateExecuting` or `OMX_StatePause` to `OMX_StateIdle`.
- The IL client flushes or disables a port.

In these cases, a component may also return a partially consumed input buffer to the IL client.

The `EmptyBufferDone` call is a blocking call that should return from within five milliseconds. Therefore, the IL client may elect not to fill the buffers during this call but queue them for processing outside this call.

The `EmptyBufferDone` call is defined as follows.

```
OMX_ERRORTYPE(* OMX_CALLBACKTYPE::EmptyBufferDone)(
 OMX_IN OMX_HANDLETYPE hComponent,
 OMX_IN OMX_PTR pAppData,
 OMX_IN OMX_BUFFERHEADERTYPE* pBuffer)
```

The parameters are as follows.

Parameter	Description
hComponent *[in]*	The handle of the component that is calling this function.
pAppData *[in]*	A pointer to IL client-defined data.
pBuffer *[in]*	A pointer to an `OMX_BUFFERHEADERTYPE` structure that was consumed or returned.

3.1.3.9.3 FillBufferDone

A component uses the `FillBufferDone` callback to pass a buffer from an output port back to the IL client. A component sets the `nOffset` and `nFilledLen` of the buffer header to reflect the portion of the buffer it filled; for example, `nFilledLen` is equal to 0 if it contains no data).

In addition to facilitating normal dataflow between an executing component and the IL client, a component uses this function to return output buffers to the IL client in the following cases:

- The IL client commands a transition from `OMX_StateExecuting` or `OMX_StatePause` to `OMX_StateIdle`.
- The IL client flushes or disables a port.

The `FillBufferDone` call is a blocking call that should return from within five milliseconds. The IL client may elect not to empty the buffers during this call but queue them for consumption outside this call.

`FillBufferDone` is defined as follows.

```
OMX_ERRORTYPE(* OMX_CALLBACKTYPE::FillBufferDone)(
OMX_IN OMX_HANDLETYPE hComponent,
OMX_IN OMX_PTR pAppData,
OMX_IN OMX_BUFFERHEADERTYPE* pBuffer)
```

The parameters are as follows.

Parameter	Description
hComponent *[in]*	The handle of the component to access. This handle is the component handle returned by the call to the `GetHandle` function.
pAppData *[in]*	A pointer to IL client-defined data
pBuffer *[in]*	A pointer to an `OMX_BUFFERHEADERTYPE` structure that was filled or returned.

3.1.3.10 OMX_PARAM_BUFFERSUPPLIERTYPE

The `OMX_PARAM_BUFFERSUPPLIERTYPE` structure is used to communicate buffer supplier settings or buffer supplier preferences during tunnel setup (see Section 3.4.1.2).

`OMX_PARAM_BUFFERSUPPLIERTYPE` is defined as follows.

```
typedef struct OMX_PARAM_BUFFERSUPPLIERTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BUFFERSUPPLIERTYPE eBufferSupplier;
} OMX_PARAM_BUFFERSUPPLIERTYPE;
```

3.1.3.10.1 Parameter Definitions

- `nPortIndex` represents the port that this structure applies to.
- `eBufferSupplier` is a field that specifies the port in the tunnel that is the supplier port (see Section 3.1.1.5).

3.1.3.11 OMX_TUNNELSETUPTYPE

The `ComponentTunnelRequest` function uses the `OMX_TUNNELSETUPTYPE` structure to pass data between two ports when an IL client connects these ports via an `OMX_SetupTunnel` call.

`OMX_TUNNELSETUPTYPE` is defined as follows.

```
typedef struct OMX_TUNNELSETUPTYPE  {
    OMX_U32 nTunnelFlags;
    OMX_BUFFERSUPPLIERTYPE eSupplier;
} OMX_TUNNELSETUPTYPE;
```

3.1.3.11.1 Parameter Definitions

- `nTunnelFlags` is an integer parameter that contains one or more bit flags applied to the port that receives this structure. Flags include:

```
#define OMX_PORTTUNNELFLAG_READONLY 0x00000001
```

  If the flag is set as read only, the input port that receives this structure cannot alter the contents of buffers supplied on the tunnel.

- The `eSupplier` field defines whether the input port or the output port provides the buffers. The exact sequence of calls to set up a tunnel is specified in section 3.4.1.2.

3.1.3.12 OMX_PARAM_PORTDEFINITIONTYPE

The `OMX_PARAM_PORTDEFINITIONTYPE` structure contains a set of generic fields that characterize each port of the component. Some of these fields are common to all domains while other fields are specific to their respective domains. The IL client uses this structure to retrieve general information from each port.

`OMX_PARAM_PORTDEFINITIONTYPE` is defined as follows.

```
typedef struct OMX_PARAM_PORTDEFINITIONTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_DIRTYPE eDir;
    OMX_U32 nBufferCountActual;
    OMX_U32 nBufferCountMin;
    OMX_U32 nBufferSize;
    OMX_BOOL bEnabled;
    OMX_BOOL bPopulated;
    OMX_PORTDOMAINTYPE eDomain;
    union {
        OMX_AUDIO_PORTDEFINITIONTYPE audio;
        OMX_VIDEO_PORTDEFINITIONTYPE video;
        OMX_IMAGE_PORTDEFINITIONTYPE image;
        OMX_OTHER_PORTDEFINITIONTYPE other;
    } format;
    OMX_BOOL bBuffersContiguous;
    OMX_U32 nBufferAlignment;
} OMX_PARAM_PORTDEFINITIONTYPE;
```

3.1.3.12.1 Parameter Definitions

- `nPortIndex` represents the port that this structure applies to. The value of `nPortIndex` is a unique 32-bit number for the component. No two ports on a single component may share the same port number, but ports on different components may have the same port number.

- `eDir` is a read-only field that indicates the direction for the port.

Table 3-13: OMX_DIRTYPE enumeration

Field Name	Description
OMX_DirInput	Input port
OMX_DirOutput	Output port

- nBufferCountActual represents the number of buffers that are required on this port before it is populated, as indicated by the bPopulated field of this structure. The component shall set a default value equal to nBufferCountMin for this field. The component shall disallow changes to this field when requested to set a value less than nBufferCountMin.
- nBufferCountMin is a read-only field that specifies the minimum number of buffers that the port requires.
- nBufferSize is a read-only field that specifies the minimum size in bytes for buffers that are allocated for this port. .
- bEnabled is a read-only Boolean field that indicates if the port is enabled. Ports default to bEnabled = OMX_TRUE and are enabled/disabled by sending the OMX_CommandPortEnable and OMX_CommandPortDisable commands with the OMX_SendCommand method. A port shall not be populated when it is not enabled.
- bPopulated is a read-only Boolean field that indicates if a port is populated. A port is populated when all of the buffers indicated by nBufferCountActual with a size of at least nBufferSize have been allocated on the port. A populated port shall be enabled. Enabled ports shall be populated on a transition to OMX_StateIdle and unpopulated on a transition to OMX_StateLoaded.
- eDomain is a read-only field that indicates the domain of the port. This field determines the contents of the format union explained in the next paragraph. Values for eDomain are defined in Table 3-14 below.

Table 3-14: Port Domains

Field Name	Description
OMX_PortDomainAudio	Specifies that the field format is a structure of the OMX_AUDIO_PORTDEFINITIONTYPE type.
OMX_PortDomainVideo	Specifies that the field format is a structure of the OMX_VIDEO_PORTDEFINITIONTYPE type.
OMX_PortDomainImage	Specifies that the field format is a structure of the OMX_IMAGE_PORTDEFINITIONTYPE type.
OMX_PortDomainOther	Specifies that the field format is a structure of the OMX_OTHER_PORTDEFINITIONTYPE type.

- The format fields are a union of domain-specific parameters. For more information on parameters for audio, video, image, and other domains, see Section 4 - OpenMAX IL Data API.

- `bBuffersContiguous` is a Boolean field that has two meanings depending on the component state and whether the port is enabled or disabled. When the port is disabled or the component is in the state `OMX_StateLoaded`, `bBuffersContiguous` shall indicate whether the port prefers each buffer to be in physically contiguous memory. During the transition from `OMX_StateLoaded` to `OMX_StateIdle`, or when enabling the port, the buffer allocator shall update this field to indicate the actual contiguity of the buffers as described in section 2.8.2.

  Note that having physically contiguous buffers is not always enough to allow processing by hardware, the exact meaning of this field may be platform specific.

- `nBufferAlignment` is a read-only field that specifies the alignment the port requires for each of its buffer addresses (`OMX_BUFFERHEADERTYPE` structure's `pBuffer` parameter) and the payload data within the buffer (`OMX_BUFFERHEADERTYPE` structure's `pBuffer + nOffset` parameters).

  For example, a value of 4 denotes the alignment shall be 4-byte aligned. A value of zero denotes that the port does not have any alignment restrictions. Buffer alignments are restricted to powers of two.

3.1.3.13 OMX_CONFIG_TUNNELEDPORTSTATUSTYPE

The `OMX_CONFIG_TUNNELEDPORTSTATUSTYPE` structure is used to communicate to a port what API calls are allowed on its tunneled port.

This structure is intended to be used in interop profile only. A component shall use this structure to inform its tunneled components about when some specific API calls become allowed on its non-supplier ports.

The index specified for this structure is `OMX_IndexConfigTunneledPortStatus`.

`OMX_CONFIG_TUNNELEDPORTSTATUSTYPE` is defined as follows:

```
typedef struct OMX_CONFIG_TUNNELEDPORTSTATUSTYPE{
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nTunneledPortStatus;
} OMX_CONFIG_TUNNELEDPORTSTATUSTYPE;
```

3.1.3.13.1 Parameters

- `nPortIndex` represents the port that this structure applies to.
- `nTunnelPortStatus` is an integer parameter that contains one or more bit flags applied to the port that receives this structure.

Table 3-15 : Port Status Flags

Flags	Description
OMX_PORTSTATUS_ACCEPTUSEBUFFER	When this flag is set for a port, it means that its tunneled port accepts OMX_UseBuffer calls
OMX_PORTSTATUS_ACCEPTBUFFEREXCHANGE	When this flag is set for a port, it means that its tunneled port would accept OMX_EmptyThisBuffer or OMX_FillThisBuffer calls.

3.1.3.13.2 OMX_StateLoaded to OMX_StateIdle Transition

When commanded to transition from OMX_StateLoaded state to OMX_StateIdle state, a component does the following:

- For each enabled non-supplier port, it shall inform its tunneled port using OMX_SetConfig(OMX_IndexConfigTunneledPortStatus) that OMX_UseBuffer calls are now allowed.

For each enabled supplier port, it should start calling OMX_UseBuffer only when it got the information that the call would be accepted by the tunneled port. Attempting to call OMX_UseBuffer prior to getting the OMX_SetConfig(OMX_IndexConfigTunneledPortStatus) telling that the call would be accepted is likely to result in the call returning OMX_ErrorIncorrectStateOperation. If one of the UseBuffer calls returns OMX_ErrorIncorrectStateOperation, the component should wait for a call to OMX_SetConfig(OMX_PORTSTATUS_ACCEPTUSEBUFFER) to resume the sequence, or for a OMX_SendCommand(CommandStateSet, OMX_StateLoaded) that cancels the current transition to OMX_StateIdle.

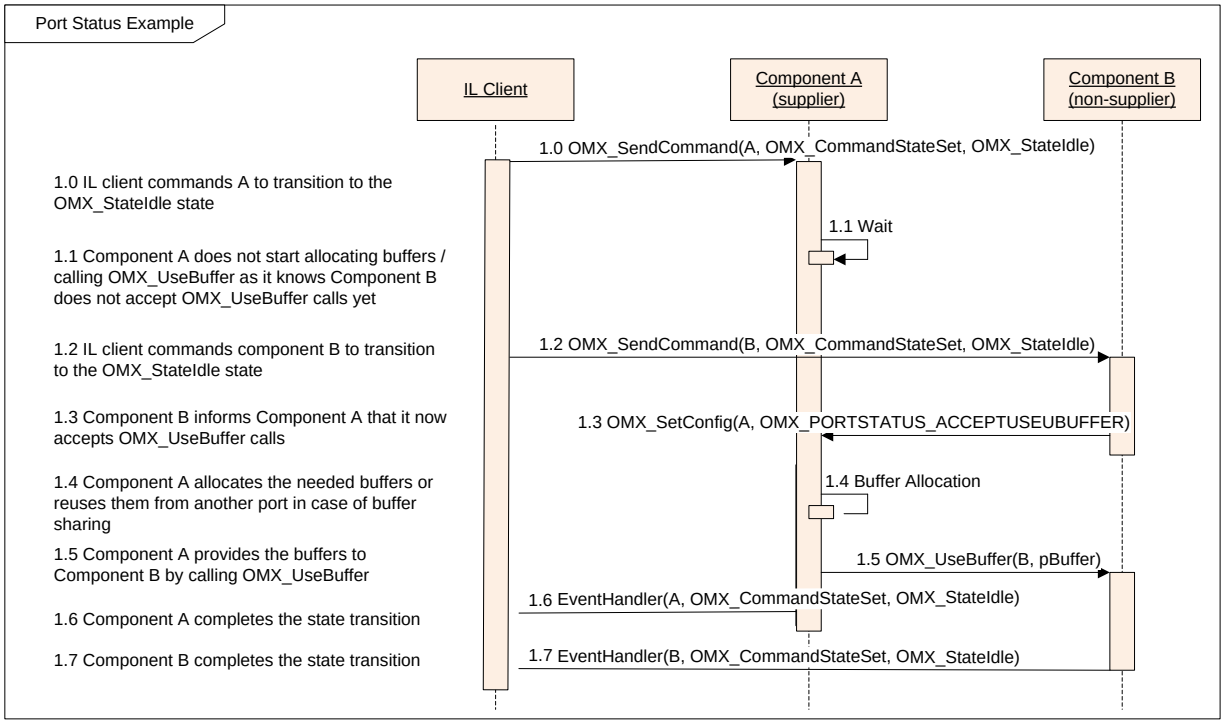


Figure 3-2 : OMX_StateLoaded to OMX_StateIdle transition while supplier first

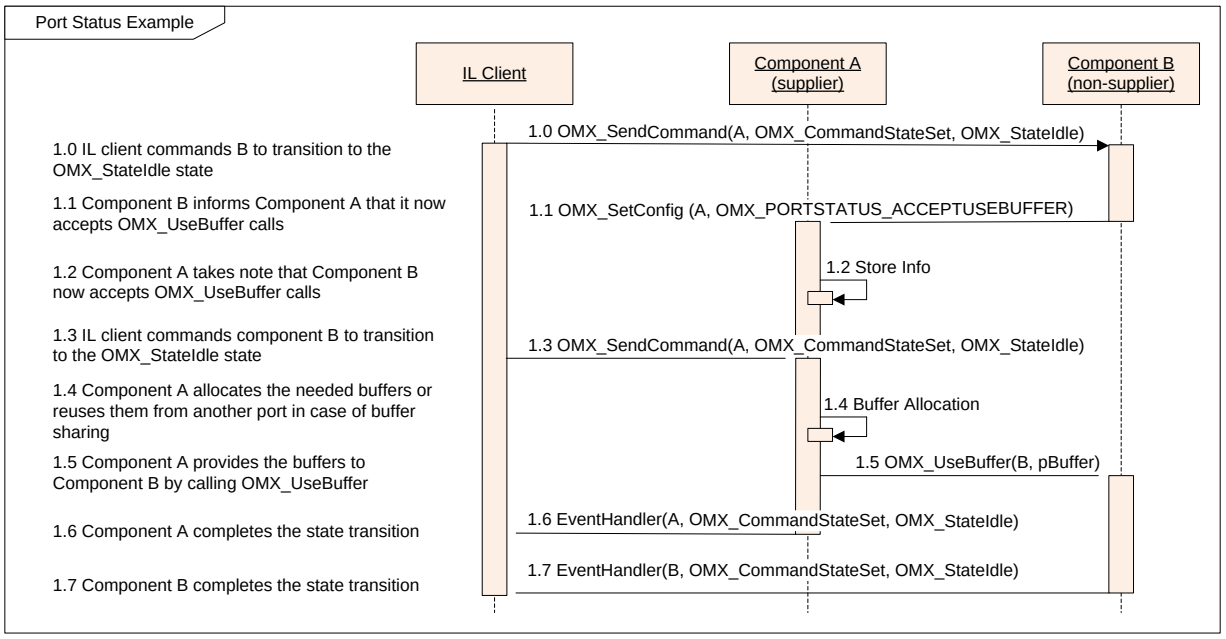


Figure 3-3: OMX_StateLoaded to OMX_StateIdle transition while non-supplier first

3.1.3.13.3 OMX_StateIdle to OMX_StateExecuting Transition

When commanded to transition from `OMX_StateIdle` state to `OMX_StateExecuting` state, a component does the following:

- For each enabled non-supplier port, it shall inform its tunneled port using `OMX_SetConfig(OMX_IndexConfigTunneledPortStatus)` that buffer exchange is now allowed.
- For each enabled supplier port, it should start calling `OMX_EmptyThisBuffer /OMX_FillThisBuffer` only when it got the information that the call would be accepted by the tunneled port. Attempting to call `OMX_EmptyThisBuffer` or `OMX_FillThisBuffer` prior to getting the `OMX_SetConfig(OMX_IndexConfigTunneledPortStatus)` telling that the call would be accepted is likely to result in the call returning `OMX_ErrorIncorrectStateOperation`.

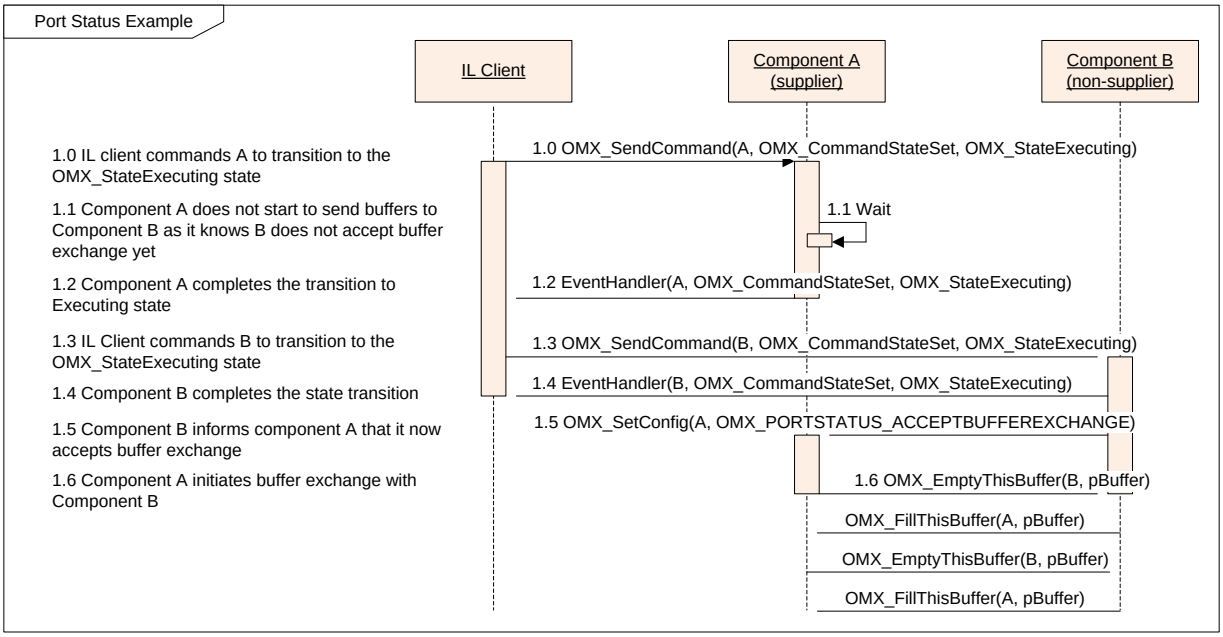


Figure 3-4: OMX_StateIdle to OMX_StateExecuting transition while supplier first

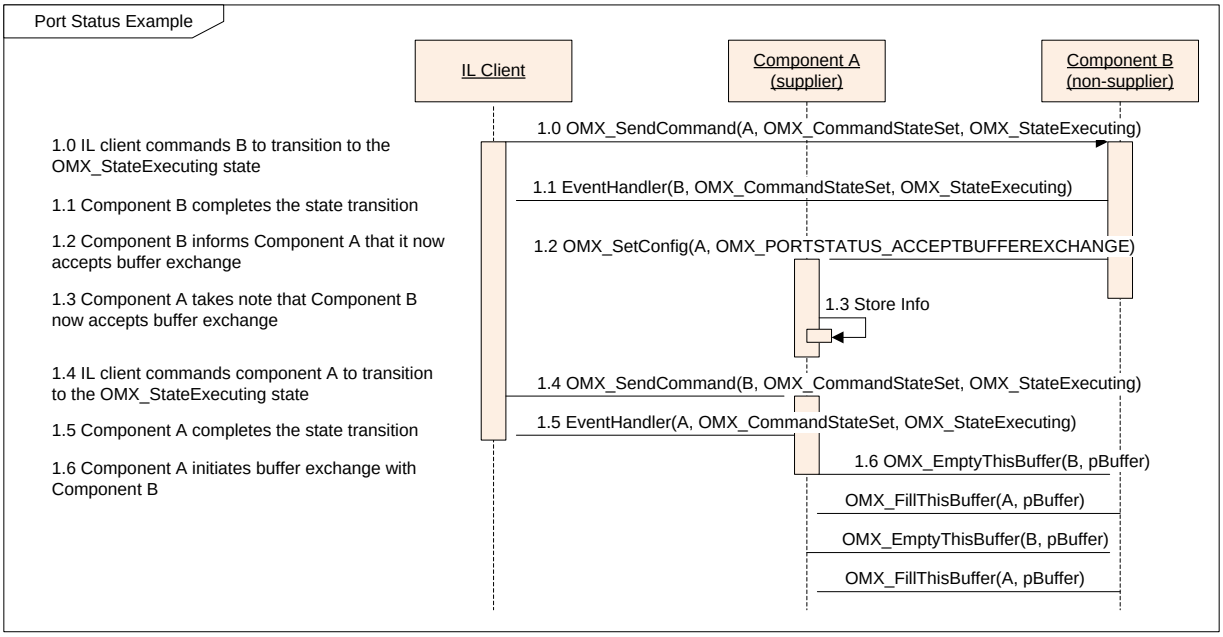


Figure 3-5: OMX_StateIdle to OMX_StateExecuting State Transition while non-supplier first

3.1.3.13.4 OMX_StateExecuting to OMX_StateIdle Transition

When commanded to transition from `OMX_StateExecuting` state to `OMX_StateIdle` state, a component does the following:

- For each enabled non-supplier port, it shall return all the buffers that the port is holding using `OMX_EmptyThisBuffer` for an output port or `OMX_FillThisBuffer` for an input port, then it shall reject any further call to `OMX_EmptyThisBuffer` for an input port or `OMX_FillThisBuffer` for an output port by returning `OMX_ErrorIncorrectStateOperation`.
- For each enabled supplier port, it shall wait for all buffers to be returned by the tunneled port before completing the transition.

A component in `OMX_StateExecuting` state may get an `OMX_ErrorIncorrectStateOperation` when calling `OMX_EmptyThisBuffer` or `OMX_FillThisBuffer` if the non-supplier component is commanded to transition to `OMX_StateIdle` before the supplier component. In this situation, it should stop buffer exchanges until it commanded to transition to `OMX_State Idle` or until it gets the `OMX_SetConfig(OMX_IndexConfigTunneledPortStatus)` telling that buffer exchanges are accepted again by the tunneled port (See Figure 3-5: OMX_StateIdle to OMX_StateExecuting State Transition while non-supplier first).

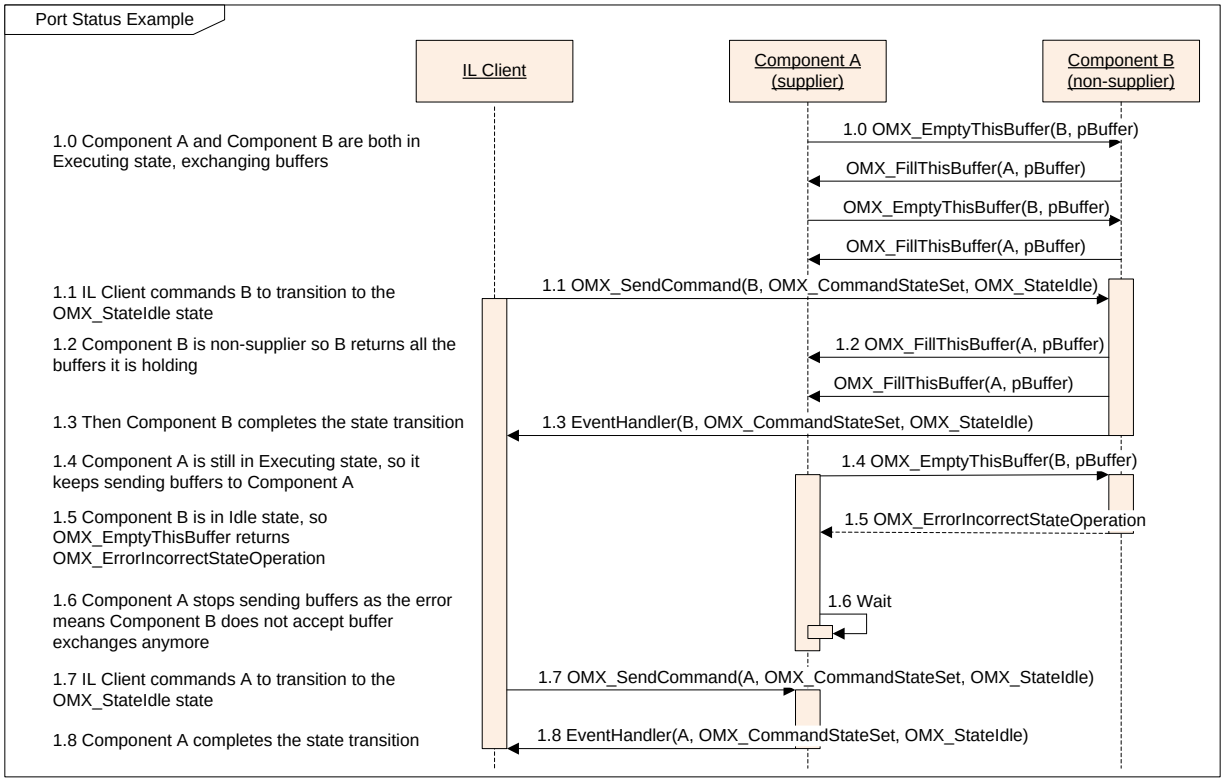


Figure 3-6: Executing to OMX_StateIdle transition while non-supplier first

3.1.3.13.5 Disabling Ports – OMX_CommandPortDisable

When commanded to disable a port, a component does the following:

- For a non-supplier port, it shall return all the buffers that the port is holding using `OMX_EmptyThisBuffer` for an output port or `OMX_FillThisBuffer` for an input port, then it shall reject any further call to `OMX_EmptyThisBuffer` for an input port or `OMX_FillThisBuffer` for an output port by returning `OMX_ErrorIncorrectStateOperation`. It shall then wait for OMX_FreeBuffer calls to complete the command.
- For a supplier port, it shall wait for all buffers to be returned by the tunneled port, then call `OMX_FreeBuffer` to free all the buffers.

A component in Executing state may get an `OMX_ErrorIncorrectStateOperation` when calling `OMX_EmptyThisBuffer` or `OMX_FillThisBuffer` if the non-supplier port gets the `OMX_CommandPortDisable` command before the supplier port. In this situation, it should stop buffer exchanges until it gets the `OMX_CommandPortDisable` command or until it gets the `OMX_SetConfig(OMX_IndexConfigTunneledPortStatus)` telling that buffer exchanges are accepted again by the tunneled port (See Figure 3-5: OMX_StateIdle to OMX_StateExecuting State Transition while non-supplier first).

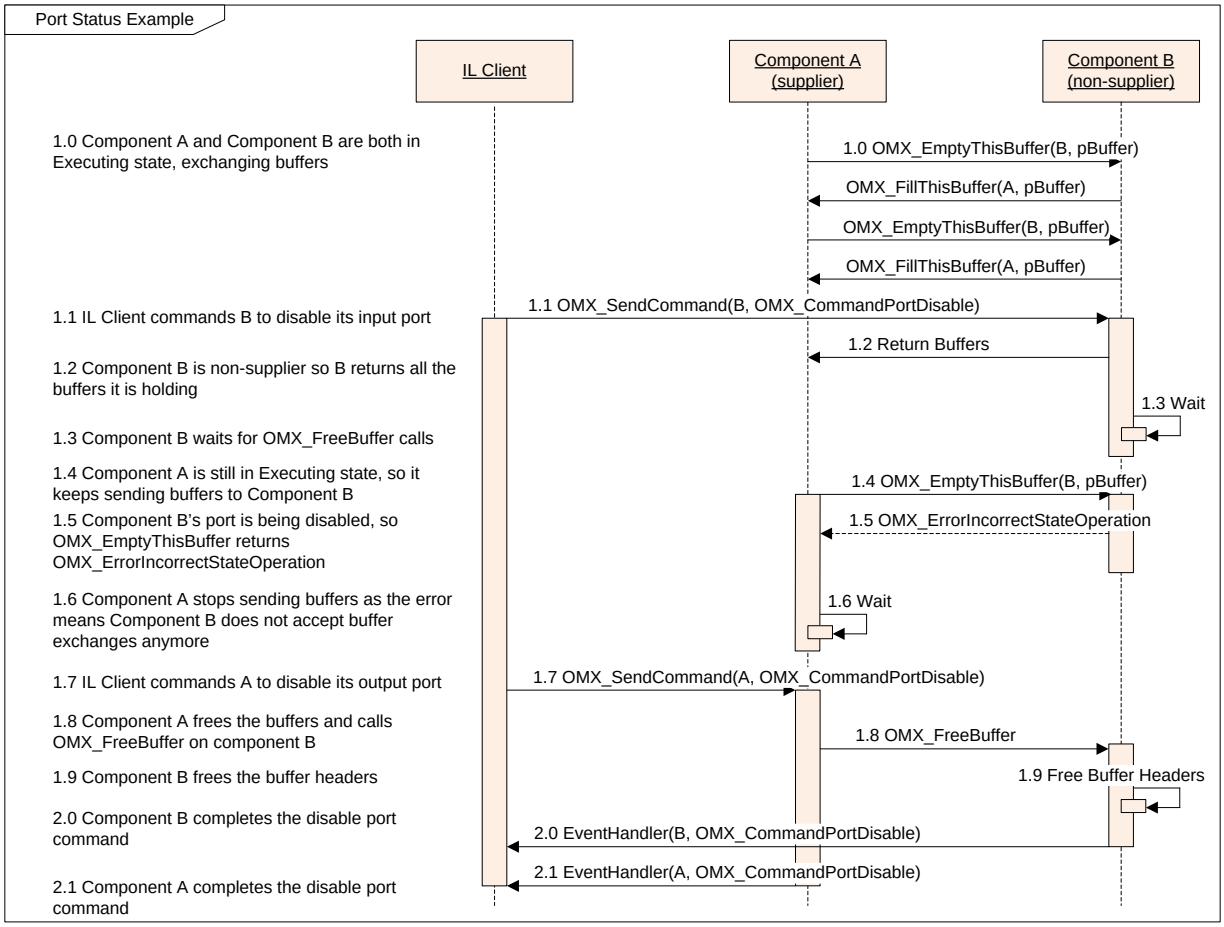


Figure 3-7: Disabling ports while non-supplier first

3.1.3.13.6 Enabling Ports – OMX_CommandPortEnable

When commanded to enable a port, a component does the following:

- For a non-supplier port, it shall inform its tunneled port using `OMX_SetConfig(OMX_IndexConfigTunneledPortStatus)` that `OMX_UseBuffer` calls are now allowed. If the component is in `OMX_StateExecuting` state, it can inform in the same call that buffer exchange will be allowed once the port is populated.

- For a supplier port, it should start calling `OMX_UseBuffer` only when it got the information that the call would be accepted by the tunneled port. Attempting to call `OMX_UseBuffer` prior to getting the `OMX_SetConfig(OMX_IndexConfigTunneledPortStatus)` telling that the call would be accepted is likely to result in the call returning `OMX_ErrorIncorrectStateOperation`. If one of `the OMX_UseBuffer` calls returns `OMX_ErrorIncorrectStateOperation`, the component should wait for a call to `OMX_SetConfig(OMX_PORTSTATUS_ACCEPTUSEBUFFER)`

to resume the sequence, or for a `OMX_SendCommand(OMX_CommandPortDisable)` that cancels the current transition to Enabled.

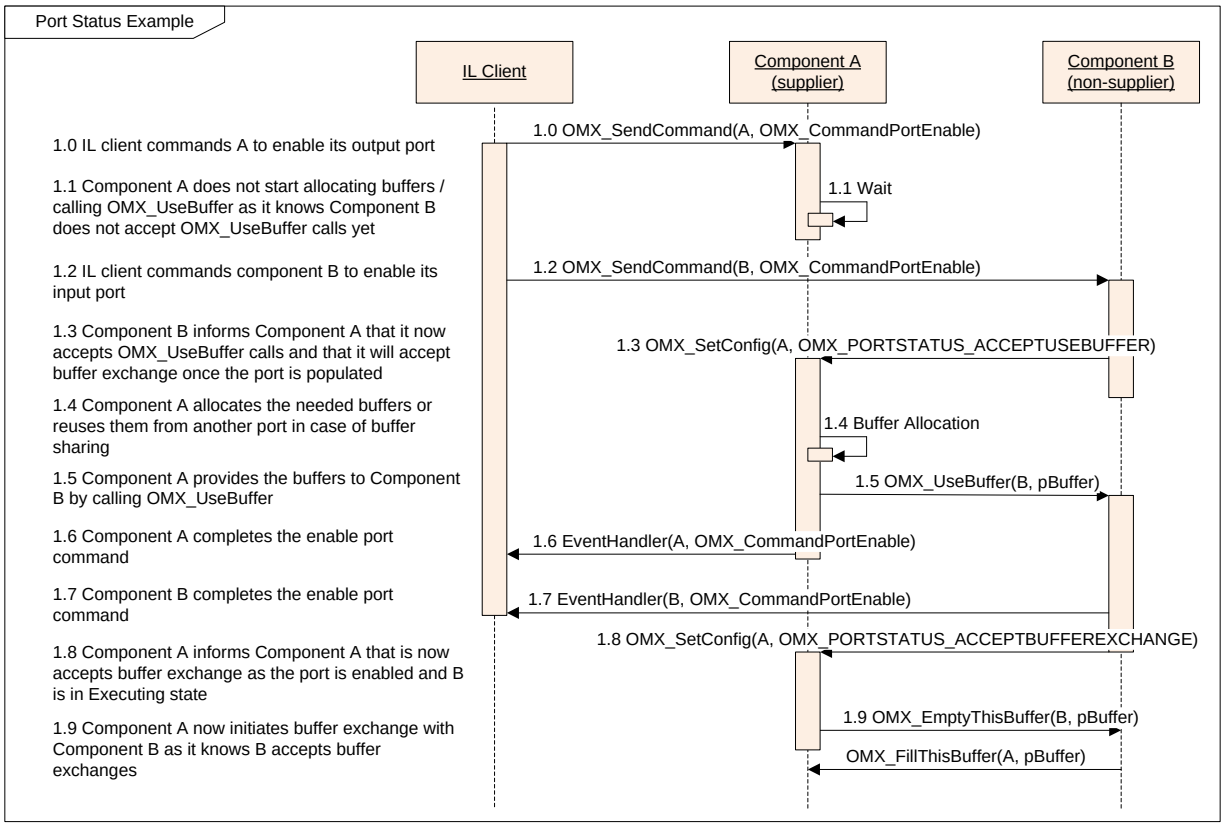


Figure 3-8: enabling ports while supplier first

3.2 OpenMAX IL Core Methods/Macros

The OpenMAX IL core implements the main interface for an IL client that wants to use OpenMAX IL components. For efficiency, OpenMAX IL defines a set of OpenMAX IL core macros that map on one-to-one basis to most OpenMAX IL component methods.

Some macros and methods recommend that the function return within either five milliseconds or 20 milliseconds, depending on the function. The 5-millisecond timeout was deemed by the standards body to be a reasonable response time for commands that may not require buffer processing. The standards body identified the 20-millisecond timeout to be a reasonable response time for commands that may require buffer processing to be completed; the assumption here is that the longest buffer processing would be less than 30 milliseconds, which corresponds to 30-frames per second video. These timeouts are intended primarily to enable component integrators to get a good idea of component response latency via conformance testing.

The macros include the following:

- Get component information (version, capabilities).
- Set the callbacks structure from the IL client to the IL component.
- Set/Get component parameters at init time.
- Set/Get component parameters at run time.
- Allocate/De-allocate buffers.
- Send a buffer full of data to an OpenMAX IL component port.
- Send an empty buffer to an OpenMAX IL component port.
- Send commands to a component.
- Get the actual state of the component.
- Get references to OpenMAX IL component-proprietary parameters.

The OpenMAX IL core also implements methods for the following:

- Initializing/de-initializing the whole OpenMAX IL core.
- Getting an OpenMAX IL component handle.
- Releasing an OpenMAX IL component handle.
- Detecting all OpenMAX IL components available on the platform at run time.
- Setting up data tunnels among OpenMAX IL components.
- Tearing down data tunnels among OpenMAX IL components.
- Acquiring content pipes.
- Querying for information on installed standard component implementations.

When a time limit for the execution of a method is specified, it is not intended as a hard restriction for the conformance of the component to the standard, but if the limit is not respected, a note shall appear in the description document related to the component.

3.2.1 Return Codes for the Functions

Table 3-16 lists all of the possible return error codes for each function. A critical error denotes an error from which the component cannot recover. The component shall remain in its current state and wait for the action of the IL client. After receiving a critical error event, the IL client shall transition the component from its current state back to `OMX_StateLoaded` (through `OMX_StateIdle`, where necessary), and unload it by call `OMX_FreeHandle`. Note that there is no gurantee that actions other than the necessary commands to transition the component to OMX_StateLoaded state and unload it will succeed after the component has signaled the critical error. Despite the critical error event, the component shall return all buffers to their suppliers where necessary for state transition, and shall complete the `OMX_FreeBuffer` calls.

All columns but the last two correspond to errors returned from a call to the component. The rightmost two columns denote errors sent asynchronously as the result of an internal error.

Table 3-16: Error Codes

	OMX_GetComponentVersion	OMX_SendCommand	OMX_GetParameter	OMX_SetParameter	OMX_GetConfig	OMX_SetConfig	OMX_GetExtensionIndex	OMX_GetState	OMX_UseBuffer	OMX_AllocateBuffer	OMX_UseEGLImage	OMX_FreeBuffer	OMX_EmptyThisBuffer	OMX_FillThisBuffer	ComponentDeInit	OMX_Init	OMX_Deinit	OMX_ComponentNameEnum	OMX_GetHandle	OMX_FreeHandle	OMX_SetupTunnel	OMX_TeardownTunnel	OMX_ComponentOfRoleEnum	OMX_RoleOfComponentEnum	OMX_GetCoreInterface	OMX_FreeCoreInterface	Sent with EventHandler	Critical error
OMX_ErrorNone	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
OMX_ErrorInsufficientResources		X							X	X	X					X			X								X	
OMX_ErrorUndefined	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X			X	X	X			X	X				X
OMX_ErrorInvalidComponentName																			X					X				
OMX_ErrorComponentNotFound																			X					X				
OMX_ErrorBadParameter	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X			X	X	X	X	X	X		X	X		
OMX_ErrorNotImplemented											X										X	X			X			
OMX_ErrorUnderflow																											X	
OMX_ErrorOverflow																											X	
OMX_ErrorHardware																											X	X
OMX_ErrorStreamCorrupt																											X	
OMX_ErrorPortsNotCompatible																					X							
OMX_ErrorResourcesLost																											X	
OMX_ErrorNoMore			X		X													X					X	X				
OMX_ErrorVersionMismatch	X	X	X	X	X	X	X	X	X	X	X	X			X	X	X	X	X	X	X	X						
OMX_ErrorNotReady			X	X	X	X																						
OMX_ErrorTimeout		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X						
OMX_ErrorSameState																											X	
OMX_ErrorResourcesPreempted																											X	
OMX_ErrorIncorrectStateTransition																											X	
OMX_ErrorIncorrectStateOperation				X					X	X	X		X	X	X						X	X						
OMX_ErrorUnsupportedSetting				X		X																						
OMX_ErrorUnsupportedIndex			X	X	X	X	X																					
OMX_ErrorBadPortIndex		X	X	X	X	X	X		X		X	X	X	X							X	X						

	OMX_GetComponentVersion	OMX_SendCommand	OMX_GetParameter	OMX_SetParameter	OMX_GetConfig	OMX_SetConfig	OMX_GetExtensionIndex	OMX_GetState	OMX_UseBuffer	OMX_AllocateBuffer	OMX_UseEGLImage	OMX_FreeBuffer	OMX_EmptyThisBuffer	OMX_FillThisBuffer	ComponentDeInit	OMX_Init	OMX_Deinit	OMX_ComponentNameEnum	OMX_GetHandle	OMX_FreeHandle	OMX_SetupTunnel	OMX_TeardownTunnel	OMX_ComponentOfRoleEnum	OMX_RoleOfComponentEnum	OMX_GetCoreInterface	OMX_FreeCoreInterface	Sent with EventHandler	Critical error
OMX_ErrorPortUnpopulated																											X	
OMX_errorComponentSuspended																												
OMX_ErrorDynamicResourcesUnavailable																											X	
OMX_ErrorMbErrorsInFrame																											X	
OMX_ErrorFormatNotDetected																											X	
OMX_ErrorSeperateTablesUsed			X																									
OMX_ErrorTunnelingUnSupported																					X							
OMX_ErrorInvalidMode						X																						
OMX_ErrorStreamCorruptStalled																											X	
OMX_ErrorStreamCorruptFatal																											X	
OMX_ErrorPortsNotConnected																						X						
OMX_ErrorContentURINotSpecified																											X	
OMX_ErrorContentURIError				X		X																					X	
OMX_ErrorCommandCanceled																											X	

3.2.2 Macros

This section describes the OpenMAX IL core macros. Note that some of these calls occur when only the caller is in the appropriate state to make the call (e.g. when tunneling) or when the component is transitioning from one state to another.

Table 3-17 defines which macros may be called on a component in each component state.

Table 3-17: Valid Component Calls

	OMX_GetComponentVersion	OMX_SendCommand	OMX_GetParameter	OMX_SetParameter	OMX_GetConfig	OMX_SetConfig	OMX_GetExtensionIndex	OMX_GetState	OMX_UseBuffer	OMX_AllocateBuffer	OMX_FreeBuffer	OMX_EmptyThisBuffer	OMX_FillThisBuffer	ComponentDeinit	OMX_SetupTunnel	OMX_TeardownTunnel	OMX_UseEGLImage	OMX_SetCallbacks	OMX_FreeHadle
OMX_StateLoaded	X	X[4]	X	X	X	X	X	X	X[1]	X[1]	X			X	X	X	X	X	X
OMX_StateIdle	X	X[4]	X		X	X	X	X			X	X[3]	X[3]	X					
OMX_StateExecuting	X	X	X		X	X	X	X			X	X[3]	X[3]	X					
OMX_StatePause	X	X	X		X	X	X	X			X	X	X	X					
OMX_StateWaitForResources	X	X[4]	X	X	X	X	X	X	X	X	X			X			X		
Disabled Port	X	X	X	X	X	X	X	X	X[2]	X[2]	X			X	X		X		

X[1]: valid if already started transition to OMX_StateIdle. In tunneled case for a non-supplier port, the component calls `OMX_SetConfig(OMX_PORTSTATUS_ACCEPTUSEBUFFER)` on its tunneled component to indicate when it can start calling UseBuffer.

X[2]: valid if already started transition to Enabled Port. In tunneled case for a non-supplier port, the component calls `OMX_SetConfig(OMX_PORTSTATUS_ACCEPTUSEBUFFER)` on its tunneled component to indicate when it can start calling UseBuffer.

X[3]: in tunneled case for a non-supplier port, the component calls `OMX_SetConfig(OMX_PORTSTATUS_ACCEPTBUFFEREXCHANGE)` on its tunneled component to indicate when it can start exchanging buffers.

X[4]: not valid for `OMX_CommandMarkBuffer`

3.2.2.1 OMX_GetComponentVersion

The `GetComponentVersion` macro will query the component and returns information about it. This is a blocking call. The component should return from this call within five milliseconds.

The macro is defined as follows.

```
#define OMX_GetComponentVersion (
  hComponent,
  pComponentName,
  pComponentVersion,
  pSpecVersion,
  pComponentUUID)
((OMX_COMPONENTTYPE*)hComponent)->GetComponentVersion(     \
        hComponent,                                        \
        pComponentName,                                    \
        pComponentVersion,                                 \
```

```
        pSpecVersion,
        pComponentUUID)
```

The parameters are as follows.

Parameter	Description
hComponent [in]	The handle of the component that executes the command.
pComponentName [out]	A pointer to a component name string. Component names are strings limited to a length up to 127 bytes plus the trailing null for a maximum length of 128 bytes. An example of a valid component name is "OMX.<vendor_name>.AUDIO.DSP.MIXER\0". Names are assigned by the vendor, but shall start with "OMX." concatenated to the vendor specified string.
pComponentVersion [out]	A pointer to an OpenMAX IL version structure that the component will populate. The component will fill in a value that indicates the component version. Note that the component version is not the same as the OpenMAX IL specification version, which is found in all structures. The vendor of the component defines the component version and establishes its value.
pSpecVersion [out]	A pointer to an OpenMAX IL version structure that the component will populate. `pSpecVersion` is the version of the specification that the component was built against.
pComponentUUID [out]	A pointer to an UUID identifier that uniquely identifies the component. A component shall not be required to provide this information, it is optional information that a component may choose to provide.

3.2.2.1.1 Prerequisites for This Method

This method has no prerequisites.

3.2.2.1.2 Sample Code Showing Calling Sequence

The following sample code shows a calling sequence.

```
/* detect mismatch between IL client's and component's spec version */
OMX_GetComponentVersion(
      hComp,
      &CompName,
      &CompVersion,
      &CompSpecVersion);
if (CompSpecVersion != IlClientVersion){
      printf("ERROR: version mismatch\n");
}
```

3.2.2.2 OMX_SendCommand

The `OMX_SendCommand` macro will invoke a command on the component. This is a non-blocking call that should, at a minimum, validate command parameters but return within five milliseconds. The component uses an event callback to notify the IL client of the results of the command once completed. If the component executes the command

successfully, the component generates an OMX_EventCmdComplete callback with error code set to OMX_ErrorNone. If the component fails to execute the command, the component generates an OMX_EventCmdComplete callback and passes the appropriate error as a parameter.

Calling OMX_SendCommand when there is already an on-going command being processed is allowed only in the following cases:

- When currently doing a state transition from OMX_StateLoaded to OMX_StateIdle, the IL client can try to cancel the transition by commanding the component to go back to OMX_StateLoaded state. Depending on timings, the on-going command will either be canceled, or it may have time to complete normally before the cancellation is received. When canceled, the on-going command will report an OMX_EventCmdComplete event with OMX_ErrorCommandCanceled error code.
- When currently enabling a port (OMX_CommandPortEnable), the IL client can try to cancel the operation by commanding the component to disable (OMX_CommandPortDisable) the port. Depending on timings, the on-going command will either be canceled, or it may have time to complete normally before the cancellation is received. When canceled, the on-going command will report an OMX_EventCmdComplete event with OMX_ErrorCommandCanceled error code.
- When currently doing a state transition from OMX_StateLoaded to OMX_StateIdle, the IL client can command the component to disable (OMX_CommandPortDisable) a port. This may allow the component to complete the transition to OMX_StateIdle if the port was the one blocking the transition. Depending on timings, the commands will either complete in the order they were sent (case where the on-going transition to OMX_StateIdle completes normally before the disable port command is received), or in the reverse order (case where the disable port command - OMX_CommandPortDisable -unblocks the transition to OMX_StateIdle).
- When currently disabling (OMX_CommandPortDisable) a port and if the port is doing buffer sharing, the component may report an event OMX_EventPortNeedsDisable to indicate to the IL client that it shall disable another port of the component to allow the current disable port command (OMX_CommandPortDisable) to complete. In this situation, the IL client shall call OMX_CommandPortDisable to disable the requested port. The commands will complete in the reverse order they were sent.
- When currently flushing (OMX_CommandFlush) a port and if the port is doing buffer sharing, the component may report an OMX_EventPortNeedsFlush event to indicate to the IL client that it shall flush another port of the component to allow the current flush command to complete. In this situation, the IL client shall call OMX_CommandFlush to flush the requested port. The commands will complete in the reverse order they were sent.

- Queuing several `OMX_CommandMarkBuffer` commands is allowed. The number of OMX_Command`MarkBuffer` commands that can be queued is implementation specific.

In any other case, `OMX_SendCommand` will return `OMX_ErrorIncorrectStateOperation` if the IL client attempts to call it when there is already an on-going command being processed.

The macro is defined as follows.

```
#define OMX_SendCommand (
  hComponent,
  Cmd,
nParam,
pCmdData)
 ((OMX_COMPONENTTYPE*)hComponent)->SendCommand(          \
        hComponent,                                      \
        Cmd,                                             \
        nParam,
        pCmdData)
```

The parameters are as follows.

Parameter	**Description**
hComponent [in]	The handle of the component that executes the command
Cmd [in]	Command for the component to execute
nParam [in]	Integer parameter for the command that is to be executed
pCmdData [in]	A pointer that contains implementation-specific data that cannot be represented with the numeric parameter `nParam`

Section 3.3—OpenMAX IL Component Methods and Structures describes the corresponding function that each component implements.

3.2.2.3 OMX_CommandStateSet

The IL client calls this command to request that the component transition into the state given in `nParam`. The component shall make the transition between the old state and the new state successfully only if it is a legal transition and all prerequisites for this transition are met. For more information on component states, see Section 3.1.1.2—OMX_STATETYPE.

If the component successfully transitions to the new state, it exposes the new internal state (e.g. if queried via `OMX_GetState()`) and then notifies the IL client of the new state via the `OMX_EventCmdComplete` event, indicating `OMX_CommandStateSet`

for nData1 and the new state for nData2, and OMX_ErrorNone for error code. If a state transition fails, the component shall notify the IL client of the error that prevented it via OMX_EventCmdComplete event indicating OMX_CommandStateSet for nData1, the attempted transition state for nData2 and an appropriate error code for pEventData. Relevant errors include but are not limited to the following:

- OMX_ErrorSameState: The component is already in the state requested.
- OMX_ErrorIncorrectStateTransition: The transition requested is not legal.
- OMX_ErrorInsufficientResources: The transition required the allocation of resources and the component failed to acquire the resources.

3.2.2.4 OMX_CommandFlush

The IL client calls this command to flush one or more component ports. nParam specifies the index of the port to flush. If the value of nParam is OMX_ALL, the component shall flush all ports. In tunneled cases, the IL client shall always flush both ends of a tunnel.

When the IL client flushes a non-tunneled port, that port shall return all the buffers it is holding to the IL client using EmptyBufferDone and FillBufferDone (appropriate for an input port or an output port, respectively) to return the buffers.

When the IL client flushes a non-supplier tunneled port, that port shall return all the buffers it is holding to its tunneled port using EmptyThisBuffer or FillThisBuffer (appropriate for an input port or an output port, respectively) to return the buffers. Output ports shall zero the nFilledLen field of the buffer header when returning buffers as part of a flush operation.

When the IL client flushes a supplier port, that port shall hold its buffers.

A non-supplier port that is sharing its buffers with another port of the component may need its sharing port to be flushed as well in order to be able to complete the flush command. In that situation, the component shall report an OMX_EventPortNeedsFlush event with the index of the sharing port to indicate to the IL client that it needs to flush another port in order for the current flush command to complete.

When a port is flushed, the component shall reset any internal state associated with the port so as to be ready to process from another location within the stream after the flush. This includes discarding any unprocessed data in the queued buffers for a supplier port.

The codec configuration shall not be reset, but the component shall be able to handle the first buffer after the flush containing a new CODEC_CONFIG indicated by the OMX_BUFFERFLAG_CODECCONFIG buffer flag.

For each port that the component flushes, the component shall send an OMX_EventCmdComplete event, indicating OMX_CommandFlush for nData1, the individual port index for nData2, even if the flush resulted from using a value of

OMX_ALL for nParam, and OMX_ErrorNone for pEventData in case of success or an appropriate error code for pEventData in case of failure.

3.2.2.5 OMX_CommandPortDisable

The OMX_CommandPortDisable command disables a port. nParam specifies the index of the port to disable. If the value of nParam is OMX_ALL, the component shall disable all ports. In tunneled cases, the IL client shall always disable both ends of the tunnel.

A disabled port has no buffers and does not allocate buffers or buffer headers on a transition from OMX_StateLoaded or OMX_StateWaitForResources to OMX_StateIdle. An IL client can change the parameters via OMX_SetParameter of a disabled port regardless of the component state. Thus the OMX_CommandPortDisable command, in co-operation with OMX_TeardownTunnel, OMX_SetupTunnel and OMX_CommandPortEnable, is useful for the dynamic reconfiguration or re-tunneling of a port.

The port shall immediately clear bEnabled in its port definition structure when it receives OMX_CommandPortDisable. If the port that the IL client is disabling is a non-supplier port, the component shall return any buffers it is holding to its supplier via OMX_EmptyThisBuffer/OMX_FillThisBuffer if tunneling or EmptyBufferDone/FillBufferDone if not tunneling. Then, the component shall wait for the supplier to free the buffers via OMX_FreeBuffer before completing the disable command. If the port that the IL client is disabling is a supplier port with buffers allocated, the component shall wait for the non-supplier port to return all buffers via OMX_EmptyThisBuffer or OMX_FillThisBuffer. Then, the component shall free the buffers via OMX_FreeBuffer before completing the disable command.

For each port that the component disables, the component shall send an OMX_EventCmdComplete event indicating OMX_CommandPortDisable for nData1, the individual port index for nData2, even if using a value of OMX_ALL for nParam caused the port to be disabled, and OMX_ErrorNone for pEventData in case of success or an appropriate error code for pEventData in case of failure.

A non-supplier port that is sharing its buffers with another port of the component may require its sharing port to be disabled as well in order to be able to complete the OMX_CommandPortDisable command. In this situation, the component shall report an OMX_EventPortNeedsDisable event with the index of the sharing port to indicate to the IL client that it needs to disable another port in order for the current OMX_CommandPortDisable command to complete.

3.2.2.6 OMX_CommandPortEnable

The OMX_CommandPortEnable command enables a port. nParam specifies the index of the port to be enabled. If the value of nParam is OMX_ALL, the component

shall enable all ports. In tunneled cases, the IL client shall always enable both ends of a tunnel.

An enabled port shall abide by all the requirements of the component's state. Thus, the port shall:

- Have no buffers allocated if the component is in the `OMX_StateLoaded` state or the `OMX_StateWaitForResources` state and all buffers are allocated otherwise.
- Allocate buffers on a transition from either the `OMX_StateLoaded` state or the `OMX_StateWaitForResources` state to the `OMX_StateIdle`.
- Transfer a buffer to facilitate data flow in the `OMX_StateExecuting` state.
- Disallow modification of its parameters via `OMX_SetParameter` in all states but `OMX_StateLoaded`.

The `OMX_CommandPortEnable` command, in co-operation with `OMX_CommandPortDisable`, is useful for the dynamic reconfiguration or re-tunneling of a port.

The port shall immediately set `bEnabled` in its port definition structure when the port receives `OMX_CommandPortEnable`. If the IL client enables a port while the component is in any state other than `OMX_StateLoaded` or `OMX_StateWaitForResources`, then that port shall allocate its buffers via the same call sequence used on a transition from `OMX_StateLoaded` to `OMX_StateIdle`. If the IL client enables while the component is in the `OMX_StateExecuting` state, then that port shall begin transferring buffers.

For each port that the component enables, the component shall send an `OMX_EventCmdComplete` event, indicating `OMX_CommandPortEnable` for `nData1`, the individual port index for `nData2`, even if using the value of OMX_ALL for `nParam` caused the enable operation, and `OMX_ErrorNone` for `pEventData` in case of success or an appropriate error code for `pEventData` in case of failure.

3.2.2.7 OMX_CommandMarkBuffer

The `OMX_CommandMarkBuffer` command instructs the given port to mark a buffer. `nParam` holds the index of the port that will perform the mark. The `pCmdData` parameter of `OMX_SendCommand` points to an `OMX_MARKTYPE` structure. The `pMarkTargetComponent` field of this structure holds a pointer to the component that will send an event after processing the marked buffer. The `pMarkData` field of this structure holds a pointer to application-specific data associated with the mark to uniquely identify the mark to the application upon a mark event (denoted the *mark data*).

When instructed to mark a buffer, the component will mark the next buffer that it receives as input after it receives the mark command. The exception is a source component, which will mark the next buffer it adds to its output buffer queue. For components other than source components, the port index value in `nParam` holds the

index of the input port that will mark its next buffer. For source components, the port index value in nParam holds the index of the output port that will mark its next buffer.

In the following cases, multiple marks may compete for a single buffer:

- A component receives two or more mark commands with no intervening buffer(s).
- Two or more input buffers, each with a mark, contribute to an output buffer (e.g., in a mixer).
- A component receives a mark command and the next buffer is already marked.

If multiple marks compete for application to the same buffer, the component uses the first mark received to mark the buffer and applies the remaining marks to subsequent buffers in the order that the component received them. If there are no subsequent buffers, the component may send the remaining marks on one or more empty buffers.

For each case where the component successfully marks a buffer, the component shall send an OMX_EventCmdComplete event indicating OMX_CommandMarkBuffer for nData1, the individual port index for nData2, and OMX_ErrorNone for pEventData. If a mark operation fails, the component shall notify the IL client of the error via OMX_EventCmdComplete event with an appropriate error code for pEventData.

A buffer header includes pMarkTargetComponent and the pMarkData fields, whose meaning is identical to those in OMX_MARKTYPE. A component marks a buffer by copying pMarkTargetComponent and the pMarkData fields from the mark command to the buffer headers. Both fields are NULL by default (i.e., before the buffer being marked). A component propagates the mark fields from an input buffer to an output buffer according to the buffer metadata rules established for buffer flags and timestamps. The target component does not propagate the mark but instead clears both fields to NULL.

When a component receives a buffer, it shall compare its own pointer to the pMarkTargetComponent. If the pointers match, the component shall send a mark event, including pMarkData as a parameter, immediately after the buffer exits the component or has been completely processed in the case where it does not exit the component.

A component shall accept an OMX_CommandMarkBuffer request when in the OMX_StateExecuting or OMX_StatePause states.

If the port associated with the OMX_CommandMarkBuffer request is disabled or the component is in the OMX_StatePause state, the component shall queue the requests and commence processing these queued requests when in the OMX_StateExecuting state and the associated port is enabled.

If a component has queued OMX_CommandMarkBuffer requests upon transitioning to OMX_StateIdle, it shall automatically flush these queued requests - complete the command processing. The component shall emit a failed OMX_EventCmdComplete

callback (with an appropriate error code for `pEventData`) as the completion status for each queued request being flushed.

`OMX_MARKTYPE` is defined as follows.

```
typedef struct OMX_MARKTYPE {
    OMX_HANDLETYPE hMarkTargetComponent;
    OMX_PTR pMarkData;
} OMX_MARKTYPE;
```

The parameters are described as follows.

Parameter	Description
hMarkTargetComponent	Identifies the component handle that shall generate a mark event upon process the mark.
nMarkData	Application specific data associated with mark sent on a mark event to disambiguate a mark from others.

3.2.2.7.1 Prerequisites for This Method

This method has no prerequisites.

3.2.2.7.2 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* instructs a component port to mark a buffer*/

OMX_MARKTYPE mark;

mark.hMarkTargetComponent = hComp;
mark.pMarkData = appData;

OMX_SendCommand(hComp, OMX_CommandMarkBuffer, portIndex, &mark);
```

3.2.2.8 OMX_GetParameter

The `OMX_GetParameter` macro will get a parameter setting from a component. The `nParamIndex` parameter indicates which structure is requested from the component. The caller shall provide memory for the structure and populate the `nSize` and `nVersion` fields before invoking this macro. If the parameter settings are for a port, the caller shall also provide a valid port number in the `nPortIndex` field before invoking this macro. All components shall support a set of defaults for each parameter so that the caller can obtain the structure populated with valid values.

This call is a blocking call. The component should return from this call within 20 milliseconds.

The `OMX_GetParameter` macro is defined as follows.

```
#define OMX_GetParameter (
  hComponent,
  nParamIndex,
pComponentParameterStructure)
((OMX_COMPONENTTYPE*)hComponent)->GetParameter(          \
        hComponent,                                      \
        nParamIndex,                                     \
        pComponentParameterStructure)
```

The parameters are described as follows.

Parameter	Description
hComponent [in]	The handle of the component that executes the call
nParamIndex [in]	The index of the structure to be filled. This value is from the `OMX_INDEXTYPE` enumeration.
pComponentParameterStructure [in,out]	A pointer to the IL client-allocated structure that the component fills

Section 3.3—OpenMAX IL Component Methods and Structures describes the corresponding function that each component implements.

3.2.2.8.1 Prerequisites for This Method

This method has no prerequisites.

3.2.2.8.2 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* disable every audio port of a component*/
OMX_GetParameter(hComp, OMX_IndexParamAudioInit, &oParam);
for (i=0;i<oParam.nPorts;i++) {
      OMX_SendCommand(
            hComp,
            OMX_CommandPortDisable,
            oParam.nStartPortNumber + i,
            0);
}
```

3.2.2.8.3 Error Conditions

The following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the parameter structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the specified parameter index is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the parameter structure does not match the expected version for the component.

- `OMX_ErrorNotReady` if an `OMX_GetParameter` operation has not completed processing. The caller should retry the `OMX_GetParameter` call.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called with a structure that includes the `nPortIndex` field and the value of `nPortIndex` exceeds the number of ports (of the appropriate domain) for the component.

3.2.2.9 OMX_SetParameter

The `OMX_SetParameter` macro will send a parameter structure to a component. The `nParamIndex` parameter indicates which structure is passed to the component.

The caller shall provide the memory for the correct structure and shall fill in the structure `nSize` and `nVersion` fields in addition to all other fields before invoking this macro. The caller is free to dispose of this structure after the call, as the component is required to copy any data it shall retain.

Some parameter structures contain read-only fields. The `OMX_SetParameter` method will preserve read-only fields, and shall not generate an error when the caller attempts to change the value of a read-only field.

This call is a blocking call. The component should return from this call within 20 milliseconds.

The `OMX_SetParameter` macro is defined as follows.

```
#define OMX_SetParameter (
  hComponent,
  nParamIndex,
pComponentParameterStructure)
((OMX_COMPONENTTYPE*)hComponent)->SetParameter(          \
        hComponent,                                      \
        nParamIndex,                                     \
        pComponentParameterStructure)
```

The parameters are as follows.

Parameter	Description
hComponent [in]	The handle of the component that executes the call.
nParamIndex [in]	The index of the structure that is to be sent. This value is from the `OMX_INDEXTYPE` enumeration.
pComponentParameterStructure [in]	A pointer to the IL client-allocated structure that the component uses for initialization.

Section 3.3.6 below describes the corresponding function that each component implements.

3.2.2.9.1 Prerequisites for This Method

The `OMX_SetParameter` macro can be invoked only when the component is in the `OMX_StateLoaded` state or on a port that is disabled.

The only exception to this prerequisite is when the component is transitioning from `OMX_StateLoaded` to `OMX_StateIdle`, or the port is being enabled. In these two cases, the component shall accept calls to `OMX_SetParameter` with the `OMX_IndexParamPortDefinition` index. This is needed to modify the value of any of the writable fields in the `OMX_PARAM_PORTDEFINITIONTYPE` structure on a non-supplier port, before any buffers have been allocated on that port. After the first buffer is allocated, `OMX_SetParameter` shall no longer be accepted by the port.

3.2.2.9.2 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* force a port to be the supplier */
OMX_GetParameter(hComp, OMX_IndexParamPortDefinition, &oPortDef);
if (oPortDef.eDir == OMX_DirInput){
      oSupplier.eBufferSupplier = OMX_BufferSupplyInput;
} else {
      oSupplier.eBufferSupplier = OMX_BufferSupplyOutput;
}
oSupplier.nPortIndex = nPortIndex;
OMX_SetParameter(hComp, OMX_IndexParamCompBufferSupplier, &oSupplier);
```

3.2.2.9.3 Error Conditions

The following error conditions can occur:

- `OMX_ErrorIncorrectStateOperation` when the `OMX_SetParameter` function is called and the component is not in the `OMX_StateLoaded` state, or the port is not disabled.
- `OMX_ErrorBadParameter` if one or more fields of the parameter structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the specified parameter index is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the parameter structure does not match the expected version for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the parameter structure is unsupported by the component during an `OMX_SetParameter` call.
- `OMX_ErrorNotReady` if an `OMX_SetParameter` operation has not completed processing. The caller should retry the `OMX_SetParameter` call.

3.2.2.10 OMX_GetConfig

The `OMX_GetConfig` macro will get a configuration structure from a component. This macro can be invoked at any time after the component has been loaded. The `nConfigIndex` parameter indicates which structure is being requested from the component. The caller shall provide the memory for the structure and populate the `nSize` and `nVersion` fields before invoking this macro. If the configuration settings are for a port, the caller shall also provide a valid port number in the `nPortIndex` field before invoking this macro. All components shall support a set of defaults for each configuration so that the caller can obtain the structure populated with valid values.

This call is a blocking call. The component should return from this call within five milliseconds.

The `OMX_GetConfig` macro is defined as follows.

```
#define OMX_GetConfig (
  hComponent,
  nConfigIndex,
pComponentConfigStructure)
((OMX_COMPONENTTYPE*)hComponent)->GetConfig(            \
        hComponent,                                     \
        nConfigIndex,                                   \
        pComponentConfigStructure)
```

The parameters are as follows.

Parameters	**Description**
hComponent [in]	The handle of the component that executes the call.
nConfigIndex [in]	The index of the structure to be filled. This value is from the `OMX_INDEXTYPE` enumeration.
pComponentConfigStructure [in,out]	A pointer to the IL client-allocated structure that the component fills.

Section 3.3.7 below describes the corresponding function that each component implements.

3.2.2.10.1 Prerequisites for This Method

This method has no prerequisites.

3.2.2.10.2 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* Wait until a certain playback position */
do {
OMX_GetConfig(hClockComp, OMX_IndexConfigTimeCurrentMediaTime,
              oMediaTime);
} while (oMediaStamp.nTimeStamp < nTargetTimeStamp);
```

3.2.2.10.3 Error Conditions

The following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the config structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the specified config index is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the config structure does not match the expected version for the component.
- `OMX_ErrorNotReady` if an `OMX_GetConfig` operation has not completed processing. The caller should retry the `OMX_GetConfig` call.
- `OMX_ErrorNoMore` when the `OMX_GetConfig` function is called with a structure that includes the `nPortIndex` field and the value of `nPortIndex` exceeds the number of ports (of the appropriate domain) for the component.

3.2.2.11 OMX_SetConfig

The `OMX_SetConfig` macro will set a component configuration value. This macro can be invoked anytime after the component has been loaded.

The caller shall provide the memory for the correct structure and fill in the structure `nSize` and `nVersion` fields in addition to all other fields before invoking this macro. The caller can dispose of this structure after the call, as the component is required to copy any data it shall retain.

Some configuration structures contain read-only fields. The `OMX_SetConfig` method will preserve read-only fields in configuration structures that contain them, and shall not generate an error when the caller attempts to change the value of a read-only field.

This call is a blocking call. The component should return from this call within five milliseconds.

The `OMX_SetConfig` macro is defined as follows.

```
#define OMX_SetConfig (
  hComponent,
  nConfigIndex,
  pComponentConfigStructure     )
((OMX_COMPONENTTYPE*)hComponent)->SetConfig(            \
        hComponent,                                     \
        nConfigIndex,                                   \
        pComponentConfigStructure)
```

The parameters are as follows.

Parameter	Description
hComponent [in]	The handle of the component that executes the call.

Parameter	Description
nConfigIndex [in]	The index of the structure that is to be sent. This value is from the OMX_INDEXTYPE enumeration.
pComponentConfigStructure [in]	A pointer to the IL client-allocated structure that the component uses for initialization.

Section 3.3.8 below describes of the corresponding function that each component implements.

3.2.2.11.1 Prerequisites for This Method

This method has no prerequisites.

3.2.2.11.2 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* Change the time scale of the clock component*/
oScale.xScale = 0x00020000; /*2x*/
OMX_SetConfig(hClockComp, OMX_IndexConfigTimeScale, (OMX_PTR)&oScale);
```

3.2.2.11.3 Error Conditions

The following error conditions can occur:

- OMX_ErrorBadParameter if one or more fields of the config structure are incorrect.
- OMX_ErrorUnsupportedIndex when the specified config index is unsupported.
- OMX_ErrorVersionMismatch when the nVersion field of the config structure does not match the expected version for the component.
- OMX_ErrorUnsupportedSetting when a field in the config structure is unsupported by the component during an OMX_SetConfig call.
- OMX_ErrorNotReady if an OMX_SetConfig operation has not completed processing. The caller should retry the OMX_SetConfig call.

3.2.2.12 OMX_GetExtensionIndex

The OMX_GetExtensionIndex macro will invoke a component to convert a vendor-specific extension string to an OpenMAX IL configuration or parameter index. The vendor is not required to support this command for the indexes already found in the OMX_INDEXTYPE enumeration, which reduces the memory footprint. The component may support any standardized OpenMAX IL or vendor-specific extension indexes that are not found in the master OMX_INDEXTYPE enumeration.

This call is a blocking call. The component should return from this call within five milliseconds.

The `OMX_GetExtensionIndex` macro is defined as follows.

```
#define OMX_GetExtensionIndex (
  hComponent,
  cParameterName,
  pIndexType     )
((OMX_COMPONENTTYPE*)hComponent)->GetExtensionIndex(     \
        hComponent,                                      \
        cParameterName,                                  \
        pIndexType)
```

The parameters are as follows.

Parameter	Description
hComponent [in]	The handle of the component that executes the call.
cParameterName [in]	An `OMX_STRING` value that shall be less than 128 characters long including the trailing null byte. The component will translate this string into a configuration index.
pIndexType [out]	A pointer to the `OMX_INDEXTYPE` structure that is to receive the index value.

Section 3.3.9 below describes the corresponding function that each component implements.

3.2.2.12.1 Prerequisites for This Method

This method has no prerequisites.

3.2.2.12.2 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* Set the vendor-specific filename parameter on a reader */
OMX_GetExtensionIndex(
      hFileReaderComp,
      "OMX.CompanyXYZ.index.param.filename",
      &eIndexParamFilename);
OMX_SetParameter(hComp, eIndexParamFilename, &oFileName);
```

3.2.2.13 OMX_GetState

The `OMX_GetState` macro will invoke the component to get the current state of the component and place the state value into the location pointed to by `pState`. The component should return from this call within five milliseconds.

The `OMX_GetState` macro is defined as follows.

```
#define OMX_GetState (
```

```
  hComponent,
  pState     )
((OMX_COMPONENTTYPE*)hComponent)->GetState(             \
        hComponent,                                     \
        pState)
```

The parameters are as follows.

Parameter	Definition
hComponent [in]	The handle of the component that executes the call.
pState [out]	A pointer to the location that receives the state. The value returned is one of the OMX_STATETYPE members.

Section 3.3.10 below describes the corresponding function that each component implements.

3.2.2.13.1 Prerequisites for This Method

This method has no prerequisites.

3.2.2.13.2 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
OMX_SendCommand(hComp, OMX_CommandStateSet, OMX_StateIdle, 0);
do {
OMX_GetState(hComp, &eState);
} while (OMX_StateIdle != eState);
```

3.2.2.14 OMX_UseBuffer

The `OMX_UseBuffer` macro requests the component to use a buffer allocated by the IL client or a buffer supplied by a tunneled component. The `OMX_UseBuffer` implementation shall allocate the buffer header, populate it with the given input parameters, and pass it back via the `ppBufferHdr` output parameter.

When populating fields within the buffer header structure, components are required to correctly initialize both `pInputPortIndex` and `pOutputPortIndex`. They are also required to initialize the `pAppPrivate` field with the `pAppPrivate` function parameter. The `pAppPrivate` parameter should also be used to initialize the `pInputPortPrivate` or `pOutputPortPrivate` field, when called on an output port or input port respectively.

When `OMX_UseBuffer` is used for pre-announcing a buffer pointer, the `pBuffer` and `nAllocLen` fields in the allocated buffer header shall be initialized with the `pBuffer` and `nSizeBytes` function parameters, respectively. When the `pBuffer` function parameter is NULL, the component shall initialize `pBuffer` and `nAllocLen` with NULL and zero, respectively.

The OMX_UseBuffer macro can be called on a component under the following conditions:

- While the component is in the OMX_StateLoaded state and has already received a command to transition to the OMX_StateIdle state.
- While the component is in the OMX_StateWaitForResources state, the resources needed are available, and the component is ready to go to the OMX_StateIdle state
- On a disabled port when the component is in one of the OMX_StateExecuting, the OMX_StatePause, or the OMX_StateIdle states and has already received a command to enable the port.
- In tunneled case for a non-supplier port, the component shall call OMX_SetConfig(OMX_PORTSTATUS_ACCEPTUSEBUFFER) on its tunneled component to indicate when it can start calling UseBuffer.

This is a blocking call. The component should return from this call within 20 milliseconds.

The OMX_UseBuffer macro is defined as follows.

```
#define OMX_UseBuffer(                                  \
     hComponent,                                        \
     ppBufferHdr,                                       \
     nPortIndex,                                        \
     pAppPrivate,                                       \
     nSizeBytes,                                        \
     pBuffer)                                           \
((OMX_COMPONENTTYPE*)hComponent->UseBuffer(             \
     hComponent,                                        \
     ppBufferHdr,                                       \
     nPortIndex,                                        \
     pAppPrivate,                                       \
     nSizeBytes,                                        \
     pBuffer)
```

The parameters are as follows.

Parameter	Description
hComponent [in]	The handle of that component that executes the call.
ppBufferHdr [out]	A pointer to a pointer of an OMX_BUFFERHEADERTYPE structure that receives the pointer to the buffer header.
nPortIndex [in]	The index of the port that will use the specified buffer. This index is relative to the component that owns the port.
pAppPrivate [in]	A pointer that refers to an implementation-specific memory area that is under responsibility of the supplier of the buffer.
nSizeBytes [in]	The buffer size in bytes.

Parameter	Description
pBuffer [in]	A pointer to the memory buffer area to be used. When non-NULL, the call will establish a one-to-one relationship between the memory buffer and the buffer header. When NULL, the actual memory buffer is transferred only via OMX_EmptyThisBuffer or OMX_FillThisBuffer calls.

Section 3.3.12 below describes the corresponding function that each component implements.

3.2.2.14.1 Prerequisites for This Method

The component shall be in the `OMX_StateLoaded` or the `OMX_StateWaitForResources` state, or the port to which the call applies shall be disabled.

3.2.2.14.2 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* supplier port allocates buffers and pass them to non-supplier */
for (i=0;i<pPort->nBufferCount;i++)
{
      pPort->pBuffer[i] = malloc(pPort->nBufferSize);
     OMX_UseBuffer(pPort->hTunnelComponent,
                   &pPort->pBufferHdr[i],
                   pPort->nTunnelPort,
                   pPort,
                   pPort->nBufferSize,
                   pPort->pBuffer[i]);
}
```

3.2.2.15 OMX_AllocateBuffer

The `OMX_AllocateBuffer` macro will request that the component allocate a new buffer and buffer header. The component will allocate the buffer and the buffer header and return a pointer to the buffer header.

When populating fields within the buffer header structure, components are required to correctly initialize both `pInputPortIndex` and `pOutputPortIndex`. They are also required to initialize the `pAppPrivate` field with the `pAppPrivate` function parameter. The `pAppPrivate` parameter should also be used to initialize the `pInputPortPrivate` or `pOutputPortPrivate` field, when called on an output port or input port respectively.

The `OMX_AllocateBuffer` macro can be called on a component under the following conditions:

- While the component is in the `OMX_StateLoaded` state and has already received a command to transition to `OMX_StateIdle`.

- While the component is in the `OMX_StateWaitForResources` state, the resources needed are available, and the component is ready to go to the `OMX_StateIdle` state.
- On a disabled port when the component is in one of the `OMX_StateExecuting`, the `OMX_StatePause`, or the `OMX_StateIdle` states and has already received a command to enable the port.

The `OMX_AllocateBuffer` macro allocates buffers on a specific port for communication with the IL client only. This macro cannot be used to allocate buffers for tunneled ports. Buffers allocated before a port was configured for tunneling will result in the component failing `OMX_SetupTunnel` calls to the port.

The component should return from this call within five milliseconds.

The `OMX_AllocateBuffer` macro is defined as follows.

```
#define OMX_AllocateBuffer(                                     \
        hComponent,                                             \
        ppBuffer,                                               \
        nPortIndex,                                             \
        pAppPrivate,                                            \
        nSizeBytes)                                             \
    ((OMX_COMPONENTTYPE*)hComponent)->AllocateBuffer(           \
        hComponent,                                             \
        ppBuffer,                                               \
        nPortIndex,                                             \
        pAppPrivate,                                            \
        nSizeBytes)
```

The parameter are as follows.

Parameter	Description
hComponent [in]	The handle of the component that executes the call.
ppBuffer [out]	A pointer to a pointer of an `OMX_BUFFERHEADERTYPE` structure that receives the pointer to the buffer header.
nPortIndex [in]	Selects the port on the component that the buffer will be used with. The port can be found by using the `nPortIndex` value as an index into the port definition array of the component.
pAppPrivate [in]	Initializes the `pAppPrivate` member of the buffer header structure.
nSizeBytes [in]	The size of the buffer to allocate.

Section 3.3.13 below describes the corresponding function that each component implements.

3.2.2.15.1 Prerequisites for This Method

The component shall be in the `OMX_StateLoaded` or the `OMX_StateWaitForResources` state, or the port to which the call applies shall be disabled.

3.2.2.15.2 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* IL client asks component to allocate buffers */
for (i=0;i<pClient->nBufferCount;i++)
{
      OMX_AllocateBuffer(hComp,
                  &pClient->pBufferHdr[i],
                  pClient->nPortIndex,
                  pClient,
                  pClient->nBufferSize);
}
```

3.2.2.16 OMX_FreeBuffer

The `OMX_FreeBuffer` macro will release a buffer and buffer header from the component. The component shall free only the buffer header if it allocated only the buffer header. The component shall free both the buffer and the buffer header if it allocated both the buffer and the buffer header. Thus, the component shall track which buffers it allocated so it can perform the corresponding de-allocation.

The call should be performed under the following conditions:

- While the component is in the `OMX_StateIdle` state and the IL client has already sent a request for the state transition to `OMX_StateLoaded` (e.g., during the stopping of the component)
- On a disabled port when the component is in the `OMX_StateExecuting`, the `OMX_StatePause`, or the `OMX_StateIdle` state.

The call can be made at any time provided the caller owns the buffer, but may result in the port sending an `OMX_ErrorPortUnpopulated` event error if the call is not performed as described.

The call is made from suppliers to release buffer headers from non-supplier ports.

This call is a blocking call. The component should return from the call within 20 milliseconds.

The `OMX_FreeBuffer` macro is defined as follows.

```
#define OMX_FreeBuffer (
  hComponent,
  nPortIndex,
  pBuffer     )
  ((OMX_COMPONENTTYPE*)hComponent)->FreeBuffer(           \
        hComponent,                                       \
```

```
        nPortIndex,                                          \
        pBuffer)
```

The parameters are as follows.

Parameter	Description
hComponent [in]	The handle of the component that executes the call
nPortIndex [in]	The index of the port that is using the specified buffer
pBuffer [in]	A pointer to an OMX_BUFFERHEADERTYPE structure used to provide or receive the pointer to the buffer header.

Section 3.3.14 describes the corresponding function that each component implements.

3.2.2.16.1 Prerequisites for This Method

The component should be in the OMX_StateIdle state or the port should be disabled.

3.2.2.16.2 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* supplier port frees buffers */
for (i=0;i<pPort->nBufferCount;i++)
{
      free(pPort->pBuffer[i]);
      pPort->pBuffer[i] = 0;
      OMX_FreeBuffer(pPort->hTunnelComponent,
                  pPort->nTunnelPort,
                  pPort->pBufferHdr[i]);
      pPort->pBufferHdr[j] = 0;
}
```

3.2.2.17 OMX_EmptyThisBuffer

The OMX_EmptyThisBuffer macro will send a filled buffer to an input port of a component. When the buffer contains data, the value of the nFilledLen field of the buffer header will not be zero. If the buffer contains no data, the value of nFilledLen is 0x0. The OMX_EmptyThisBuffer macro shall be executed to pass buffers containing data when the component is in one of the OMX_StateExecuting or OMX_StatePause states. In tunneled case, a component shall call OMX_SetConfig(OMX_PORTSTATUS_ACCEPTBUFFEREXCHANGE) on its tunneled component to indicate when it can start exchanging buffers.

When a port is non-tunneled, buffers sent using OMX_EmptyThisBuffer are returned to the IL client with the EmptyBufferDone callback.

When a port is tunneled, buffers sent using `OMX_EmptyThisBuffer` can be returned using `OMX_FillThisBuffer`.

This call is a non-blocking call since the component will queue the buffer and return immediately. The buffer will be emptied later at the proper time. If the parameter `nInputPortIndex` in the buffer header does not specify a valid input port, the component returns `OMX_ErrorBadPortIndex.` The component should return from this call within five milliseconds.

The `OMX_EmptyThisBuffer` macro is defined as follows.

```
#define OMX_EmptyThisBuffer (
  hComponent,
  pBuffer     )
  ((OMX_COMPONENTTYPE*)hComponent)->EmptyThisBuffer(      \
        hComponent,                                       \
        pBuffer)
```

The parameters are as follows.

Parameter	**Description**
hComponent [in]	The handle of the component that executes the call.
pBuffer [in]	A pointer to an `OMX_BUFFERHEADERTYPE` structure that is used to provide or receive the pointer to the buffer header. The buffer header shall specify the index of the input port that receives the buffer

Section 3.3.15 below describes the corresponding function that each component implements.

3.2.2.17.1 Prerequisites for This Method

The component shall be in the appropriate state as shown in Table 3-17.

3.2.2.17.2 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* deliver full buffer */
if (pPort->hTunnelComponent)
OMX_EmptyThisBuffer(pPort->hTunnelComponent, pBuffer);
else
      pCallbacks->FillBufferDone(hComp, pBuffer,
pPort->pCallbackAppData);
```

3.2.2.18 OMX_FillThisBuffer

The `OMX_FillThisBuffer` macro will send an empty buffer to an output port of a component. The `OMX_FillThisBuffer` macro shall be executed to pass buffers containing no data when the component is in one of the `OMX_StateExecuting` or `OMX_StatePause` states. In tunneled case, a component shall call

`OMX_SetConfig(OMX_PORTSTATUS_ACCEPTBUFFEREXCHANGE)` on its tunneled component to indicate when it can start exchanging buffers.

When a port is non-tunneled, buffers sent using `OMX_FillThisBuffer` return to the IL client with the `FillBufferDone` callback.

When a port is tunneled, buffers sent using `OMX_FillThisBuffer` can be returned using `OMX_EmptyThisBuffer`..

This call is a non-blocking call since the component will queue the buffer and return immediately. The buffer will be filled later at the proper time. If the parameter `nOutputPortindex` in the buffer header does not specify a valid output port, the component returns `OMX_ErrorBadPortIndex`. The component should return from this call within five milliseconds.

The `OMX_FillThisBuffer` macro is defined as follows.

```
#define OMX_FillThisBuffer (
  hComponent,
  pBuffer     )
  ((OMX_COMPONENTTYPE*)hComponent)->FillThisBuffer(       \
        hComponent,                                       \
        pBuffer)
```

The parameters are as follows.

Parameter	Description
hComponent [in]	The handle of the component that executes the call.
pBuffer [in]	A pointer to an `OMX_BUFFERHEADERTYPE` structure used to provide or receive the pointer to the buffer header. The buffer header shall specify the index of the output port that receives the buffer.

Section 3.3.16 below describes the corresponding function that each component implements.

3.2.2.18.1 Prerequisites for This Method

The component shall be in the appropriate state as shown in Table 3-17.

3.2.2.18.2 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* On a port enable, if tunneling and an input and not supplier */
/* then give buffers to supplier port */
if (pPort->hTunnelComponent &&
    (pPort->oPortDef.eDir == OMX_DirInput) &&
    (pPort->eSupplierSetting == OMX_BufferSupplyInput) )
{
   for (i=0;i<pPort->nBuffers;i++){
OMX_FillThisBuffer(pPort->hTunnelComponent,
pPort->ppBufferHdrs[i]);
```

```
    }
}
```

3.2.2.19 OMX_UseEGLImage

`OMX_UseEGLImage` enables an OMX IL component to use as a buffer, the image already allocated via EGL. EGLImages are designed for sharing data between rendering based EGL interfaces, such as OpenGL ES and OpenVG. The format of an EGLImage is opaque to the EGL's client by design, so any memory allocated through this macro are not accessible directly by the IL client.

A method for this interface shall be provided by the component, but may not be implemented, by returning `OMX_ErrorNotImplemented`. Components should inspect the EGLImage provided to the method, and determine if the EGLImage is compatible with the port configuration.

The `OMX_UseEGLImage` macro requests that the component use an EGLImage provided by EGL, in place of using the `OMX_UseBuffer` method. The `OMX_UseEGLImage` implementation shall allocate the buffer header, populate it with the given input parameters, and pass it back via the `ppBufferHdr` output parameter. The `pBuffer` field of the pBufferHdr parameter shall be 0x0, because the format of the EGLImage is opaque to the IL client.

The `OMX_UseEGLImage` macro shall be executed under the following conditions:

- While the component is in the `OMX_StateLoaded` state and has already sent a request for the state transition to `OMX_StateIdle`.
- While the component is in the `OMX_StateWaitForResources` state, the resources needed are available, and the component is ready to go to the `OMX_StateIdle` state.
- On a disabled port when the component is in the `OMX_StateExecuting`, the `OMX_StatePause`, or the `OMX_StateIdle` state.

This is a blocking call. The component should return from this call within 20 milliseconds.

The `OMX_UseEGLImage` macro is defined as follows.

```
#define OMX_UseEGLImage(                                    \
      hComponent,                                           \
      ppBufferHdr,                                          \
      nPortIndex,                                           \
      pAppPrivate,                                          \
      eglImage)                                             \
((OMX_COMPONENTTYPE*)hComponent->UseEGLImage(               \
      hComponent,                                           \
      ppBufferHdr,                                          \
      nPortIndex,                                           \
      pAppPrivate,                                          \
      eglImage)
```

The parameters are as follows.

Parameter	Description
hComponent [in]	The handle of that component that executes the call.
ppBufferHdr [out]	A pointer to a pointer of an OMX_BUFFERHEADERTYPE structure that receives the pointer to the buffer header.
nPortIndex [in]	The index of the port that will use the specified buffer. This index is relative to the component that owns the port.
pAppPrivate [in]	A pointer that refers to an implementation-specific memory area that is under responsibility of the supplier of the buffer.
eglImage [in]	The handle of the EGLImage to use as a buffer on the specified port. The component is expected to validate properties of the EGLImage against the configuration of the port to ensure the component can use the EGLImage as a buffer.

Section 3.3.19 below describes the corresponding function that each component implements.

3.2.2.19.1 Prerequisites for This Method

The component shall be in the OMX_StateLoaded or the OMX_StateWaitForResources state, or the port to which the call applies shall be disabled.

3.2.2.20 OMX_SetCallbacks

The OMX_SetCallbacks macro will transfer new callbacks information from the IL client to the component. The OMX_SetCallbacks macro is invoked to pass a pointer to an OMX_CALLBACKTYPE structure containing the callbacks that the component will use for this IL client. Also a pointer to an IL client-defined value is passed. This value shall be returned to the IL client during callbacks so that the IL client can determine the context and/or the source of the callback.

In addition, the component shall update the pApplicationPrivate field of the OMX_COMPONENTTYPE structure during the call to OMX_SetCallbacks.

The component shall guarantee that once that this method returns, the pointers for any previously existing callbacks and IL client-defined values will no longer be used in future callbacks.

This call is a blocking call. The component should return from this call within 20 milliseconds.

The OMX_SetCallbacks macro is defined as follows.

```
#define OMX_SetCallbacks (\
  hComponent,\
```

```
  pCallbacks,\
  pAppData)\
  ((OMX_COMPONENTTYPE*)hComponent)->SetCallbacks(        \
        hComponent,                                      \
        pCallbacks,                                      \
        pAppData)
```

The parameters are as follows.

Parameter	Description
hComponent [in]	The handle of the component that executes the call.
pCallbacks [in]	A pointer to an OMX_CALLBACKTYPE structure that is used to provide the callback information to the component.
pAppData [in]	A pointer to a value that the IL client has defined (for example, a pointer to a data structure) that allows the callback in the IL client to determine the context of the call.

3.2.2.20.1 Prerequisites for This Method

The component shall be in the OMX_StateLoaded state.

3.2.2.20.2 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* On GetHandle (for statically linked components):
   create component, initialize it, and set its callbacks */
pComp = (OMX_COMPONENTTYPE *)malloc(sizeof(OMX_COMPONENTTYPE));
hHandle = (OMX_HANDLETYPE)pComp;
pComp->nVersion = version_1_2_0;
pComp->nSize = sizeof(OMX_COMPONENTTYPE);
OMX_ComponentRegistered[i].pInitialize(hHandle);
OMX_SetCallbacks(hHandle, pCallBacks, pAppData);
```

3.2.3 Functions

This section describes the functions in the OpenMAX IL API.

3.2.3.1 OMX_Init

The OMX_Init method initializes the OpenMAX IL core. Each OpenMAX IL client shall use OMX_Init as their first call into OpenMAX IL and this client shall later make a paired OMX_Deinit call, as the last call into OpenMAX IL. The OpenMAX IL core implementation shall keep track of OMX_Init and OMX_Deinit calls. The core should return from this call within 20 milliseconds.

The usage of OMX_Init() is as follows.

```
OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_Init()
```

3.2.3.1.1 Prerequisites for This Method

This method has no prerequisites.

3.2.3.1.2 Results/Outputs for This Method

If the command successfully executes, the return code will be `OMX_ErrorNone`. Otherwise, the appropriate OpenMAX IL error will be returned. The OpenMAX IL core functions are ready to be used when this function returns successfully.

3.2.3.1.3 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* Initialize OpenMAX IL and create some components */
OMX_Init();
OMX_GetHandle(hMp3Decoder, "OMX.CompanyXYZ.mp3.decoder",
      pAppData, pCallbacks);
OMX_GetHandle(hAudioMixer, "OMX.CompanyXYZ.audio.mixer",
      pAppData, pCallbacks);
```

3.2.3.2 OMX_Deinit

The final `OMX_Deinit` call de-initializes the OpenMAX IL core. An OpenMAX IL client shall use `OMX_Deinit` as their last call into OpenMAX IL, after all OpenMAX IL-related resources have been released. The client shall only call `OMX_Deinit` once per `OMX_Init` call. The OpenMAX IL core implementation shall keep track of `OMX_Init` and `OMX_Deinit` calls. If there are valid component handles when the final call to `OMX_Deinit` is made, the `OMX_Deinit` shall fail and return `OMX_ErrorIncorrectStateOperation`. The core should return from this call within 20 milliseconds.

The `OMX_Deinit` method usage is as follows.

```
OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_Deinit()
```

3.2.3.2.1 Prerequisites for This Method

The use of `OMX_Deinit` requires that all component handles acquired by the IL client in the system have been released, implying that all resources associated with components have been freed.

3.2.3.2.2 Results/Outputs for This Method

The use of `OMX_Deinit` returns `OMX_ERRORTYPE`. If the command successfully executes, the return code will be `OMX_ErrorNone`. Otherwise, the appropriate OpenMAX IL error will return.

3.2.3.2.3 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* Determine if a component of a particular name exists. */
OMX_Init();
eError = OMX_ErrorNone;
for (i=0; OMX_ErrorNone == eError; i++)
{
      eError = OMX_ComponentNameEnum(szCompEnumName, 256, i);
      if ((OMX_ErrorNone == eError) &&
          (!strcmp(szCompEnumName, szComponentName))
      {
            OMX_Deinit();
            return OMX_TRUE;
      }
}
OMX_Deinit();
return OMX_FALSE;
```

3.2.3.3 OMX_ComponentNameEnum

The `OMX_ComponentNameEnum` method will allow the IL client to enumerate through all the names of recognized components in the system to detect all the components in the system run-time. There is no strict ordering to the enumeration of component names, although each name shall be enumerated only once. If the OpenMAX IL core supports run-time installation of new components, it is required to detect newly installed components only when the first call to enumerate component names occurs (i.e., when the value of `nIndex` is 0x0).

The `OMX_ComponentNameEnum` method is defined as follows.

```
OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_ComponentNameEnum(
  OMX_OUT OMX_STRING    cComponentName,
  OMX_IN OMX_U32        nNameLength,
  OMX_IN OMX_U32        nIndex
 )
```

The parameters are as follows.

Parameter	**Description**
cComponentName [out]	A pointer to a null-terminated string with the component name. Component names are strings limited to less than 127 bytes in length plus the trailing null for a maximum length of 128 bytes. An example of a valid component name is "OMX.<vendor_name>.AUDIO.DSP.MIXER\0". The name shall start with "OMX." concatenated to a vendor-specified string.
nNameLength [in]	The number of characters in the `cComponentName` string. Since all component name strings are restricted to less than 128 characters, not including the trailing null, the caller should provide an input string of at least 128 characters.
nIndex [in]	A number containing the enumeration index for the component. Multiple calls to `OMX_ComponentNameEnum` with increasing values of `nIndex`

Parameter	Description
	will enumerate through the component names in the system until `OMX_ErrorNoMore` returns. The value of `nIndex` is 0 to N-1, where N is the number of installed components in the system.

3.2.3.3.1 Prerequisites for This Method

`OMX_ComponentNameEnum` can be called after the `OMX_Init` function.

3.2.3.3.2 Results/Outputs for This Method

If `OMX_ComponentNameEnum` successfully executes, the return code will be `OMX_ErrorNone`. When the value of `nIndex` exceeds the number of components in the system minus 1, `OMX_ErrorNoMore` will be returned. Otherwise, the appropriate OpenMAX IL error will be returned.

3.2.3.3.3 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* print a list of all components */
eError = OMX_ErrorNone;
for (i=0; OMX_ErrorNoMore != eError; i++)
{
      eError = OMX_ComponentNameEnum(szCompName, 256, i);
      if (OMX_ErrorNone == eError)
            printf("Component %i: %s\n", szCompName);
}
```

3.2.3.4 OMX_GetHandle

The `OMX_GetHandle` method will locate the component specified by the component name given, load that component into memory, and validate it. If the component is valid, `OMX_GetHandle` will invoke the component's methods to fill the component handle and set up the callbacks. The `OMX_GetHandle` method will allocate the actual `OMX_HANDLETYPE` structure, ensures it is populated correctly, and then updates the value of `*pHandle` with a pointer to the newly created handle. The component should return from this call within 20 milliseconds.

Each time the `OMX_GetHandle` function returns successfully, a new component instance is created. The IL client shall configure the newly created component, which is in the `OMX_StateLoaded` state, before the component can be used. After creating a new component all ports shall default to enabled and not connected to any other ports via a tunnel.

Since components are requested by name, a naming convention is defined. OpenMAX IL component names are zero terminated strings with the following format:

"OMX.<vendor_name>.<vendor_specified_convention>".

For example:

OMX.CompanyABC.MP3Decoder.productXYZ

No standardization among component names is dictated across different vendors.

`OMX_GetHandle` is defined as follows.

```
OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_GetHandle(
  OMX_OUT OMX_HANDLETYPE *    pHandle,
  OMX_IN OMX_STRING           cComponentName,
  OMX_IN OMX_PTR              pAppData,
  OMX_IN OMX_CALLBACKTYPE *   pCallBacks
 )
```

The parameters are as follows.

Parameter	Description
pHandle *[out]*	A pointer to `OMX_HANDLETYPE` to be filled in by this method.
cComponentName *[in]*	A pointer to a null-terminated string with the component name. Component names are strings limited to less than 128 bytes in length plus the trailing null for a maximum length of 128 bytes. An example of a valid component name is “OMX.<vendor_name>.AUDIO.DSP.MIXER\0”. The name shall start with “OMX.” concatenated to a vendor-specified string.
pAppData *[in]*	A pointer to an IL client-defined value that will be returned during callbacks so that the IL client can identify the source of the callback.
pCallBacks *[in]*	A pointer to an `OMX_CALLBACKTYPE` structure containing the callbacks that the component will use for this IL client.

3.2.3.4.1 Prerequisites for This Method

The OpenMAX IL core shall be initialized.

3.2.3.4.2 Results/Outputs for This Method

If successful, the function returns a valid component handle to the IL client.

3.2.3.4.3 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* determine maximum number of instantiations of a component */
eError = OMX_ErrorNone;
for (i=0; OMX_ErrorNone == eError; i++)
{
      eError = OMX_GetHandle(&hComp[i],
                  szComponentName,
                  pAppData,
                  pCallbacks);
}
printf("Created %i instantiations.\n",i);
```

3.2.3.5 OMX_FreeHandle

The `OMX_FreeHandle` method will free a handle allocated by the `OMX_GetHandle` method. The component should return from this call within 20 milliseconds. The IL client should call `OMX_FreeHandle` only when the component is in the `OMX_StateLoaded` and when all the ports are not connected via any tunnels.

`OMX_FreeHandle` is defined as follows.

```
OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_FreeHandle(
   OMX_IN OMX_HANDLETYPE    hComponent   )
```

The single parameter is as follows.

Parameter	Description
hComponent [in]	The handle of the component to be freed.

3.2.3.5.1 Prerequisites for This Method

The component should be in the `OMX_StateLoaded` state when this method is called.

3.2.3.5.2 Results/Outputs for This Method

All resources associated with the components are freed.

3.2.3.5.3 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* stop executing component and clean up component */
OMX_SendCommand(hComp, OMX_CommandStateSet, OMX_StateIdle, 0);
OMX_SendCommand(hComp, OMX_CommandStateSet, OMX_StateLoaded, 0);
do {
      OMX_GetState(hComp, &eState);
} while (OMX_StateLoaded != eState);
OMX_FreeHandle(hComp);
```

3.2.3.6 OMX_SetupTunnel

The `OMX_SetupTunnel` method sets up tunneled communication between an output port and an input port. This method is an actual method and not a defined macro. The `OMX_SetupTunnel` method will make calls to the components' `ComponentTunnelRequest()` method to set up the tunnel.

By default all ports are created not connected to any other port via a tunnel. After `OMX_SetupTunnel` returns successfully, both ports are connected together. Only after calling `OMX_TeardownTunnel` are these ports no longer connected. `OMX_SetupTunnel` shall only be called on ports that are not connected to any other ports.

When setting up tunneled communication between an output port and an input port, the method first issues a call to `ComponentTunnelRequest()` on the component with the output port. If the call is successful, a second call to `ComponentTunnelRequest()` on the component with the input port is made. Should either call to `ComponentTunnelRequest()` fail, the method will set up both the output and input ports for non-tunneled communication.

It is the responsibility of the input port to check that it is compatible with the output port.

The compatibility check shall be performed:

- At tunnel setup time when the `ComponentTunnelRequest()` method is called on the input port. `OMX_ErrorPortsNotCompatible` shall be returned if the compatibility check fails.
- When the input port is enabled. The component shall issue an `OMX_EventError` event with the value `OMX_ErrorPortsNotCompatible` when the compatibility check fails.
- When the component is transitioning from `OMX_StateLoaded` to `OMX_StateIdle` state. The component shall issue an `OMX_EventError` event with the value `OMX_ErrorPortsNotCompatible` when the compatibility check fails.

When checking the compatibility between two ports, the following rules shall be used:

- The domains (`eDomain` field within the `OMX_PARAM_PORTDEFINITIONTYPE` structure) of the two ports shall be compatible. In certain use cases, `OMX_PortDomainImage` and `OMX_PortDomainVideo` domains may be considered as compatible.
- When the domain is `OMX_PortDomainAudio`, the values of the `eEncoding` field within the `OMX_AUDIO_PORTDEFINITIONTYPE` structures of the two ports shall either be the same or be equal to `OMX_AUDIO_CodingUnused`.
- When the domain is `OMX_PortDomainVideo` or `OMX_PortDomainImage`, the values of the `nFrameWidth, nFrameHeight, nStride, nSliceHeight, nBitrate,` and `xFramerate fields` of the `OMX_VIDEO_PORTDEFINITIONTYPE` or `OMX_IMAGE_PORTDEFINITIONTYPE` structures of the two ports shall be the same. The `eCompressionFormat` of both ports shall be the same or eColorFormat of both ports, of the `OMX_VIDEO_PORTDEFINITIONTYPE` or `OMX_IMAGE_PORTDEFINITIONTYPE` structures, shall be the same.
- When the domain is `OMX_PortDomainOther`, the values of the `eFormat` field of the `OMX_OTHER_FORMATTYPE` structures of the two ports shall be the same.

- In case the `eEncoding` or `eCompressionFormat` or `eFormat` or `eColorFormat` fields of both ports are different from "unused" value (e.g. `OMX_AUDIO_CodingUnused`), then the component shall check the detailed settings of the current format by doing a `OMX_GetParameter` on the output port and comparing with the values of the input port.

  For example if the ports are of the audio domain and `eEncoding` is `OMX_AUDIO_CodingAAC`, then the component shall do a `OMX_GetParameter` with `OMX_IndexParamAudioAac` index to retrieve the detailed AAC settings of the output port and check that they are compatible with the settings of the input port. If `OMX_GetParameter` returns `OMX_ErrorUnsupportedIndex` or if the values of the fields are either unknown, don't care or variable, the component should assume that the tunneled component does not know the detailed settings, and the compatibility check shall succeed.

The components may negotiate proprietary communication in place of tunneled communication so long as both the output and input ports can support proprietary communication. An IL client cannot disambiguate between tunneled and proprietary communication.

The core should return from this call within 20 milliseconds.

The IL client may use `OMX_SetupTunnel` to establish proprietary communication between base profile components (given than both components support it) but not to establish a tunnel between them. An IL client may only establish tunnels between Interop profile components.

If this method fails because the `OMX_SetupTunnel` implementation supports neither tunneling nor proprietary communication then it shall return `OMX_ErrorNotImplemented`.

If this method fails because `OMX_SetupTunnel` supports proprietary communication but not tunneling and proprietary communication does not apply to the given components then is shall return `OMX_ErrorTunnelingUnsupported`.

`OMX_SetupTunnel` may only return `OMX_ErrorNotImplemented` or `OMX_ErrorTunnelingUnsupported` when operating on one or more base profile components; these errors do not apply when operating on two Interop profile components.

For a detailed description of the process to set up a data tunnel between two components, see section 3.4.1.2.

`OMX_SetupTunnel` is defined as follows.

```
OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_SetupTunnel(
  OMX_IN OMX_HANDLETYPE       hOutput,
  OMX_IN OMX_U32              nPortOutput,
  OMX_IN OMX_HANDLETYPE       hInput,
  OMX_IN OMX_U32              nPortInput
 )
```

The parameters are as follows.

Parameter	Description
hOutput [in]	The handle of the component containing the output port used in the tunnel, where the output port is identified by the `nPortOutput` parameter. By definition, an output port has the direction `OMX_DirOutput`. This parameter shall be a valid component handle.
nPortOutput [in]	Indicates the output port of the component specified by `hOutput` that is to be used for tunneled or proprietary communication.
hInput [in]	The handle of the component containing the input port used in the tunnel, where the input port is identified by the `nPortInput` parameter. By definition, an input port has the direction `OMX_DirInput`. This parameter shall be a valid component handle.
nPortInput [in]	Indicates the input port of the component specified by `hInput` that is to be used for tunneled or proprietary communication.

3.2.3.6.1 Prerequisites for This Method

Each component that is being tunneled shall be in the `OMX_StateLoaded` state, or its port shall be disabled, and both ports shall not be currently connected via any current tunnel.

3.2.3.6.2 Results/Outputs for This Method

If the method returns successfully, tunneled or proprietary communication has been set up between the specified output and input ports, and both ports are now connected together. If any error is returned then both ports are not connected via a tunnel.

3.2.3.6.3 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* set up tunnel between two components then transition to idle */
OMX_SetupTunnel(hCompA, nCompAOutPort, hCompB, nCompBInPort);
OMX_SendCommand(hCompA, OMX_CommandStateSet, OMX_StateIdle, 0);
OMX_SendCommand(hCompB, OMX_CommandStateSet, OMX_StateIdle, 0);
```

3.2.3.7 OMX_TeardownTunnel

The `OMX_TeardownTunnel` method clears tunneled communication between an output port and an input port. This method is an actual method and not a defined macro. The `OMX_TeardownTunnel` method will make calls to the components' `ComponentTunnelRequest()` methods to tear down the tunnel. `OMX_TeardownTunnel` shall only be called on a pair of ports that are current connected together. After `OMX_TeardownTunnel` returns successfully these ports are no longer connected together.

The core should return from this call within 20 milliseconds.
The IL client may use `OMX_TeardownTunnel` to disconnect components currently using proprietary communication.

`OMX_TeardownTunnel` is defined as follows.

```
OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_TeardownTunnel(
  OMX_IN OMX_HANDLETYPE       hOutput,
  OMX_IN OMX_U32              nPortOutput,
  OMX_IN OMX_HANDLETYPE       hInput,
  OMX_IN OMX_U32              nPortInput
 )
```

The parameters are as follows.

Parameter	Description
hOutput [in]	The handle of the component containing the output port used in the tunnel, where the output port is identified by the `nPortOutput` parameter. By definition, an output port has the direction `OMX_DirOutput`. This parameter shall be a valid component handle.
nPortOutput [in]	Indicates the output port of the component specified by `hOutput` that is currently being used for tunneled or proprietary communication.
hInput [in]	The handle of the component containing the input port used in the tunnel, where the input port is identified by the `nPortInput` parameter. By definition, an input port has the direction `OMX_DirInput`. This parameter shall be a valid component handle.
nPortInput [in]	Indicates the input port of the component specified by `hInput` that is currently being used for tunneled or proprietary communication.

3.2.3.7.1 Prerequisites for This Method

Each component that is being tunneled shall be in the `OMX_StateLoaded` state, or its port shall be disabled, and both ports shall be currently connected together.

3.2.3.7.2 Results/Outputs for This Method

If the method returns successfully both ports are no longer connected, either via a tunnel or using proprietary communication. If the method returns unsuccessfully then both ports retain their previous connection status.

3.2.3.7.3 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* tear down a tunnel between two components */
OMX_TeardownTunnel(hCompA, nCompAOutPort, hCompB, nCompBInPort);
```

3.2.3.8 OMX_GetCoreInterface

The `OMX_GetCoreInterface` method returns a new core extension interface by an extension name. This method will allocate the actual extension interface structure, ensures it is populated correctly, and updates the value of *ppItf with a pointer to the newly created interface. The core should return from this call within 20 milliseconds.

Since interfaces are requested by name, a naming convention is defined. OpenMAX IL interface names are zero terminated strings with the following format:

"OMX.<vendor_name>.<vendor_specified_convention>".

Vendor name "Khronos" is reserved for interfaces defined by Khronos.

OMX_GetCoreInterface is defined as follows.

```
OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_GetCoreInterface (
  OMX_OUT void ** ppItf,
  OMX IN OMX_STRING cExtensionName)
```

The parameters are as follows.

Parameter	Description
cExtensionName [in]	An OMX_STRING for the name of the extension interface to be instantiated by the IL core. Extension names are strings limited to a length up to 127 characters plus the trailing null for a maximum length of 128 characters.
ppItf [out]	Pointer to an interface pointer to be filled in by the IL core. The caller has to cast the value to the interface type specific to the named extension. In case this method returns an error, the pointer value is unspecified.

3.2.3.8.1 Prerequisites for this Method

The OpenMAX IL core shall be initialized.

3.2.3.8.2 Results/Outputs for this Method

If successful, the function returns a valid interface pointer to the IL client. The method shall return `OMX_ErrorNotImplemented` if the extension requested by the IL client is not supported.

3.2.3.9 OMX_FreeCoreInterface

The `OMX_FreeCoreInterface` method will free an interface allocated by the `OMX_GetCoreInterface` method. The exact state the interface needs to be in, when this method is called, is extension-dependent and shall be documented in extension documentation. The core should return from this call within 20 milliseconds.

OMX_FreeCoreInterface is defined as follows.

```
OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_FreeCoreInterface (
  OMX_IN void * pItf)
```

The single parameter is defined as follows.

Parameter	Description
pItf [in]	The pointer to the interface to be freed.

3.2.3.9.1 Prerequisites for this Method

The input parameter pItf shall contain a valid pointer to an interface allocated by the OMX_GetCoreInterface method.

3.2.3.9.2 Results/Outputs for this Method

The method has freed the interface.

3.3 OpenMAX IL Component Methods and Structures

OpenMAX IL components are defined in the OMX_Component.h header file. The structure OMX_COMPONENTTYPE holds the data fields and function entry points for a component.

OMX_COMPONENTTYPE is defined as follows.

```
typedef struct OMX_COMPONENTTYPE
{
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_PTR pComponentPrivate;
    OMX_PTR pApplicationPrivate;

    OMX_ERRORTYPE (*GetComponentVersion)(
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_OUT OMX_STRING pComponentName,
            OMX_OUT OMX_VERSIONTYPE* pComponentVersion,
            OMX_OUT OMX_VERSIONTYPE* pSpecVersion,
            OMX_OUT OMX_UUIDTYPE* pComponentUUID);

    OMX_ERRORTYPE (*SendCommand)(
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_COMMANDTYPE Cmd,
            OMX_IN  OMX_U32 nParam1,
            OMX_IN  OMX_PTR pCmdData);

    OMX_ERRORTYPE (*GetParameter)(
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_INDEXTYPE nParamIndex,
            OMX_INOUT OMX_PTR pComponentParameterStructure);
```

```
    OMX_ERRORTYPE (*SetParameter)(
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_INDEXTYPE nIndex,
            OMX_IN  OMX_PTR pComponentParameterStructure);

    OMX_ERRORTYPE (*GetConfig)(
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_INDEXTYPE nIndex,
            OMX_INOUT OMX_PTR pComponentConfigStructure);

    OMX_ERRORTYPE (*SetConfig)(
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_INDEXTYPE nIndex,
            OMX_IN  OMX_PTR pComponentConfigStructure);

    OMX_ERRORTYPE (*GetExtensionIndex)(
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_STRING cParameterName,
            OMX_OUT OMX_INDEXTYPE* pIndexType);

    OMX_ERRORTYPE (*GetState)(
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_OUT OMX_STATETYPE* pState);

    OMX_ERRORTYPE (*ComponentTunnelRequest)(
        OMX_IN  OMX_HANDLETYPE hComp,
        OMX_IN  OMX_U32 nPort,
        OMX_IN  OMX_HANDLETYPE hTunneledComp,
        OMX_IN  OMX_U32 nTunneledPort,
        OMX_INOUT  OMX_TUNNELSETUPTYPE* pTunnelSetup);

    OMX_ERRORTYPE (*UseBuffer)(
            OMX_IN OMX_HANDLETYPE hComponent,
            OMX_INOUT OMX_BUFFERHEADERTYPE** ppBufferHdr,
            OMX_IN OMX_U32 nPortIndex,
            OMX_IN OMX_PTR pAppPrivate,
            OMX_IN OMX_U32 nSizeBytes,
            OMX_IN OMX_U8* pBuffer);

    OMX_ERRORTYPE (*AllocateBuffer)(
            OMX_IN OMX_HANDLETYPE hComponent,
            OMX_INOUT OMX_BUFFERHEADERTYPE** ppBuffer,
            OMX_IN OMX_U32 nPortIndex,
            OMX_IN OMX_PTR pAppPrivate,
            OMX_IN OMX_U32 nSizeBytes);

    OMX_ERRORTYPE (*FreeBuffer)(
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_U32 nPortIndex,
            OMX_IN  OMX_BUFFERHEADERTYPE* pBuffer);

    OMX_ERRORTYPE (*EmptyThisBuffer)(
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_BUFFERHEADERTYPE* pBuffer);

    OMX_ERRORTYPE (*FillThisBuffer)(
```

```
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_BUFFERHEADERTYPE* pBuffer);

    OMX_ERRORTYPE (*SetCallbacks)(
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_CALLBACKTYPE* pCallbacks,
            OMX_IN  OMX_PTR pAppData);

    OMX_ERRORTYPE (*ComponentDeInit)(
            OMX_IN  OMX_HANDLETYPE hComponent);

    OMX_ERRORTYPE (*UseEGLImage)(
            OMX_IN OMX_HANDLETYPE hComponent,
            OMX_INOUT OMX_BUFFERHEADERTYPE** ppBufferHdr,
            OMX_IN OMX_U32 nPortIndex,
            OMX_IN OMX_PTR pAppPrivate,
            OMX_IN void* eglImage);

    OMX_ERRORTYPE (*ComponentRoleEnum)(
        OMX_IN OMX_HANDLETYPE hComponent,
            OMX_OUT OMX_U8 *cRole,
            OMX_IN OMX_U32 nIndex);

} OMX_COMPONENTTYPE;
```

3.3.1 pComponentPrivate

`pComponentPrivate` is a pointer to the component private data area. The component allocates and initializes this member when the component is first loaded. The application should not access this data area.

3.3.2 pApplicationPrivate

`pApplicationPrivate` is a pointer to the application private data area. The component initializes this field during the call to `SetCallbacks`, as this field is provided back to the IL client when the component issues callbacks.

3.3.3 GetComponentVersion

The IL client calls the `GetComponentVersion` component method via the `OMX_GetComponentVersion` core macro. See the definition of `OMX_GetComponentVersion` in section 3.2.2.1 above for a description of its semantics.

3.3.4 SendCommand

The IL client calls the `SendCommand` component method via the `OMX_SendCommand` core macro. See the definition of `OMX_SendCommand` in section 3.2.2.2 above for a description of its semantics.

3.3.5 GetParameter

The IL client or a tunneled component calls the `GetParameter` component method via the `OMX_GetParameter` core macro. See the definition of `OMX_GetParameter` in section 3.2.2.8 above for a description of its semantics.

3.3.6 SetParameter

The IL client or a tunneled component calls the `SetParameter` component method via the `OMX_SetParameter` core macro. See the definition of `OMX_SetParameter` in section 3.2.2.8.3 above for a description of its semantics.

3.3.7 GetConfig

The IL client calls the `GetConfig` component method via the `OMX_GetConfig` core macro. See the definition of `OMX_GetConfig` in section 3.2.2.9.3 above for a description of its semantics.

3.3.8 SetConfig

The IL client calls the `SetConfig` component method via the `OMX_SetConfig` core macro. See the definition of `OMX_SetConfig` in section 3.2.2.10.3 above for a description of its semantics.

3.3.9 GetExtensionIndex

The IL client calls the `GetExtensionIndex` component method via the `OMX_GetExtensionIndex` core macro. See the definition of `OMX_GetExtensionIndex` in section 3.2.2.12 for a description of its semantics.

3.3.10 GetState

The IL client calls the `GetState` component method via the `OMX_GetState` core macro. See the definition of `OMX_GetState` in section 3.2.2.13 above for a description of its semantics.

3.3.11 ComponentTunnelRequest

The `ComponentTunnelRequest` method will interact with another OpenMAX IL component to determine if tunneling is possible and to set up the tunneling if it is possible. The return codes for this method can determine if tunneling is not possible or if proprietary communication or tunneling is used. The `ComponentTunnelRequest` method will also be used to disconnect a port that was previously connected via tunneling or proprietary communication.

The interop profile-conformant component shall support tunneling to a component with compatible parameters (refer to 3.2.3.6 for information regarding compatibility

checking) . The component may also support proprietary communication. If proprietary communication is supported, the negotiation of proprietary communication is performed in a vendor-specific way. The only requirement is that the proper result be returned. The details of the proprietary communication setup are left to the vendor's component implementer.

The `ComponentTunnelRequest` method is invoked on both components that support the tunneling communication. When this method is invoked on the component that provides the output port, the component will do the following:

1. Indicate its supplier preference in `pTunnelSetup`.

When this method is invoked on the component that provides the input port, the component will do the following:

1. Check the data compatibility between the ports using one or more `GetParameter` calls.
2. Review the buffer supplier preferences of the output port and use `OMX_SetParameter` with index `OMX_IndexParamCompBufferSupplier` to inform the output port of which port supplies the buffers.

If this method is invoked with a NULL value for the `pTunnelComp` parameter, the port should be set up for non-tunneled communication with the IL client, and is no longer connected to another port.

If this method is invoked with a non-NULL value for the `pTunnelComp` parameter, and this port is already connected then this call shall fail.

The component should return from this call within five milliseconds.

The parameters for this method are defined as follows.

Parameter	**Description**
hComp [in]	The handle of the target component of the `RequestTunnel` call and one of the components that will participate in the tunnel.
nPort [in]	The index of the port belonging to `hComp` that will participate in the tunnel.
hTunneledComp [in]	The handle of the other component that participates in the tunnel. When this parameter is NULL, the port specified in `nPort` should be configured for non-tunneled communication with the IL client and not connected to another port.
nTunneledPort [in]	The index of the port belonging to `hTunneledComp` that participates in the tunnel.
pTunnelSetup [in,out]	The structure that contains data for the tunneling negotiation between components. The supplier field can be filled by both components. The read-only flag can be applied by both components.

3.3.11.1 Prerequisites for This Method

The component shall be in the `OMX_StateLoaded` state or its port shall be disabled.

3.3.11.2 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* Translate a SetupTunnel call to two ComponentTunnelRequest calls */
pCompOut = (OMX_COMPONENTTYPE *)hOutput;
pCompIn = (OMX_COMPONENTTYPE *)hInput;
pCompOut->ComponentTunnelRequest(hOutput, nPortOutput, hInput,
      nPortInput, &oTunnelSetup);
pCompIn->ComponentTunnelRequest(hInput, nPortInput, hOutput,
      nPortOutput, &oTunnelSetup);
```

3.3.12 UseBuffer

The IL client or a tunneled component calls the `UseBuffer` component method via the `OMX_UseBuffer` core macro. See the definition of `OMX_UseBuffer` in section 3.2.2.14 above for a description of its semantics.

3.3.13 AllocateBuffer

The IL client calls the `AllocateBuffer` component method via the `OMX_AllocateBuffer` core macro. See the definition of `OMX_AllocateBuffer` in section 3.2.2.15 above for a description of its semantics.

3.3.14 FreeBuffer

The IL client or a tunneled component calls the `FreeBuffer` component method via the `OMX_FreeBuffer` core macro. See the definition of `OMX_FreeBuffer` in section 3.2.2.16 above for a description of its semantics.

3.3.15 EmptyThisBuffer

The IL client or a tunneled component calls the `EmptyThisBuffer` component method via the `OMX_EmptyThisBuffer` core macro. See the definition of `OMX_EmptyThisBuffer` in section 3.2.2.17 above for a description of its semantics.

3.3.16 FillThisBuffer

The IL client or a tunneled component calls the `FillThisBuffer` component method via the `OMX_FillThisBuffer` core macro. See the definition of `OMX_FillThisBuffer` in section 3.2.2.18 above for a description of its semantics.

3.3.17 SetCallbacks

The IL client calls the `SetCallbacks` component method via the `OMX_SetCallbacks` core macro. The `SetCallbacks` method will allow the core to transfer the callback structure from the IL client to the component. This is a blocking call. See the definition of `OMX_SetCallbacks` in section 3.2.2.20 for a description of its semantics.

3.3.18 ComponentDeinit

The core calls the `ComponentDeinit` function when the core needs to dispose of a component.

The single parameter for this method is as follows.

Parameter	Description
hComponent [in]	The handle of the component that executes the call.

3.3.18.1 Prerequisites for This Method

The IL client shall call this method the `OMX_FreeHandle` macro only when the component is in `OMX_StateLoaded` state.

3.3.18.2 Sample Code Showing Calling Sequence

The following sample code shows the calling sequence.

```
/* On FreeHandle: de-initialize component and destroy it */
pComp = (OMX_COMPONENTTYPE*)hComponent;
(pComp->ComponentDeinit)(hComponent);
```

3.3.19 UseEGLImage

The IL client or a tunneled component calls the `UseEGLImage` component method via the `OMX_UseEGLImage` core macro. See the definition of `OMX_UseEGLImage` in section 3.2.2.19 above for a description of its semantics.

3.4 Calling Sequences

This section describes how the IL client, the OpenMAX IL core, and the components dynamically interact in a few meaningful use cases, namely initialization, de-initialization, data flow, data tunneling setup, and data flow in the case of data tunneling and dynamic port reconfiguration. The interaction between the core, the components, and the possible implementation of a resource manager is also described.

3.4.1 Initialization

This section describes the operations for initializing the OpenMAX IL components. The components can be handled directly by the IL client, can be tunneled to each other, or both. The tunneled and non-tunneled cases are distinguished for clarity, but the two cases can be both present in the component framework.

3.4.1.1 Non-tunneled Initialization

Figure 3-9 shows how an IL client should initialize an OpenMAX IL component.

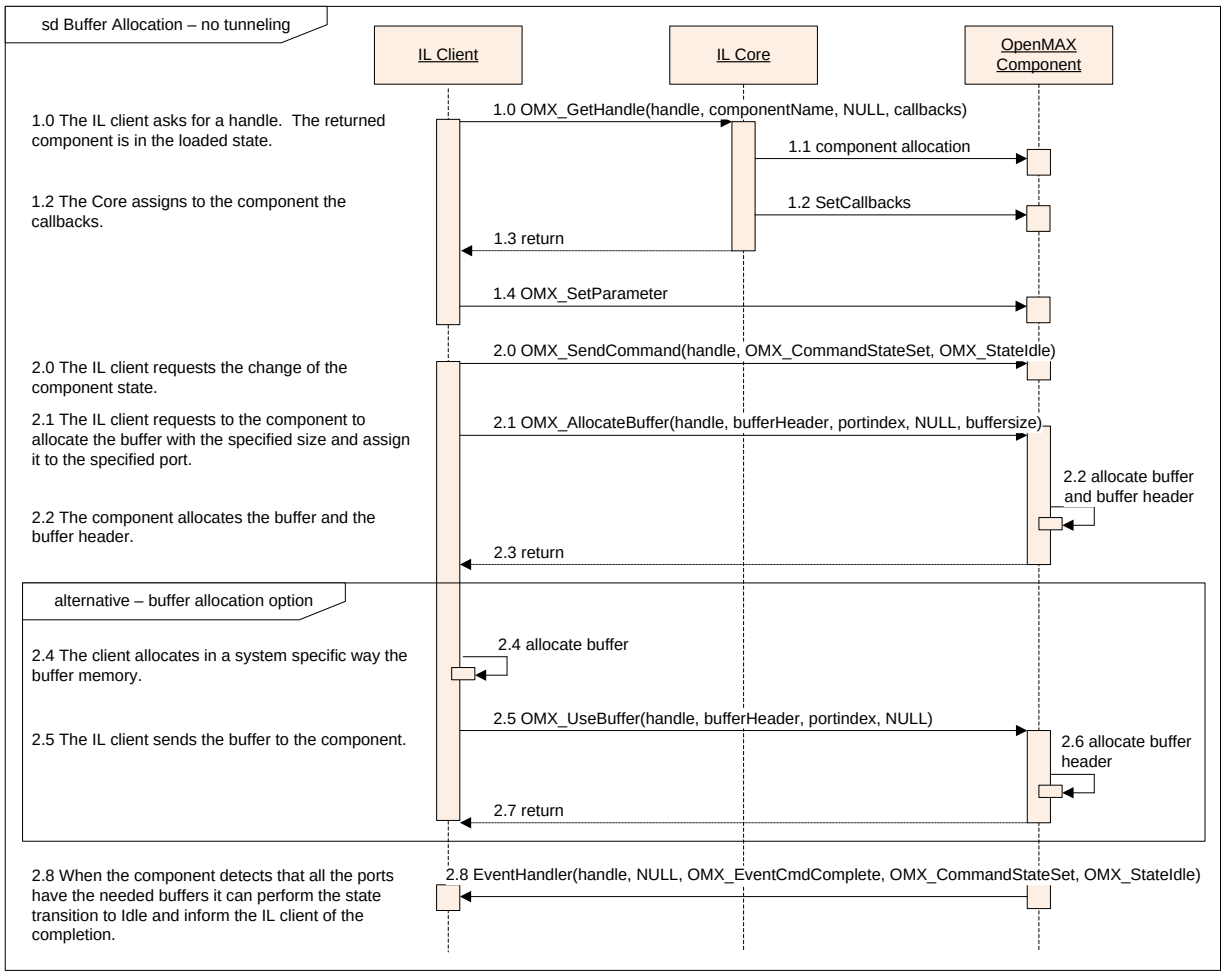


Figure 3-9. Component Initialization

First, the IL client shall call the `OMX_GetHandle` function, which activates the actual component creation (1.1) by the core. Also, all of the configuration resources of the component are loaded into memory. The core passes IL client callback functions to the component by means of the `SetCallbacks` method (1.2). If previous steps are successful, a valid handle is returned in step 1.3 and the component will be in the `OMX_StateLoaded` state.

The IL client shall configure the component and its ports. For this purpose, the IL core macro `OMX_SetParameter` shall be used; it may be called multiple times (step 1.4) if needed.

When the client has completed the configuration phase, it can request the component to make the state transition to `OMX_StateIdle`. Only after this request shall the IL client set up buffers for the component to use for all of its ports. The IL client shall use either `OMX_AllocateBuffer` or `OMX_UseBuffer` to set up buffers. If the IL client asks components for a tunnel, it does not allocate buffers because the tunneled components allocate any buffers. See section 3.4.1.2 for more details on tunneling.

This process may be repeated multiple times, depending on the number of ports and the total number of buffers needed on each port. If `OMX_UseBuffer` is used, the IL client may pre-announce an allocated buffer to the component, or may defer the alloation and announce a NULL buffer pointer. Alternatively, the IL client may ask the component to allocate a buffer and a buffer header using the `OMX_AllocateBuffer` method. In the latter case, the component will allocate both a buffer and its related header and return it to the IL client by reference.

As soon as these initial configuration steps are completed, the component shall complete the state transition and return an event to the client for the `SendCommand` request completion (step 2.8).

The component is now ready to be used by the IL client.

3.4.1.2 Tunneled Initialization

To avoid moving data buffers back and forth among the IL client and OpenMAX IL components, data tunnels can be set up so that the output buffer of one component is passed directly to the input port of the next component in the chain.

Consider the example shown in Figure 3-10, where an IL client generates data for a chain of three tunneled components identified as A, B, and C. Component C is a sink and does not return data to the IL client.

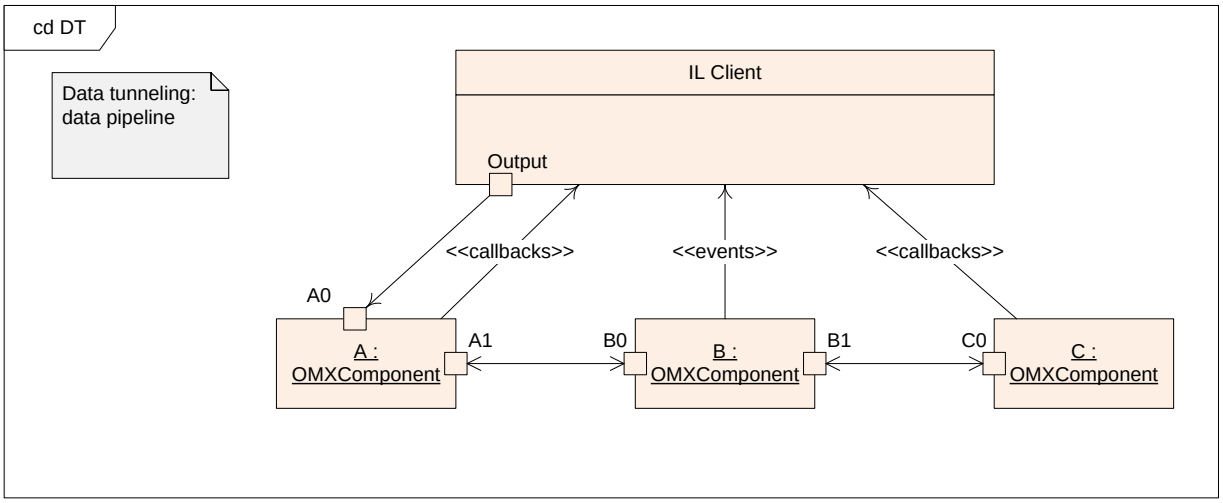


Figure 3-10. Example of Data Tunneling Among OpenMAX IL Components

Note that all callbacks are always directed to and managed by the IL client when ports communicate using proprietary or tunneled communication. The tunneling setup and initialization require a detailed description, based on the following steps:

- The components are constructed with the calls to `OMX_GetHandle`.
- The components are tunneled, linking an output port of the first component to an input port of the second component. The port that shall supply the buffer is decided in this phase.
- The IL client may override the ports' choice of buffer supplier after `OMX_SetupTunnel` has completed by setting the buffer supplier into the input port, which in turn will reprogram the supplier to the output port.

During the transition from `OMX_StateLoaded` to `OMX_StateIdle`, each component shall not transition until the required buffer headers on all enabled ports have been allocated.

`OMX_SetupTunnel` shall be executed only when the components are in the `OMX_StateLoaded` state or when ports are disabled, and when those ports are not connected to any others. Figure 3-11 illustrates the setup process:

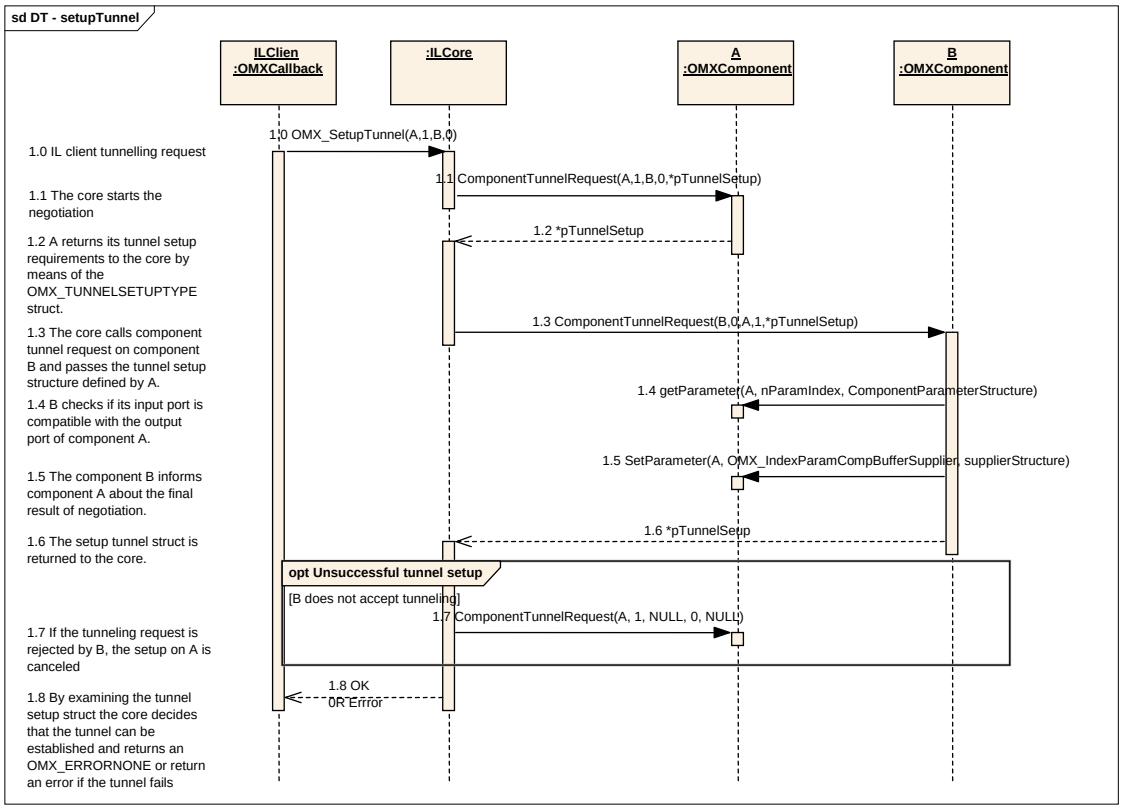


Figure 3-11. Tunnel Setup

The IL client shall start the data setup process by calling the `OMX_SetupTunnel` function of the IL core when the components that are being tunneled are in the `OMX_StateLoaded` state (step 1.0).

As a result, the IL core shall call the `ComponentTunnelRequest` methods of component A and B in sequence. The structure `OMX_TUNNELSETUPTYPE` defined in section 3.1.3.10 shall be passed by the IL core to the component with the output port first. The component receiving such a call shall fill in the structure and return it to the core. If the `ComponentTunnelRequest` call returns successfully, the IL core shall call the same function on the second component (1.3), passing the `OMX_TUNNELSETUPTYPE` structure that was filled in by the first component. The component also shall check that the output port of the peer component is compatible with its input port (i.e., the data type should be the same) (1.4). If the tunnel setup parameters included in the structure are agreed to by the second component, the `ComponentTunnelRequest` call will send back to the first component the result of negotiation (1.5) and returns successfully (1.6). The IL core shall check that both calls of `ComponentTunnelRequest` did not return errors. If so, the initial `OMX_SetupTunnel` will return successfully.

If the call to `ComponentTunnelRequest` on component B fails, component A will be set to not tunnel by a second call to `ComponentTunnelRequest` with a pointer to NULL in place of the component B handle and `pTunnelSetup` parameter.

After the successful tunnel setup, the IL client may override the buffer supplier negotiation with the procedure illustrated in Figure 3-12:

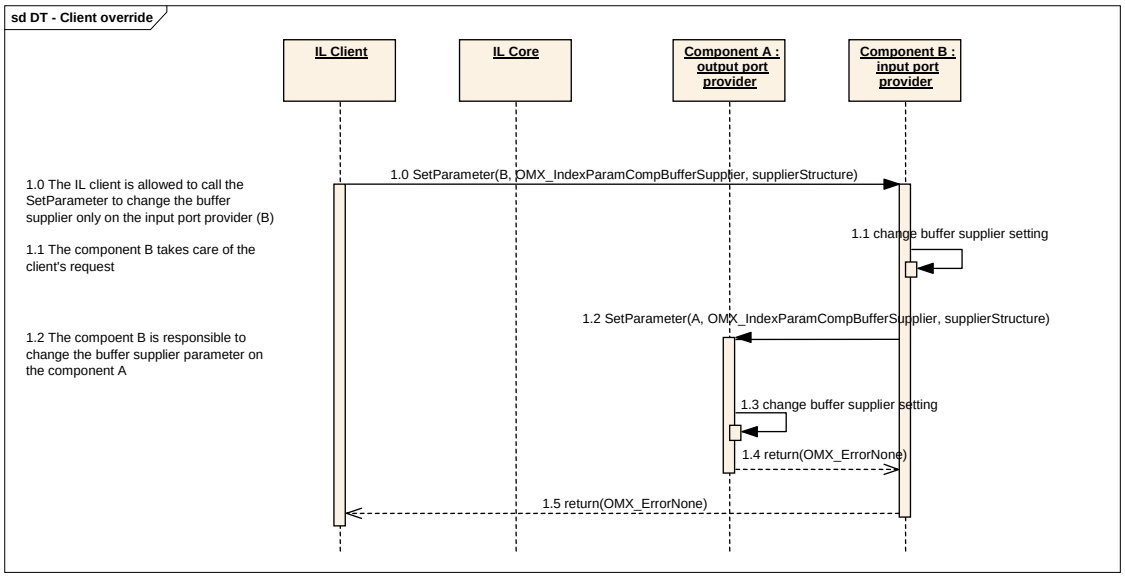


Figure 3-12. IL Client Buffer Supplier Override

If the IL client wants to override the negotiation of tunneled components that specifies which component is the buffer supplier, it shall call the function `SetParameter` on the component that provides the input port. That component is responsible for signaling to the other tunneled component the new buffer supplier, with the same call to `SetParameter`.

Note that if the IL client calls `SetParameter` before the tunnel setup, an IL component shall return `OMX_ErrorIncorrectStateOperation` as changing the buffer supplier preference before a tunnel is actually setup is not supported.

The last step of the tunnel initialization phase is the state transition from `OMX_StateLoaded` to `OMX_StateIdle` that also involves the buffer allocation and assignment. Figure 3-13 illustrates the state transition behavior in which the tunnels are already created and configured.

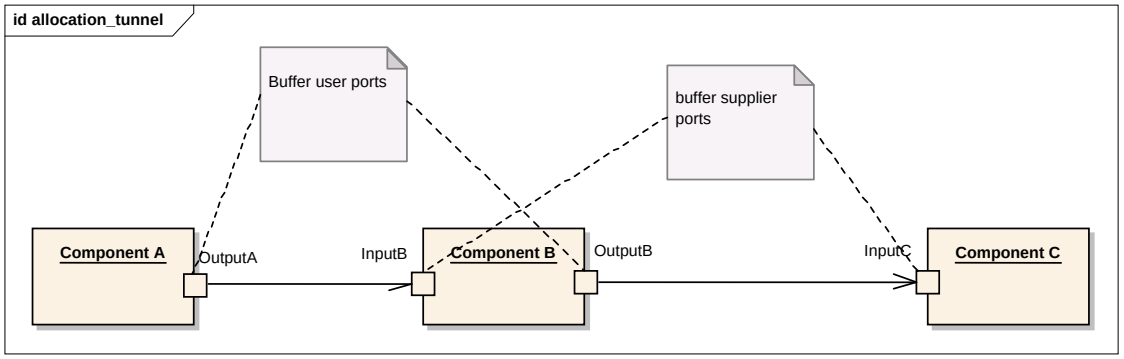


Figure 3-13. Tunneling Example

Component A is tunneled with component B, and component B is the buffer supplier. Component B is tunneled with component C, and component C is the buffer supplier.

Figure 3-14 illustrates the behavior of each tunneled component during the state transition.

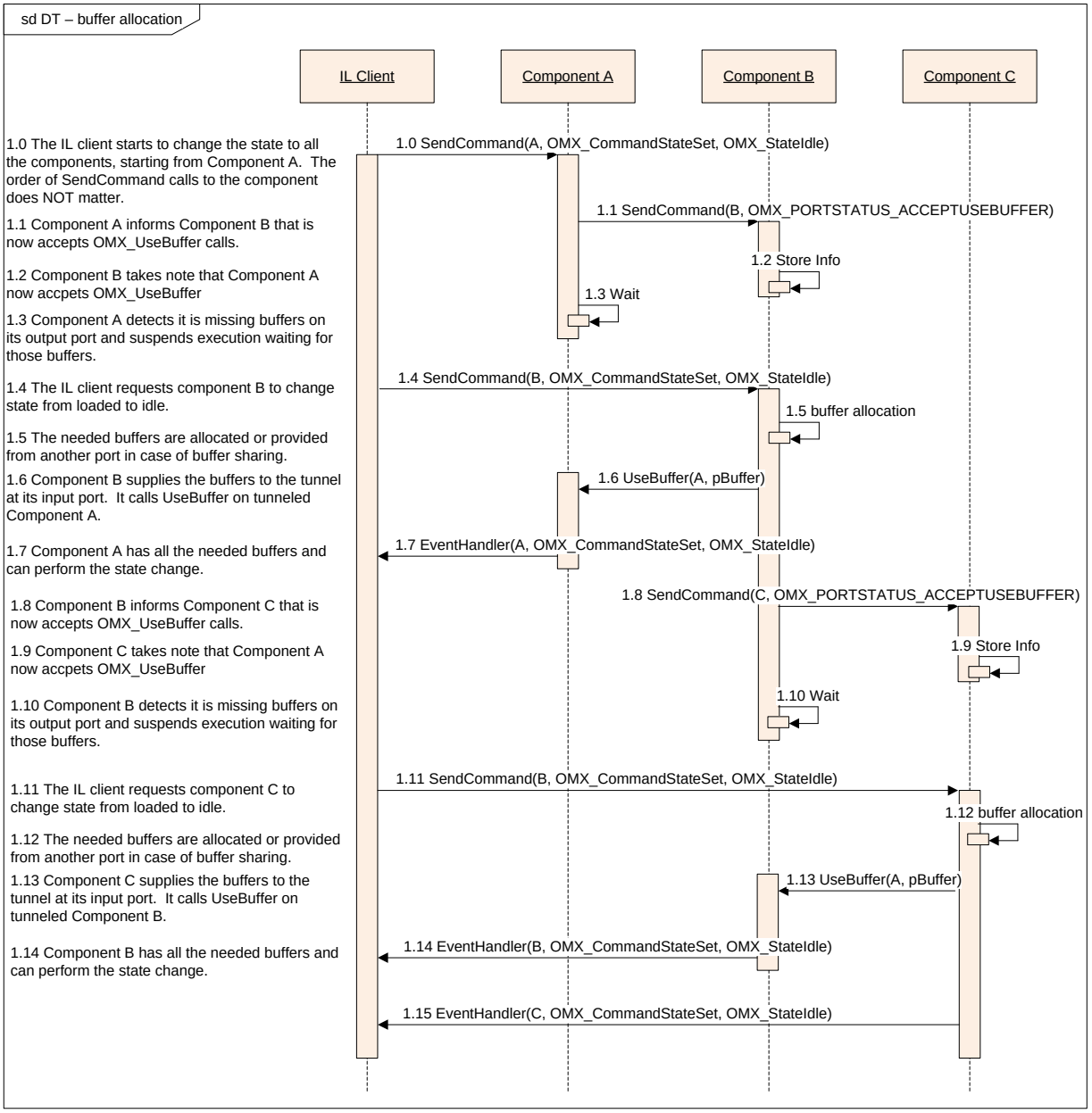


Figure 3-14. State Transition to Idle in the Case of Tunneled Components

Each supplier port on a component shall pass its buffers or NULL pointers to the non-supplier port it is tunneling with via `OMX_UseBuffer`. After all of its supplier ports have passed buffers, the component waits until all of its non-supplier ports have received all of their buffers via `OMX_UseBuffer`.

In Figure 3-14, component A receives the state transition request from the IL client. Component A is tunneled with component B. The input port of B is set as buffer supplier

for the tunnel. In this case, component A shall wait until its output port receives all of the needed `OMX_UseBuffer` calls.

Meanwhile, the IL client asks component B to change its state. In this case, component B has a port that is a buffer supplier, the input port, and it shall call `OMX_UseBuffer` on the output port of component A to allocate the buffer headers necessary for data transfer. Component B may allocate the buffers at this point and can pre-announce them, or may defer buffer allocation and will use NULL buffer pointers. Then, component B waits for all of the `OMX_UseBuffer` calls on its output port.

Now component A has allocated all of the needed buffer headers, so it can perform the state transition to `OMX_StateIdle`. The exact sequence of transitions can be different, since it depends on the platform, the operating system, and the implementation. The only rule is to wait until all the resources are available.

The IL client requests that component C change its state. Component C behaves like component B: Component C makes the necessary `OMX_UseBuffer` calls on the output port of component B, and then can change its state, since it does not need any other buffers.

Finally, component B can change its state to `OMX_StateIdle` since it has allocated all of the needed buffer headers.

3.4.2 Data Flow

OpenMAX IL defines two means of data communication:

- Tunneled communication, where a port exchanges data directly with a port on another component
- Non-tunneled communication, where a port exchanges data only with the IL client

A port may implement data tunneling via proprietary communication, taking advantage of platform-specific features. The following sections describe the data flow inherent to each means of communication.

3.4.2.1 Non-tunneled Data Flow

An IL client that has a data buffer to deliver to a component input port shall issue an `OMX_EmptyThisBuffer` call.

Conversely, for the component output port, the IL client shall initially provide one or more empty buffers into which the component can write output data; the `OMX_FillThisBuffer` call accomplishes this task. As soon as one buffer is available from the component output port, the component shall send a `FillBufferDone` callback. The component is aware of the callback entry point from the earlier `SetBacks` call.

Note that the IL client is entirely responsible for moving data buffers among components if data tunneling is not used.

Figure 3-15 illustrates the dynamic behavior related to data flow.

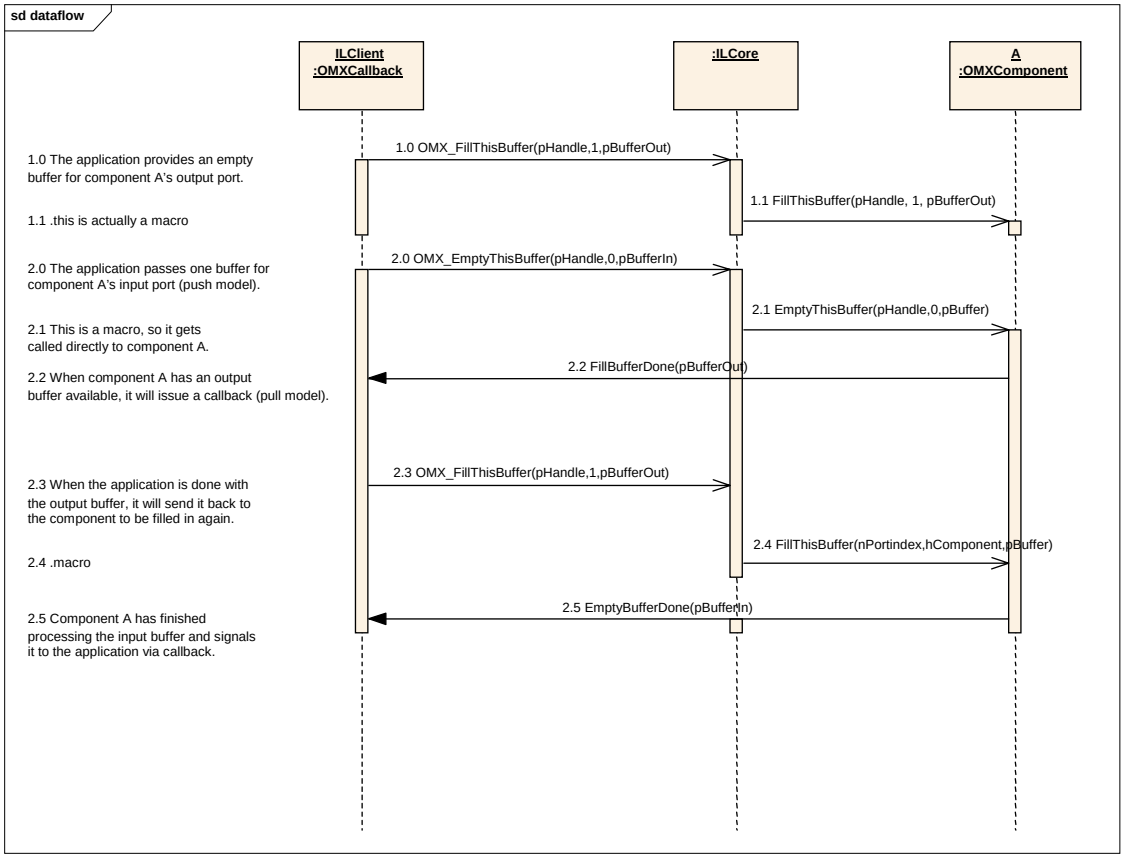


Figure 3-15. Data Flow Between Non-tunneled Components

3.4.2.2 Tunneled Data Flow

In data tunneling, OpenMAX IL components directly pass data buffers among themselves without returning them to the IL client. This data flow uses a different convention from the situation where all data buffers are exchanged with the IL client.

If the buffer supplier is the output component, it shall call `OMX_EmptyThisBuffer` on the other tunneled component to pass the buffer that is to be emptied. When the input component has terminated the operation, it shall return the buffer to the output component by calling `OMX_FillThisBuffer` on it.

If the buffer supplier is the input component, the communication mechanism is the same but is initiated by calling `OMX_FillThisBuffer` on the output component. Figure 3-16 illustrates this process.

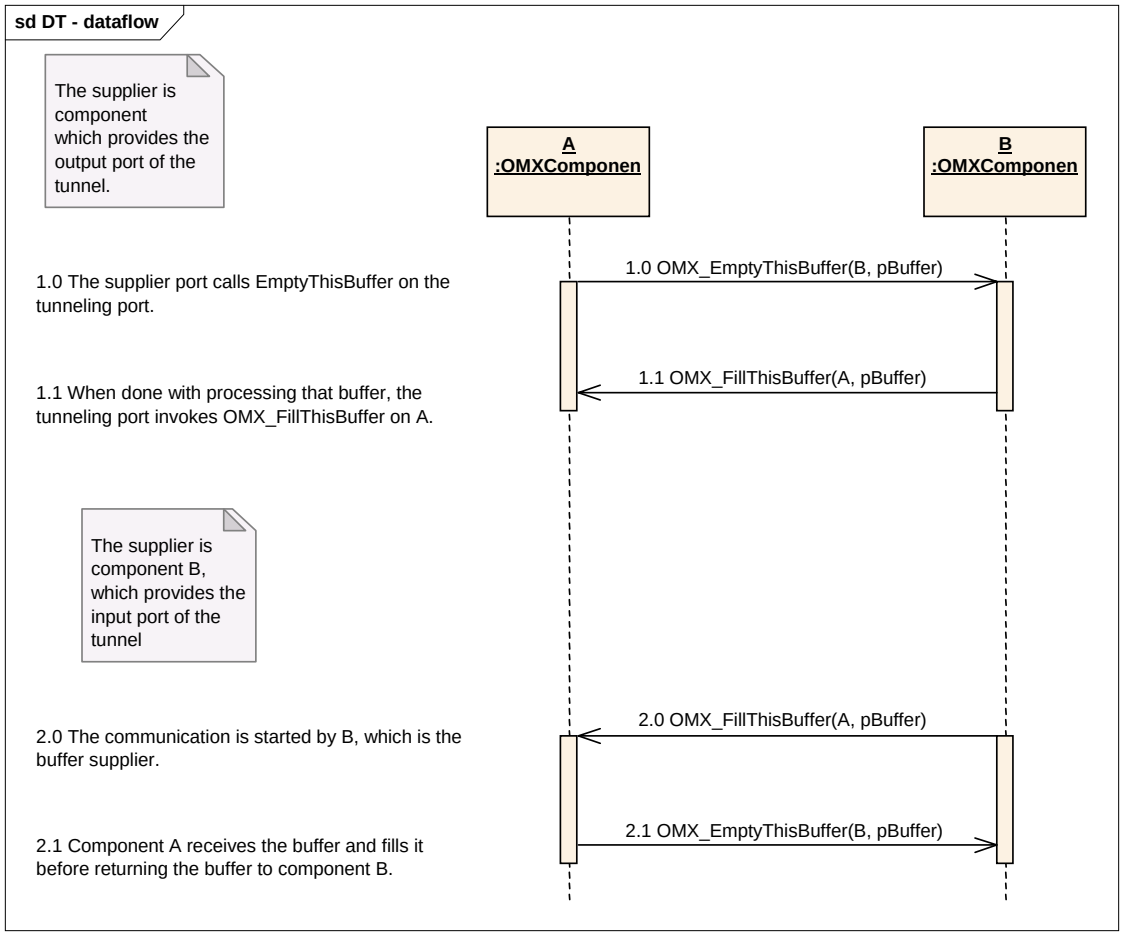


Figure 3-16. Data Flow Between Tunneled Components

3.4.2.3 Proprietary Communication

On some platforms data tunneling among components can be optimized by proprietary communication mechanisms, which can be based on specific hardware such as DMA or shared memory. Such resources are set up in a proprietary manner during the standard data tunneling setup phase. Although the IL client uses the standard `OMX_SetupTunnel` call, platform-specific optimizations can prepare optimized transport channels among components.

Assuming a chain of components A, B, and C that support proprietary communication, the resulting data flow would appear as illustrated in Figure 3-17.

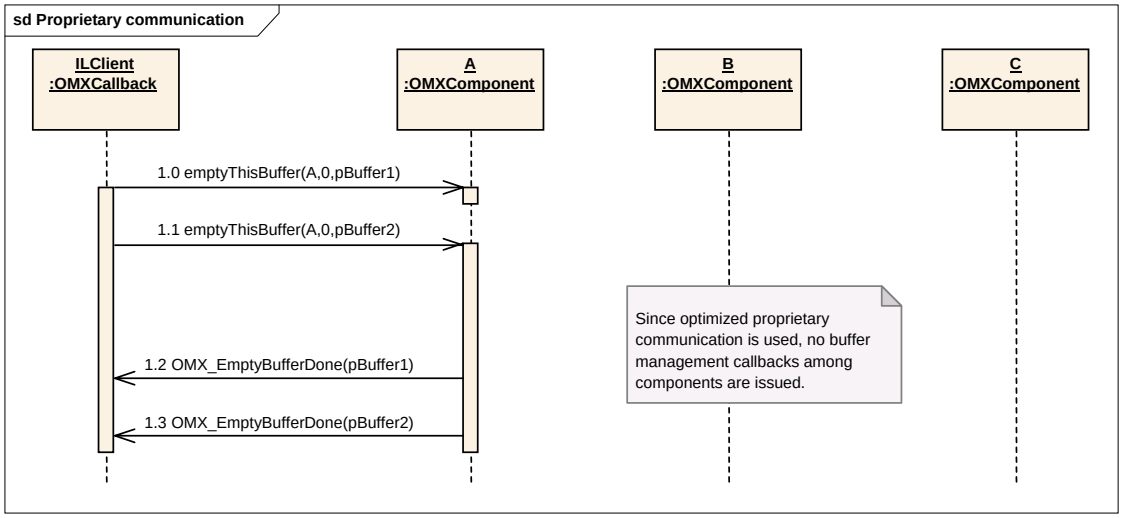


Figure 3-17. Data Flow with Proprietary Communication Between Components

Assuming that all components are in the `OMX_StateExecuting` state, the IL client sends two buffers to component A using the `OMX_EmptyThisBuffer` call (steps 1.0 and 1.1). Given the data tunnel setup, the output of component A is sent to the input port of component B. The output of component B is sent to the input port of component C, which is the sink.

No callbacks will be invoked since the components will use their proprietary mechanisms to move data.

The `EmptyBufferDone` callback will be issued to the IL client only when component A has finished processing buffers.

Even though buffer-related callbacks are not used in this use case, note that components may still generate events to the IL client using the `EventHandler` callback entry point.

3.4.3 De-Initialization

This section describes tunneled and non-tunneled component de-initialization.

3.4.3.1 Non-tunneled De-initialization

When the IL client decides to stop the execution and dispose of the components, it should first switch the components to the `OMX_StateIdle` state so that all buffers are returned to their suppliers.

When the transition to `OMX_StateIdle` is completed, the IL client can request the component to change its state to `OMX_StateLoaded`. The IL client shall free all of the component's buffers by calling `OMX_FreeBuffer` for each buffer. The `OMX_FreeBuffer` function requires that the component remove the specified buffer from the specified port. If the component allocated the buffer with an `OMX_AllocateBuffer` call, the component shall also free the buffer memory. If the IL client allocated the buffer and assigned it to the component with an

`OMX_UseBuffer` call, then the IL client shall de-allocate the buffer memory after calling `OMX_FreeBuffer`. This procedure is performed for each non-tunneled port.

When all of the buffers have been freed, the component shall complete the state transition. Finally, the IL client calls the `OMX_FreeHandle` function that disposes of the component.

This procedure is performed for each non-tunneled port.

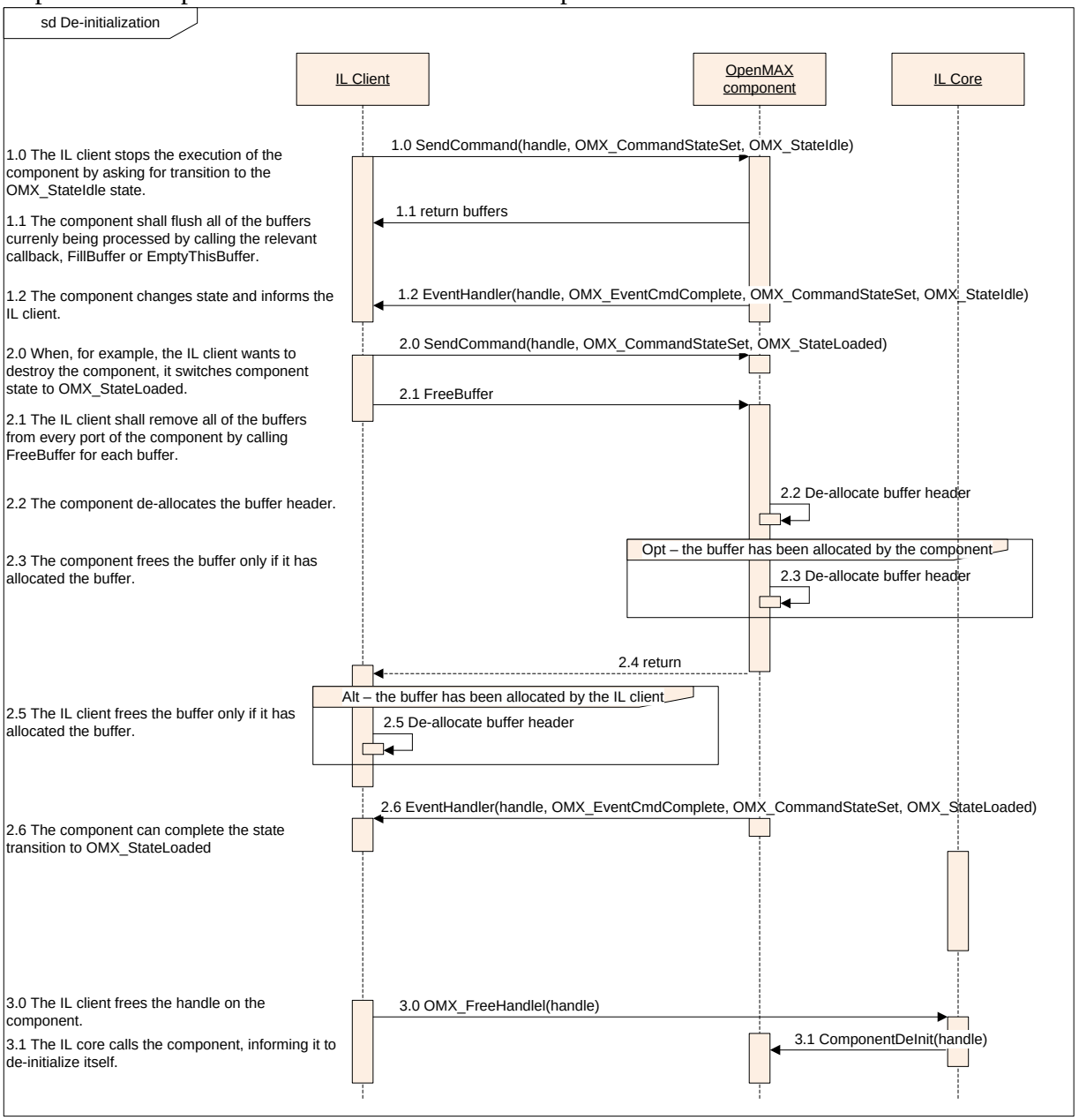


Figure 3-18. illustrates non-tunneled de-initialization

.

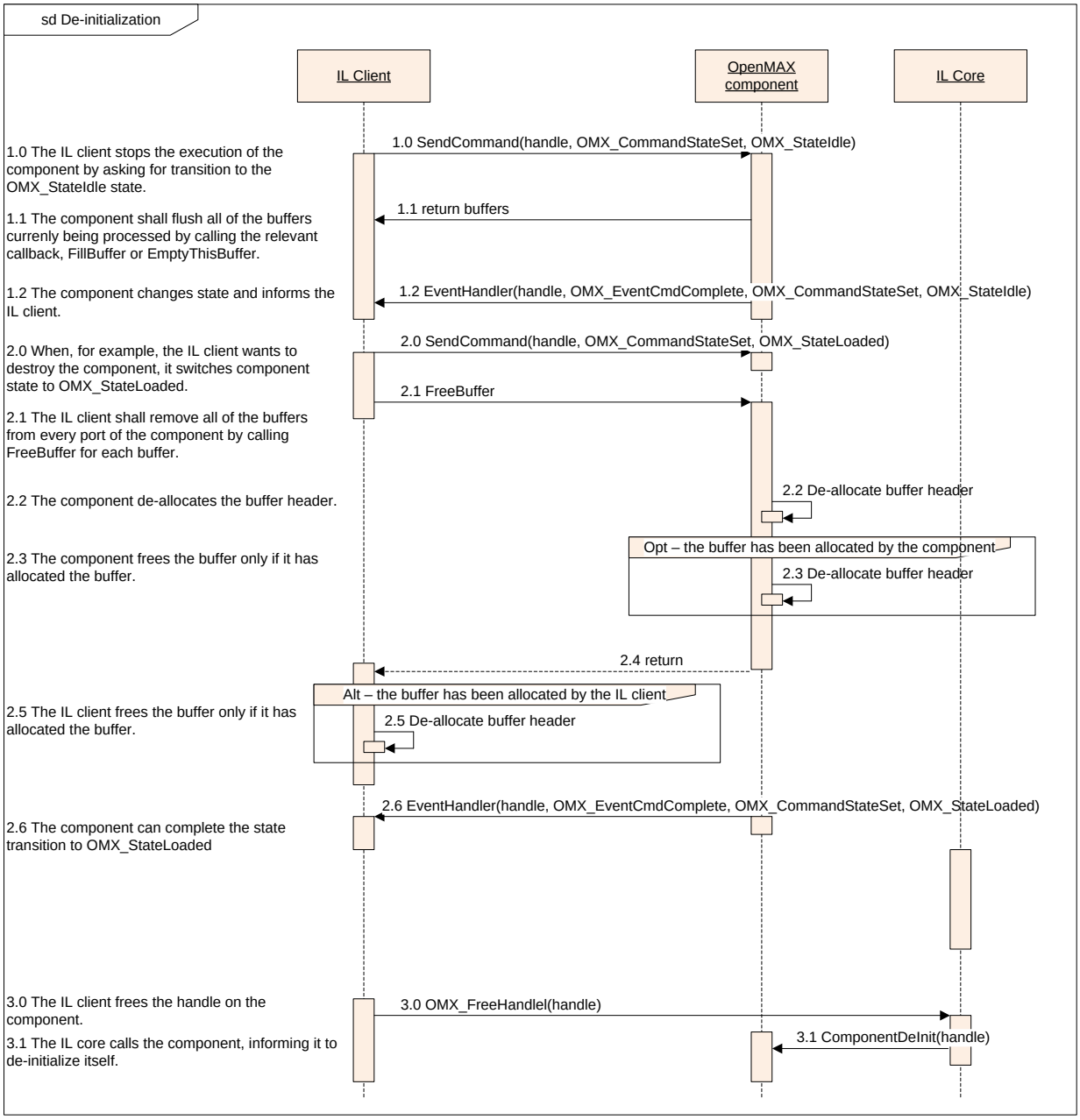


Figure 3-19. De-initialization of Non-tunneled Components

A port that is tunneled shall follow the component de-initialization procedure illustrated in Section 3.4.3.2.

3.4.3.2 Tunneled De-Initialization

When the IL client decides to stop the execution and dispose of two components that are tunneled, it may first either disable all ports or transition the component to `OMX_StateLoaded` (via `OMX_StateIdle` if necessary). The following example illustrates the second method, where it first switches the components to the `OMX_StateIdle` state so that all buffers are returned to their suppliers. After all

buffers are returned to supplier ports, the components shall complete the transition to `OMX_StateIdle`.

The IL client should then start the state transition to `OMX_StateLoaded`. When the supplier component frees all the buffers, the components complete the state transition to `OMX_StateLoaded`.

After this point the tunnel shall be disconnected using `OMX_TeardownTunnel`, inside this function the IL core will call `ComponentTunnelRequest` on each component port to disconnect the tunnel. After this has been completed the IL client can call `OMX_FreeHandle` to dispose of the components.

Figure 3-19 illustrates the component de-initialization for a port that is tunneled – the expectation is that the two components have a single port each, and these ports are tunneled together.

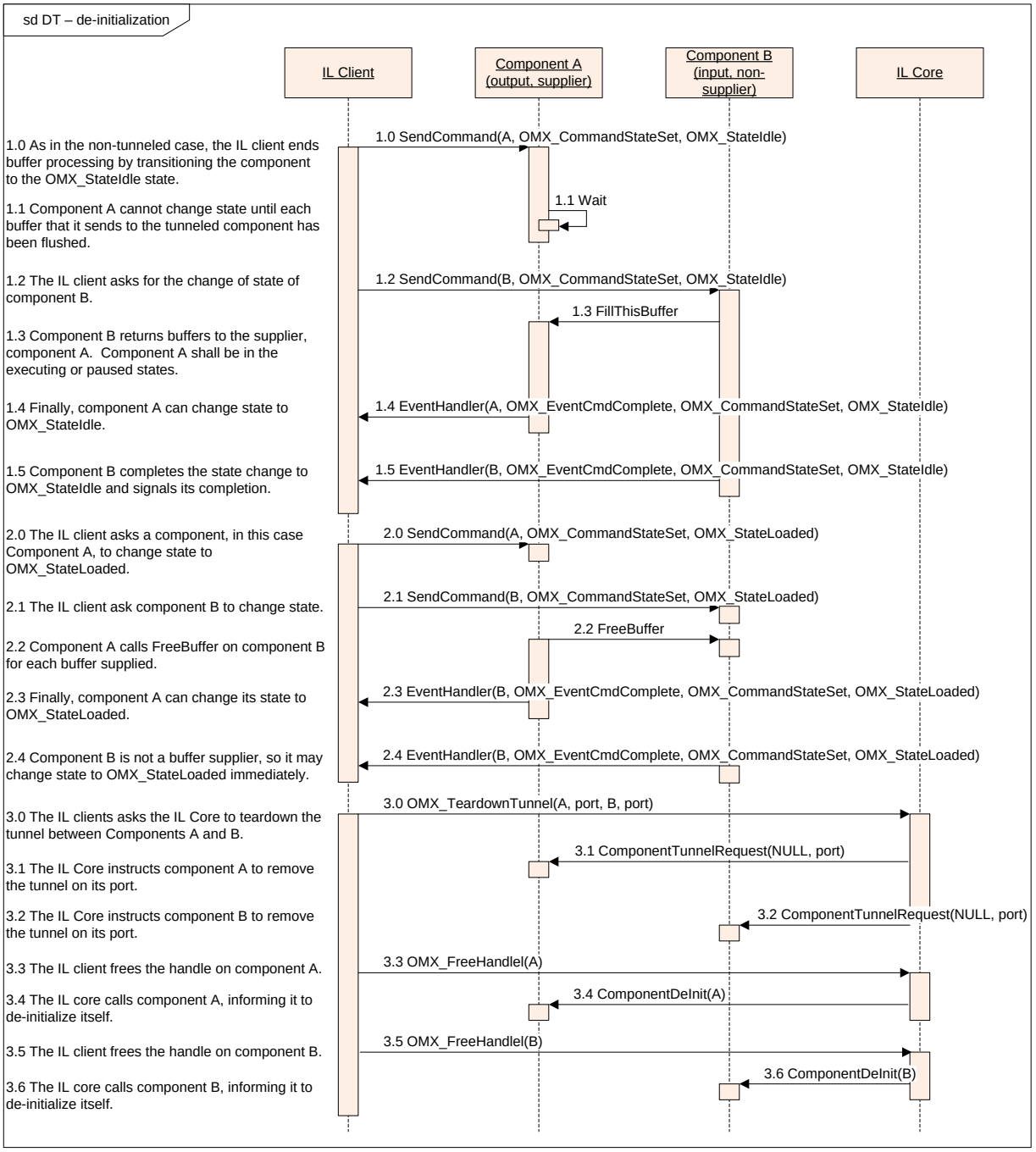


Figure 3-20: De-initialization of Tunneled Components

3.4.4 Port Disablement and Enablement

Disabling a port causes it to behave as if its component transitioned to the `OMX_StateLoaded` state. Thus, all of the port's buffers are returned to their suppliers, and any buffers the disabled port allocated are freed. The act of enabling a port inverts this process, putting a port that is effectively in the `OMX_StateLoaded` state into the component's state. Thus, if the component is in a state where its ports have buffers, then

an enabled port will acquire buffers. Likewise, if the component is exchanging buffers, an enabled port will begin exchanging buffers.

Note that if a port is disabled when the component is in the `OMX_StateLoaded` state, the port's effective state is still made disjoint from the component's state. Thus, when a component transitions from `OMX_StateLoaded` to `OMX_StateIdle`, any disabled port will not acquire buffers but, instead, will effectively remain in `OMX_StateLoaded`.

The description of port disablement and enablement is divided into tunneling and non-tunneling cases.

3.4.4.1 Tunneled Ports Disablement and Enablement

Figure 3-20 illustrates the behavior of enabling and disabling tunneled ports.

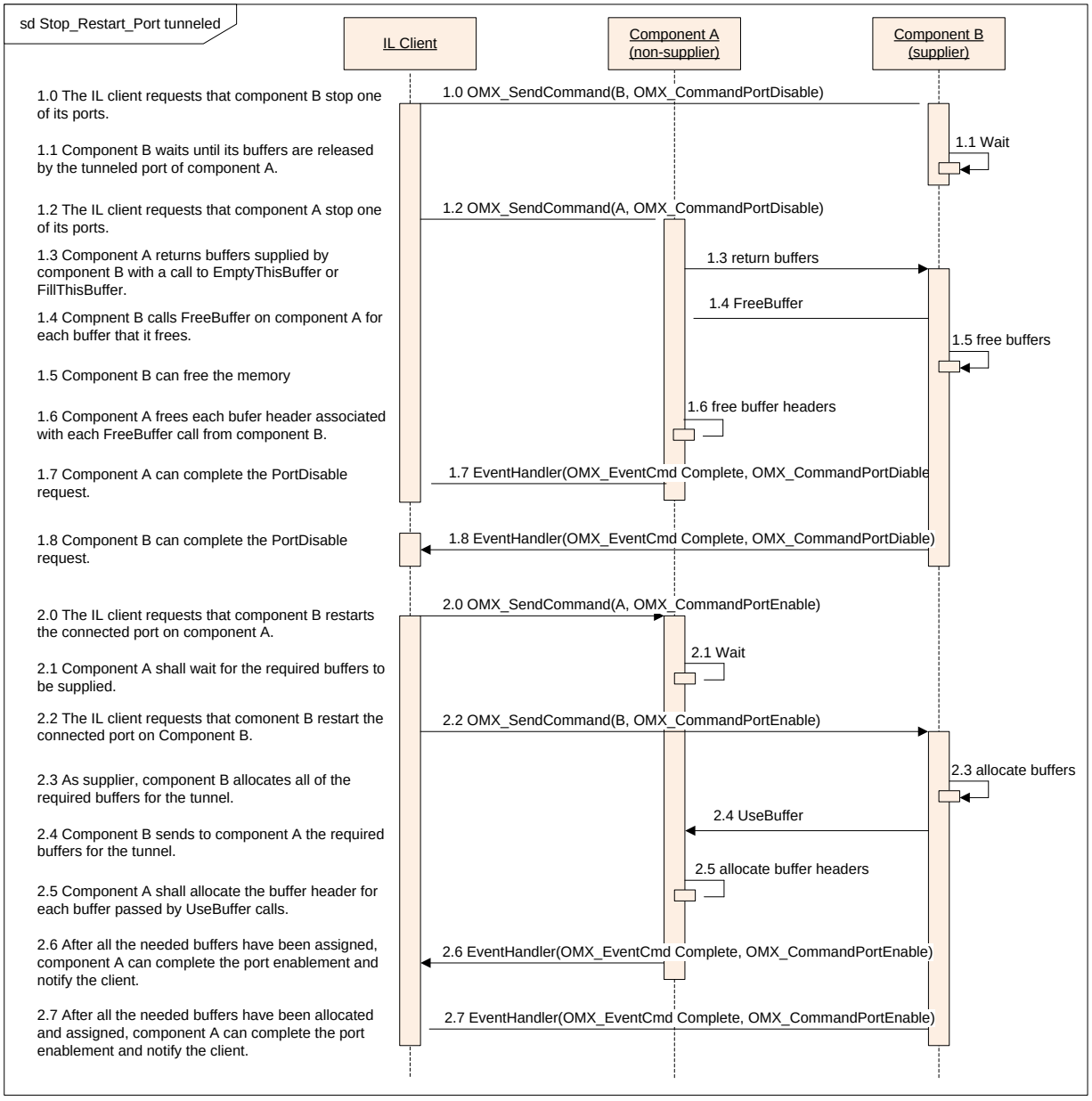


Figure 3-21. Disablement and Enablement of Tunneled Ports

3.4.4.2 Non-tunneled Port Disablement and Enablement

Figure 3-21 illustrates the case of the disablement and enablement procedure for a non-tunneled port. A detailed discussion of `OMX_AllocateBuffer`, `OMX_UseBuffer`, and `OMX_FreeBuffer` is omitted here; for more detailed descriptions of the use of these functions, see sections 3.3.13, 3.3.12, and 3.3.14, respectively.

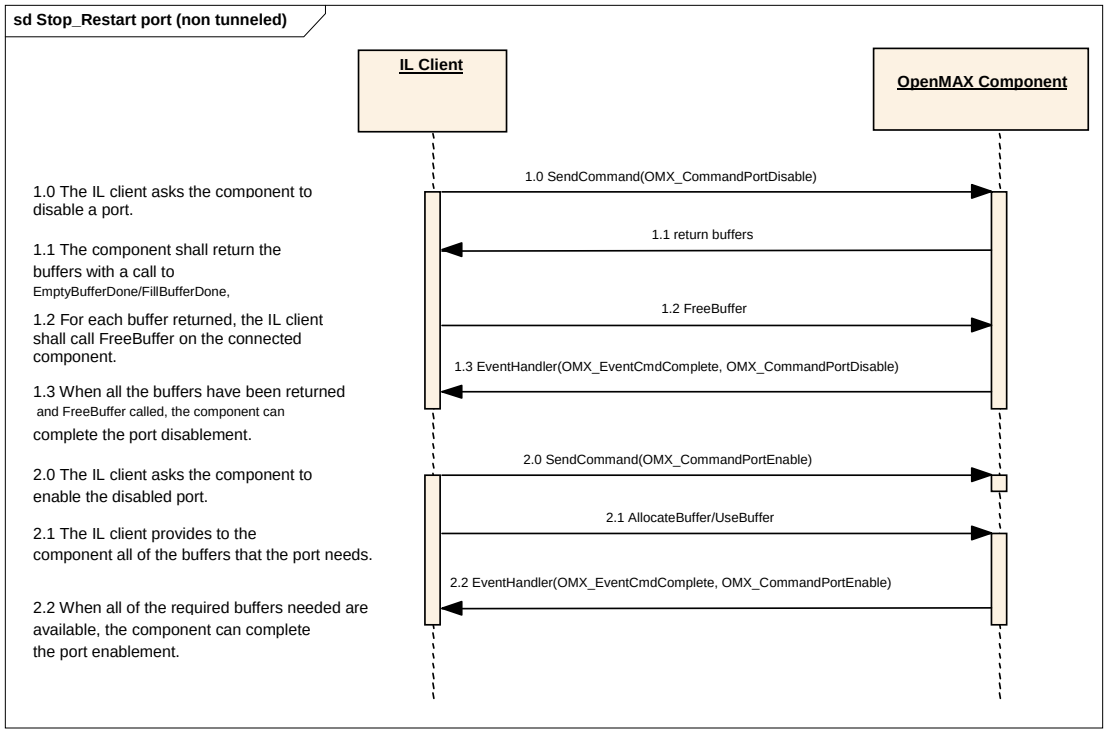


Figure 3-22. Disablement and Enablement of Non-tunneled Ports

3.4.5 Dynamic Port Reconfiguration

This section describes how a component may change its port settings dynamically.

The following examples show where this functionality is typically needed:

- A video decoder parses a sequence header and discovers the frame size of the output pictures, so buffers associated with its output ports shall be rearranged.
- The parameters of an audio stream vary dynamically, and a decoder should change its port settings.

Figure 3-22 shows how a video decoder and a video renderer, both of which exchange data through the IL client, should dynamically change their port settings.

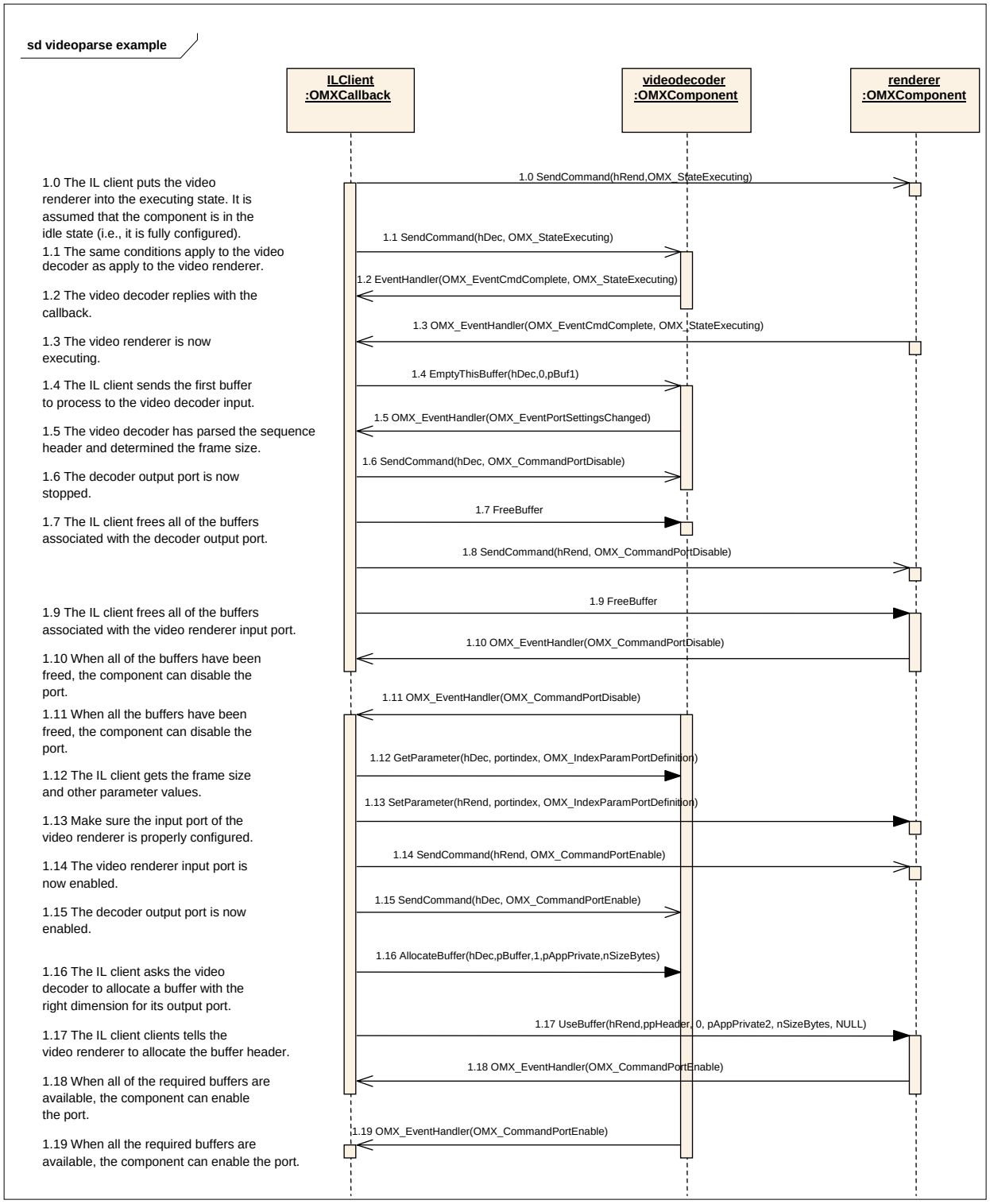


Figure 3-23. Dynamic Port Reconfiguration

The sequence starts with the IL client putting a video renderer and a video decoder in the `OMX_StateExecuting` state (1.0 through 1.3). At this stage, the output port of the video decoder and the input port of the renderer are not yet configured, since the

dimension of the output frame is unknown *a priori*. The decoder needs to start parsing the input bit stream to derive such information.

In fact, the IL client sends the first buffer to the decoder in step 1.4. Assuming that the video sequence header is included in that first buffer, the OpenMAX IL decoder component will parse it and change its output port settings accordingly.

The OpenMAX IL decoder component shall then notify the IL client by generating the `OMX_EventPortSettingsChanged` event (step 1.5). As soon as the IL client receives this callback, it shall disable the output port of the video decoder and the input port of the video renderer (steps 1.6 through 1.11).

The IL client then reads the new port settings with `OMX_GetParameter` and allocates one or more buffers with the right dimensions for the output port. Once the buffers are allocated, the IL client uses `OMX_UseBuffer` (1.17). The buffer pointers cannot be pre-announced in this example, because buffers allocated by OMX_AllocateBuffer (1.16) may change during execution. The input port of the video renderer shall also be set up with `OMX_SetParameter` (1.18).

Finally, ports can be enabled and normal processing resumes.

3.4.6 Autodetect Port Reconfiguration

This section describes how a component may change its autodetect port settings.

The following examples show where this functionality is typically needed:

- A file reader parses a media container such as a 3GPP file and discovers the video and audio decoders required to decode the elementary streams.
- The encoding types of a media container may vary so a file reader should change its port settings once the formats are determined.

Figure 3-23 Autodetect Port Reconfiguration shows how a file reader, an audio decoder and a video decoder should connect after the autodetect ports have determined the required port settings.

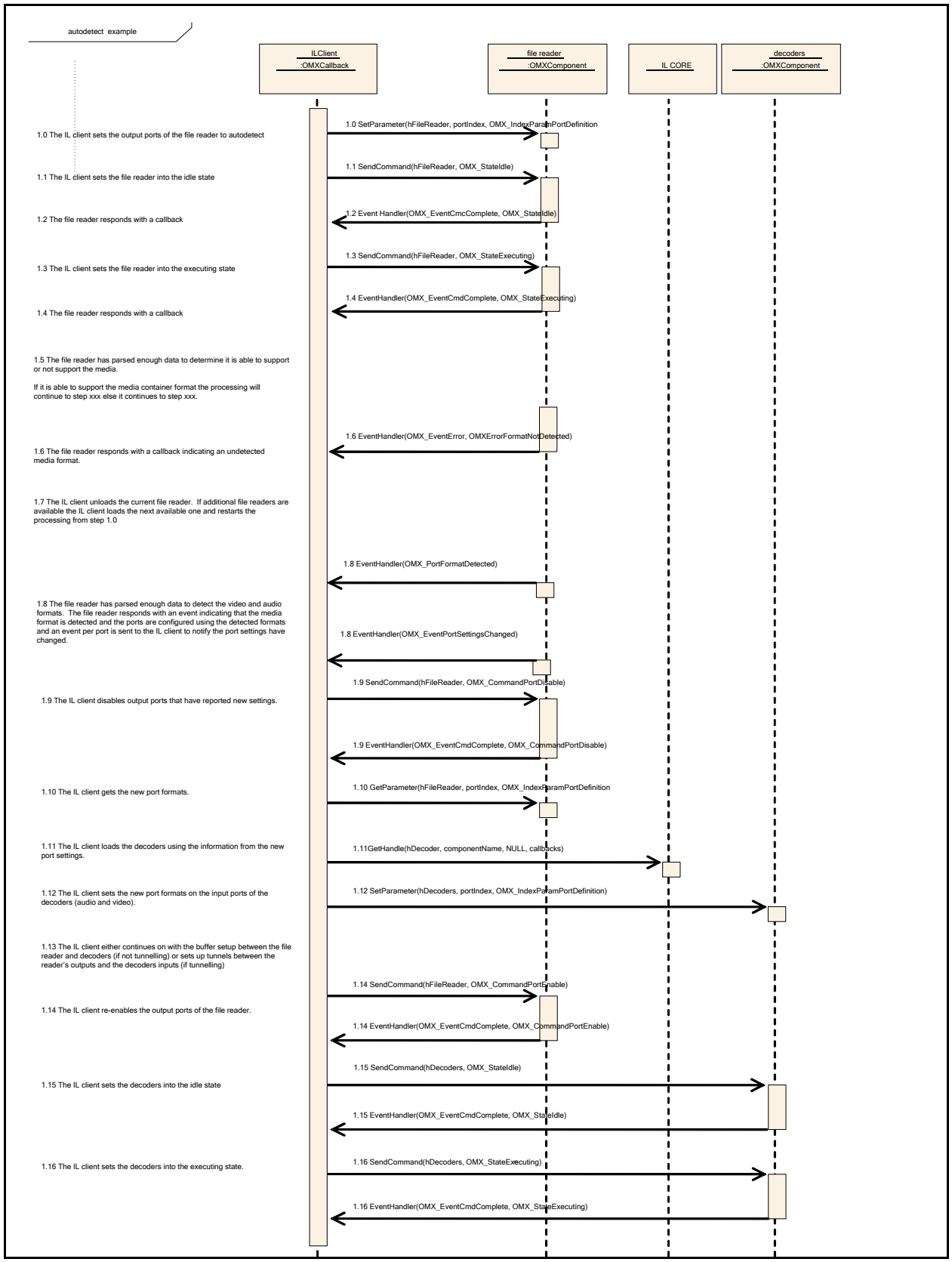


Figure 3-24 Autodetect Port Reconfiguration

The sequence starts with the IL client setting the output port formats (`OMX_IndexParamVideoPortFormat and OMX_IndexParamAudioPortFormat`) of the file reader to autodetect.

Initially the IL client instantiates only the file reader, lets all output ports communicate with the IL client, and sets all output ports to autodectect. The IL client then commands the file reader to transition into `OMX_StateIdle` thereby allocating all of its buffers. The IL client then commands the file reader to transition into `OMX_StateExecuting`.

The file reader now reads and parses data until it determines if it is able to detect the format of the media container. If the file reader is not able to detect the media container format it will notify the IL client by generating an `OMX_ErrorFormatNotDetected` error (step 1.6). Since the media container format was not detected, the IL client can return to step 1.0 with another file reader component and execute the same sequence. This continues until either the media container format is detected or no more file reader components exist that have not attempted to detect the media container format.

If the file reader is able to detect the media container format and the format of the streams it will emit on the output ports, the file reader component will change its output port settings accordingly and notify the IL client by generating the `OMX_EventPortFormatDetected` and `OMX_EventPortSettingsChanged` events (step 1.8) for each output port where the format has been changed. As soon as the IL client receives this callback, it shall disable the changed output ports of the file reader (step 1.9).

The IL client shall then read the new file reader port settings for all output ports whose settings have changed with `OMX_GetParameter.` Based on these settings the IL client shall select appropriate decoder components and call the `OMX_GetHandle` function for each. If previous step is successful, valid handles are returned in step 1.11 and the decoder components will be in the `OMX_StateLoaded` state.

The IL client shall configure the decoder components and its ports (including the format settings discovered from the parser). For this purpose, the IL core macro `OMX_SetParameter` shall be used; it may be called multiple times (step 1.12) if needed.

At this point the IL client may setup the components for either non-tunneled communication (by setting up the buffers itself) or tunneled communication (by setting up tunnels and letting the components set up the buffers)

Finally the IL client re-enables the reader's output ports and transitions the decoders into `OMX_StateIdle` then `OMX_StateExecuting`. At this point processing resumes.

3.4.7 Resource Management

This section describes the entry points for resource management. The interface between components and the resource manager are presented only as an example. Only the interface between the IL client and the components is part of the OpenMAX IL standard definition. An IL client may use the resource manager entry points.

Figure 3-24 proposes the behavior of an IL client is agnostic of the resource manager. The resource manager handles the component internally only, and the IL client has to take no special action.

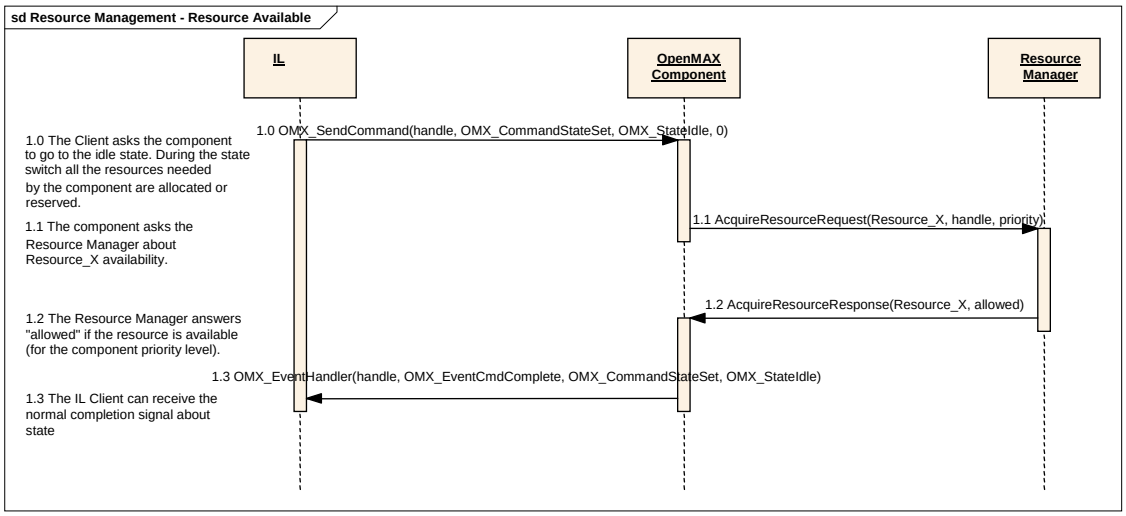


Figure 3-25. Transition from Loaded to Idle with Resource Management

In Figure 3-24, the IL client is unaware of the existence of a resource manager. In the implementation of the OpenMAX IL component, an asynchronous call to the resource manager is implemented.

The OpenMAX IL component provides a callback to the resource manager, which receives the signal for the completion of the request.

Figure 3-24 represents a possible implementation of a resource manager, and shows how it can be transparent to the client. The functions `AcquireResourceRequest` and `AcquireResourceResponse` are examples. This specification is concerned only about the interface between the IL client and the components. Details of the interactions between the components and the vendor/specific manager(s) are outside the scope of this specification.

Figure 3-25 presents a more complex use case.

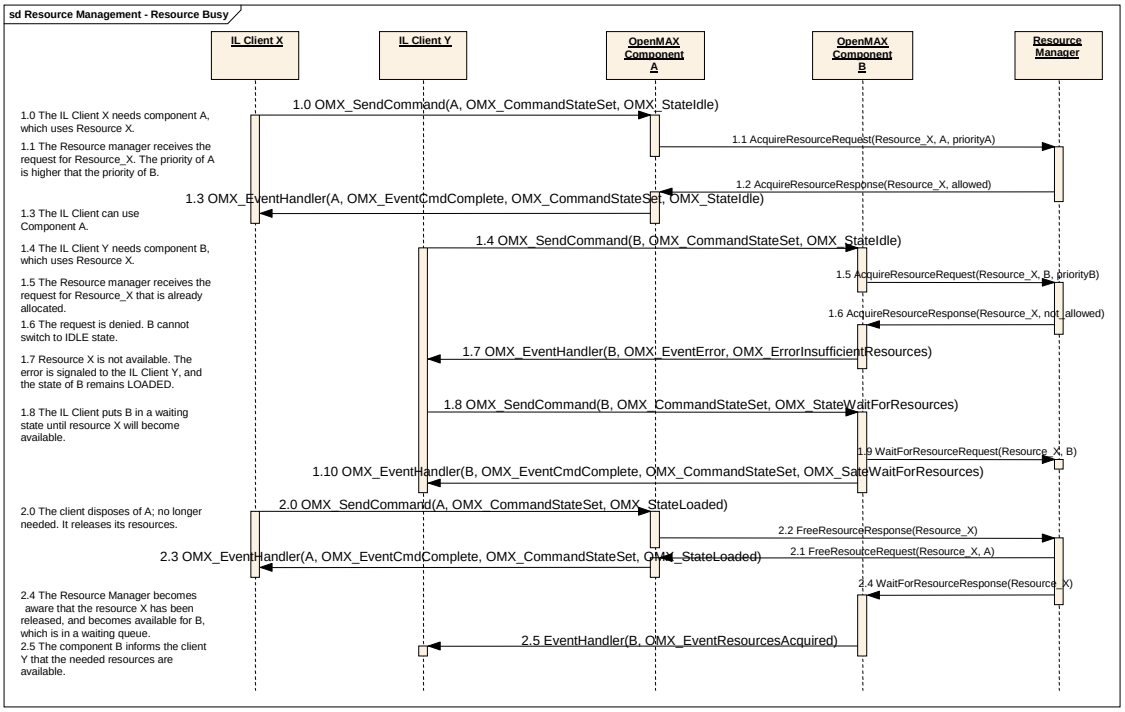


Figure 3-26. Busy Resource Management

In Figure 3-25, two different OpenMAX IL components, A and B, need the same resource to work, and they have different priorities. Here, as in the preceding example, the IL clients use the standard transition from `OMX_StateLoaded` to `OMX_StateIdle` to set up the component and allocate all of the required resources.

The first component, component A, takes ownership of the resource, requesting it from the resource manager. Component A switches to `OMX_StateIdle` and is ready to execute.

The second component, component B, asks for the same resource, but in this case the resource manager denies it since a higher priority component, component A, has that resource. This event is reported to the IL client with an error message including the value `OMX_ErrorInsufficientResources`. If IL client Y decides that it needs to be notified when this resource becomes available again, it may direct component B to change state to `OMX_StateWaitForResources.` This action puts component B in a waiting queue until the resource X will become available. Alternatively, IL client Y may request component B to switch back to `OMX_StateLoaded`.

Figure 3-25 also shows the behavior of components when resource X becomes available. Component A changes state to `OMX_StateLoaded` and releases all of the resources. The resource manager becomes aware of the available resource and calls Component B, which is already in the waiting queue.

When the resource manager provides the component with all the resources it is waiting on, the component informs the IL client that all resources needed are available with an `OMX_EventResourcesAcquired` event. The IL client shall now provide all of the needed buffers to the component. Then, the component can change state by itself to

`OMX_StateIdle` and alert the client about the state change. This waiting queue represents a unique case of automatic state change.

In Figure 3-25, the priorities of components A and B are not compared within the IL layer, and no preemption mechanism is implemented or proposed; an external policy manager, which should communicate with the resource manager, should have this responsibility. The description of such a policy manager is outside the scope of this document and the OpenMAX IL standard in general.

Figure 3-26 presents an example of a client that actively uses the resource management API.

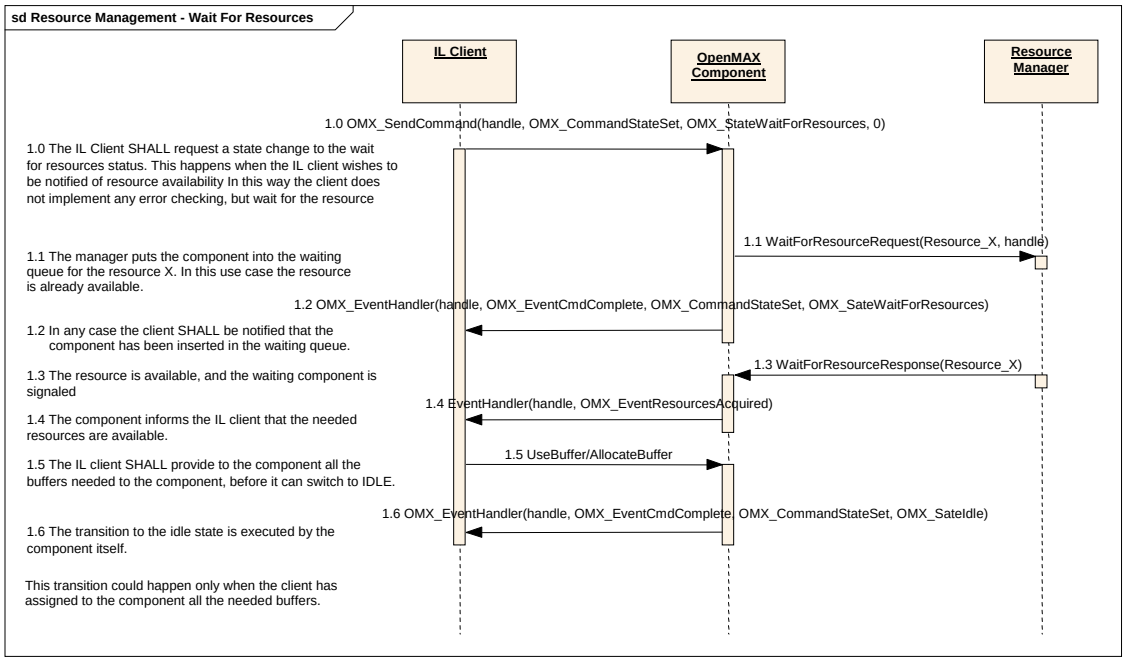


Figure 3-27. State Change from OMX_StateLoaded to OMX_StateWaitForResources

The IL client may request a state change from `OMX_StateLoaded` to `OMX_StateWaitForResources` in case the IL client wants to be notified when the resource becomes available again. For an explanation of `OMX_StateWaitForResources`, see section 3.1.1.2.5.

In this case, the client puts the component into a waiting queue, handled by the resource manager; the change to the idle state happens effectively when the resource will become available or if it is available immediately. In any case, the client receives two different `EventHandler` callbacks that correspond to two different state changes.

The two functions `WaitForResourceRequest` and `WaitForResourceResponse` in Figure 3-26 are not defined in this specification but are examples of an interaction between components and the resource manager.

The IL client may decide to stop waiting at a certain time. In this case, it shall request the component to change state back to `OMX_StateLoaded`, as shown in Figure 3-27.

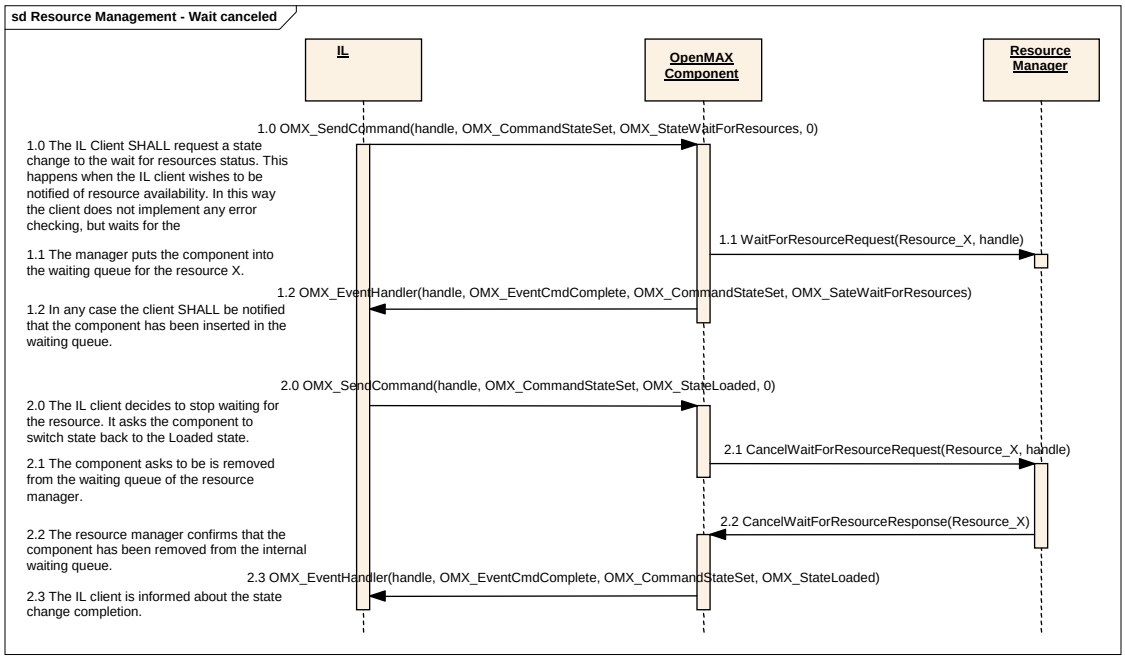


Figure 3-28. Remove Component from Waiting Status

3.4.8 Component Suspension

This section shows an example of component suspension due to lack of dynamic resources, and comprise an example of two components, MP3 decoder and AAC decoder, requiring access to a common resource.

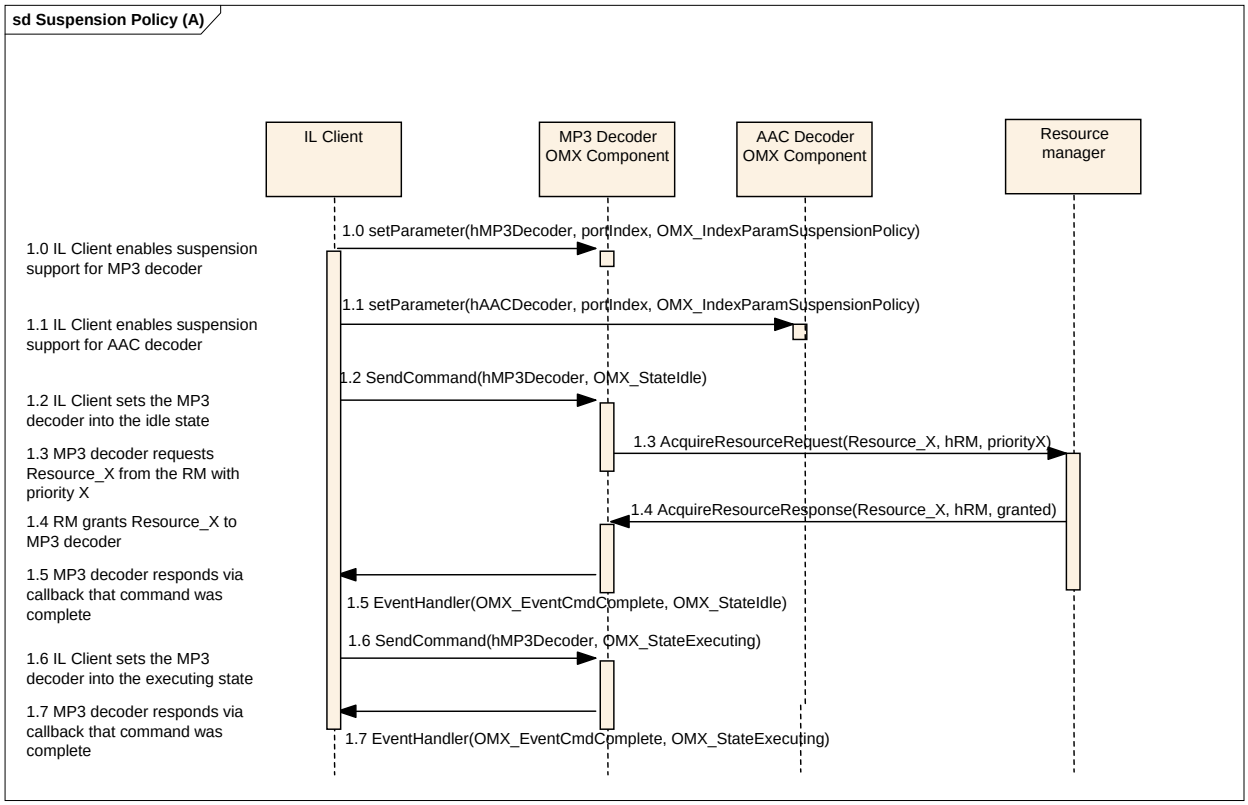


Figure 3-29: Suspension Policy (A)

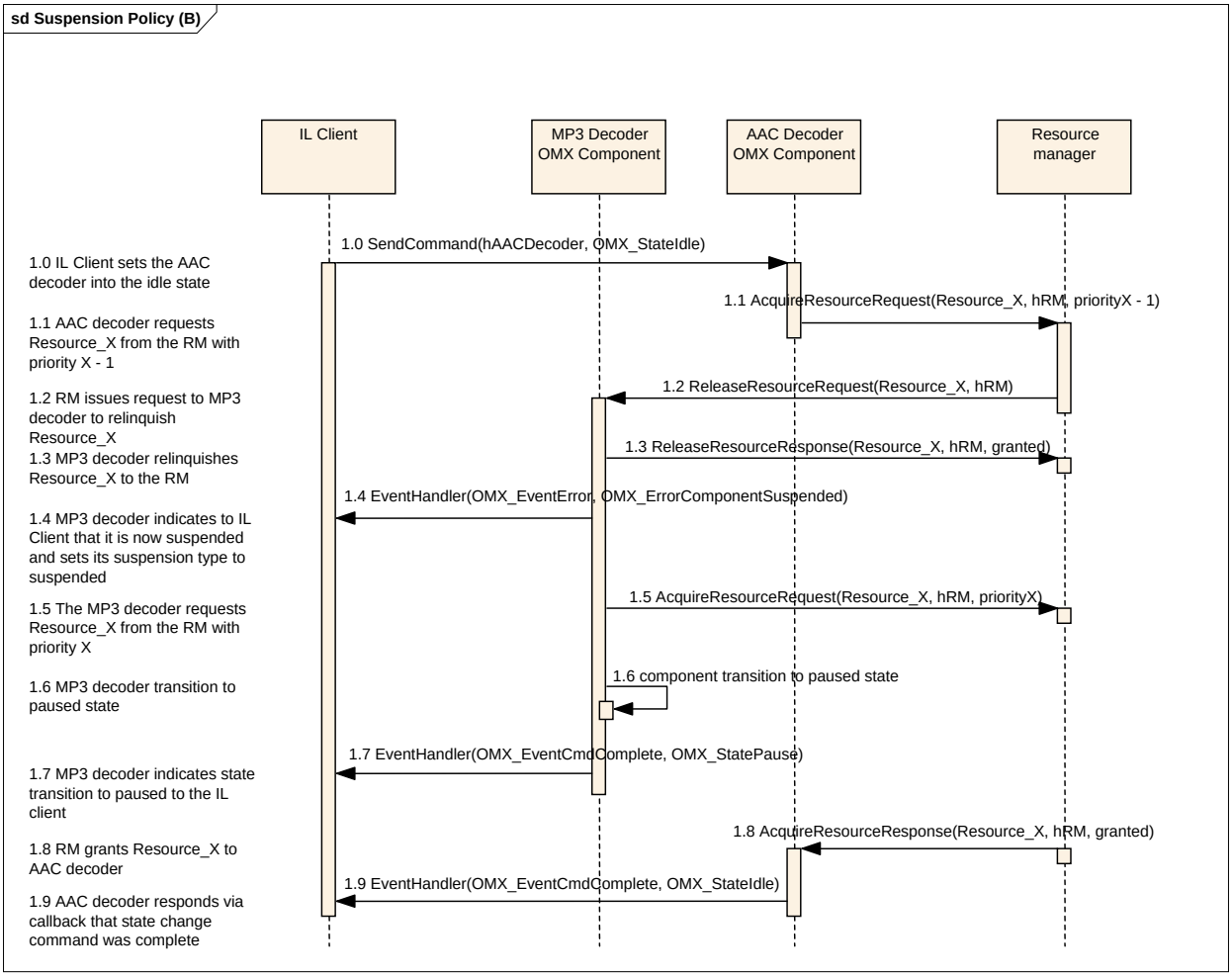


Figure 3-30: Suspension Policy (B)

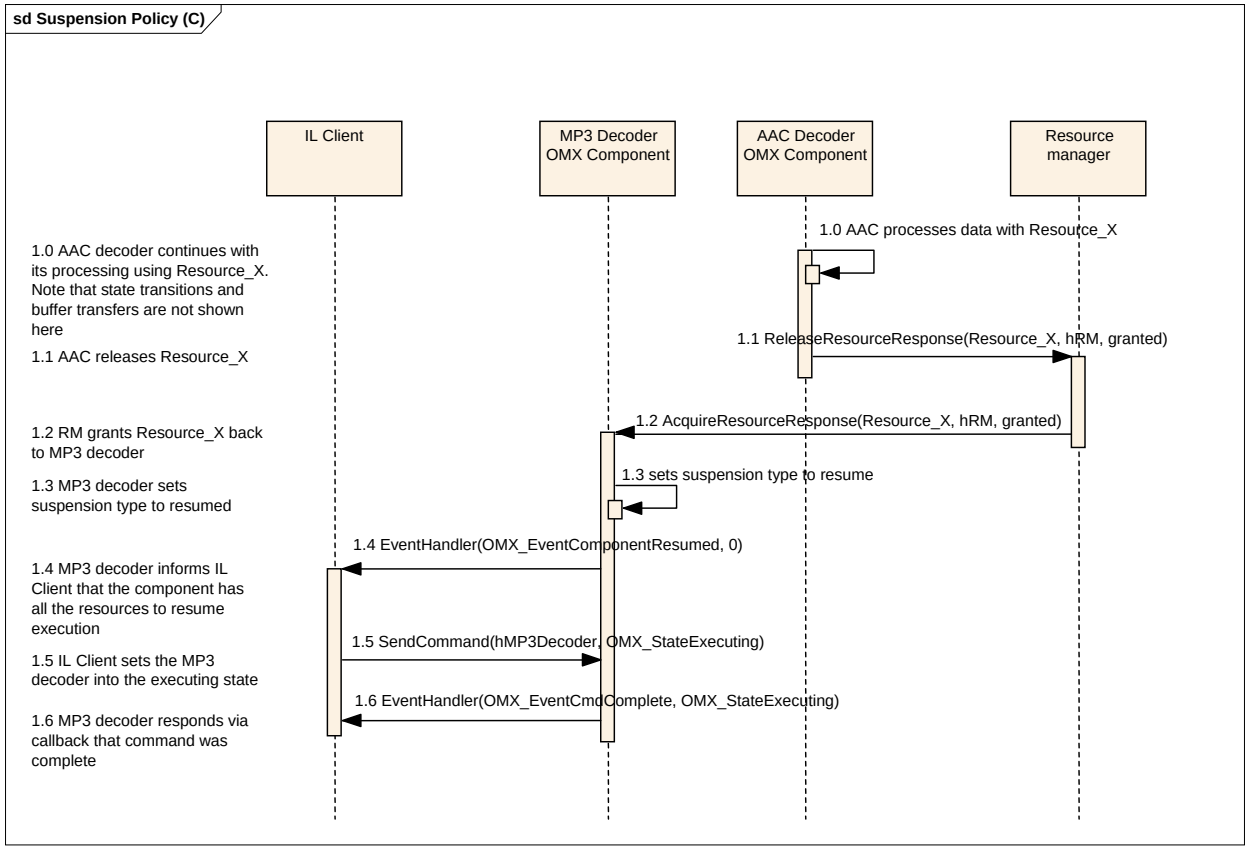


Figure 3-31: Component Suspension Due to Pre-emption of Resources

The example assume that each component needs to process a set of compressed buffers to be decoded. The IL client sets the components to support the suspension mechanism (1.0 A and 1.1 A) so that any loss of resources while processing the streams can be resumed.

The IL client transitions the MP3 decoder into `OMX_StateIdle` (1.2 A). At this time the MP3 decoder issues a request to the resource manager (RM) for Resource_X (1.3 A). The RM responds to the request by granting Resource_X to the MP3 decoder (1.4 A). The MP3 decoder is then transitioned to start processing of stream buffers. (Note the buffer transfers are not shown in the diagram for simplicity).

Next the IL client transitions the AAC decoder into `OMX_StateIdle` (1.0 B). The AAC decoder issues a request for Resource_X with as a higher priority client to the RM (1.1 B). The RM in turn issues a request to the MP3 decoder to release Resource_X (1.2 B). The MP3 decoder complies and releases Resource_X to the RM (1.3 B).

The MP3 decoder at this point sends an error to the IL client to indicate that the component is suspended (1.4 B). The MP3 decoder issues an acquire resource request for Resource_x (1.5 B) which of course the RM cannot fulfill since it is a lower priority request but the RM will track this resource request for the MP3 decoder.

The next step for the MP3 decoder is to transition to `OMX_StatePause` (1.6 B) and then emit a command complete paused event to the IL client (1.7 B). At this point the MP3 decoder is in a paused suspension state.

Concurrently, the RM may also grant Resource_X to the AAC decoder after being released by the MP3 decoder (1.7 B). The AAC decoder completes the state change to `OMX_StateIdle` by issuing a command complete to the IL client. Assuming the IL client transitions the AAC decoder to executing and after processing a number of buffers (1.0 C) the AAC decoder releases Resource_X (1.1 C).

The RM then grants Resource_X to the MP3 component (1.2 C) base on its earlier request (1.5 B). The MP3 decoder then sets its suspension type to resume (1.3 C) and then issues an `OMX_EventComponentResumed` message to the IL client (1.4 C). The IL client transitions the MP3 component out of `OMX_StatePause` to `OMX_StateExecuting` to resume the stream processing (1.5 C-1.6 C).

3.5 Slaving Behavior for Port Settings

Some components have some port settings which are common between their input port and their output port. When these components are not able to perform conversion for these common settings, there is an implicit slaving behavior that the component shall implement to make sure the common settings are always kept the same between both ports.

This slaving behavior is defined as follows:

- when issuing `OMX_SetParameter()` on the input port, if the value of the settings in common with the output port are changed, then the component shall update the output port settings and emit a `OMX_EventPortSettingsChanged` event on the output port.
- when issuing `OMX_SetParameter()`on the output port, the component shall return the error `OMX_ErrorBadParameter` if the settings in common with the input port are changed.
- if the settings in common between both ports change during runtime (for example after parsing the stream), then the component shall emit `OMX_EventPortSettingsChanged` events on both ports.

Example of components which shall implement this slaving behavior:

- audio decoder, encoder, and processor class have the sampling rate and number of channels in common between input and output ports.
- video decoder, encoder, and IV processor class have the width and height between input and output ports.

4 OpenMAX IL Data API

This section describes the typical component usage for the audio, video, image, and other domains. This section also details all of the structures, parameters, and configurations that apply to ports for each of the domains and provides use case examples where appropriate.

4.1 Audio

This section describes the structures, parameters, and configuration details for ports in the audio domain. These parameter and configurations details are specified in the OMX_Audio.h header.

4.1.1 Audio Use Case Examples

Figure 4-1 illustrates an example of an audio playback processing chain. Two sound sources are played simultaneously and are mixed with effects added to both the individual processing paths and the mixed signal. Only OpenMAX IL standard components are shown in this example.

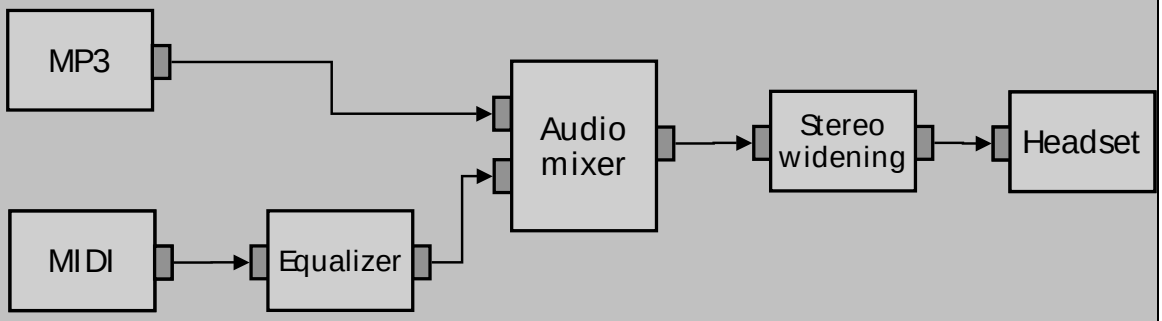


Figure 4-1. Audio Playback Processing Chain

Figure 4-2 illustrates a simple example of speech processing chains with echo cancellation added for an uplink speech path. Speech codecs can be any specified OpenMAX IL codecs.

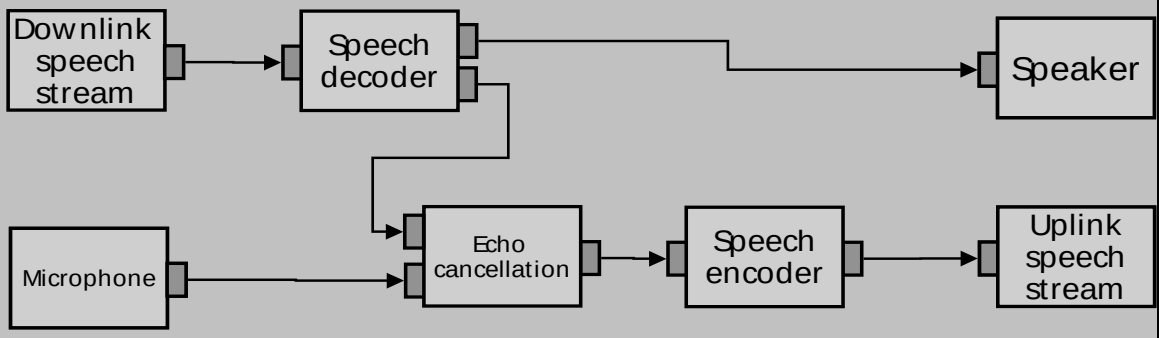


Figure 4-2. Speech Processing Chain

4.1.2 Minimum Buffer Payload Size for Uncompressed Data

OpenMAX IL has specified a minimum buffer payload sizes for all types of uncompressed data. The minimum payload size for pulse code modulation (PCM) audio is five milliseconds. This means that an output port of a PCM component shall produce at least five milliseconds of audio data for each buffer. The minimum payload size is applied only for PCM (i.e., `OMX_AUDIO_CodingPCM`) and not for any other formats.

4.1.3 Whole-file Buffering for MIDI Formats

Most MIDI content formats contain multiple parallel tracks of media data that appear in the file in serial track order rather than interleaved in real-time execution order. In addition, the MIDI state is deterministic only from the beginning of file playback, and thus seeks within any MIDI file require that at least some part of the file be re-processed from the beginning. For these reasons, callers shall provide the full length of the MIDI file data to OpenMAX IL components processing MIDI using the `nFileSize` field of the `OMX_AUDIO_PARAM_MIDITYPE` structure. For more information on the `OMX_AUDIO_PARAM_MIDITYPE` structure, see section 4.1.32.

4.1.4 General Enumerations

`OMX_AUDIO_CODINGTYPE` is the enumeration used to define the possible audio encoding types. If `OMX_AUDIO_CodingUnused` is selected, the coding selection shall be done in a vendor-specific way. Table 4-1 shows the contents of `OMX_AUDIO_CODINGTYPE`.

Table 4-1: Audio Coding Types

Field Name	Description	References to Standard(s)
`OMX_AUDIO_CodingUnused`	Placeholder value when coding is not available	Not available
`OMX_AUDIO_CodingAutoDetect`	Auto detection of audio format	Not available
`OMX_AUDIO_CodingPCM`	Any variant of PCM coding	PCM
`OMX_AUDIO_CodingADPCM`	Any variant of ADPCM encoded data	ADPCM
`OMX_AUDIO_CodingAMR`	Any variant of AMR encoded data	AMR-NB , AMR-WB , AMR-WB+
`OMX_AUDIO_CodingGSMFR`	Any variant of GSM Full-Rate (i.e., GSM610)	GSM-FR

Field Name	Description	References to Standard(s)
OMX_AUDIO_CodingGSMEFR	Any variant of GSM Enhanced Full-Rate encoded data	GSM-EFR
OMX_AUDIO_CodingGSMHR	Any variant of GSM Half-Rate encoded data	GSM-HR
OMX_AUDIO_CodingPDCFR	Any variant of PDC Full-Rate encoded data	PDC-FR
OMX_AUDIO_CodingPDCEFR	Any variant of PDC Enhanced Full-Rate encoded data	PDC-EFR
OMX_AUDIO_CodingPDCHR	Any variant of PDC Half-Rate encoded data	PDC-HR
OMX_AUDIO_CodingTDMAFR	Any variant of TDMA Full-Rate encoded data (TIA/EIA-136-420)	TDMA-FR
OMX_AUDIO_CodingTDMAEFR	Any variant of TDMA Enhanced Full-Rate encoded data (TIA/EIA-136-410)	TDMA-EFR
OMX_AUDIO_CodingQCELP8	Any variant of QCELP 8 kbps encoded data	QCELP8
OMX_AUDIO_CodingQCELP13	Any variant of QCELP 13 kbps encoded data	QCELP13
OMX_AUDIO_CodingEVRC	Any variant of EVRC encoded data	EVRC
OMX_AUDIO_CodingSMV	Any variant of SMV encoded data	SMV
OMX_AUDIO_CodingG711	Any variant of G.711 encoded data	G.711
OMX_AUDIO_CodingG723	Any variant of G.723.1 encoded data	G.723.1

Field Name	Description	References to Standard(s)
OMX_AUDIO_CodingG726	Any variant of G.726 encoded data	G.726
OMX_AUDIO_CodingG729	Any variant of G.729 encoded data	G.729
OMX_AUDIO_CodingAAC	Any variant of AAC encoded data	MPEG-2 AAC , MPEG-4 AAC HE-AAC v1 , HE-AAC v2
OMX_AUDIO_CodingMP3	Any variant of MP3 encoded data	MPEG-1 Audio , MPEG-2 Audio
OMX_AUDIO_CodingSBC	Any variant of SBC encoded data	SBC
OMX_AUDIO_CodingVORBIS	Any variant of VORBIS encoded data	VORBIS
OMX_AUDIO_CodingWMA	Any variant of WMA encoded data	WMA
OMX_AUDIO_CodingRA	Any variant of RA encoded data	RA
OMX_AUDIO_CodingMIDI	Any variant of MIDI encoded data	SP-MIDI, DLS 1, DLS 2 General MIDI, General MIDI 2 , GM Lite , XMF type 0 and 1, Mobile XMF

4.1.5 Parameter and Configuration Indexes

The header OMX_Index.h contains the enumeration `OMX_INDEXTYPE`, which contains all standard index values used with the core functions `OMX_GetParameter`, `OMX_SetParameter`, `OMX_GetConfig`, and `OMX_SetConfig`. Table 4-2 shows the indices and associated structures that relate to audio. The structures are described in the following sections.

Table 4-2: Audio-specific indices and associated structures.

OpenMAX IL Indices (OMX_Index.h)	Corresponding OpenMAX IL Audio Structures (OMX_Audio.h)
OMX_IndexParamPortDefinition	OMX_PARAM_PORTDEFINITIONTYPE with OMX_AUDIO_PORTDEFINITIONTYPE
OMX_IndexParamAudioPortFormat	OMX_AUDIO_PARAM_PORTFORMATTYPE
OMX_IndexParamAudioPcm	OMX_AUDIO_PARAM_PCMMODETYPE
OMX_IndexParamAudioMp3	OMX_AUDIO_PARAM_MP3TYPE
OMX_IndexParamAudioAac	OMX_AUDIO_PARAM_AACPROFILETYPE
OMX_IndexParamAudioVorbis	OMX_AUDIO_PARAM_VORBISTYPE
OMX_IndexParamAudioWma	OMX_AUDIO_PARAM_WMATYPE
OMX_IndexParamAudioRa	OMX_AUDIO_PARAM_RATYPE
OMX_IndexParamAudioSbc	OMX_AUDIO_PARAM_SBCTYPE
OMX_IndexParamAudioAdpcm	OMX_AUDIO_PARAM_ADPCMTYPE
OMX_IndexParamAudioG723	OMX_AUDIO_PARAM_G723TYPE
OMX_IndexParamAudioG726	OMX_AUDIO_PARAM_G726TYPE
OMX_IndexParamAudioG729	OMX_AUDIO_PARAM_G729TYPE
OMX_IndexParamAudioAmr	OMX_AUDIO_PARAM_AMRTYPE
OMX_IndexParamAudioGsm_FR	OMX_AUDIO_PARAM_GSMFRTYPE
OMX_IndexParamAudioGsm_EFR	OMX_AUDIO_PARAM_GSMEFRTYPE
OMX_IndexParamAudioGsm_HR	OMX_AUDIO_PARAM_GSMHRTYPE
OMX_IndexParamAudioTdma_FR	OMX_AUDIO_PARAM_TDMAFRTYPE
OMX_IndexParamAudioTdma_EFR	OMX_AUDIO_PARAM_TDMAEFRTYPE
OMX_IndexParamAudioPdc_FR	OMX_AUDIO_PARAM_PDCFRTYPE
OMX_IndexParamAudioPdc_EFR	OMX_AUDIO_PARAM_PDCEFRTYPE
OMX_IndexParamAudioPdc_HR	OMX_AUDIO_PARAM_PDCHRTYPE
OMX_IndexParamAudioQcelp8	OMX_AUDIO_PARAM_QCELP8TYPE
OMX_IndexParamAudioQcelp13	OMX_AUDIO_PARAM_QCELP13TYPE
OMX_IndexParamAudioEvrc	OMX_AUDIO_PARAM_EVRCTYPE
OMX_IndexParamAudioSmv	OMX_AUDIO_PARAM_SMVTYPE
OMX_IndexParamAudioMidi	OMX_AUDIO_PARAM_MIDITYPE
OMX_IndexParamAudioMidiLoadUserSound	OMX_AUDIO_PARAM_MIDILOADUSERSOUNDTYPE
OMX_IndexConfigAudioMidiImmediateEvent	OMX_AUDIO_CONFIG_MIDIIMMEDIATEEVENTTYPE
OMX_IndexConfigAudioMidiSoundBankProgram	OMX_AUDIO_CONFIG_MIDISOUNDBANKPROGRAMTYPE

OpenMAX IL Indices (OMX_Index.h)	Corresponding OpenMAX IL Audio Structures (OMX_Audio.h)
OMX_IndexConfigAudioMidiControl	OMX_AUDIO_CONFIG_MIDICONTROLTYPE
OMX_IndexConfigAudioMidiStatus	OMX_AUDIO_CONFIG_MIDISTATUSTYPE
OMX_IndexConfigAudioMidiMetaEvent	OMX_AUDIO_CONFIG_MIDIMETAEVENTTYPE
OMX_IndexConfigAudioMidiMetaEventData	OMX_AUDIO_CONFIG_MIDIMETAEVENTDATATYPE
OMX_IndexConfigAudioVolume	OMX_AUDIO_CONFIG_VOLUMETYPE
OMX_IndexConfigAudioChannelVolume	OMX_AUDIO_CONFIG_CHANNELVOLUMETYPE
OMX_IndexConfigAudioBalance	OMX_AUDIO_CONFIG_BALANCETYPE
OMX_IndexConfigAudioMute	OMX_AUDIO_CONFIG_MUTETYPE
OMX_IndexConfigAudioChannelMute	OMX_AUDIO_CONFIG_CHANNELMUTETYPE
OMX_IndexConfigAudioLoudness	OMX_AUDIO_CONFIG_LOUDNESSTYPE
OMX_IndexConfigAudioBass	OMX_AUDIO_CONFIG_BASSTYPE
OMX_IndexConfigAudioTreble	OMX_AUDIO_CONFIG_TREBLETYPE
OMX_IndexConfigAudioEqualizer	OMX_AUDIO_CONFIG_EQUALIZERTYPE
OMX_IndexConfigAudioStereoWidening	OMX_AUDIO_CONFIG_STEREOWIDENINGTYPE
OMX_IndexConfigAudioChorus	OMX_AUDIO_CONFIG_CHORUSTYPE
OMX_IndexConfigAudioReverberation	OMX_AUDIO_CONFIG_REVERBERATIONTYPE
OMX_IndexConfigAudioEchoCancelation	OMX_AUDIO_CONFIG_ECHOCANCELATIONTYPE
OMX_IndexConfigAudioNoiseReduction	OMX_AUDIO_CONFIG_NOISEREDUCTIONTYPE
OMX_IndexConfigAudio3DOutput	OMX_AUDIO_CONFIG_3DOUTPUTTYPE
OMX_IndexConfigAudio3DLocation	OMX_AUDIO_CONFIG_3DLOCATIONTYPE
OMX_IndexParamAudio3DDopplerMode	OMX_AUDIO_PARAM_3DDOPPLERMODETYPE
OMX_IndexConfigAudio3DDopplerSettings	OMX_AUDIO_CONFIG_3DDOPPLERSETTINGSTYPE
OMX_IndexConfigAudio3DLevels	OMX_AUDIO_CONFIG_3DLEVELSTYPE
OMX_IndexConfigAudio3DDistanceAttenuation	OMX_AUDIO_CONFIG_3DDISTANCEATTENUATIONTYPE
OMX_IndexConfigAudio3DDirectivitySettings	OMX_AUDIO_CONFIG_3DDIRECTIVITYSETTINGSTYPE
OMX_IndexConfigAudio3DDirectivityOrientation	OMX_AUDIO_CONFIG_3DDIRECTIVITYORIENTATIONTYPE

OpenMAX IL Indices (OMX_Index.h)	Corresponding OpenMAX IL Audio Structures (OMX_Audio.h)
OMX_IndexConfigAudio3DMacroscopicOrientation	OMX_AUDIO_CONFIG_3DMACROSCOPICORIENTATIONTYPE
OMX_IndexConfigAudio3DMacroscopicSize	OMX_AUDIO_CONFIG_3DMACROSCOPICSIZETYPE
OMX_IndexParamAudioQueryChannelMapping	OMX_AUDIO_CHANNELMAPPINGTYPE
OMX_IndexConfigAudioSbcBitpool	OMX_AUDIO_SBCBITPOOLTYPE
OMX_IndexConfigAudioAmrMode	OMX_AUDIO_AMRMODETYPE
OMX_IndexConfigAudioBitrate	OMX_AUDIO_CONFIG_BITRATETYPE
OMX_IndexConfigAudioAMRISFIndex	OMX_AUDIO_CONFIG_AMRISFTYPE
OMX_IndexParamAudioFixedPoint	OMX_AUDIO_FIXEDPOINTTYPE

4.1.6 OMX_AUDIO_PORTDEFINITIONTYPE

The OMX_AUDIO_PORTDEFINITIONTYPE structure is used to define all of the parameters necessary for the compliant component to set up an input or an output audio path. If additional information is needed to define the parameters of the port, such as frequency, additional structures such as the OMX_AUDIO_PARAM_PCMMODETYPE structure shall be sent to supply the extra parameters for the port. The number of audio paths for input and output will vary by the type of the audio component.

OMX_Component.h contains common port definition structures for all media domains.

The OMX_AUDIO_PORTDEFINITIONTYPE structure can query the current definition of an audio port or set the definition of an audio port for a component. The OMX_AUDIO_PORTDEFINITIONTYPE structure is included as part of the OMX_PARAM_PORTDEFINITIONTYPE structure, it is accessed via the OMX_GetParameter function or the OMX_GetParameter function using the OMX_IndexParamPortDefinition index.

OMX_AUDIO_PORTDEFINITIONTYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PORTDEFINITIONTYPE {
    OMX_NATIVE_DEVICETYPE pNativeRender;
    OMX_BOOL bFlagErrorConcealment;
    OMX_AUDIO_CODINGTYPE eEncoding;
} OMX_AUDIO_PORTDEFINITIONTYPE;
```

The parameters for OMX_AUDIO_PORTDEFINITIONTYPE are defined as follows.

- pNativeRender is the platform-specific reference for an output device; otherwise this field is 0.

- `bFlagErrorConcealment` turns on error concealment if it is supported by the OpenMAX IL component.
- `eEncoding` is the type of data expected for this port (e.g., PCM, AMR, MP3, and so forth).

4.1.7 OMX_AUDIO_PARAM_PORTFORMATTYPE

`OMX_AUDIO_PARAM_PORTFORMATTYPE` is the structure for the port format parameter. This structure enumerates the various data formats that the port supports.

This parameter call can be used with both `OMX_GetParameter` and `OMX_SetParameter`. In the `OMX_GetParameter` case, the caller specifies all fields and the `OMX_GetParameter` call returns the value of `eEncoding`. The value of `nIndex` goes from 0 to N-1, where N is the number of formats supported by the port. The port does not need to report N as the caller can determine N by enumerating all the formats supported by the port. Each port shall support at least one format. If there are no more formats, `OMX_GetParameter` returns `OMX_ErrorNoMore` (i.e., `nIndex` is supplied where the value is N or greater). Ports supply formats in order of preference: Higher preference formats are provided with lower values of `nIndex.`

For `OMX_SetParameter`, the `nIndex` field is ignored. If the format is supported, it is set as the format of the port, and the default values for the format are programmed into the port definition type as a side effect. This allows the caller to query the default values for the format without having to know them in advance.

`OMX_AUDIO_PARAM_PORTFORMATTYPE` is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_PORTFORMATTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nIndex;
    OMX_AUDIO_CODINGTYPE eEncoding;
} OMX_AUDIO_PARAM_PORTFORMATTYPE;
```

The parameters for `OMX_AUDIO_PARAM_PORTFORMATTYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `nIndex` indicates the enumeration index for the format from 0 to N-1.
- `eEncoding` is the type of data expected for this port (e.g., PCM, AMR, MP3, and so forth).

4.1.8 OMX_AUDIO_PARAM_PCMMODETYPE

The `OMX_AUDIO_PARAM_PCMMODETYPE` structure is used to set or query the current settings for PCM audio using the `OMX_GetParameter` function. It is also used to set the parameters for PCM audio using the `OMX_SetParameter` function. When calling

either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioPcm.

Note that the minimum buffer payload size is applied to all modes of PCM audio. The payload size is defined by OMX_MIN_PCMPAYLOAD_MSEC and is five milliseconds.

OMX_AUDIO_PARAM_PCMMODETYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_PCMMODETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_NUMERICALDATATYPE eNumData;
    OMX_ENDIANTYPE eEndian;
    OMX_BOOL bInterleaved;
    OMX_U32 nBitPerSample;
    OMX_U32 nSamplingRate;
    OMX_AUDIO_PCMMODETYPE ePCMMode;
    OMX_AUDIO_CHANNELTYPE eChannelMapping[OMX_AUDIO_MAXCHANNELS];
} OMX_AUDIO_PARAM_PCMMODETYPE;
```

4.1.8.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_PCMMODETYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of channels of audio (mono, stereo, multi-channel).
- eNumData indicates whether the PCM data is signed or unsigned.

Table 4-3: OMX_NUMERICALDATATYPE enumeration

Field Name	Description
OMX_NumericalDataSigned	Signed data
OMX_NumericalDataUnsigned	Unsigned data

- eEndian indicates whether PCM data is in little- or big-endian order.

Table 4-4: OMX_ENDIANTYPE enumeration

Field Name	Description
OMX_EndianBig	Big-endian data
OMX_EndianLittle	Little-endia data

- bInterleaved indicates whether the data is normal interleaved or non-interleaved. True represents normal interleaved data, and false represents non-interleaved data such as block data.
- nBitPerSample is the number of bits per sample.
- nSamplingRate is the sampling rate of the source data. Use the value 0 for variable or unknown sampling rate.

- `ePCMMode` is the PCM mode enumeration. Table 4-5 identifies the PCM mode.

Table 4-5: PCM Mode

Field Name	Description
`OMX_AUDIO_PCMModeLinear`	Linear PCM encoded data
`OMX_AUDIO_PCMModeALaw`	A law PCM encoded data (G.711)
`OMX_AUDIO_PCMModeMULaw`	μ law PCM encoded data (G.711)

- `eChannelMapping` is the audio channel mapping enumeration. A component will indicate the order of the audio channels as shown in Table 4-6. A component should use the default channel mapping (standard RIFF/WAV mapping as present in standard multi-channel WAV files: FRONT_LEFT FRONT_RIGHT FRONT_CENTER LOW_FREQUENCY BACK_LEFT BACK_RIGHT .) if possible.

Table 4-6: Audio Channel Mapping

Field Name	Description
`OMX_AUDIO_ChannelNone`	Unused or empty
`OMX_AUDIO_ChannelUnknown`	Unknown channel mapping
`OMX_AUDIO_ChannelLF`	Left front
`OMX_AUDIO_ChannelRF`	Right front
`OMX_AUDIO_ChannelCF`	Center front
`OMX_AUDIO_ChannelLS`	Left surround
`OMX_AUDIO_ChannelRS`	Right surround
`OMX_AUDIO_ChannelLFE`	Low frequency effects
`OMX_AUDIO_ChannelCS`	Back surround
`OMX_AUDIO_ChannelLR`	Left rear
`OMX_AUDIO_ChannelRR`	Right rear
`OMX_AUDIO_ChannelLCF`	Left of center front
`OMX_AUDIO_ChannelRCF`	Right of center front
`OMX_AUDIO_ChannelLHS`	Left (Hand) side
`OMX_AUDIO_ChannelRHS`	Right (Hand) side
`OMX_AUDIO_ChannelCT`	Center top
`OMX_AUDIO_ChannelFLT`	Front left top
`OMX_AUDIO_ChannelFCT`	Front center top
`OMX_AUDIO_ChannelFRT`	Front right top
`OMX_AUDIO_ChannelBLT`	Back left top
`OMX_AUDIO_ChannelBCT`	Back center top
`OMX_AUDIO_ChannelBRT`	Back right top

4.1.8.2 Functionality

The OMX_AUDIO_PARAM_PCMMODETYPE structure sets the parameters of PCM audio.

4.1.9 OMX_AUDIO_PARAM_MP3TYPE

The OMX_AUDIO_PARAM_MP3TYPE structure is used to set or query the current settings for the MPEG Layer-3 (MP3) codec component using the OMX_GetParameter function. It is also used to set the parameters of the MP3 codec component using the OMX_SetParameter function. The index specified for this structure is OMX_IndexParamAudioMp3 when calling either the OMX_GetParameter or the OMX_SetParameter functions.

OMX_AUDIO_PARAM_MP3TYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_MP3TYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_U32 nBitRate;
    OMX_U32 nSampleRate;
    OMX_U32 nAudioBandWidth;
    OMX_AUDIO_CHANNELMODETYPE eChannelMode;
    OMX_AUDIO_MP3STREAMFORMATTYPE eFormat;
} OMX_AUDIO_PARAM_MP3TYPE;
```

4.1.9.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_MP3TYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of channels of audio (mono, stereo, multi-channel).
- nBitRate is the bit rate of the encoded MP3 audio. If the bit rate is variable or unknown, this parameter has the value 0.
- nSampleRate is the sample rate of the encoded or decoded audio.
- nAudioBandWidth is the audio bandwidth in Hz to which an encoder should limit the audio signal. Use the value 0 to let encoder decide.
- eChannelMode is the enumeration of OMX_AUDIO_CHANNELMODETYPE for the audio channel mode. AAC, MP3, and SBC use this value, although the names are most appropriate for MP3. Table 4-7 shows the values.

Table 4-7: Audio Channel Mode

Mode	Description
OMX_AUDIO_ChannelModeStereo	Two channels. The bit rate allocation between the two channels changes according to each channel's information.
OMX_AUDIO_ChannelModeJointStereo	A mode that takes advantage of what is common between the two channels for higher compression gain.
OMX_AUDIO_ChannelModeDual	Two mono channels. Each channel is encoded with half the bit rate of the overall bit rate.
OMX_AUDIO_ChannelModeMono	Mono channel mode.

- eFormat is the stream format type supported for encoding and decoding MP3 content. Table 4-8 shows the possible MP3 audio stream format types for OMX_AUDIO_MP3STREAMFORMATTYPE.

Table 4-8: MP3 Stream Format Values

Field Name	Description
OMX_AUDIO_MP3StreamFormatUnknown	Unknown, unused or not required format setting.
OMX_AUDIO_MP3StreamFormatMP1Layer3	MPEG1 Layer 3 stream format.
OMX_AUDIO_MP3StreamFormatMP2Layer3	MPEG2 Layer 3 stream format.
OMX_AUDIO_MP3StreamFormatMP2_5Layer3	MPEG2.5 Layer 3 stream format.

4.1.9.2 Functionality

The OMX_AUDIO_PARAM_MP3TYPE structure sets the parameters of the MP3 codec.

4.1.10 OMX_AUDIO_PARAM_AACPROFILETYPE

The OMX_AUDIO_PARAM_AACPROFILETYPE structure is used to set or query the current settings for the MPEG AAC codec component using the OMX_GetParameter function. It is also used to set the parameters of the AAC codec component using the OMX_SetParameter function. The index specified for this structure is OMX_IndexParamAudioAac when calling either the OMX_GetParameter or the OMX_SetParameter functions.

OMX_AUDIO_PARAM_AACPROFILETYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_AACPROFILETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_U32 nSampleRate;
    OMX_U32 nBitRate;
    OMX_U32 nAudioBandWidth;
    OMX_U32 nFrameLength;
    OMX_U32 nAACtools;
    OMX_U32 nAACERtools;
    OMX_AUDIO_AACPROFILETYPE eAACProfile;
    OMX_AUDIO_AACSTREAMFORMATTYPE eAACStreamFormat;
    OMX_AUDIO_CHANNELMODETYPE eChannelMode;
} OMX_AUDIO_PARAM_AACPROFILETYPE;
```

4.1.10.1 Parameter Definitions

The parameters for the `OMX_AUDIO_PARAM_AACPROFILETYPE` structure are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `nChannels` is the number of channels of audio (mono, stereo, multi-channel).
- `nSampleRate` is the sample rate of the encoded or decoded audio.
- `nBitRate` is the bit rate of the encoded AAC audio. If the bit rate is variable or unknown, this parameter has the value 0.
- `nAudioBandWidth` is the audio bandwidth in Hz to which an encoder should limit the audio signal. Use the value 0 to let the encoder decide.
- `nFrameLength` is the frame length of the codec in audio samples per channel. The value can be 1024 (AAC) or 960 (AAC-LC), 2048 (HE-AAC), 512 or 480 (AAC-LD). Use the value 0 to let encoder decide.
- `nAACtools` is the AAC tool usage. Table 4-9 shows the preprocessor defines that should be used to signal the use of AAC coding tools. Use `OMX_AUDIO_AACToolAll` to let the encoder decide. Preprocessor defines are used to allow parameter passing in the following fashion:
  `"AACtoolParam = OMX_AUDIO_AACToolMS + OMX_AUDIO_AACToolTNS;"`

Table 4-9: AAC Tool Usage

Define Name	Description
`OMX_AUDIO_AACToolNone`	No AAC tools allowed (encoder configuration) or active (optional decoder information output).
`OMX_AUDIO_AACToolMS`	Mid/Side (MS) joint coding tool.
`OMX_AUDIO_AACToolIS`	Intensity Stereo (IS) tool.
`OMX_AUDIO_AACToolTNS`	Temporal Noise Shaping (TNS) tool.

Define Name	Description
OMX_AUDIO_AACToolPNS	MPEG-4 Perceptual Noise Substitution (PNS) tool.
OMX_AUDIO_AACToolLTP	MPEG-4 Long Term Prediction (LTP) tool.
OMX_AUDIO_AACToolAll	All AAC tools allowed or active.

- nAACERtools is the AAC Error Resilience tool usage. Table 4-10 shows the preprocessor defines that should be used to signal the use of AAC Error Resilience tools. Use OMX_AUDIO_AACERAll to let encoder decide. Preprocessor defines are used to allow parameter passing in the following fashion:
"AACERtoolParam = OMX_AUDIO_AACERRVLC + OMX_AUDIO_AACERHCR;"

Table 4-10: AAC Error Resilience Tool Usage

Define Name	Description
OMX_AUDIO_AACERNone	No AAC ER tools allowed/used
OMX_AUDIO_AACERVCB11	Virtual Code Books for AAC section data (VCB11)
OMX_AUDIO_AACERRVLC	Reversible Variable Length Coding (RVLC)
OMX_AUDIO_AACERHCR	Huffman Codeword Reordering (HCR)
OMX_AUDIO_AACERAll	All AAC ER tools allowed/used

- eAACProfile is the enumeration of OMX_AUDIO_AACPROFILETYPE for the AAC profile type. The term *profile* is used in the MPEG-2 AAC standard and the terms *object type* and *profile* are used in the MPEG-4 AAC standard. Table 4-11 shows the values and descriptions.

Table 4-11: AAC Profile Type

Field Name	Description
OMX_AUDIO_AACObjectUnknown	Unknown, unused or not required setting.
OMX_AUDIO_AACObjectNull	Null - not used
OMX_AUDIO_AACObjectMain	AAC Main object/profile
OMX_AUDIO_AACObjectLC	AAC Low Complexity object/profile (MPEG-4: AAC profile)
OMX_AUDIO_AACObjectSSR	AAC Scalable Sample Rate object/profile
OMX_AUDIO_AACObjectLTP	AAC Long Term Prediction object
OMX_AUDIO_AACObjectHE	High Efficiency AAC (object type SBR, MPEG-4: HE-AAC profile)

Field Name	Description
OMX_AUDIO_AACObjectScalable	AAC Scalable object
OMX_AUDIO_AACObjectERLC	ER AAC Low Complexity object (Error Resilient AAC-LC)
OMX_AUDIO_AACObjectLD	AAC Low Delay object (Error Resilient)
OMX_AUDIO_AACObjectHE_PS	AAC High Efficiency with Parametric Stereo coding (HE-AAC v2, object type PS)

- eAACStreamFormat is the enumeration of OMX_AUDIO_AACSTREAMFORMATTYPE for the AAC stream format. Table 4-12 shows the field names and values.

Table 4-12: AAC Stream Format Type

Field Name	Description
OMX_AUDIO_AACStreamFormatUnknown	Unknown, unused or not required format setting.
OMX_AUDIO_AACStreamFormatMP2ADTS	MPEG-2 AAC Audio Data Transport Stream format
OMX_AUDIO_AACStreamFormatMP4ADTS	MPEG-4 AAC Audio Data Transport Stream format
OMX_AUDIO_AACStreamFormatMP4LOAS	Low Overhead Audio Stream format
OMX_AUDIO_AACStreamFormatMP4LATM	Low Overhead Audio Transport Multiplex
OMX_AUDIO_AACStreamFormatADIF	Audio Data Interchange Format
OMX_AUDIO_AACStreamFormatMP4FF	AAC inside MPEG-4/ISO File Format
OMX_AUDIO_AACStreamFormatRAW	AAC Raw Format (access units)

- eChannelMode is the enumeration for the audio channel mode used by AAC and MP3, although the names are more appropriate for MP3. For more information on MP3, see section 4.1.9.

4.1.10.2 Functionality

The OMX_AUDIO_PARAM_AACPROFILETYPE structure sets the parameters of the AAC codec.

4.1.11 OMX_AUDIO_PARAM_VORBISTYPE

The OMX_AUDIO_PARAM_VORBISTYPE structure is used to set or query the current settings for the Vorbis codec component of the Ogg Vorbis format using the OMX_GetParameter function. It is also used to set the parameters of the Vorbis codec

component using the OMX_SetParameter function. The index specified for this structure is OMX_IndexParamAudioVorbis when calling either the OMX_GetParameter or the OMX_SetParameter functions.

OMX_AUDIO_PARAM_VORBISTYPE is defined as follows.

```
typedef struct OMX_AUDIO_PARAM_VORBISTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_U32 nBitRate;
    OMX_U32 nMinBitRate;
    OMX_U32 nMaxBitRate;
    OMX_U32 nSampleRate;
    OMX_U32 nAudioBandWidth;
    OMX_S32 nQuality;
    OMX_BOOL bManaged;
    OMX_BOOL bDownmix;
} OMX_AUDIO_PARAM_VORBISTYPE;
```

4.1.11.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_VORBISTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of channels of audio (mono, stereo, multi-channel).
- nBitRate is the bit rate of the encoded Vorbis audio. If the bit rate is variable or unknown, this parameter has the value 0. Encoding is set to the bit rate closest to the specified value in bits per second (bps).
- nMinBitRate sets the minimum bit rate in bps.
- nMaxBitRate sets the maximum bit rate in bps.
- nSampleRate is the sample rate of the encoded or decoded audio. Use the value 0 for variable or unknown sampling rate.
- nAudioBandWidth is the audio bandwidth in Hz to which an encoder should limit the audio signal. Use the value 0 to let encoder decide.
- nQuality sets the encoding quality between -1 (low) and 10 (high). In the default mode of operation, the quality level is 3. The normal quality range is 0-10.
- bManaged sets the bit rate management mode. This turns off the normal variable bit rate (VBR) encoding but allows the encoder to enforce hard or soft bit rate constraints. This mode can be slower and may also be of lower quality; it is primarily useful for streaming.
- bDownmix sets the downmix input from stereo to mono. This parameter has no effect on non-stereo streams. This parameter is useful for lower bit-rate encoding.

4.1.11.2 Functionality

The OMX_AUDIO_PARAM_VORBISTYPE structure sets the parameters of the Vorbis codec.

4.1.12 OMX_AUDIO_PARAM_WMATYPE

The OMX_AUDIO_PARAM_WMATYPE structure is used to set or query the current settings for the Windows Media® audio codec component using the OMX_GetParameter function. It is also used to set the parameters of the Windows Media audio codec component using the OMX_SetParameter function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioWma.

OMX_AUDIO_PARAM_WMATYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_WMATYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U16 nChannels;
    OMX_U32 nBitRate;
    OMX_AUDIO_WMAFORMATTYPE eFormat;
    OMX_AUDIO_WMAPROFILETYPE eProfile;
    OMX_U32 nSamplingRate;
    OMX_U16 nBlockAlign;
    OMX_U16 nEncodeOptions;
    OMX_U32 nSuperBlockAlign;
    OMX_U32 nBitsPerSample ;
    OMX_U32 nAdvancedEncodeOpt;
    OMX_U32 nAdvancedEncodeOpt2 ;
} OMX_AUDIO_PARAM_WMATYPE;
```

4.1.12.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_WMATYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of channels of audio (mono, stereo).
- nBitRate is the bit rate of the encoded Windows Media audio. If the bit rate is variable or unknown, this parameter has a value 0.
- eFormat is the enumeration for the version of the Windows Media audio codec. Table 4-13 shows the field names and values.

Table 4-13: Windows Media Audio Codec Version

Field Name	Description
OMX_AUDIO_WMAFormatUnknown	Unknown, unused or not required format setting.

Field Name	Description
OMX_AUDIO_WMAFormatUnused	The version of the Windows Media audio codec is either not applicable or is unknown.
OMX_AUDIO_WMAFormat7	Windows Media audio version 7.
OMX_AUDIO_WMAFormat8	Windows Media audio version 8.
OMX_AUDIO_WMAFormat9	Windows Media audio version 9.
OMX_AUDIO_WMAFormat9_Professional	Windows Media audio version 9 Professional.
OMX_AUDIO_WMAFormat9_Lossless	Windows Media audio version 9 Lossless.
OMX_AUDIO_WMAFormat9_Voice	Windows Media audio version 9 Voice.
OMX_AUDIO_WMAFormat10_Professional	Windows Media audio version 10 Professional.
OMX_AUDIO_WMAFormat10_Voice	Windows Media audio version 10 Voice.

- `eProfile` is the enumeration for the profile of the Windows Media audio codec. Table 4-14 shows the field names and values.

Table 4-14: Windows Media Audio Codec Profile

Field Name	Description
OMX_AUDIO_WMAProfileUnknown	Unknown, unused or not required setting.
OMX_AUDIO_WMAProfileUnused	The profile of the Windows Media audio codec is either not applicable or is unknown.
OMX_AUDIO_WMAProfileL1	Windows Media audio version 9 profile L1.
OMX_AUDIO_WMAProfileL2	Windows Media audio version 9 profile L2.
OMX_AUDIO_WMAProfileL3	Windows Media audio version 9 profile L3.
OMX_AUDIO_WMAProfileM0	Windows Media audio Pro profile M0.
OMX_AUDIO_WMAProfileM1	Windows Media audio Pro profile M1.
OMX_AUDIO_WMAProfileM2	Windows Media audio Pro profile M2.
OMX_AUDIO_WMAProfileM3	Windows Media audio Pro profile M3.
OMX_AUDIO_WMAProfileN1	Windows Media audio Lossless profile N1.
OMX_AUDIO_WMAProfileN2	Windows Media audio Lossless profile N2.
OMX_AUDIO_WMAProfileN3	Windows Media audio Lossless profile N3.
OMX_AUDIO_WMAProfileS1	Windows Media audio Voice profile S1.
OMX_AUDIO_WMAProfileS2	Windows Media audio Voice profile S2.

- `nSamplingRate` is the sampling rate of the source data.
- `nBlockAlign` is the block alignment, or block size, in bytes of the audio codec.

- nEncodeOptions is WMA Type-specific data.
- nSuperBlockAlign is WMA Type-specific data.
- nBitsPerSample refers to the stream can be encoded for 24-bit or 16-bit. This parameter is required to support wma lossless AND wma pro configuration. This parameter is specified as a per channel value.
- nAdvancedEncodeOpt is WMA Type-specific data. It refers to bit packed words indicating the features supported for LBR bitstream. This parameter is valid for OMX_AUDIO_WMAFormat9_Lossless wma lossless and OMX_AUDIO_WMAFormat10_Professional for both encoders and decoders.
- nAdvancedEncodeOpt2 is WMA Type-specific data. It refers to bit packed words indicating the features supported for LBR bitstream. This parameter is valid for OMX_AUDIO_WMAFormat9_Lossless wma lossless and OMX_AUDIO_WMAFormat10_Professional for both encoders and decoders.

4.1.13 OMX_AUDIO_PARAM_RATYPE

The OMX_AUDIO_PARAM_RATYPE structure is used to set or query the current settings for the RealAudio® codec component using the OMX_GetParameter function. It is also used to set the parameters of the codec component using the OMX_SetParameter function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioRa.

OMX_AUDIO_PARAM_RATYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_RATYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_U32 nSamplingRate;
    OMX_U32 nBitsPerFrame;
    OMX_U32 nSamplePerFrame;
    OMX_U32 nCouplingQuantBits;
    OMX_U32 nCouplingStartRegion;
    OMX_U32 nNumRegions;
    OMX_AUDIO_RAFORMATTYPE eFormat;
} OMX_AUDIO_PARAM_RATYPE;
```

4.1.13.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_RATYPE are defined as follows.

- nPortIndex: represents the port that this structure applies to.
- nChannels is the number of audio channels.

- `nSamplingRate` is the sampling rate of the source data.
- `nBitsPerFrame` is the value for bits per frame.
- `nSamplePerFrame` is the value for samples per frame.
- `nCouplingQuantBits` is the number of coupling quantization bits in the stream.
- `nCouplingStartRegion` is the coupling start region in the stream.
- `nNumRegions` is the number of regions value.
- `eFormat` is the audio format. Table 4-15 shows the possible RealAudio format types in `OMX_AUDIO_RAFORMATTYPE`. See https://community.helixcommunity.org/realcodecs/#RealAudio for further details.

Table 4-15: Supported RealAudio Format Types

Field Name	RA Format Descriptions
`OMX_AUDIO_RAFormatUnknown`	Unknown, unused or not required setting.
`OMX_AUDIO_RAFormatUnused`	Format unused or unknown
`OMX_AUDIO_RA8`	RealAudio 8 audio codec
`OMX_AUDIO_RA9`	RealAudio 9 audio codec
`OMX_AUDIO_RA10_AAC`	MPEG-4 AAC codec for bitrates of more than 128kbps
`OMX_AUDIO_RA10_CODEC`	RealAudio codec for bitrates less than 128 kbps
`OMX_AUDIO_RA10_LOSSLESS`	RealAudio Lossless
`OMX_AUDIO_RA10_MULTICHANNEL`	RealAudio Multichannel
`OMX_AUDIO_RA10_VOICE`	RealAudio Voice for bitrates below 15 kbps.

4.1.13.2 Functionality

The `OMX_AUDIO_PARAM_RATYPE` structure sets the parameters of the RealAudio codec.

4.1.14 OMX_AUDIO_PARAM_SBCTYPE

The Subband codec (SBC) is a mandatory audio codec for applications that support the Bluetooth™ Advance Audio Distribution Profile (A2DP). The A2DP codec algorithm is designed to obtain high quality audio at medium bit rates with a low computational complexity.

The `OMX_AUDIO_PARAM_SBCTYPE` structure is used to set or query the current settings for the codec component using the `OMX_GetParameter` function. It is also used to set the parameters of the codec component using the `OMX_SetParameter`

function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioSbc.

OMX_AUDIO_PARAM_SBCTYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_SBCTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_U32 nBitRate;
    OMX_U32 nSampleRate;
    OMX_U32 nBlocks;
    OMX_U32 nSubbands;
    OMX_U32 nBitPool;
    OMX_BOOL bEnableBitrate;
    OMX_AUDIO_CHANNELMODETYPE eChannelMode;
    OMX_AUDIO_SBCALLOCMETHODTYPE eSBCAllocType;
} OMX_AUDIO_PARAM_SBCTYPE;
```

4.1.14.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_SBCTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of audio channels.
- nBitRate is the bit rate of the encoded SBC audio. If the bit rate is variable or unknown, this parameter has the value 0.
- nSampleRate is the sample rate of the source data. If the sample rate is variable or unknown, this parameter has the value 0.
- nBlocks is the block length with which the stream has been encoded.
- nSubbands is the number of frequency subbands.
- nBitPool is the size of the bit allocation pool used for encoding the stream.
- bEnableBitrate is the Boolean value to use nBitRate or nBitPool.
- eChannelMode is the audio channel mode.
- eSBCAllocType is the enumeration of the adaptive bit allocation algorithm. Table 4-16 shows the field names and values.

Table 4-16: Adaptive Bit Allocation Algorithm Values

Field Name	Description
OMX_AUDIO_SBCAllocMethodLoudness	Loudness allocation method
OMX_AUDIO_SBCAllocMethodSNR	Signal-to-noise ratio (SNR) allocation method

4.1.14.2 Functionality

This OMX_AUDIO_PARAM_SBCTYPE structure configures the parameters of the SBC codec.

4.1.15 OMX_AUDIO_PARAM_ADPCMTYPE

Adaptive Differential PCM (ADPCM) is a waveform coding generic algorithm. It can be implemented in many ways and with different rates.

The OMX_AUDIO_PARAM_ADPCMTYPE structure is used to set or query the current settings for the ADPCM codec component using the OMX_GetParameter function. It is also used to set the parameters of the ADPCM codec component using the OMX_SetParameter function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioAdpcm.

OMX_AUDIO_PARAM_ADPCMTYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_ADPCMTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_U32 nBitsPerSample;
    OMX_U32 nSampleRate;
    OMX_U32 nBlockSize;
} OMX_AUDIO_PARAM_ADPCMTYPE;
```

4.1.15.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_ADPCMTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of channels of audio (mono, stereo).
- nBitsPerSample is the number of bits per sample of audio.
- nSampleRate is the sampling rate of the source data. Use the value 0 for variable or unknown sampling rate.
- nBlockSize is the ADPCM block coding size.

4.1.15.2 Functionality

The OMX_AUDIO_PARAM_ADPCMTYPE structure sets the parameters of a generic ADPCM codec.

4.1.16 OMX_AUDIO_PARAM_G723TYPE

ITU G.723.1 is a standard speech codec that has two rates, 5.3 and 6.3 kbps, and is used in video telephony. The input sampling rate is 8 kHz.

The OMX_AUDIO_PARAM_G723TYPE structure is used to set or query the current settings for the codec component using the OMX_GetParameter function. It is also used to set the parameters of the codec component using the OMX_SetParameter function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioG723.

OMX_AUDIO_PARAM_G723TYPE is defined as follows.

```
typedef struct OMX_AUDIO_PARAM_G723TYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_BOOL bDTX;
    OMX_AUDIO_G723RATE eBitRate;
    OMX_BOOL bHiPassFilter;
    OMX_BOOL bPostFilter;
} OMX_AUDIO_PARAM_G723TYPE;
```

4.1.16.1 Parameter Definitions

The parameters of OMX_AUDIO_PARAM_G723TYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of channels of audio (mono, stereo).
- bDTX enables Discontinuous Transmission according to Annex A of the standard.
- eBitRate is the bit rate of the encoded speech. Table 4-17 identifies bit rate values.

Table 4-17: G.723 Bit Rate Values

Field Name	Description
OMX_AUDIO_G723ModeUnused	Rate unused or unknown
OMX_AUDIO_G723ModeLow	5.3 kbps
OMX_AUDIO_G723ModeHigh	6.3 kbps

- bHiPassFilter enables high-pass filter preprocessing in the encoder.
- bPostFilter enables post filter processing.

4.1.16.2 Functionality

The OMX_AUDIO_PARAM_G723TYPE structure sets the parameters of the ITU-G.723.1 codec.

4.1.17 OMX_AUDIO_PARAM_G726TYPE

ITU G.726 is a standard ADPCM waveform codec having four rates. The rate of 32 kbps is the most used rate and identical to an older standard, ITU G.721. The input sampling rate is 8 kHz.

The OMX_AUDIO_PARAM_G726TYPE structure is used to set or query the current settings for the codec component using the OMX_GetParameter function. It is also used to set the parameters of the codec component using the OMX_SetParameter function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioG726.

OMX_AUDIO_PARAM_G726TYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_G726TYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
     OMX_AUDIO_G726MODE eG726Mode;
} OMX_AUDIO_PARAM_G726TYPE;
```

4.1.17.1 Parameter Definitions

The parameters of OMX_AUDIO_PARAM_G726TYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of channels of audio (mono, stereo).
- eG726Mode is the bit rate of the encoded speech. Table 4-18 identifies the bit rate values.

Table 4-18: G.726 Bit Rate Values

Field Name	Description
OMX_AUDIO_G726ModeUnused	Rate unused or unknown
OMX_AUDIO_G726Mode16	16 kbps
OMX_AUDIO_G726Mode24	24 kbps
OMX_AUDIO_G726Mode32	32 kbps (equals G.721)
OMX_AUDIO_G726Mode40	40 kbps

4.1.17.2 Functionality

The OMX_AUDIO_PARAM_G726TYPE structure sets the parameters of the ITU-G.726 codec.

4.1.18 OMX_AUDIO_PARAM_G729TYPE

ITU G.729 is a standard speech codec with a coding rate of 8 kbps that is used in various applications. The input sampling rate is 8 kHz. A bit-compatible, low-complexity version

is called G.729 appendix A (or G.729A). Support for DTX is described in annex B of the G.729 standard.

The OMX_AUDIO_PARAM_G729TYPE structure is used to set or query the current settings for the codec component using the OMX_GetParameter function. It is also used to set the parameters of the codec component using the OMX_SetParameter function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioG729.

OMX_AUDIO_PARAM_G729TYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_G729TYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_BOOL bDTX;
    OMX_AUDIO_G729TYPE eBitType;
} OMX_AUDIO_PARAM_G729TYPE;
```

4.1.18.1 Parameter Definitions

The parameters of OMX_AUDIO_PARAM_G729TYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of channels of audio (mono, stereo).
- bDTX enables Discontinuous Transmission when Annex B of the standard is used.
- eBitType identifies the standard annexes used. Table 4-19 identifies the standard annexes.

Table 4-19: Standard Annexes

Field Name	Description
OMX_AUDIO_G729	G.729 without annexes
OMX_AUDIO_G729A	G.729 with annex A
OMX_AUDIO_G729B	G.729 with annex B
OMX_AUDIO_G729AB	G.729 with annexes A and B

4.1.18.2 Functionality

The OMX_AUDIO_PARAM_G729TYPE structure sets the parameters of the ITU-G.729 codec.

4.1.19 OMX_AUDIO_PARAM_AMRTYPE

The Adaptive Multi-Rate coder is defined in 3GPP standards as having three main versions:

- Narrow Band (AMR-NB), where the sampling rate is 8 kHz. It is defined in standards 26.07x and 26.09x. This version is used in cellular phones and other wireless devices mainly for speech conversation.
- Wide Band (AMR-WB), where the sampling rate is 16 kHz. It is defined in standards 26.17x and 26.19x, and in ITU G.722.2. This version is used in cellular phones and other wireless devices mainly for streaming and voice-over-IP (VoIP) communication.
- Extended Wide Band (AMR-WB+). New features include extended audio bandwith, support for stereophonic audio, and the option to operate on and switch between several internal samping frequences (ISF). AMR-WB+ decoder can output signals at a variety of sampling rates (8kHz, 11.025kHz, 16kHz, 32kHz, 22.05kHz, 24kHz, 32kHz, 44.1kHz, 48kHz). Note that the desired PCM output sampling rate in case of AMR-WB+ decoder should be specified as the output port PCM `nSamplingRate` parameter. The AMR-WB+ encoder should be notified of the PCM input sampling rate through the input port PCM `nSamplingRate` parameter.AMR-WB+ is defined in standards 3GPP TS 26.290 and in RFC 4352. The expected key application for AMR-WB+ is streaming.

The `OMX_AUDIO_PARAM_AMRTYPE` structure is used to set or query the current settings for the codec component using the `OMX_GetParameter` function. It is also used to set the parameters of the codec component using the `OMX_SetParameter` function. When calling either the `OMX_GetParameter` or the `OMX_SetParameter` functions, the index specified for this structure is `OMX_IndexParamAudioAmr`.

`OMX_AUDIO_PARAM_AMRTYPE` is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_AMRTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_U32 nBitRate;
    OMX_AUDIO_AMRBANDMODETYPE eAMRBandMode;
    OMX_AUDIO_AMRDTXMODETYPE  eAMRDTXMode;
    OMX_AUDIO_AMRFRAMEFORMATTYPE eAMRFrameFormat;
    OMX_AUDIO_AMRISFINDEXTYPE eAMRISFIndex;
} OMX_AUDIO_PARAM_AMRTYPE;
```

4.1.19.1 Parameter Definitions

The parameters of `OMX_AUDIO_PARAM_AMRTYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `nChannels` is the number of channels of audio (mono, stereo).
- `nBitrate` is the bit rate of the encoded AMR audio. This parameter is a read only parameter used to query the current bitrate of the audio. If the bit rate is variable or unknown, this parameter has the value 0.

- `eAMRBandMode` is the bit rate of the encoded speech. Table 4-20 shows the bit rate values. Note that bit rate values for AMR-WB+ entries `OMX_AUDIO_AMRBandModeWBP16` - `OMX_AUDIO_AMRBandModeWBP47` assume internal sampling frequency (ISF) of 25600 Hz.

Table 4-20: Adaptive Multi-Rate Bit Rate Values

Field Name	Description
`OMX_AUDIO_AMRBandModeUnused`	Rate unused or unknown
`OMX_AUDIO_AMRBandModeNB0`	4.75 kbps
`OMX_AUDIO_AMRBandModeNB1`	5.15 kbps
`OMX_AUDIO_AMRBandModeNB2`	5.9 kbps
`OMX_AUDIO_AMRBandModeNB3`	6.7 kbps
`OMX_AUDIO_AMRBandModeNB4`	7.4 kbps
`OMX_AUDIO_AMRBandModeNB5`	7.95 kbps
`OMX_AUDIO_AMRBandModeNB6`	10.2 kbps
`OMX_AUDIO_AMRBandModeNB7`	12.2 kbps
`OMX_AUDIO_AMRBandModeWB0`	6.6 kbps
`OMX_AUDIO_AMRBandModeWB1`	8.85 kbps
`OMX_AUDIO_AMRBandModeWB2`	12.65 kbps
`OMX_AUDIO_AMRBandModeWB3`	14.25 kbps
`OMX_AUDIO_AMRBandModeWB4`	15.85 kbps
`OMX_AUDIO_AMRBandModeWB5`	18.25 kbps
`OMX_AUDIO_AMRBandModeWB6`	19.85 kbps
`OMX_AUDIO_AMRBandModeWB7`	23.05 kbps
`OMX_AUDIO_AMRBandModeWB8`	23.85 kbps
`OMX_AUDIO_AMRBandModeWBP0`	6.6 kbps
`OMX_AUDIO_AMRBandModeWBP1`	8.85 kbps
`OMX_AUDIO_AMRBandModeWBP2`	12.65 kbps
`OMX_AUDIO_AMRBandModeWBP3`	14.25 kbps
`OMX_AUDIO_AMRBandModeWBP4`	15.85 kbps
`OMX_AUDIO_AMRBandModeWBP5`	18.25 kbps
`OMX_AUDIO_AMRBandModeWBP6`	19.85 kbps
`OMX_AUDIO_AMRBandModeWBP7`	23.05 kbps
`OMX_AUDIO_AMRBandModeWBP8`	23.85 kbps
`OMX_AUDIO_AMRBandModeWBP9`	SID
`OMX_AUDIO_AMRBandModeWBP10`	13.6 kbps Mono
`OMX_AUDIO_AMRBandModeWBP11`	18 kbps Stereo
`OMX_AUDIO_AMRBandModeWBP12`	24 kbps Mono
`OMX_AUDIO_AMRBandModeWBP13`	24 kbps Stereo

Field Name	Description
OMX_AUDIO_AMRBandModeWBP14	FRAME_ERASURE
OMX_AUDIO_AMRBandModeWBP15	NO_DATA
OMX_AUDIO_AMRBandModeWBP16	10.4 kbps
OMX_AUDIO_AMRBandModeWBP17	12 kbps
OMX_AUDIO_AMRBandModeWBP18	13.6 kbps
OMX_AUDIO_AMRBandModeWBP19	15.2 kbps
OMX_AUDIO_AMRBandModeWBP20	16.8 kbps
OMX_AUDIO_AMRBandModeWBP21	19.2 kbps
OMX_AUDIO_AMRBandModeWBP22	20.8 kbps
OMX_AUDIO_AMRBandModeWBP23	24 kbps
OMX_AUDIO_AMRBandModeWBP24	12.4 kbps
OMX_AUDIO_AMRBandModeWBP25	12.8 kbps
OMX_AUDIO_AMRBandModeWBP26	14 kbps
OMX_AUDIO_AMRBandModeWBP27	14.4 kbps
OMX_AUDIO_AMRBandModeWBP28	15.2 kbps
OMX_AUDIO_AMRBandModeWBP29	16 kbps
OMX_AUDIO_AMRBandModeWBP30	16.4 kbps
OMX_AUDIO_AMRBandModeWBP31	17.2 kbps
OMX_AUDIO_AMRBandModeWBP32	18 kbps
OMX_AUDIO_AMRBandModeWBP33	18.4 kbps
OMX_AUDIO_AMRBandModeWBP34	19.2 kbps
OMX_AUDIO_AMRBandModeWBP35	20 kbps
OMX_AUDIO_AMRBandModeWBP36	20.4 kbps
OMX_AUDIO_AMRBandModeWBP37	21.2 kbps
OMX_AUDIO_AMRBandModeWBP38	22.4 kbps
OMX_AUDIO_AMRBandModeWBP39	23.2 kbps
OMX_AUDIO_AMRBandModeWBP40	24 kbps
OMX_AUDIO_AMRBandModeWBP41	25.6 kbps
OMX_AUDIO_AMRBandModeWBP42	26 kbps
OMX_AUDIO_AMRBandModeWBP43	26.8 kbps
OMX_AUDIO_AMRBandModeWBP44	28.8 kbps
OMX_AUDIO_AMRBandModeWBP45	29.6 kbps
OMX_AUDIO_AMRBandModeWBP46	30 kbps
OMX_AUDIO_AMRBandModeWBP47	32 kbps
OMX_AUDIO_AMRBandModeAuto	Allow encoder to select mode

- eAMRDTXMode identifies the AMR Discontinuous Transmission mode and voice activity detection (VAD) type. Table 4-21 describes the modes and types.

Table 4-21: Adaptive Multi-Rate Discontinuous Transmission Mode and VAD Type

Field Name	Description
OMX_AUDIO_AMRDTXModeOff	DTX not used
OMX_AUDIO_AMRDTXModeOnVAD1	Use Type 1 VAD
OMX_AUDIO_AMRDTXModeOnVAD2	Use Type 2 VAD
OMX_AUDIO_AMRDTXModeOnAuto	VAD type automatic
OMX_AUDIO_AMRDTXasEFR	DTX frames as EFR (3GPP 26.101, frame type equals 8,9,10)

- eAMRFrameFormat identifies the encoded frame format. Table 4-22 shows the frame formats.

Table 4-22: Encoded Frame Format

Field Name	Description
OMX_AUDIO_AMRFrameFormatConformance	Standard test-sequence format (3GPP 26.074)
OMX_AUDIO_AMRFrameFormatIF1	Interface format 1 (NB- 3GPP 26.101, sec. 4 WB- 3GPP 26.201, sec. 4)
OMX_AUDIO_AMRFrameFormatIF2	Interface format 2 (NB- 3GPP 26.101, annex A WB- 3GPP 26.201, annex A)
OMX_AUDIO_AMRFrameFormatFSF	File Storage format (RFC 4867, sec. 5)
OMX_AUDIO_AMRFrameFormatRTPPayloadFull	RTP payload format (RFC 4867, sec. 4)
OMX_AUDIO_AMRFrameFormatITU	ITU frame format
OMX_AUDIO_AMRFrameFormatRTPPayloadConstrained	RTP payload format (RFC 4867, sec. 4.4)
OMX_AUDIO_AMRFrameFormatWBPlusTIF	AMR-WB+ Transport Interface Format (3GPP TS 26.290, sec. 8.2)
OMX_AUDIO_AMRFrameFormatWBPlusFSF	AMR-WB+ File Storage Format (3GPP TS 26.290, sec. 8.3)
OMX_AUDIO_AMRFrameFormatWBPlusRTPPayloadBasic	RTP payload basic format (RFC 4352, sec. 4.2 and 4.3.2.1)

Field Name	Description
OMX_AUDIO_AMRFrameFormatWBPlusRTPPayloadInterleaved	RTP payload intearleaved format (RFC 4352, sec. 4.2 and 4.3.2.2)

OMX_AUDIO_AMRFrameFormatRTPPayloadFull format shall be used to specify AMR RTP payloads that do not satisfy one or more of the constraints that apply to OMX_AUDIO_AMRFrameFormatRTPPayloadConstrained format specified below.

OMX_AUDIO_AMRFrameFormatRTPPayloadConstrained format is reserved for the most commonly used AMR RTP payload format that satisfies all of the following constraints:

- RTP payload data is single-channel
- RTP payload data is in octet-aligned mode
- RTP payload data is not robust-sort-ordered
- RTP payload data is non-interleaved
- RTP payload data does not include frame CRCs

Furthermore, in case of OMX_AUDIO_AMRFrameFormatRTPPayloadConstrained format, the payload of a compressed data buffer delivered to the AMR decoder component shall omit the RTP header and payload header, and shall consist of the payload table of contents (TOC) followed by the speech data in normal order, as described in sections 4.4.2, 4.4.3 and 4.4.4 of the RFC 4867 document.

OMX_AUDIO_AMRFrameFormatWBPlusRTPPayloadBasic and OMX_AUDIO_AMRFrameFormatWBPlusRTPPayloadInterleaved format - the payload of a compressed data buffer delivered to the AMR decoder component shall contain the payload header, table of contents (TOC) followed by audio data as described in sections 4.2. and 4.3 or RFC 4352 document.

- eAMRISFIndex identifies the internal sampling frequency (ISF) index for AMR-WB+ given in 3GPP TS 26.290 document. In case of encoder, when the decision about the value of ISF is left to the encoder component, the parameter is to be set to OMX_AUDIO_AMRISFIndexAuto. If the ISF is unknown or variable, the parameter has the value OMX_AUDIO_AMRISFIndexUnknown. This parameter shall be ignored for AMR-NB and AMR-WB audio. Table 4-23 describes the values of the parameter.

Table 4-23: Adaptive Multi-Rate Extended Wide Band Internal Sampling Frequency Index

Field Name	Description
OMX_AUDIO_AMRISFIndex0	N/A

Field Name	Description
OMX_AUDIO_AMRISFIndex1	12800 Hz
OMX_AUDIO_AMRISFIndex2	14400 Hz
OMX_AUDIO_AMRISFIndex3	16000 Hz
OMX_AUDIO_AMRISFIndex4	17067 Hz
OMX_AUDIO_AMRISFIndex5	19200 Hz
OMX_AUDIO_AMRISFIndex6	21333 Hz
OMX_AUDIO_AMRISFIndex7	24000 Hz
OMX_AUDIO_AMRISFIndex8	25600 Hz
OMX_AUDIO_AMRISFIndex9	28800 Hz
OMX_AUDIO_AMRISFIndex10	32000 Hz
OMX_AUDIO_AMRISFIndex11	34133 Hz
OMX_AUDIO_AMRISFIndex12	36000 Hz
OMX_AUDIO_AMRISFIndex13	38400 Hz
OMX_AUDIO_AMRISFIndexAuto	Allow encoder to select ISF
OMX_AUDIO_AMRISFIndexUknown	N/A

4.1.19.2 Functionality

The OMX_AUDIO_PARAM_AMRTYPE structure sets the parameters of the AMR codec.

4.1.20 OMX_AUDIO_PARAM_GSMFRTYPE

The GSM Full-Rate codec is defined in ETSI standards 06.1x and 06.3x, which became 3GPP standards 26.01x and 26.03x.

The GSM Full-Rate coder is used in legacy GSM cellular phones. The sampling rate is 8 kHz. The encoded speech has a rate of 13 kbps, or 260 bits per frame of 20 milliseconds. The coding algorithm is RPE-LTP.

The OMX_AUDIO_PARAM_GSMFRTYPE structure is used to set or query the current settings for the codec component using the OMX_GetParameter function. It is also used to set the parameters of the codec component using the OMX_SetParameter function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioGsm_FR.

OMX_AUDIO_PARAM_GSMFRTYPE is defined as follows.

```
typedef struct OMX_AUDIO_PARAM_GSMFRTYPE {
   OMX_U32 nSize;
   OMX_VERSIONTYPE nVersion;
   OMX_U32 nPortIndex;
   OMX_BOOL bDTX;
   OMX_BOOL bHiPassFilter ;
```

```
} OMX_AUDIO_PARAM_GSMFRTYPE;
```

4.1.20.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_GSMFRTYPE as defined as follows.

- nPortIndex represents the port that this structure applies to.
- bDTX enables Discontinuous Transmission (3GPP 46.031, 46.032).
- bHiPassFilter enables high-pass filter processing

4.1.20.2 Functionality

The OMX_AUDIO_PARAM_GSMFRTYPE structure sets the parameters of the GSM Full-Rate codec.

4.1.21 OMX_AUDIO_PARAM_GSMEFRTYPE

The GSM Enhanced Full-Rate codec is defined in ETSI standards 06.5x, 06.6x, and 06.8x; these standards became 3GPP standards 26.05x, 26.06x, and 26,08x.

The GSM Enhanced Full-Rate codec is used in GSM cellular phones. The sampling rate is 8 kHz. The encoded speech has a rate of 12.2 kbps, or 244 bits per frame of 20 milliseconds. Each coded frame is augmented by 16 error-protection bits that provide the complement of 260 bits, which is the same as the Full Rate codec. However this augmentation is performed outside of the speech coder. The coding algorithm is ACELP.

The OMX_AUDIO_PARAM_GSMEFRTYPE structure is used to set or query the current settings for the codec component using the OMX_GetParameter function. It is also used to set the parameters of the codec component using the OMX_SetParameter function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioGsm_EFR.

OMX_AUDIO_PARAM_GSMEFRTYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_GSMEFRTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bDTX;
    OMX_BOOL bHiPassFilter;
} OMX_AUDIO_PARAM_GSMEFRTYPE;
```

4.1.21.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_GSMEFRTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bDTX enables Discontinuous Transmission (3GPP 46.041, 46.042).
- bHiPassFilter enables High-Pass filter preprocessing in the encoder.

4.1.21.2 Functionality

The OMX_AUDIO_PARAM_GSMEFRTYPE structure sets the parameters of the GSM Enhanced Full-Rate codec.

4.1.22 OMX_AUDIO_PARAM_GSMHRTYPE

The GSM Half-Rate codec is defined in ETSI standards 06.2x and 06.4x; these standards became 3GPP standards 26.02x and 26.04x.

The GSM Half-Rate codec is used in GSM cellular phones. The sampling rate is 8 kHz. The encoded speech has a rate of 5.6 kbps, or 112 bits per frame of 20 milliseconds. The coding algorithm is VSELP.

The OMX_AUDIO_PARAM_GSMHRTYPE structure is used to set or query the current settings for the codec component using the OMX_GetParameter function. It is also used to set the parameters of the codec component using the OMX_SetParameter function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioGsm_HR.

OMX_AUDIO_PARAM_GSMHRTYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_GSMHRTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bDTX;
    OMX_BOOL bHiPassFilter;
} OMX_AUDIO_PARAM_GSMHRTYPE;
```

4.1.22.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_GSMHRTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bDTX enables Discontinuous Transmission (3GPP 46.041, 46.042).
- bHiPassFilter enables High-Pass filter preprocessing in the encoder.

4.1.22.2 Functionality

The OMX_AUDIO_PARAM_GSMHRTYPE structure sets the parameters of the GSM Half-Rate codec.

4.1.23 OMX_AUDIO_PARAM_TDMAFRTYPE

The TDMA Full-Rate codec is defined in theTIA/EIA-136-420 American cellular standard, also referred to as IS-136. It is a legacy codec used in the American cellular standard known as DAMPS.

The sampling rate is 8 kHz. The encoded speech has a rate of 7.95 kbps, or 159 bits per frame of 20 milliseconds. The coding algorithm is VSELP.

The OMX_AUDIO_PARAM_TDMAFRTYPE structure is used to set or query the current settings for the codec component using the OMX_GetParameter function. It is also used to set the parameters of the codec component using the OMX_SetParameter function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioTdma_FR.

OMX_AUDIO_PARAM_TDMAFRTYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_TDMAFRTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_BOOL bDTX;
    OMX_BOOL bHiPassFilter;
} OMX_AUDIO_PARAM_TDMAFRTYPE;
```

4.1.23.1 Parameter Definitions

The parameters of OMX_AUDIO_PARAM_TDMAFRTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of audio channels.
- bDTX enables Discontinuous Transmission.
- bHiPassFilter enables High-Pass filter preprocessing in the encoder.

4.1.23.2 Functionality

The OMX_AUDIO_PARAM_TDMAFRTYPE structure sets the parameters of the TDMA Full-Rate codec.

4.1.24 OMX_AUDIO_PARAM_TDMAEFRTYPE

The TDMA Enhanced Full-Rate codec is defined in the TIA/EIA-136-410 American cellular standard, which is also referred to as IS-641, DAMPS-EFR. It is the codec used in the American cellular standard known as DAMPS.

The sampling rate is 8 kHz. The encoded speech has a rate of 7.4 kbps, or 148 bits per frame of 20 milliseconds. The coding algorithm is ACELP.

The OMX_AUDIO_PARAM_TDMAEFRTYPE structure is used to set or query the current settings for the codec component using the OMX_GetParameter function. It is also used to set the parameters of the codec component using the OMX_SetParameter function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioTdma_EFR.

OMX_AUDIO_PARAM_TDMAEFRTYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_TDMAEFRTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_BOOL bDTX;
    OMX_BOOL bHiPassFilter;
} OMX_AUDIO_PARAM_TDMAEFRTYPE;
```

4.1.24.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_TDMAEFRTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of audio channels.
- bDTX enables Discontinuous Transmission.
- bHiPassFilter enables High-Pass filter preprocessing in the encoder.

4.1.24.2 Functionality

The OMX_AUDIO_PARAM_TDMAEFRTYPE structure sets the parameters of the TDMA Enhanced Full-Rate codec.

4.1.25 OMX_AUDIO_PARAM_PDCFRTYPE

The PDC Full-Rate codec is defined in ARIB standard RCR-27B. It is the legacy codec used in the Japanese cellular system.

The sampling rate is 8 kHz. The encoded speech has a rate of 6.7 kbps, or 134 bits per frame of 20 milliseconds. The coding algorithm is VSELP.

The OMX_AUDIO_PARAM_PDCFRTYPE structure is used to set or query the current settings for the codec component using the OMX_GetParameter function. It is also used to set the parameters of the codec component using the OMX_SetParameter function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioPdc_FR.

OMX_AUDIO_PARAM_PDCFRTYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_PDCFRTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_BOOL bDTX;
    OMX_BOOL bHiPassFilter;
} OMX_AUDIO_PARAM_PDCFRTYPE;
```

4.1.25.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_PDCFRTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of audio channels.
- bDTX enables Discontinuous Transmission.
- bHiPassFilter enables High-Pass filter preprocessing in the encoder.

4.1.25.2 Functionality

The OMX_AUDIO_PARAM_PDCFRTYPE structure sets the parameters of the PDC Full-Rate codec.

4.1.26 OMX_AUDIO_PARAM_PDCEFRTYPE

The PDC Full-Rate codec is defined in ARIB standard RCR-27H. The codec is used in the Japanese cellular system.

The sampling rate is 8 kHz. The encoded speech has a rate of 6.7 kbps, or 134 bits per frame of 20 milliseconds. The coding algorithm is ACELP.

The OMX_AUDIO_PARAM_PDCEFRTYPE structure is used to set or query the current settings for the codec component using the OMX_GetParameter function. It is also used to set the parameters of the codec component using the OMX_SetParameter function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioPdc_EFR.

OMX_AUDIO_PARAM_PDCEFRTYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_PDCEFRTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_BOOL bDTX;
    OMX_BOOL bHiPassFilter;
} OMX_AUDIO_PARAM_PDCEFRTYPE;
```

4.1.26.1 Parameter Definitions

The parameters of OMX_AUDIO_PARAM_PDCEFRTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of audio channels.
- bDTX enables Discontinuous Transmission.
- bHiPassFilter enables High-Pass filter preprocessing in the encoder.

4.1.26.2 Functionality

The OMX_AUDIO_PARAM_PDCEFRTYPE structure sets the parameters of the PDC Enhanced Full-Rate codec.

4.1.27 OMX_AUDIO_PARAM_PDCHRTYPE

The PDC Full-Rate codec is defined in ARIB standard RCR-27C. The codec is used in the Japanese cellular system.

The sampling rate is 8 kHz. The encoded speech has a rate of 3.45 kbps, or 138 bits per frame of 40 milliseconds. The coding algorithm is PSI-CELP.

The OMX_AUDIO_PARAM_PDCHRTYPE structure is used to set or query the current settings for the codec component using the OMX_GetParameter function. It is also used to set the parameters of the codec component using the OMX_SetParameter function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioPdc_HR.

OMX_AUDIO_PARAM_PDCHRTYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_PDCHRTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_BOOL bDTX;
    OMX_BOOL bHiPassFilter;
} OMX_AUDIO_PARAM_PDCHRTYPE;
```

4.1.27.1 Parameter Definitions

The parameters of OMX_AUDIO_PARAM_PDCHRTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of audio channels.
- bDTX enables Discontinuous Transmission.
- bHiPassFilter enables High-Pass filter preprocessing in the encoder.

4.1.27.2 Functionality

The OMX_AUDIO_PARAM_PDCHRTYPE structure sets the parameters of the PDC Full-Rate codec.

4.1.28 OMX_AUDIO_PARAM_QCELP8TYPE

The QCELP (lower rate) variable rate codec is defined in the TIA/EIA-96 standard. It is the legacy codec used in the CDMA cellular standard, mainly in Korea and North America.

The sampling rate is 8 kHz. The encoded speech has a maximal rate called Rate 1 of 8 kbps, or 160 bits per frame of 20 milliseconds. The codec can work on lower rates, namely Rates 1/2, 1/4, and 1/8, depending on the speech activity and channel capacity. Rate 1 adds 11 parity bits per frame, so its rate becomes 8.55 kbps.

The coding algorithm is QCELP.

The OMX_AUDIO_PARAM_QCELP8TYPE structure is used to set or query the current settings for the codec component using the OMX_GetParameter function. It is also used to set the parameters of the codec component using the OMX_SetParameter function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioQcelp8.

OMX_AUDIO_PARAM_QCELP8TYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_QCELP8TYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_U32 nBitRate;
    OMX_AUDIO_CDMARATETYPE eCDMARate;
    OMX_U32 nMinBitRate;
    OMX_U32 nMaxBitRate;
} OMX_AUDIO_PARAM_QCELP8TYPE;
```

4.1.28.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_QCELP8TYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of audio channels.
- nBitRate is the bit rate of the audio stream. If the bit rate is unknown, this parameter has the value 0.
- eCDMARate is the frame rate or type. Table 4-24 shows the frame rate values.

Table 4-24: CDMA Frame Rate Values

Field Name	Description
OMX_AUDIO_CDMARateBlank	Blank frame
OMX_AUDIO_CDMARateFull	Rate 1
OMX_AUDIO_CDMARateHalf	Rate ½
OMX_AUDIO_CDMARateQuarter	Rate ¼
OMX_AUDIO_CDMARateEighth	Rate 1/8
OMX_AUDIO_CDMARateErasure	Erasure frame (due to channel errors)

- nMinBitRate is the minimal restriction on the encoder for the current frame. The value is 1, 2, 3, or 4. The default value is 1.

- nMaxBitRate is the maximal restriction on the encoder for the current frame. The value is 1, 2, 3, or 4. This value shall be greater than or equal to the minimal rate. The default value is 4.

4.1.28.2 Functionality

The OMX_AUDIO_PARAM_QCELP8TYPE structure sets the parameters of the QCELP8 codec.

4.1.29 OMX_AUDIO_PARAM_QCELP13TYPE

The QCELP (high-rate) variable rate codec is defined in the TIA/EIA-733 standard. It is the codec that is used in the high-rate service option of CDMA cellular standard, mainly in Korea and North America.

The sampling rate is 8 kHz. The encoded speech has a maximal rate called Rate 1 of 13.3 kbps, or 266 bits per frame of 20 milliseconds. The codec can work on lower rates, namely Rates 1/2, 1/4, and 1/8, depending on the capacity of the speech activity channel.

The coding algorithm is QCELP.

The OMX_AUDIO_PARAM_QCELP13TYPE structure is used to set or query the current settings for the codec component using the OMX_GetParameter function. It is also used to set the parameters of the codec component using the OMX_SetParameter function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioQcelp13.

OMX_AUDIO_PARAM_QCELP13TYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_QCELP13TYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_AUDIO_CDMARATETYPE eCDMARate;
    OMX_U32 nMinBitRate;
    OMX_U32 nMaxBitRate;
} OMX_AUDIO_PARAM_QCELP13TYPE;
```

4.1.29.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_QCELP13TYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of audio channels.
- eCDMARate is the frame rate or type. Table 4-24 in section 4.1.28.1 shows the frame rate values.
- nMinBitRate is the minimal restriction on the encoder for the current frame. The value is 1, 2, 3, or 4. The default value is 1.

- nMaxBitRate is the maximal restriction on the encoder for the current frame. The value is 1, 2, 3, or 4. The value shall be greater than or equal to the minimal rate. The default value is 4.

4.1.29.2 Functionality

The OMX_AUDIO_PARAM_QCELP13TYPE structure sets the parameters of the QCELP13 codec.

4.1.30 OMX_AUDIO_PARAM_EVRCTYPE

The Enhanced Variable Speech Coder is defined in the TIA/EIA-127 standard. It is the codec used in the CDMA cellular standard, mainly in Korea and North America.

The sampling rate is 8 kHz. The encoded speech has a maximal rate, called Rate 1, of 8.55 kbps, or 171 bits per frame of 20 milliseconds. The codec can work on lower rates, namely Rate 1/2 and 1/8, depending on the speech activity and the channel capacity.

The coding algorithm is RCELP.

The OMX_AUDIO_PARAM_EVRCTYPE structure is used to set or query the current settings for the codec component using the OMX_GetParameter function. It is also used to set the parameters of the codec component using the OMX_SetParameter function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioEvrc.

OMX_AUDIO_PARAM_EVRCTYPE is defined as follows.

```
 typedef struct OMX_AUDIO_PARAM_EVRCTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_AUDIO_CDMARATETYPE eCDMARate;
    OMX_BOOL bRATE_REDUCon;
    OMX_U32 nMinBitRate;
    OMX_U32 nMaxBitRate;
    OMX_BOOL bHiPassFilter;
    OMX_BOOL bNoiseSuppressor;
    OMX_BOOL bPostFilter;
} OMX_AUDIO_PARAM_EVRCTYPE;
```

4.1.30.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_EVRCTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of audio channels.
- eCDMARate is the frame rate or type. Table 4-24 in section 4.1.28.1 shows the frame rate values.

- bRATE_REDUCon specifies if rate reduction is required
- nMinBitRate is the minimal restriction on the encoder for the current frame. The value is 1, 3, or 4. The default value is 1.
- nMaxBitRate is the maximal restriction on the encoder for the current frame. The value is 1, 3, or 4. The value shall be greater than or equal to the minimal rate. The default value is 4.
- bHiPassFilter enables high-pass filter processing.
- bNoiseSuppressor enables the encoder's noise suppressor preprocessing as a part of the encoder.
- bPostFilter enables post filter processing.

4.1.30.2 Functionality

The OMX_AUDIO_PARAM_EVRCTYPE structure sets the parameters of the Enhanced Variable Speech Coder (EVRC) speech codec.

4.1.31 OMX_AUDIO_PARAM_SMVTYPE

The Selectable Mode Vocoder (SMV) is defined in 3GPP2 standard C.S0030-2. It is the codec used in the CDMA2000 cellular standard.

The sampling rate is 8 kHz. The encoded speech has a maximal rate, called Rate 1, of 8.55 kbps, or 171 bits per frame of 20 milliseconds. It can work on lower rates, namely Rates 1/2, 1/4, and 1/8, depending on the speech activity and the channel capacity.

The coding algorithm is eX-CELP.

The OMX_AUDIO_PARAM_SMVTYPE structure is used to set or query the current settings for the codec component using the OMX_GetParameter function. It is also used to set the parameters of the codec component using the OMX_SetParameter function. When calling either the OMX_GetParameter or the OMX_SetParameter functions, the index specified for this structure is OMX_IndexParamAudioSmv.

OMX_AUDIO_PARAM_SMVTYPE is defined as follows.

```
typedef struct OMX_AUDIO_PARAM_SMVTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_AUDIO_CDMARATETYPE eCDMARate;
    OMX_BOOL bRATE_REDUCon;
    OMX_U32 nMinBitRate;
    OMX_U32 nMaxBitRate;
    OMX_BOOL bHiPassFilter;
    OMX_BOOL bNoiseSuppressor;
    OMX_BOOL bPostFilter;
} OMX_AUDIO_PARAM_SMVTYPE;
```

4.1.31.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_SMVTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of audio channels.
- eCDMARate is the frame rate or type. Table 4-24 in section 4.1.28.1 identifies the frame rate values.

Table 4-25: Selectable Mode Vocoder Frame Rate Values

Field Name	Description
OMX_AUDIO_CDMARateBlank	Blank frame
OMX_AUDIO_CDMARateFull	Rate 1
OMX_AUDIO_CDMARateHalf	Rate ½
OMX_AUDIO_CDMARateEighth	Rate 1/8
OMX_AUDIO_CDMARateErasure	Erasure frame (due to channel errors)

- bRATE_REDUCon specifies if rate reduction is required
- nMinBitRate is the minimal restriction on the encoder for the current frame. The value is 1, 3, or 4. The default value is 1.
- nMaxBitRate is the maximal restriction on the encoder for current frame. The value is 1, 3, or 4. The value shall be greater than or equal to the minimal rate. The default value is 4.
- bHiPassFilter enables high-pass filter processing.
- bNoiseSuppressor enables the encoder's noise suppressor preprocessing as a part of the encoder.
- bPostFilter enables post filter processing.

4.1.31.2 Functionality

The OMX_AUDIO_PARAM_SMVTYPE structure sets the parameters of the Selectable Mode Vocoder codec.

4.1.32 OMX_AUDIO_PARAM_MIDITYPE

The OMX_AUDIO_PARAM_MIDITYPE structure is used to set or query the initial basic parameters of the MIDI engine. The parameters define the number of output channels of PCM audio, the maximum polyphony that the device supports, and whether the default soundbank is loaded at initialization.

OMX_AUDIO_PARAM_MIDITYPE is defined as follows.

```
typedef struct OMX_AUDIO_PARAM_MIDITYPE {
    OMX_U32 nSize;
```

```
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nFileSize;
    OMX_BU32 sMaxPolyphony;
    OMX_BOOL bLoadDefaultSound;
    OMX_AUDIO_MIDIFORMATTYPE eMidiFormat;
} OMX_AUDIO_PARAM_MIDITYPE;
```

4.1.32.1 Parameter Definitions

The parameters for `OMX_AUDIO_PARAM_MIDITYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `nFileSize` is the size of the MIDI file data in bytes. This field shall be specified by the IL client or the component configuring this port before data is accepted.
- `sMaxPolyphony` specifies the range of simultaneous polyphonic voices that are supported. Since this parameter is of type `OMX_BU32` (a bounded, unsigned 32-bit integer), it allows the querying and setting of minimum, nominal, and maximum values. A value of zero indicates that the default polyphony of the device is used.
- `bLoadDefaultSound` is a Boolean value that indicates whether the default soundbank is it to be loaded at initialization.
- `eMidiFormat` is an enumeration for the format of the MIDI file. Table 4-26 shows the MIDI file format.

Table 4-26: MIDI File Format

Field Name	Description
`OMX_AUDIO_MIDIFormatUnknown`	MIDI format is unknown, not used or not required.
`OMX_AUDIO_MIDIFormatSMF0`	Standard MIDI File format 0
`OMX_AUDIO_MIDIFormatSMF1`	Standard MIDI File format 1
`OMX_AUDIO_MIDIFormatSMF2`	Standard MIDI File format 2
`OMX_AUDIO_MIDIFormatSPMIDI`	SP-MIDI
`OMX_AUDIO_MIDIFormatXMF0`	XMF type 0
`OMX_AUDIO_MIDIFormatXMF1`	XMF type 1
`OMX_AUDIO_MIDIFormatMobileXMF`	Mobile XMF (XMF type 2)

4.1.33 OMX_AUDIO_PARAM_MIDILOADUSERSOUNDTYPE

The `OMX_AUDIO_PARAM_MIDILOADUSERSOUNDTYPE` structure is used to set or query the parameters required for loading and unloading user-specified MIDI downloadable soundbanks (DLS). This structure contains a major exception to the

memory rules used in OpenMAX IL: It includes a pointer to data, namely the DLS, which is outside the structure. This is because DLS soundbanks can grow to upwards of 400 kB in some cases. Without this exception, the implementations would be forced to make redundant copies of these large soundbanks, wasting valuable system resources.

OMX_AUDIO_PARAM_MIDILOADUSERSOUNDTYPE is defined as follows.

```
typedef struct OMX_AUDIO_PARAM_MIDILOADUSERSOUNDTYPE {
  OMX_U32 nSize;
  OMX_VERSIONTYPE nVersion;
  OMX_U32 nPortIndex;
  OMX_U32 nDLSIndex;
  OMX_U32 nDLSSize;
  OMX_PTR pDLSData;
  OMX_AUDIO_MIDISOUNDBANKTYPE eMidiSoundBank;
OMX_AUDIO_MIDISOUNDBANKLAYOUTTYPE eMidiSoundBankLayout;
} OMX_AUDIO_PARAM_MIDILOADUSERSOUNDTYPE;
```

4.1.33.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_MIDILOADUSERSOUNDTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nDLSIndex is the DLS file index to be loaded.
- nDLSSize is the size of the DLS in bytes.
- pDLSData is the pointer to the DLS file data.
- eMidiSoundBank is an enumeration for the various types of MIDI DLS soundbanks. Table 4-27 identifies the MIDI soundbanks.

Table 4-27: MIDI Soundbanks

Field Name	Description
OMX_AUDIO_MIDISoundBankUnused	Unused/unknown soundbank type
OMX_AUDIO_MIDISoundBankDLS1	DLS 1
OMX_AUDIO_MIDISoundBankDLS2	DLS 2
OMX_AUDIO_MIDISoundBankMobileDLSBase	Mobile DLS, using the base functionality
OMX_AUDIO_MIDISoundBankMobile DLSPlusOptions	Mobile DLS, using the specification-defined optional feature set

- eMidiSoundBankLayout is an enumeration for the various layouts of MIDI DLS soundbanks. Bank layout describes how the bank most significant bit (MSB) and least significant bit (LSB) are used in the DLS instrument definitions soundbank Table 4-28 shows the MIDI soundbank layouts.

Table 4-28: MIDI Soundbank Layouts

Field Name	Description
OMX_AUDIO_MIDISoundBankLayoutUnused	Unknown/unused soundbank layout type.

Field Name	Description
OMX_AUDIO_MIDISoundBankLayoutGM	GS layout based on bank MSB 0x00.
OMX_AUDIO_MIDISoundBankLayoutGM2	General MIDI 2 layout using MSB 0x78/0x79, LSB 0x00.
OMX_AUDIO_MIDISoundBankLayoutUser	Does not conform to any bank numbering standards.

4.1.34 OMX_AUDIO_CONFIG_MIDIIMMEDIATEEVENTTYPE

The OMX_AUDIO_CONFIG_MIDIIMMEDIATEEVENTTYPE structure is used to set the parameters for live MIDI events and Maximum Instantaneous Polyphony (MIP) messages, which are part of the SP-MIDI standard. The OMX_AUDIO_CONFIG_MIDIIMMEDIATEEVENTTYPE structure does not specify the format of MIDI events or MIP messages; it simply provides an array for the MIDI events or the MIP message buffer. The MIDI engine can parse this array and process it appropriately.

OMX_AUDIO_CONFIG_MIDIIMMEDIATEEVENTTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_MIDIIMMEDIATEEVENTTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nMidiEventSize;
    OMX_U8 nMidiEvents[1];
} OMX_AUDIO_CONFIG_MIDIIMMEDIATEEVENTTYPE;
```

4.1.34.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_MIDIIMMEDIATEEVENTTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nMidiEventSize is the size of the immediate MIDI events or MIP message in bytes.
- nMidiEvents is the MIDI event array to be rendered immediately, or an array for the MIP message buffer, where the size is indicated by nMidiEventSize.

4.1.34.2 Post-processing Conditions

The live MIDI event array is rendered by the MIDI engine, or the MIP message contained in the buffer is processed.

4.1.35 OMX_AUDIO_CONFIG_MIDISOUNDBANKPROGRAMTYPE

The OMX_AUDIO_CONFIG_MIDISOUNDBANKPROGRAMTYPE structure is used to query and set the parameters for soundbank/program pairs in a given MIDI channel. It will be called once for each of the 16 MIDI channels. Note that the entire MIDI stream

goes to a single port. One-to-one mapping does not occur between ports and MIDI channels.

OMX_AUDIO_CONFIG_MIDISOUNDBANKPROGRAMTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_MIDISOUNDBANKPROGRAMTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannel;
    OMX_U16 nIDProgram;
    OMX_U16 nIDSoundBank;
    OMX_U32 nUserSoundBankIndex;
} OMX_AUDIO_CONFIG_MIDISOUNDBANKPROGRAMTYPE;
```

4.1.35.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_MIDISOUNDBANKPROGRAMTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannel refers to a MIDI channel. Valid channel values are 1 to 16.
- nIDProgram refers to a MIDI program. Valid program ID range is 1 to 128.
- nIDSoundBank is the soundbank ID.
- nUserSoundBankIndex is the user soundbank index. The index makes access to soundbanks easier if multiple banks are present.

4.1.35.2 Post-processing Conditions

The specified MIDI channel has a soundbank and program associated with it.

4.1.36 OMX_AUDIO_CONFIG_MIDICONTROLTYPE

The OMX_AUDIO_CONFIG_MIDICONTROLTYPE structure is used to query and set the parameters for controlling the rate and the looping (repeated playback) of MIDI playback.

OMX_AUDIO_CONFIG_MIDICONTROLTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_MIDICONTROLTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BS32 sPitchTransposition;
    OMX_BU32 sPlayBackRate;
    OMX_BU32 sTempo ;
    OMX_U32 nMaxPolyphony;
    OMX_U32 nNumRepeat;
    OMX_U32 nStopTime;
    OMX_U16 nChannelMuteMask;
    OMX_U16 nChannelSoloMask;
    OMX_U32 nTrack0031MuteMask;
```

```
    OMX_U32 nTrack3263MuteMask;
    OMX_U32 nTrack0031SoloMask;
    OMX_U32 nTrack3263SoloMask;
} OMX_AUDIO_CONFIG_MIDICONTROLTYPE;
```

4.1.36.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_MIDICONTROLTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- sPitchTransposition is the pitch transposition in semitones, stored as signed Q21.10 format, based on the Java MMAPI (JSR-135) requirement. As it is a bounded value type (OMX_BS32), it allows the querying and setting of a range of values, including minimum, actual, and maximum.
- sPlayBackRate is the relative playback rate, stored as an unsigned Q15.17 fixed-point number based on the JSR-135 requirement. As it is a bounded value type (OMX_BU32), it allows the querying and setting of a range of values, including minimum, actual, and maximum.
- sTempo is the tempo in beats per minute (BPM), stored as an unsigned Q22.10 fixed-point number based on the JSR-135 requirement. As it is a bounded value type (OMX_BU32), it allows the querying and setting of a range of values, including minimum, actual, and maximum.
- nMaxPolyphony specifies the maximum number of simultaneous polyphonic voices, which is the maximum run-time polyphony. A value of zero indicates that the default polyphony of the device is used.
- nNumRepeat specifies the number of times to repeat the playback.
- nStopTime is the time in milliseconds to indicate when playback will stop automatically. This value is set to zero if not used.
- nChannelMuteMask is a 16-bit mask for channel mute status.
- nChannelSoloMask is a 16-bit mask for channel solo status.
- nTrack0031MuteMask is a 32-bit mask for track mute status for tracks 0-31.
- nTrack3263MuteMask is a 32-bit mask for track mute status for tracks 32-63.
- nTrack0031SoloMask is a 32-bit mask for track solo status for tracks 0-31.
- nTrack3263SoloMask is a 32-bit mask for track mute status for tracks 32-63.

4.1.36.2 Post-processing Conditions

In case of a OMX_SetConfig call using the OMX_AUDIO_CONFIG_MIDICONTROLTYPE structure, the parameters required to control MIDI playback are set. In case of a OMX_GetConfig call using the

OMX_AUDIO_CONFIG_MIDICONTROLTYPE structure, the MIDI IL client can determine the parameters controlling MIDI playback.

4.1.37 OMX_AUDIO_CONFIG_MIDISTATUSTYPE

The OMX_AUDIO_CONFIG_MIDISTATUSTYPE structure is used to query the current status of the MIDI playback. As such, it can be used only by an OMX_GetConfig call. The OMX_AUDIO_CONFIG_MIDISTATUSTYPE structure returns all of the parameters that characterize the current status of the MIDI engine.

OMX_AUDIO_CONFIG_MIDISTATUSTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_MIDISTATUSTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U16 nNumTracks;
    OMX_U32 nDuration;
    OMX_U32 nPosition;
    OMX_BOOL bVibra;
    OMX_U32 nNumMetaEvents;
    OMX_U32 nNumActiveVoices;
    OMX_AUDIO_MIDIPLAYBACKSTATETYPE eMIDIPlayBackState;
} OMX_AUDIO_CONFIG_MIDISTATUSTYPE;
```

4.1.37.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_MIDISTATUSTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nNumTracks is a read-only field that identifies the number of MIDI tracks in the file. Note that this parameter will have a valid value only when the entire file has been parsed and buffered. An OMX_GetConfig call issued before the entire file has been processed will not contain the correct number of MIDI tracks.
- nDuration is the length of the currently open MIDI resource in milliseconds. As with nNumTracks, this parameter will have a meaningful value only after the entire file has been buffered.
- nPosition is the current position in milliseconds of the MIDI resource being played.
- bVibra is a Boolean value that indicates if a vibra track exists in the file. This parameter will return a meaningful value only after the entire file has been buffered. The value returned when in the middle of the file cannot be relied upon.
- nNumMetaEvents is the total number of MIDI meta events in the currently open MIDI resource. This parameter will return a valid value only after the entire file is buffered. The value returned when in the middle of the file cannot be relied upon.

- `nNumActiveVoices` is the number of active voices in the currently playing MIDI resource, or the current polyphony level. This parameter may not return a meaningful value until the entire file is parsed and buffered.
- `eMIDIPlayBackState` is the enumeration for the MIDI playback state. Table 4-29 describes the payback states.

Table 4-29: MIDI Playback States

Field Name	Description
`OMX_AUDIO_MIDIPlayBackStateUnknown`	Unknown/unused MIDI playback state, or state does not map to one of the defined states.
`OMX_AUDIO_MIDIPlayBackStateClosed Engaged`	No MIDI resource is currently open. The MIDI engine is currently processing MIDI events.
`OMX_AUDIO_MIDIPlayBackStateParsing`	A MIDI resource is open and is being primed. The MIDI engine is currently processing MIDI events.
`OMX_AUDIO_MIDIPlayBackStateOpen Engaged`	A MIDI resource is open and primed but not playing. The MIDI engine is currently processing MIDI events. The transition to this state is only possible from the `OMX_AUDIO_MIDIPlayBackStatePlaying` state when the 'playback head' reaches the end of media data or the playback stops due to a stop time setting.
`OMX_AUDIO_MIDIPlayBackStatePlaying`	A MIDI resource is open and currently playing. The MIDI engine is currently processing MIDI events.
`OMX_AUDIO_MIDIPlayBackStatePlaying Partially`	Best-effort playback due to SP-MIDI/DLS resource constraints
`OMX_AUDIO_MIDIPlayBackStatePlaying Silently`	Due to system resource constraints and SP-MIDI content constraints, there is currently no audible MIDI content during playback. The situation may change if resources are freed later.

4.1.38 OMX_AUDIO_CONFIG_MIDIMETAEVENTTYPE

MIDI meta events are like audio metadata, except that they are interspersed with the MIDI content throughout the file and not localized in the header. As such, it is necessary to retrieve information about these meta-events from the engine as it encounters these meta events within the MIDI content. Component vendors are not required to enumerate

all types of meta events; vendors can choose the meta events they want to support. Meta events are enumerated in the same order that they are detected in the MIDI file. Meta event data will always be provided as binary data, as it is present in the MIDI file.

The OMX_AUDIO_CONFIG_MIDIMETAEVENTTYPE structure is used to query the meta event, its track number, and the size of the meta event data using OMX_GetConfig. This allows the application to quickly determine meta events of interest. If the application requires the meta event data, the OMX_AUDIO_CONFIG_MIDIMETAEVENTDATATYPE structure, which is defined in section 4.1.39, needs to be used in a second OMX_GetConfig call.

OMX_AUDIO_CONFIG_MIDIMETAEVENTTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_MIDIMETAEVENTTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nIndex;
    OMX_U8 nMetaEventType;
    OMX_U32 nMetaEventSize;
    OMX_U32 nTrack;
    OMX_U32 nPosition;
} OMX_AUDIO_CONFIG_MIDIMETAEVENTTYPE;
```

4.1.38.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_MIDIMETAEVENTTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nIndex is the index of the meta event. Meta events will be numbered 0 to N-1, where N is the number of meta events that the MIDI decoder reports.
- nMetaEventType is the meta event type. The values are 0-127.
- nMetaEventSize is the size of the meta event in bytes.
- nTrack is the track number for the meta event.
- nPosition is the position of the meta event in milliseconds.

4.1.39 OMX_AUDIO_CONFIG_MIDIMETAEVENTDATATYPE

The OMX_AUDIO_CONFIG_MIDIMETAEVENTDATATYPE structure is typically used by the IL client via an OMX_GetConfig call to retrieve the meta event data, after the type, size and track number of the meta event have been determined by a previous OMX_GetConfig call using the OMX_AUDIO_CONFIG_MIDIMETAEVENTTYPE structure defined in section 4.1.38 above. The IL client is responsible for sizing the structure appropriately so that it can hold the meta event data.

OMX_AUDIO_CONFIG_MIDIMETAEVENTDATATYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_MIDIMETAEVENTDATATYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nIndex;
    OMX_U32 nMetaEventSize;
    OMX_U8 nData[1];
} OMX_AUDIO_CONFIG_MIDIMETAEVENTDATATYPE;
```

4.1.39.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_MIDIMETAEVENTDATATYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nIndex is the index of the meta event. Meta events are numbered 0 to N-1, where N is the number of meta events that the MIDI decoder reports.
- nMetaEventSize is the size of the meta event in bytes.
- nData is an array of one or more bytes of meta data as indicated by the nMetaEventSize field.

4.1.40 OMX_AUDIO_CONFIG_VOLUMETYPE

The OMX_AUDIO_CONFIG_VOLUMETYPE structure is used to adjust the audio volume for a port.

OMX_AUDIO_CONFIG_VOLUMETYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_VOLUMETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bLinear;
    OMX_BS32 sVolume;
} OMX_AUDIO_CONFIG_VOLUMETYPE;
```

4.1.40.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_VOLUMETYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bLinear is a Boolean to indicate if the volume is to be set on a linear (0-100) or a logarithmic scale (millibel, which is abbreviated mB).
- sVolume is the linear volume setting in the range 0-100, or the logarithmic volume setting for this port. The values for volume are in millibel (abbreviated mB, where 1 millibel = 1/100 decibel) relative to a gain of 1 (i.e., the output is the same as the input level). Values are in mB from nMax (maximum volume) to nMin (minimum volume, typically negative). Since the volume is voltage and not

a power, it takes a setting of -600 mB to decrease the volume by half. If a component cannot accurately set the volume to the requested value, it shall set the volume to the closest value below the requested value. When getting the volume setting, the current actual volume shall be returned.

4.1.41 OMX_AUDIO_CONFIG_CHANNELVOLUMETYPE

The `OMX_AUDIO_CONFIG_CHANNELVOLUMETYPE` structure is used to adjust the audio volume for a channel via the `OMX_IndexConfigAudioChannelVolume` config.

`OMX_AUDIO_CONFIG_CHANNELVOLUMETYPE` is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_CHANNELVOLUMETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannel;
    OMX_BOOL bLinear;
    OMX_BS32 sVolume;
    OMX_BOOL bIsMIDI;
} OMX_AUDIO_CONFIG_CHANNELVOLUMETYPE;
```

4.1.41.1 Parameter Definitions

The parameters for `OMX_AUDIO_CONFIG_CHANNELVOLUMETYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `nChannel` is the channel to select in the range 0 to N-1. Use OMX_ALL to apply the volume setting to all channels.
- `bLinear` is the volume to be set on a linear scale (0-100) or a logarithmic scale (mB).
- `sVolume` is the linear volume setting in the range 0-100 or the logarithmic volume setting for this port. The values for volume are in millibel (abbreviated mB, where 1 millibel = 1/100 dB) relative to a gain of 1 (i.e., the output is the same as the input level). Values are in mB from nMax (maximum volume) to nMin (minimum volume, typically negative). Since the volume is voltage and not a power, it takes a setting of -600 mB to decrease the volume by half. If a component cannot accurately set the volume to the requested value, it shall set the volume to the closest value below the requested value. When getting the volume setting, the current actual volume shall be returned.
- `bIsMIDI` is OMX_TRUE if `nChannel` refers to a MIDI channel, or OMX_FALSE otherwise.

4.1.42 OMX_AUDIO_CONFIG_BALANCETYPE

The OMX_AUDIO_CONFIG_BALANCETYPE structure defines the audio left-right balance adjustment for a port.

OMX_AUDIO_CONFIG_BALANCETYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_BALANCETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_S32 nBalance;
} OMX_AUDIO_CONFIG_BALANCETYPE;
```

4.1.42.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_BALANCETYPE are as follows.

- nPortIndex represents the port that this structure applies to. Select the input port to set just that port's balance. Select the output port to adjust the master balance.
- nBalance is the balance setting for this port. The values are -100 to 100, where -100 indicates all left, and no right.

4.1.43 OMX_AUDIO_CONFIG_MUTETYPE

The OMX_AUDIO_CONFIG_MUTETYPE structure adjusts the audio mute for a port.

OMX_AUDIO_CONFIG_MUTETYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_MUTETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bMute;
} OMX_AUDIO_CONFIG_MUTETYPE;
```

4.1.43.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_MUTETYPE are defined as follows.

- nPortIndex represents the port that this structure applies to. Select the input port to set just that port's mute setting. Select the output port to adjust the master mute.
- bMute identifies whether the port is muted (OMX_TRUE) or playing normally (OMX_FALSE).

4.1.44 OMX_AUDIO_CONFIG_CHANNELMUTETYPE

The OMX_AUDIO_CONFIG_CHANNELMUTETYPE structure adjusts the audio mute for a channel.

OMX_AUDIO_CONFIG_CHANNELMUTETYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_CHANNELMUTETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannel;
    OMX_BOOL bMute;
    OMX_BOOL bIsMIDI;
} OMX_AUDIO_CONFIG_CHANNELMUTETYPE;
```

4.1.44.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_CHANNELMUTETYPE are defined as follows.

- nPortIndex represents the port that this structure applies to. Select the input port to set just that port's mute setting. Select the output port to adjust the master mute.
- nChannel is the channel to select in the range 0 to N-1. Use OMX_ALL to apply the audio mute setting to all channels.
- bMute identifies whether port is muted (OMX_TRUE) or playing normally (OMX_FALSE).
- bIsMIDI identifies whether the channel is a MIDI channel. The values are OMX_TRUE if nChannel refers to a MIDI channel, OMX_FALSE if otherwise.

4.1.45 OMX_AUDIO_CONFIG_LOUDNESSTYPE

The OMX_AUDIO_CONFIG_LOUDNESSTYPE structure is used to enable or disable the loudness audio effect, which boosts the bass and the high frequencies to compensate for the limited hearing range of humans at the extreme ends of the audio spectrum. The setting can be changed using the OMX_SetConfig function. The current state can be queried using the OMX_GetConfig function. When calling either OMX_SetConfig or OMX_GetConfig, the index specified for this structure is OMX_IndexConfigAudioLoudness.

OMX_AUDIO_CONFIG_LOUDNESSTYPE is defined as follows.

```
 typedef struct OMX_AUDIO_CONFIG_LOUDNESSTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bLoudness;
} OMX_AUDIO_CONFIG_LOUDNESSTYPE;
```

4.1.45.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_LOUDNESSTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bLoudness enables the loudness if set to OMX_TRUE or disables the loudness effect if set to OMX_FALSE.

4.1.46 OMX_AUDIO_CONFIG_BASSTYPE

The OMX_AUDIO_CONFIG_BASSTYPE structure is used to enable or disable the low-frequency level (bass) audio effect, and to set or query the current bass level. The setting can be changed using the OMX_SetConfig function, and the current state can be queried using the OMX_GetConfig function. When calling either function, the index specified for this structure is OMX_IndexConfigAudioBass.

OMX_AUDIO_CONFIG_BASSTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_BASSTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bEnable;
    OMX_S32 nBass;
} OMX_AUDIO_CONFIG_BASSTYPE;
```

4.1.46.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_BASSTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bEnable enables the bass-level setting if set to OMX_TRUE or disables the bass-level setting if set to OMX_FALSE.
- nBass is the bass-level setting for the port, as a continuous value from -100 to 100. The value –100 means minimum bass level, zero means no change in level, and 100 represents the maximum low-frequency boost.

4.1.47 OMX_AUDIO_CONFIG_TREBLETYPE

The OMX_AUDIO_CONFIG_TREBLETYPE structure is used to enable or disable the high-frequency level (treble) audio effect, and to set or query the current level. The setting can be changed using the OMX_SetConfig function, and the current state can be queried using the OMX_GetConfig function. When calling either function, the index specified for this structure is OMX_IndexConfigAudioTreble.

OMX_AUDIO_CONFIG_TREBLETYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_TREBLETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bEnable;
    OMX_S32  nTreble;
```

```
} OMX_AUDIO_CONFIG_TREBLETYPE;
```

4.1.47.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_TREBLETYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bEnable enables the treble level setting if set to OMX_TRUE or disables the treble level setting if set to OMX_FALSE.
- nTreble is the treble-level setting for the port, as a continuous value from -100 to 100. The value -100 means minimum high-frequency level, zero means no change in level, and 100 represents the maximum high-frequency boost.

4.1.48 OMX_AUDIO_CONFIG_EQUALIZERTYPE

The OMX_AUDIO_CONFIG_EQUALIZERTYPE structure is used to set or query the current parameters of the graphic equalizer (EQ) effect. The settings can be changed using the OMX_SetConfig function, and the current state can be queried using the OMX_GetConfig function. When calling either function, the index specified for this structure is OMX_IndexConfigAudioEqualizer.

An equalizer modifies the audio signal by frequency-dependent amplification or attenuation. A graphic EQ typically lets the user control the character of sound by controlling the levels of several fixed-frequency bands. The bands are characterized by their center and crossover frequencies.

In practice, the calling application or framework is often first interested in the number of bands that the EQ implementation supports. This number can be queried by a single call to OMX_GetConfig with sBandIndex set to zero. The query results in the same data structure with the maximum value of sBandIndex filled with N-1, where N is the number of frequency bands. The same structure will also contain the frequency and level limits for the first band. Similar queries for the rest of the bands yield the information needed, for example, to construct a user interface for the equalizer.

OMX_AUDIO_CONFIG_EQUALIZERTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_EQUALIZERTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bEnable;
    OMX_BU32 sBandIndex;
    OMX_BU32 sCenterFreq;
    OMX_BS32 sBandLevel;
} OMX_AUDIO_CONFIG_EQUALIZERTYPE;
```

4.1.48.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_EQUALIZERTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bEnable enables the EQ effect if set to OMX_TRUE or disables the EQ effect if set to OMX_FALSE.
- sBandIndex is the index of the band to be set or retrieved. The upper limit is N-1, where N is the number of bands. The lower limit is 0.
- sCenterFreq is the center frequencies in Hz. This is a read-only element and is used by the caller to determine the lower, center, and upper frequency of this band.
- sBandLevel is the band level in millibels.

4.1.49 OMX_AUDIO_CONFIG_STEREOWIDENINGTYPE

The OMX_AUDIO_CONFIG_STEREOWIDENINGTYPE structure is used to enable or disable the stereo widening audio effect, and to set or query the current strength of the effect. The setting can be changed using the OMX_SetConfig function, and the current state can be queried using the OMX_GetConfig function. When calling either function, the index specified for this structure is OMX_IndexConfigAudioStereoWidening.

Stereo widening is a special case of the "audio virtualizer" effect, and is designed to remove the inside-the-head effect in headphone listening, or to extend the stereo image beyond the physical loudspeaker span in loudspeaker reproduction.

OMX_AUDIO_CONFIG_STEREOWIDENINGTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_STEREOWIDENINGTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bEnable;
    OMX_AUDIO_STEREOWIDENINGTYPE eWideningType;
    OMX_U32 nStereoWidening;
} OMX_AUDIO_CONFIG_STEREOWIDENINGTYPE;
```

4.1.49.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_STEREOWIDENINGTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bEnable enables the stereo widening effect if set to OMX_TRUE or disables the stereo widening effect if set to OMX_FALSE.
- eWideningType is the stereo widening processing type, as shown in Table 4-30.

Table 4-30: Stereo Widening Processing Type

Field Name	Description
OMX_AUDIO_StereoWideningHeadphones	Stereo widening for headphones.
OMX_AUDIO_StereoWideningLoudspeakers	Stereo widening for two closely spaced loudspeakers.

- nStereoWidening is the stereo widening setting for the port, as a continuous value from 0 (minimum effect) to 100 (maximum effect). If the component can implement only a discrete set of presets (say, only on or off), it may round the value to a nearest available setting. When getting the setting, the exact current value shall be returned.

4.1.50 OMX_AUDIO_CONFIG_CHORUSTYPE

The OMX_AUDIO_CONFIG_CHORUSTYPE structure is used to enable or disable the chorus audio effect, and to set or query the current parameters of the effect. The settings can be changed using the OMX_SetConfig function, and the current state can be queried using the OMX_GetConfig function. When calling either function, the index specified for this structure is OMX_IndexConfigAudioChorus.

Chorus is an audio effect that presents a sound, such as a vocal track, as though it was performed by two or more singers simultaneously. The effect is produced by feeding the sound through one or more delay lines with time-variant lengths, and summing the delayed signals with the original, non-delayed sound. The length of each delay line is modulated by a low-frequency signal. Modulation waveform and stereo output details are implementation dependent.

OMX_AUDIO_CONFIG_CHORUSTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_CHORUSTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bEnable;
    OMX_BU32 sDelay;
    OMX_BU32 sModulationRate;
    OMX_U32 nModulationDepth;
    OMX_BU32 nFeedback;
} OMX_AUDIO_CONFIG_CHORUSTYPE;
```

4.1.50.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_CHORUSTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bEnable enables the chorus effect if set to OMX_TRUE or disables the chorus effect if set to OMX_FALSE.

- sDelay is the average delay in milliseconds.
- sModulationRate is the rate of modulation in mHz.
- nModulationDepth is the depth of modulation as a percentage of delay zero-to-peak. The range of values is 0-100.
- nFeedback is the feedback from the chorus output to the input in percentage.

4.1.51 OMX_AUDIO_CONFIG_REVERBERATIONTYPE

The OMX_AUDIO_CONFIG_REVERBERATIONTYPE structure is used to enable or disable the reverberation effect, and to set or query the current parameters of the effect. The settings can be changed using the OMX_SetConfig function, and the current state can be queried using the OMX_GetConfig function. When calling either function, the index specified for this structure is OMX_IndexConfigAudioReverberation.

The reverberation effect models the effect of a room (room response) to the sound. The room response is divided into three sections: direct path, early reflections, and late reverberation. This division and the effect parameters are essentially the same as used in the Interactive 3D Audio Rendering Guidelines – Level 2.0 by the Interactive Audio Special Interest Group (IASIG) of the MIDI Manufacturers Association (MMA). For more information on this specification, see http://www.iasig.org/pubs/3dl2v1a.pdf.

OMX_AUDIO_CONFIG_REVERBERATIONTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_REVERBERATIONTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bEnable;
    OMX_BS32 sRoomLevel;
    OMX_BS32 sRoomHighFreqLevel;
    OMX_BS32 sReflectionsLevel;
    OMX_BU32 sReflectionsDelay;
    OMX_BS32 sReverbLevel;
    OMX_BU32 sReverbDelay;
    OMX_BU32 sDecayTime;
    OMX_BU32 nDecayHighFreqRatio;
    OMX_U32 nDensity;
    OMX_U32 nDiffusion;
    OMX_BU32 sReferenceHighFreq;
} OMX_AUDIO_CONFIG_REVERBERATIONTYPE;
```

4.1.51.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_REVERBERATIONTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bEnable enables the reverberation effect if set to OMX_TRUE or disables the reverberation effect if set to OMX_FALSE.

- sRoomLevel is the intensity level for the whole room effect, including both early reflections and late reverberation, in millibels.
- sRoomHighFreqLevel is the attenuation in millibels at high frequencies relative to the intensity at low frequencies.
- sReflectionsLevel is the intensity level of early reflections, which are relative to the room level value, in millibels.
- sReflectionsDelay is the time delay in milliseconds of the first reflection relative to the direct path.
- sReverbLevel is the intensity level in millibels of late reverberation relative to the room level.
- sReverbDelay is the time delay in milliseconds from the first early reflection to the beginning of the late reverberation section.
- sDecayTime is the late reverberation decay time in milliseconds at low frequencies, defined as the time needed for the reverberation to decay by 60 dB.
- nDecayHighFreqRatio is the ratio of high-frequency decay time relative to low-frequency decay time as percentage in the range 0–100.
- nDensity is the modal density in the late reverberation decay as a percentage. The range of values is 0-100.
- nDiffusion is the echo density in the late reverberation decay as a percentage. The range of values is 0-100.
- sReferenceHighFreq is the reference high frequency in Hertz. This is the frequency used as the reference for all of the high-frequency parameter settings.

4.1.52 OMX_AUDIO_CONFIG_ECHOCANCELATIONTYPE

The OMX_AUDIO_CONFIG_ECHOCANCELATIONTYPE structure is used to enable or disable echo canceling, which removes undesired echo from speech or audio. The setting can be changed using the OMX_SetConfig function, and the current state can be queried using the OMX_GetConfig function. When calling either function, the index specified for this structure is OMX_IndexConfigAudioEchoCancelation.

OMX_AUDIO_CONFIG_ECHOCANCELATIONTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_ECHOCANCELATIONTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_AUDIO_ECHOCANTYPE eEchoCancelation;
} OMX_AUDIO_CONFIG_ECHOCANCELATIONTYPE;
```

4.1.52.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_ECHOCANCELATIONTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- eEchoCancelation is the enumeration for enabling/disabling echo cancellation and selecting the mode, as shown in Table 4-31.

Table 4-31: Echo Cancellation Values

Field Name	Description
OMX_AUDIO_EchoCanOff	Echo cancellation is disabled.
OMX_AUDIO_EchoCanNormal	Echo cancellation normal operation; echo from handset plastics and face.
OMX_AUDIO_EchoCanHFree	Echo cancellation optimized for hands-free operation.
OMX_AUDIO_EchoCanCarKit	Echo cancellation optimized for car kit (longer echo).

4.1.53 OMX_AUDIO_CONFIG_NOISEREDUCTIONTYPE

The OMX_AUDIO_CONFIG_NOISEREDUCTIONTYPE structure is used to enable or disable noise reduction processing, which removes undesired noise from audio. The setting can be changed using the OMX_SetConfig function, and the current state can be queried using the OMX_GetConfig function. When calling either function, the index specified for this structure is OMX_IndexConfigAudioNoiseReduction.

OMX_AUDIO_CONFIG_NOISEREDUCTIONTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_NOISEREDUCTIONTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bNoiseReduction;
} OMX_AUDIO_CONFIG_NOISEREDUCTIONTYPE;
```

4.1.53.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_NOISEREDUCTIONTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bNoiseReduction enables noise reduction processing if set to OMX_TRUE or disables noise reduction processing if set to OMX_FALSE.

4.1.54 OMX_AUDIO_CONFIG_3DOUTPUTTYPE

The OMX_AUDIO_CONFIG_3DOUTPUTTYPE structure is used to set or query the output type of the 3D processing. The setting can be changed using the OMX_SetConfig function, and the current state can be queried using the OMX_GetConfig function. When calling either function, the index specified for this structure is OMX_IndexConfigAudio3DOutput.

OMX_AUDIO_CONFIG_3DOUTPUTTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_3DOUTPUTTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_AUDIO_3DOUTPUTTYPE e3DOutputType;
} OMX_AUDIO_CONFIG_3DOUTPUTTYPE;
```

4.1.54.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_3DOUTPUTTYPE are defined as follows.

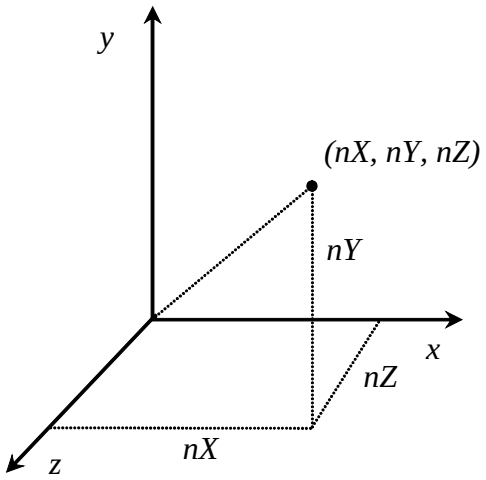
- nPortIndex represents the port that this structure applies to.
- e3DOutputType is the positional 3D audio processing type, as shown below

Field Name	Description
OMX_AUDIO_3DOutputHeadphones	Positional 3D audio for headphones.
OMX_AUDIO_3DOutputLoudspeakers	Positional 3D audio for two closely spaced loudspeakers.
OMX_AUDIO_3DOutputMax	Allowance for expansion in the number of positional 3D audio types.

4.1.55 OMX_AUDIO_CONFIG_3DLOCATIONTYPE

The OMX_AUDIO_CONFIG_3DLOCATIONTYPE structure is used to set the virtual location for the 3D sound source. The settings can be changed using the OMX_SetConfig function, and the current state can be queried using the OMX_GetConfig function. When calling either function, the index specified for this structure is OMX_IndexConfigAudio3DLocation.

The location is set by using right-handed coordinates relative to the listener. The listener is stationary: located in origin and pointing towards negative Z-axis up direction being the positive Y-axis.



OMX_AUDIO_CONFIG_3DLOCATIONTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_3DLOCATIONTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_S32 nX;
    OMX_S32 nY;
    OMX_S32 nZ;
} OMX_AUDIO_CONFIG_3DLOCATIONTYPE;
```

4.1.55.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_3DLOCATIONTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nX is the X coordinate in millimeters.
- nY is the Y coordinate in millimeters.
- nZ is the Z coordinate in millimeters.

4.1.56 OMX_AUDIO_PARAM_3DDOPPLERMODETYPE

The OMX_AUDIO_PARAM_3DDOPPLERMODETYPE structure is used to switch the Doppler effect on and off. The settings can be changed using the OMX_SetParameter function, and the current state can be queried using the OMX_GetParameter function. When calling either function, the index specified for this structure is OMX_IndexParamAudio3DDopplerMode.

OMX_AUDIO_PARAM_3DDOPPLERMODETYPE is defined as follows.

```
typedef struct OMX_AUDIO_PARAM_3DDOPPLERMODETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bEnabled;
```

```
} OMX_AUDIO_PARAM_3DDOPPLERMODETYPE;
```

4.1.56.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_3DDOPPLERMODETYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bEnabled: if true, the Doppler effect for this port is enabled; if false, the Doppler effect for this port is disabled.

4.1.57 OMX_AUDIO_CONFIG_3DDOPPLERSETTINGSTYPE

The OMX_AUDIO_CONFIG_3DDOPPLERSETTINGSTYPE structure is used to set the Doppler behavior of the 3D sound source. The settings can be changed using the OMX_SetConfig function, and the current state can be queried using the OMX_GetConfig function. When calling either function, the index specified for this structure is OMX_IndexConfigAudio3DDopplerSettings.

OMX_AUDIO_CONFIG_3DDOPPLERSETTINGSTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_3DDOPPLERSETTINGSTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nSoundSpeed;
    OMX_S32 nSourceVelocity;
    OMX_S32 nListenerVelocity;
 } OMX_AUDIO_CONFIG_3DDOPPLERSETTINGSTYPE;
```

The Doppler coefficent is usually calculated as:

$$Doppler = \frac{nSoundSpeed + nListenerVelocity}{nSoundSpeed - nSourceVelocity}$$

4.1.57.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_3DDOPPLERSETTINGSTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nSoundSpeed defines the speed of sound in millimeters per second.
- nSourceVelocity defines the speed of the source in the direction of the listener in millimeters per second. This is usually calculated as the velocity vector of the sound source projected on the line between source and listener positions.
- nListenerVelocity defines the speed of the listener in the direction of the source in millimeters per second. This is usually calculated as the velocity vector of the listener projected on the line between source and listener positions.

4.1.58 OMX_AUDIO_CONFIG_3DLEVELSTYPE

The OMX_AUDIO_CONFIG_3DLEVELSTYPE structure is used to set the direct path and room levels for the 3D sound source. The settings can be changed using the OMX_SetConfig function, and the current state can be queried using the OMX_GetConfig function. When calling either function, the index specified for this structure is OMX_IndexConfigAudio3DLevels.

OMX_AUDIO_CONFIG_3DLEVELSTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_3DLEVELSTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BS32 sDirectLevel;
    OMX_BS32 sRoomLevel;
} OMX_AUDIO_CONFIG_3DLEVELSTYPE;
```

4.1.58.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_3DLEVELSTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- sDirectLevel is the level for direct path signal in millibels.
- sRoomLevel is the level for room signal in millibels.

4.1.59 OMX_AUDIO_CONFIG_3DDISTANCEATTENUATIONTYPE

The OMX_AUDIO_CONFIG_3DDISTANCEATTENUATIONTYPE structure is used to set the distance attenuation behavior for the 3D sound source. The settings can be changed using the OMX_SetConfig function, and the current state can be queried using the OMX_GetConfig function. When calling either function, the index specified for this structure is OMX_IndexConfigAudio3DDistanceAttenuation.

OMX_AUDIO_CONFIG_3DDISTANCEATTENUATIONTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_3DDISTANCEATTENUATIONTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BS32 sMinDistance;
    OMX_BS32 sMaxDistance;
    OMX_BS32 sRollOffFactor;
    OMX_BS32 sRoomRollOffFactor;
    OMX_AUDIO_ROLLOFFMODELTYPE eRollOffModel;
    OMX_BOOL bMuteAfterMax;
} OMX_AUDIO_CONFIG_3DDISTANCEATTENUATIONTYPE;
```

4.1.59.1 Parameter Definitions

The parameters for `OMX_AUDIO_CONFIG_3DDISTANCEATTENUATIONTYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `sMinDistance` is the d_{min} in the formulae below.
- `sMaxDistance` is the d_{max} in the formulae below.
- `sRollOffFactor` is the *rolloffFactor* in thousandths in the formulae below when calculating the distance attenuation for the direct path signal output.
- `sRoomRollOffFactor` is the *rolloffFactor* in thousandths in the formulae below, but when calculating the distance attenuation for the room signal output.
- `eRollOffModel` is the roll-off model defined below.
- `sMuteAfterMax` is the *rollOffMaxDistanceMute* in the formulae below.

Two distance roll-off models are supported. The exponential distance rolloff model is defined as:

$$gain(d) = \begin{cases} 1 & \text{if } d < d_{\min} \\ 0 & \text{if } d \geq d_{\max} \text{ and } rolloffMaxDistanceMute = true \\ \left(\frac{d_{\min}}{d_{\max}}\right)^{rolloffFactor} & \text{if } d \geq d_{\max} \text{ and } rolloffMaxDistanceMute = false \\ \left(\frac{d_{\min}}{d}\right)^{rolloffFactor} & \text{otherwise} \end{cases}$$

And, the linear distance rolloff model is defined as:

$$gain(d) = \begin{cases} 1 & \text{if } d < d_{\min} \\ 0 & \text{if } d \geq d_{\max} \text{ and } rolloffMaxDistanceMute = true \\ \max(0, 1 - rolloffFactor) & \text{if } d \geq d_{\max} \text{ and } rolloffMaxDistanceMute = false \\ \max\left(0, 1 - \left(rolloffFactor \times \frac{d - d_{\min}}{d_{\max} - d_{\min}}\right)\right) & \text{otherwise} \end{cases}$$

where d is the distance of the sound source from the origin (listener).

4.1.60 OMX_AUDIO_CONFIG_3DDIRECTIVITYSETTINGSTYPE

The `OMX_AUDIO_CONFIG_3DDIRECTIVITYSETTINGSTYPE` structure is used to set the directivity behavior of the 3D sound source. The settings can be changed using the `OMX_SetConfig` function, and the current state can be queried using the

OMX_GetConfig function. When calling either function, the index specified for this structure is OMX_IndexConfigAudio3DDirectivitySettings.

OMX_AUDIO_CONFIG_3DDIRECTIVITYSETTINGSTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_3DDIRECTIVITYSETTINGSTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BS32 sInnerAngle;
    OMX_BS32 sOuterAngle;
    OMX_BS32 sOuterLevel;
} OMX_AUDIO_CONFIG_3DDIRECTIVITYSETTINGSTYPE;
```

4.1.60.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_3DDIRECTIVITYSETTINGSTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- sInnerAngle defines the inner cone in millidegrees. Within the inner cone the source radiates at its "full" level.
- sOuterAngle defines the outer cone in millidegrees. Outside of the outer cone the source radiates at *outerLevel* level relative to the full level.
- sOuterLevel defines the *outerLevel* defined above in millibels.

4.1.61 OMX_AUDIO_CONFIG_3DDIRECTIVITYORIENTATIONTYPE

The OMX_AUDIO_CONFIG_3DDIRECTIVITYORIENTATIONTYPE structure is used to set the orientation of the directivity of the 3D sound source. The settings can be changed using the OMX_SetConfig function, and the current state can be queried using the OMX_GetConfig function. When calling either function, the index specified for this structure is OMX_IndexConfigAudio3DDirectivityOrientation.

OMX_AUDIO_CONFIG_3DDIRECTIVITYORIENTATIONTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_3DDIRECTIVITYORIENTATIONTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_S32 nXFront;
    OMX_S32 nYFront;
    OMX_S32 nZFront;
} OMX_AUDIO_CONFIG_3DDIRECTIVITYORIENTATIONTYPE;
```

4.1.61.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_3DDIRECTIVITYORIENTATIONTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nXFront is the X component of the front direction vector of the sound source.
- nYFront is the Y component of the front direction vector of the sound source.
- nZFront is the Z component of the front direction vector of the sound source.

4.1.62 OMX_AUDIO_CONFIG_3DMACROSCOPICORIENTATIONTYPE

The OMX_AUDIO_CONFIG_3DMACROSCOPICORIENTATIONTYPE structure is used to set the orientation of the macroscopicity of the 3D sound source. The settings can be changed using the OMX_SetConfig function, and the current state can be queried using the OMX_GetConfig function. When calling either function, the index specified for this structure is OMX_IndexConfigAudio3DMacroscopicOrientation.

OMX_AUDIO_CONFIG_3DMACROSCOPICORIENTATIONTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_3DMACROSCOPICORIENTATIONTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_S32 nXFront;
    OMX_S32 nYFront;
    OMX_S32 nZFront;
    OMX_S32 nXAbove;
    OMX_S32 nYAbove;
    OMX_S32 nZAbove;
} OMX_AUDIO_CONFIG_3DMACROSCOPICORIENTATIONTYPE;
```

4.1.62.1 Parameter Definitions

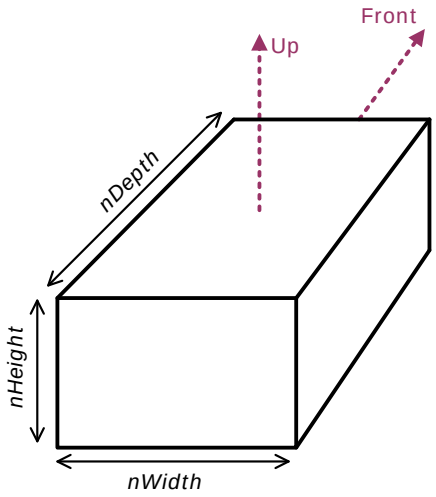
The parameters for OMX_AUDIO_CONFIG_3DMACROSCOPICORIENTATIONTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nXFront is the X component of the front direction vector of the sound source.
- nYFront is the Y component of the front direction vector of the sound source.
- nZFront is the Z component of the front direction vector of the sound source.
- nXAbove is the X component of the above direction vector of the sound source.
- nYAbove is the Y component of the above direction vector of the sound source.
- nZAbove is the Z component of the above direction vector of the sound source.

4.1.63 OMX_AUDIO_CONFIG_3DMACROSCOPICSIZETYPE

The OMX_AUDIO_CONFIG_3DMACROSCOPICSIZETYPE structure is used to set the size of the macroscopicity of the 3D sound source. The settings can be changed using the

OMX_SetConfig function, and the current state can be queried using the OMX_GetConfig function. When calling either function, the index specified for this structure is OMX_IndexConfigAudio3DMacroscopicSize.



OMX_AUDIO_CONFIG_3DMACROSCOPICSIZETYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_3DMACROSCOPICSIZETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_S32 nWidth;
    OMX_S32 nHeight;
    OMX_S32 nDepth;
} OMX_AUDIO_CONFIG_3DMACROSCOPICSIZETYPE;
```

4.1.63.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_3DMACROSCOPICSIZETYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nWidth is the width of the macroscopic sound source in millimeters.
- nHeight is the height of the macroscopic sound source in millimeters.
- nDepth is the depth of the macroscopic sound source in millimeters.

4.1.64 OMX_AUDIO_CHANNELMAPPINGTYPE

The OMX_AUDIO_PARAM_CHANNELMAPPINGTYPE structure is used to query the channel mapping information of the audio stream.

OMX_AUDIO_PARAM_CHANNELMAPPINGTYPE is defined as follows.

```
typedef struct OMX_AUDIO_PARAM_CHANNELMAPPINGTYPE {
```

```
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nChannels;
    OMX_AUDIO_CHANNELTYPE nChannelsMapping[OMX_AUDIO_MAXCHANNELS];
} OMX_AUDIO_PARAM_CHANNELMAPPINGTYPE;
```

4.1.64.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_CHANNELMAPPINGTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nChannels is the number of channels of audio (mono, stereo, multi-channel).
- nChannelsMapping identifies the channel mappings available within the stream.

4.1.65 OMX_AUDIO_SBCBITPOOLTYPE

The OMX_AUDIO_SBCBITPOOLTYPE structure is used to set or query the SBC codec bit-pool parameter.

OMX_AUDIO_SBCBITPOOLTYPE is defined as follows.

```
 typedef struct OMX_AUDIO_SBCBITPOOLTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nNewBitPool;
} OMX_AUDIO_SBCBITPOOLTYPE;
```

4.1.65.1 Parameter Definitions

The parameters for OMX_AUDIO_SBCBITPOOLTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nNewBitPool is the size of the bit allocation pool used for encoding the stream.

4.1.66 OMX_AUDIO_AMRMODETYPE

The OMX_AUDIO_AMRMODETYPE structure is used to set or query the AMR codec mode and bitrate settings.

OMX_AUDIO_AMRMODETYPE is defined as follows.

```
 typedef struct OMX_AUDIO_AMRMODETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
```

```
    OMX_U32 nBitRate;
    OMX_AUDIO_AMRBANDMODETYPE eAMRBandMode ;
} OMX_AUDIO_AMRMODETYPE;
```

4.1.66.1 Parameter Definitions

The parameters for OMX_AUDIO_AMRMODETYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nBitRate is the bitrate of the encoded AMR audio
- eAMRBandMode is the bit rate of the encoded speech. Table 4-20 shows the bit rate values.

4.1.67 OMX_AUDIO_CONFIG_BITRATETYPE

The audio encoder's bit rate setting may be updated while the audio encoder is actively encoding, the OMX_AUDIO_CONFIG_BITRATETYPE structure contains the parameters for updating the audio bit rate.

OMX_AUDIO_CONFIG_BITRATETYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_BITRATETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nEncodeBitrate;
   } OMX_AUDIO_CONFIG_BITRATETYPE;
```

4.1.67.1 Parameters

The parameters for OMX_AUDIO_CONFIG_BITRATETYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nEncodeBitrate is the target bit rate for the audio encoding in units of bits per second. Encoding is set to the bitrate closest to the specified value. Use 0 to let the encoder decide on the appropriate bitrate value

4.1.68 OMX_AUDIO_CONFIG_AMRISFTYPE

The AMR WB+ encoder's sampling frequency may be updated while the audio encoder is actively encoding.

OMX_AUDIO_CONFIG_AMRISFTYPE is defined as follows.

```
typedef struct OMX_AUDIO_CONFIG_AMRISFTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_AUDIO_AMRISFINDEXTYPE  eTargetAMRISFIndex;
} OMX_AUDIO_CONFIG_AMRISFTYPE;
```

4.1.68.1 Parameters

The parameters for OMX_AUDIO_CONFIG_AMRISFTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- eTargetAMRISFIndex is the target internal-sampling-frequency index for AMR-WB+ audio taking on values defined in Table 4-23. Use OMX_AUDIO_AMRISFIndexAuto to let the encoder decide on the appropriate ISF value. This parameter shall be ignored for formats other than AMR-WB+.

4.1.69 OMX_AUDIO_FIXEDPOINTTYPE

The OMX_AUDIO_FIXEDPOINTTYPE structure is used to set or query the current settings for the packing of PCM data within the elements specified by OMX_AUDIO_FIXEDPOINTTYPE.

OMX_AUDIO_FIXEDPOINTTYPE is defined as follows.

```
typedef struct OMX_AUDIO_FIXEDPOINTTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nValueStartBit;
    OMX_U32 nValueBits;
    OMX_U32 nSignExtensionBits;
    OMX_S32 nValuePointPosition;
} OMX_AUDIO_FIXEDPOINTTYPE;
```

4.1.69.1 Parameters

The parameters for OMX_AUDIO_FIXEDPOINTTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nValueStartBit is the bit position of the lowest valid bit.
- nValueBits is the number of bits for each sample.

  The value is contained in bits nValueStartBit + nValueBits – 1 ..

  nValueStartBit inclusive.

  When extracting the bit field, sign extension will be required if, and only if, OMX_AUDIO_PARAM_PCM_MODETYPE specifies OMX_NumericalDataSigned.
- nSignExtensionBits is the number of additional sign bits. These shall be a copy of the sign bit. An implementation may extract the bit field with or without these bits, as the result is guaranteed to be identical.

- nValuePointPosition is the bit position of the fixed point. This may be outside the valid bits, which requires implicit bits to be added. For non-fixed point samples, this shall be nValueStartBit.

4.1.69.2 Functionality

The OMX_AUDIO_FIXEDPOINTTYPE structure sets / gets position of the PCM sample bit field and its fixed point interpretation. Setting OMX_AUDIO_PARAM_PCMMODETYPE shall implicitly set OMX_AUDIO_FIXEDPOINTTYPE to:

Field Name	OMX_AUDIO_PARAM_PCMMODETYPE Derived Value
nValueStartBit	0
nValueBits	nBitsPerSample
nSignExtensionBits	0
nValuePointPosition	nBitsPerSample-1

The following table shows examples of common PCM formats:

	Typical 16b	Packed 24b (in 32b)
OMX_AUDIO_FIXEDPOINTTYPE		
nValueStartBit	0	0
nValueBits	16	24
nSignExtensionBits	0	8
nValuePointPosition	15	23
OMX_AUDIO_PARAM_PCMMODETYPE		
nBitPerSample	16	32
eNumData	OMX_NumericalDataSigned	

4.2 Image and Video Common

This section describes the parameter and configuration details for ports in the video and image domains. These parameter and configuration details are specified in the OMX_IVCommon.h header.

4.2.1 Uncompressed Data Formats

Both image and video ports operate on compressed and uncompressed data. The formats for uncompressed pixel data are common to both image and video. Table 4-32 lists the uncompressed formats.

Table 4-32: Uncompressed Data Formats

OMX_COLOR_FORMATTYPE	Description
OMX_COLOR_FormatUnused	Placeholder value when format is unknown, or specified using a vendor-specific means.
OMX_COLOR_FormatMonochrome	1 bit per pixel monochrome.
OMX_COLOR_FormatL2	2 bit per pixel luminance.
OMX_COLOR_FormatL4	4 bit per pixel luminance.
OMX_COLOR_FormatL8	8 bit per pixel luminance.
OMX_COLOR_FormatL16	16 bit per pixel luminance.
OMX_COLOR_FormatL24	24 bit per pixel luminance.
OMX_COLOR_FormatL32	32 bit per pixel luminance.
OMX_COLOR_Format8bitRGB332	8 bits per pixel RGB format with colors stored as Red 7:5, Green 4:2, and Blue 1:0.
OMX_COLOR_Format8bitBGR233	8 bits per pixel BGR format with colors stored as Blue 7:6, Green 5:3, and Red 2:0.
OMX_COLOR_Format12bitRGB444	12 bits per pixel RGB format with colors stored as Red 11:8, Green 7:4, and Blue 3:0.
OMX_COLOR_Format12bitBGR444	12 bits per pixel BGR format with colors stored as Blue 11:8, Green 7:4, and Red 3:0.
OMX_COLOR_Format16bitARGB4444	16 bits per pixel ARGB format with colors stored as Alpha 15:12, Red 11:8, Green 7:4, and Blue 3:0.
OMX_COLOR_Format16bitBGRA4444	16 bits per pixel BGRA format with colors stored as Blue 15:12, Green 11:8, Red 7:4, and Alpha 3:0.
OMX_COLOR_Format16bitARGB1555	16 bits per pixel ARGB format with colors stored as Alpha 15, Red 14:10, Green 9:5, and Blue 4:0.
OMX_COLOR_Format16bitBGRA5551	16 bits per pixel BGRA format with colors stored as Blue 15:11, Green 10:6, Red 5:1, and Alpha 0.

OMX_COLOR_FORMATTYPE	Description
OMX_COLOR_Format16bitRGB565	16 bits per pixel RGB format with colors stored as Red 15:11, Green 10:5, and Blue 4:0.
OMX_COLOR_Format16bitBGR565	16 bits per pixel BGR format with colors stored as Blue 15:11, Green 10:5, and Red 4:0.
OMX_COLOR_Format18bitRGB666	18 bits per pixel RGB format with colors stored as Red 17:12, Green 11:6, and Blue 5:0.
OMX_COLOR_Format18BitBGR666	18 bits per pixel BGR format with colors stored as Blue 17:12, Green 11:6, and Red 5:0.
OMX_COLOR_Format18bitARGB1665	18 bits per pixel ARGB format with colors stored as Alpha 17, Red 16:11, Green 10:5, and Blue 4:0.
OMX_COLOR_Format18bitBGRA5661	18 bits per pixel BGRA format with colors stored as Blue 17:13, Green 12:7, Red 6:1, and Alpha 0.
OMX_COLOR_Format19bitARGB1666	19 bits per pixel ARGB format with colors stored as Alpha 18, Red 17:12, Green 11:6, and Blue 5:0.
OMX_COLOR_Format19bitBGRA6661	19 bits per pixel BGRA format with colors stored as Blue 18:13, Green 12:7, Red 6:1, and Alpha 0.
OMX_COLOR_Format24bitRGB888	24 bits per pixel RGB format with colors stored as Red 23:16, Green 15:8, and Blue 7:0.
OMX_COLOR_Format24bitBGR888	24 bits per pixel BGR format with colors stored as Blue 23:16, Green 15:8, and Red 7:0.
OMX_COLOR_Format24bitARGB1887	24 bits per pixel ARGB format with colors stored as Alpha 23, Red 22:15, Green 14:7, and Blue 6:0.
OMX_COLOR_Format24bitBGRA7881	24 bits per pixel BGRA format with colors stored as Blue 23:17, Green 16:9, Red 8:1, and Alpha 0.
OMX_COLOR_Format24BitARGB6666	24 bits per pixel ARGB format with colors stored as Alpha 23:18, Red 17:12, Green 11:6, and Blue 5:0
OMX_COLOR_Format24BitABGR6666	24 bits per pixel ARGB format with colors stored as Alpha 23:18, Blue 17:12, Green 11:6, and Red 5:0

OMX_COLOR_FORMATTYPE	Description
OMX_COLOR_Format24BitBGRA6666	24 bits per pixel BGRA format with colors stored as Blue 23:18, Green 17:12, Red 11:6, and Alpha 5:0.
OMX_COLOR_Format24BitRGBA6666	24 bits per pixel RGBA format with colors stored as Red 23:18, Green 17:12, Blue 11:6, and Alpha 5:0.
OMX_COLOR_Format25bitARGB1888	25 bits per pixel ARGB format with colors stored as Alpha 24, Red 23:16, Green 15:8, and Blue 7:0.
OMX_COLOR_Format25bitBGRA8881	25 bits per pixel BGRA format with colors stored as Blue 24:17, Green 16:9, Red 8:1, and Alpha 0.
OMX_COLOR_Format32bitBGRA8888	32 bits per pixel BGRA format with colors stored as Blue 31:24 Green 23:16, Red 15:8, and Alpha 7:0.
OMX_COLOR_Format32bitARGB8888	24 bits per pixel ARGB format with colors stored as Alpha 31:24, Red 23:16, Green 15:8, and Blue 7:0.
OMX_COLOR_Format32bitABGR8888	32 bits per pixel ABGR format with colors stored as Alpha 31:24, Blue 23:16, Green 15:8, and Red 7:0.
OMX_COLOR_FormatYUV411Planar	YUV planar format, organized with three separate planes for each color component, namely Y, U, and V appearing in this order. U and V pixels are sub-sampled by a factor of four both horizontally and vertically.
OMX_COLOR_FormatYUV411PackedPlanar	YUV planar format, organized with three separate planes for each color component, namely Y, U, and V. U and V pixels are sub-sampled by a factor of four both horizontally and vertically. This format differs from OMX_COLOR_FormatYUV411Planar in that each slice of data shall contain a plane of Y, U, and V data in this order, whereas the OMX_COLOR_FormatYUV411Planar format transfers each plane in its entirety.
OMX_COLOR_FormatYUV420Planar	YUV planar format, organized with three separate planes for each color component, namely Y, U, and V appearing in this order. U and V pixels are sub-sampled by a factor of two both horizontally and vertically.

OMX_COLOR_FORMATTYPE	Description
OMX_COLOR_FormatYUV420PackedPlanar	YUV planar format, organized with three separate planes for each color component, namely Y, U, and V. U and V pixels are sub-sampled by a factor of two both horizontally and vertically. This format differs from OMX_COLOR_FormatYUV420Planar in that each slice of data shall contain a plane of Y, U, and V data in this order, whereas the OMX_COLOR_FormatYUV420Planar format transfers each plane in its entirety.
OMX_COLOR_FormatYUV420SemiPlanar	YUV planar format, organized with a first plane containing Y pixels, and a second plane containing U and V pixels interleaved with the first U value first. U and V pixels are sub-sampled by a factor of two both horizontally and vertically.
OMX_COLOR_FormatYUV420PackedSemiPlanar	YUV planar format, organized with a first plane containing Y pixels, and a second plane containing U and V pixels interleaved with the first U value first. U and V pixels are sub-sampled by a factor of two both horizontally and vertically. This format differs from OMX_COLOR_FormatYUV420SemiPlanar in that each slice of data shall contain a plane of Y, U and V data, whereas the OMX_COLOR_FormatYUV420SemiPlanar format transfers each plane in its entirety.
OMX_COLOR_FormatYVU420Planar	YVU planar format, organized with three separate planes for each color component, namely Y, V, and U appearing in this order. V and U pixels are sub-sampled by a factor of two both horizontally and vertically.

OMX_COLOR_FORMATTYPE	Description
OMX_COLOR_FormatYVU420PackedPlanar	YVU planar format, organized with three separate planes for each color component, namely Y, V, and U. V and U pixels are sub-sampled by a factor of two both horizontally and vertically. This format differs from OMX_COLOR_FormatYVU420Planar in that each slice of data shall contain a plane of Y, V, and U data in this order, whereas the OMX_COLOR_FormatYVU420Planar format transfers each plane in its entirety.
OMX_COLOR_FormatYVU420SemiPlanar	YVU planar format, organized with a first plane containing Y pixels, and a second plane containing V and U pixels interleaved with the first V value first. V and U pixels are sub-sampled by a factor of two both horizontally and vertically.
OMX_COLOR_FormatYVU420PackedSemiPlanar	YVU planar format, organized with a first plane containing Y pixels, and a second plane containing V and U pixels interleaved with the first V value first. V and U pixels are sub-sampled by a factor of two both horizontally and vertically. This format differs from OMX_COLOR_FormatYVU420SemiPlanar in that each slice of data shall contain a plane of Y, V and U data, whereas the OMX_COLOR_FormatYVU420SemiPlanar format transfers each plane in its entirety.
OMX_COLOR_FormatYUV422Planar	YUV planar format, organized with three separate planes for each color component, namely Y, U, and V appearing in this order. U and V pixels are sub-sampled by a factor of two horizontally.

OMX_COLOR_FORMATTYPE	Description
OMX_COLOR_FormatYUV422PackedPlanar	YUV planar format, organized with three separate planes for each color component, namely Y, U, and V. U and V pixels are sub-sampled by a factor of two horizontally. This format differs from OMX_COLOR_FormatYUV422Planar in that each slice of data shall contain a plane of Y, U, and V data in this order, whereas the OMX_COLOR_FormatYUV422Planar format transfers each plane in its entirety.
OMX_COLOR_FormatYUV422SemiPlanar	YUV planar format, organized with a first plane containing Y pixels and a second plane containing U and V pixels interleaved with the first U value first. U and V pixels are sub-sampled by a factor of two horizontally.
OMX_COLOR_FormatYUV422PackedSemiPlanar	YUV planar format, organized with a first plane containing Y pixels, and a second plane containing U and V pixels interleaved with the first U value first. U and V pixels are sub-sampled by a factor of two horizontally. This format differs from OMX_COLOR_FormatYUV422SemiPlanar in that each slice of data shall contain a plane of Y, U and V data, whereas the OMX_COLOR_FormatYUV422SemiPlanar format transfers each plane in its entirety.
OMX_COLOR_FormatYVU422Planar	YVU planar format, organized with three separate planes for each color component, namely Y, V, and U appearing in this order. V and U pixels are sub-sampled by a factor of two horizontally.

OMX_COLOR_FORMATTYPE	Description
OMX_COLOR_FormatYVU422PackedPlanar	YVU planar format, organized with three separate planes for each color component, namely Y, V, and U appearing in this order. V and U pixels are sub-sampled by a factor of two horizontally. This format differs from OMX_COLOR_FormatYVU422Planar in that each slice of data shall contain a plane of Y, V, and U data in this order, whereas the OMX_COLOR_FormatYVU422Planar format transfers each plane in its entirety.
OMX_COLOR_FormatYVU422SemiPlanar	YVU planar format, organized with a first plane containing Y pixels and a second plane containing V and U pixels interleaved with the first V value first. V and U pixels are sub-sampled by a factor of two horizontally.
OMX_COLOR_FormatYVU422PackedSemiPlanar	YVU planar format, organized with a first plane containing Y pixels, and a second plane containing V and U pixels interleaved with the first V value first. V and U pixels are sub-sampled by a factor of two horizontally. This format differs from OMX_COLOR_FormatYVU422SemiPlanar in that each slice of data shall contain a plane of Y, V and U data, whereas the OMX_COLOR_FormatYVU422SemiPlanar format transfers each plane in its entirety.
OMX_COLOR_FormatYCbYCr	16 bits per pixel YUV interleaved format organized as YUYV (i.e., YCbYCr).
OMX_COLOR_FormatYCrYCb	16 bits per pixel YUV interleaved format organized as YVYU (i.e., YCrYCb).
OMX_COLOR_FormatCbYCrY	16 bits per pixel YUV interleaved format organized as UYVY (i.e., CbYCrY).

OMX_COLOR_FORMATTYPE	Description
OMX_COLOR_FormatCrYCbY	16 bits per pixel YUV interleaved format organized as VYUY (i.e., CrYCbY).
OMX_COLOR_FormatYUV444Interleaved	12 bits per pixel YUV format with colors stores as Y 11:8, U 7:4, and V 3:0.
OMX_COLOR_FormatRawBayer8bit	SMIA 8-bit raw Bayer pattern camera format.
OMX_COLOR_FormatRawBayer10bit	SMIA 10-bit raw Bayer pattern camera format.
OMX_COLOR_FormatRawBayer8bitcompressed	SMIA compressed 8-bit camera output format.

4.2.2 Minimum Buffer Payload Size for Uncompressed Data

Uncompressed image and video data have a minimum buffer payload size. The minimum buffer payload size is determined by the `nSliceHeight` and `nStride` fields of the port definition structure. `nStride` indicates the width of a span in bytes; when negative, it indicates the data is bottom-up instead of the top-down). `nSliceHeight` indicates the number of spans in a slice.

The minimum buffer payload size can be easily calculated as the absolute value of `(nSliceHeight * nStride)`.

4.2.3 Buffer Payload Requirements for Uncompressed Data

Each image or video port on a component shall meet several requirements for buffer payloads of uncompressed image and video data. These requirements are in place to enable components from different vendors to inter-operate together correctly, and are collectively referred to as *inter-op*.

The requirements are as follows:

- Each non-empty buffer payload shall contain at least one full slice, unless it contains the end of the image (which may not be aligned to an integer multiple of slice height). For example, if the image height is 100 and the slice height is 16, the last slice of the image will contain only four spans.
- Each non-empty buffer payload shall contain an integer multiple of slice height.
- When the uncompressed image data format is planar, data from two different planes cannot reside in the same buffer payload. This means that a component shall pass a full plane in its entirety in one or more buffers, followed by another plane starting in a different buffer.

  An exception to the above requirement exists for the packed planar uncompressed formats, `OMX_COLOR_FormatYUV420PackedPlanar`, `OMX_COLOR_FormatYUV420PackedSemiPlanar`,

OMX_COLOR_FormatYVU420PackedPlanar, OMX_COLOR_FormatYVU420PackedSemiPlanar, OMX_COLOR_FormatYUV411PackedPlanar, OMX_COLOR_FormatYUV422PackedPlanar, OMX_COLOR_FormatYUV422PackedSemiPlanar,OMX_COLOR_FormatYVU422PackedPlanar, and OMX_COLOR_FormatYVU422PackedSemiPlanar.

For each of these uncompressed formats, each buffer payload contains a slice of the Y, U, and V planes. The slices are always ordered Y, U, and V or Y, V and U - depending on their color format definition.. The nSliceHeight refers to the slice height of the Y plane. The slice height of the U and V planes are derived from the slice height for the Y plane based upon for the format. For example, for OMX_COLOR_FormatYUV420PackedPlanar with an nSliceHeight of 16, a buffer payload shall contain 16 spans of Y followed by 8 spans of U (half the stride) and 8 spans of V (half the stride). This enables ports that process planar data in slices to operate on all three planes simultaneously, instead of forcing the ports to buffer the entire image before processing can begin.

4.2.4 Parameter and Configuration Indexes

The header OMX_Index.h contains the enumeration OMX_INDEXTYPE, which contains all of the standard index values used with the functions OMX_GetParameter, OMX_SetParameter, OMX_GetConfig, and OMX_SetConfig. Table 4-33 describes the index values that relate to video.

Table 4-33: Index Values for Video

Index	Description
OMX_IndexParamCommonDeblocking	Used with OMX_GetParameter and OMX_SetParameter to access OMX_PARAM_DEBLOCKINGTYPE. De-blocking reduces the appearance of block-like artifacts that appear in compressed images or video streams.
OMX_IndexParamCommonSensorMode	Used with OMX_GetParameter and OMX_SetParameter to access OMX_PARAM_SENSORMODETYPE. The mode of the sensor controls the resolution (via OMX_FRAMESIZETYPE) and frame rate of data captured by a camera.
OMX_IndexParamCommonInterleave	Used with OMX_GetParameter and OMX_SetParameter to access OMX_PARAM_INTERLEAVETYPE. This feature is used to interleave plane or input port data.

Index	Description
OMX_IndexConfigCommonColorFormat Conversion	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_COLORCONVERSIONTYPE. Color conversion programs the coefficients used when converting pixel data from RGB to YUV and visa-versa.
OMX_IndexConfigCommonScale	Used with OMX_GetConfig and OMX_SetConfig to access the OMX_CONFIG_SCALEFACTORTYPE. Scaling stretches or shrinks a rectangular region of pixels.
OMX_IndexConfigCommonImageFilter	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_IMAGEFILTERTYPE. Image filtering applies digital effects to a video or image stream.
OMX_IndexConfigCommonColorEnhancement	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_COLORENHANCEMENTTYPE. Color enhancement replaces U and V values of a YUV image with specified constant values to apply a color effect to an image or video stream.
OMX_IndexConfigCommonColorKey	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_COLORKEYTYPE. Color keying performs per-pixel selection between two sources with mixing image or video data.
OMX_IndexConfigCommonColorBlend	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_COLORBLENDTYPE. Color blending performs arithmetic operations between two sources.
OMX_IndexConfigCommonFrame Stabilisation	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_FRAMESTABTYPE.
OMX_IndexConfigCommonRotate	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_ROTATIONTYPE. Rotation rotates video or image frames clockwise by a specified angle.

Index	Description
OMX_IndexConfigCommonMirror	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_MIRRORTYPE. Mirroring reflects video or image frames along the horizontal and vertical axes.
OMX_IndexConfigCommonOutputPosition	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_POINTTYPE. The output position indicates the location of a video or image stream relative to another image or video stream. The output position is also used to indicate the location of a video or image stream relative to an output device such as an LCD display.
OMX_IndexConfigCommonInputCrop	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_RECTTYPE. Crops the image or video stream to the specified rectangle.
OMX_IndexConfigCommonOutputCrop	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_RECTTYPE. Crops the image or video stream to the specified rectangle.
OMX_IndexConfigCommonDigitalZoom	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_SCALEFACTORTYPE. Digital zoom implements a camera zoom feature digitally.
OMX_IndexConfigCommonOpticalZoom	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_SCALEFACTORTYPE. Optical zoom “zooms” an image in or out using a lens on a camera.
OMX_IndexConfigCommonWhiteBalance	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_WHITEBALCONTROLTYPE. White balance performs color correction so that a white object appears truly white and not a tint of the color of the light source.

Index	Description
OMX_IndexConfigCommonExposure	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_EXPOSURECONTROLTYPE. Exposure controls the image sensor exposure when capturing images or streaming video.
OMX_IndexConfigCommonContrast	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_CONTRASTTYPE. Contrast controls the relative difference between pixels in video or image data.
OMX_IndexConfigCommonBrightness	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_BRIGHTNESSTYPE. Brightness controls the luminosity of the pixels in video or image data.
OMX_IndexConfigCommonBacklight	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_BACKLIGHTTYPE. Bacgklight controls the strength of the backlight, and the time that the backlight is turned on.
OMX_IndexConfigCommonGamma	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_GAMMATYPE. Gamma corrects for the non-linear intensity of pixels on a display relative to the digital value of the pixel for video or image data.
OMX_IndexConfigCommonSaturation	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_SATURATIONTYPE. Saturation controls the hue intensity of video or image data.
OMX_IndexConfigCommonLightness	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_LIGHTNESSTYPE. Lightness controls the non-linear response to the brightness of pixels in video or image data.

Index	Description
OMX_IndexConfigCommonExclusionRect	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_RECTTYPE. This feature enables a component to exclude a specific region from rendering to save on processing, resulting in higher performance and lower power consumption. This configuration is often used in video conferencing where a section of the decoded input stream is covered by a preview of the viewer's image.
OMX_IndexConfigCommonPlaneBlend	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_PLANEBLENDTYPE. This feature controls the blending of multiple input sources or ports into a single destination.
OMX_IndexConfigCommonTransitionEffect	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_TRANSITIONEFFECTTYPE.
OMX_IndexConfigCommonDithering	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_DITHERTYPE. Dithering is used when performing color space conversion from a color format that has a higher precision to a color format with a lower precision.
OMX_IndexConfigCommonExposureValue	OMX_CONFIG_EXPOSUREVALUETYPE Query or config the exposure value of the camera.
OMX_IndexConfigCommonOutputSize	OMX_FRAMESIZETYPE Query or config the frame size of an output video sink region.
OMX_IndexParamCommonExtraQuantData	OMX_OTHER_EXTRADATATYPE Used to enable or query the generation of extra payload information consisting of quantization information.
OMX_IndexConfigCaptureMode	OMX_CONFIG_CAPTUREMODETYPE Query or config the capture mode of a camera.

Index	Description
OMX_IndexAutoPauseAfterCapture	OMX_CONFIG_BOOLEANTYPE Query or config the auto pause mechanism after capturing is complete for a camera.
OMX_IndexConfigCapturing	OMX_CONFIG_BOOLEANTYPE Query a component if it is capturing data.
OMX_IndexConfigSharpness	OMX_SHARPNESSTYPE Increasing negative values indicate increased blurriness while increasing positive values indicate increased sharpness.
OMX_IndexConfigCommonExtDigitalZoom	OMX_CONFIG_ZOOMFACTORTYPE. Digital zoom implements a camera zoom feature digitally.
OMX_IndexConfigCommonExtOpticalZoom	OMX_CONFIG_ZOOMFACTORTYPE. Optical zoom implements a camera zoom using optical lens. .
OMX_IndexConfigCommonCenterFieldOfView	OMX_CONFIG_POINTTYPE Configures the field of view that is associated with Digital zoom. Allows setting and querying the center of field of view in case of digital zoom, relatively to the center of the observed scene. By default, the center of the field of view is the center of the observed scene. See OMX_CONFIG_ZOOMFACTORTYPE for more details.
OMX_IndexConfigImageExposureLock	OMX_IMAGE_CONFIG_LOCKTYPE Allows locking the exposure.
OMX_IndexConfigImageWhiteBalanceLock	OMX_IMAGE_CONFIG_LOCKTYPE Allows locking the white balance.
OMX_IndexConfigImageFocusLock	OMX_IMAGE_CONFIG_LOCKTYPE Allows locking the focus.
OMX_IndexConfigCommonFocusRange	OMX_CONFIG_FOCUSRANGETYPE Allows setting the focus range.
OMX_IndexConfigImageFlashStatus This is a read only config.	OMX_IMAGE_CONFIG_FLASHSTATUSTYPE Provides status of the flash (read only).

Index	Description
OMX_IndexConfigCommonExtCaptureMode	OMX_CONFIG_EXTCAPTUREMODETYPE Configures extended capture mode settings.
OMX_IndexConfigCommonNDFilterControl	OMX_CONFIG_NDFILTERCONTROLTYPE Allows controlling the ND Filter functionality.
OMX_IndexConfigCommonAFAssistantLight	OMX_CONFIG_AFASSISTANTLIGHTTYPE Allows controlling a light assistant during camera focusing.
OMX_IndexConfigCommonPortCapturing	OMX_CONFIG_PORTBOOLEANTYPE. Query a component if it is capturing data.
OMX_IndexConfigCommonFocusRegionStatus	OMX_CONFIG_FOCUSREGIONSTATUSTYPE Allows retrieving the status of focusing areas.
OMX_IndexConfigCommonFocusRegionControl	OMX_CONFIG_FOCUSREGIONCONTROLTYPE Allows setting/getting the control of focusing areas.
OMX_IndexParamInterlaceFormat	OMX_INTERLACEFORMATTYPE Used to query the supported interlace formats.
OMX_IndexConfigDeInterlace	OMX_DEINTERLACETYPE Used to enable\disable deinterlacing support.
OMX_IndexConfigStreamInterlaceFormats	OMX_STREAMINTERLACEFORMATTYPE Used to query if the stream contains interlace or progressive content.

4.2.5 OMX_PARAM_DEBLOCKINGTYPE

De-blocking is used to reduce the appearance of block-like artifacts that appear in compressed images or video streams.

OMX_PARAM_DEBLOCKINGTYPE is defined as follows.

```
typedef struct OMX_PARAM_DEBLOCKINGTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bDeblocking;
} OMX_PARAM_DEBLOCKINGTYPE;
```

4.2.5.1 Parameters

The parameters for OMX_PARAM_DEBLOCKINGTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bDeblocking is a Boolean value that enables or disables de-blocking.

4.2.6 OMX_PARAM_INTERLEAVETYPE

Interleaving is used to interleave or de-interleave pixel data between multiple ports. When interleaving, a component uses pixel data from multiple input ports to merge into a single output port. When de-interleaving, a component uses pixel data from a single input port, splitting the color channels into separate output ports.

For example, an input port receiving 16-bit RGB can de-interleave R, G, and B color channels to three separate output ports, where the output ports are formatted as monochrome.

Similarly, a component could interleave three luminance ports containing Y, U, and V data into a single output port formatted as YUV420.

The OMX_PARAM_INTERLEAVETYPE structure interleaves pixel data. OMX_PARAM_INTERLEAVETYPE is defined as follows.

```
typedef struct OMX_PARAM_INTERLEAVETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bEnable;
    OMX_U32 nInterleavePortIndex;
} OMX_PARAM_INTERLEAVETYPE;
```

4.2.6.1 Parameters

The parameters for OMX_PARAM_INTERLEAVETYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bEnable is a Boolean value that enables interleaving.
- nInterleavePortIndex indicates the port to interleave or de-interleave with. When nPortIndex is an input port, nInterleavePortIndex contains the

output port to interleave with. When `nPortIndex` is an output port, `nInterleavePortIndex` contains the input port to de-interleave with.

4.2.7 OMX_PARAM_SENSORMODETYPE

The sensor mode is used to specify the frame rate and resolution that an image sensor or camera uses to capture image or video. The sensor mode is distinctly separate from the port on a video source, which may modify the resolution of the data produced by the image sensor.

`OMX_PARAM_SENSORMODETYPE` is defined as follows.

```
typedef struct OMX_PARAM_SENSORMODETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nFrameRate;
    OMX_BOOL bOneShot;
    OMX_FRAMESIZETYPE sFrameSize;
} OMX_PARAM_SENSORMODETYPE;
```

4.2.7.1 Parameters

The parameters for `OMX_PARAM_SENSORMODETYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `nFrameRate` is the frame rate is in frames per second. This value is represented in Q16 format. The value 0x0 is used to indicate the frame rate is unknown, variable, or is not needed.
- `bOneShot` is a Boolean value that enables or disables one shot mode.
- `sFrameSize` is the resolution of the image sensor mode provided in the form of OMX_FRAMESIZETYPE.

4.2.8 OMX_FRAMESIZETYPE

Frame size is a generic structure used to indicate the size of a frame. This structure is referred to by the `OMX_PARAM_SENSORMODETYPE` structure.

`OMX_FRAMESIZETYPE` is defined as follows.

```
typedef struct OMX_FRAMESIZETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nWidth;
    OMX_U32 nHeight;
} OMX_FRAMESIZETYPE;
```

4.2.8.1 Parameters

The parameters for `OMX_FRAMESIZETYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `nWidth` is the width of the rectangle in pixels.
- `nHeight` is the height of the rectangle in pixels.

4.2.9 OMX_CONFIG_COLORCONVERSIONTYPE

Color conversion is used to specify the coefficients when converting image or video pixel data from YUV to RGB and visa-versa.

Converting from RGB to YUV format uses the following standard formulae:

$$Y = 0.299R + 0.587G + 0.114B$$

$$U = -0.147R - 0.289G + 0.436B$$

$$V = 0.615R - 0.515G - 0.100B$$

Converting from YUV to RGB format uses the following standard formulae:

$$R = Y + 1.140V$$

$$G = Y - 0.395U - 0.581V$$

$$B = Y + 2.032U$$

The color matrix and color offset specified in the color conversion allow for the coefficients used when converting from RGB to YUV and visa-versa to be programmed explicitly.

`OMX_CONFIG_COLORCONVERSIONTYPE` is defined as follows.

```
typedef struct OMX_CONFIG_COLORCONVERSIONTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_S32 xColorMatrix[3][3];
    OMX_S32 xColorOffset[4];
}OMX_CONFIG_COLORCONVERSIONTYPE;
```

4.2.9.1 Parameters

The parameters for `OMX_CONFIG_COLORCONVERSIONTYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `xColorMatrix[3][3]` is the color conversion matrix when converting from RGB to YUV in Q16 format with the following standard formulae:

$$Y = Yr*R + Yg*G + Yb*B$$
$$U = Ur*R – Ug*G + Ub*B$$
$$V = Vr*R – Vg*G – Vb*B$$

Each constant is represented in the 3x3 matrix as:

Yr Yg Yb
Ur Ug Ub
Vr Vg Vb

Y constants are in the first row, followed by U and V constants in subsequent rows. All constants multiplied against red color values are in the first column followed by green and blue color constants, as follows

```
xColorMatrix[1][1] = Yr
xColorMatrix[3][3] = Vb,
xColorMatrix[1][3] = Yb
```

- `xColorOffset[4]` is the color conversion vector when converting from YUV to RGB in Q16 format. The standard formulae are as follows:

$$R = Y + C1*U$$
$$G = Y – C2*U – C3*V$$
$$B = Y – C4*V$$

Each constant is represented in the array:

C1 C2 C3 C4

4.2.10 OMX_CONFIG_SCALEFACTORTYPE

Scaling is used to stretch or shrink video or image data on the specified input or output port.

`OMX_CONFIG_SCALEFACTORTYPE` is defined as follows.

```
typedef struct OMX_CONFIG_SCALEFACTORTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_S32 xWidth;
    OMX_S32 xHeight;
}OMX_CONFIG_SCALEFACTORTYPE;
```

4.2.10.1 Parameters

The parameters for `OMX_CONFIG_SCALEFACTORTYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `xWidth` is the scaling in the horizontal direction in Q16 format (i.e., signed 15.16 fixed pointed format). For example, a scaling factor of 0x10000 would not change the width, but a scaling factor of 0x8000 would scale the width by 50%.

- `xHeight` is the scaling in the vertical direction in Q16 format (i.e., signed 15.16 fixed pointed format). For example, a scaling factor of 0x10000 would not change the height, but a scaling factor of 0x20000 would scale the height by 200%.

4.2.11 OMX_CONFIG_IMAGEFILTERTYPE

Image filtering is used to apply digital effects to video or image data on the specified port.

`OMX_CONFIG_IMAGEFILTERTYPE` is defined as follows.

```
typedef struct OMX_CONFIG_IMAGEFILTERTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_IMAGEFILTERTYPE eImageFilter;
} OMX_CONFIG_IMAGEFILTERTYPE;
```

4.2.11.1 Parameters

The parameters for `OMX_CONFIG_IMAGEFILTERTYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `eImageFilter` is the enumerated valued indicating the image filter used. Table 4-34 details the values that can be selected for the image filter.

Table 4-34: Image Filter Values

`OMX_IMAGEFILTERTYPE` Enumerated Value	**Description**
`OMX_ImageFilterNone`	Used to disable image filtering.
`OMX_ImageFilterNoise`	Filters data to remove noise from the image.
`OMX_ImageFilterEmboss`	Filters data to give an embossed appearance (stamped from the rear for a raised effect along edges).
`OMX_ImageFilterNegative`	Filters data to negate colors.
`OMX_ImageFilterSketch`	Filters data to give the appearance of having been sketched by an artist.
`OMX_ImageFilterOilPaint`	Filters data to appear as if it were hand painted using a brush with oil paints.
`OMX_ImageFilterHatch`	Filters data to appear as if it were printed on a material with a grain.
`OMX_ImageFilterGpen`	Filters data to appear as if it were drawn with a pen.
`OMX_ImageFilterAntialias`	Filters data to anti-alias pixels so as to sharpen edges in the image or video stream.
`OMX_ImageFilterDeRing`	Filters data to remove erroneous artifacts introduced by inherent limitations of the numerical processing of digital image data.
`OMX_ImageFilterSolarize`	Filters data to create a solarization effect.

OMX_IMAGEFILTERTYPE Enumerated Value	Description
OMX_ImageFilterPastel	Filters data to provide a pastel appearance.
OMX_ImageFilterMosaic	Filters data to provide a mosaic appearance.
OMX_ImageFilterPosterize	Filters data to replace gradual transitions of tone with abrupt changes in grade and shading.
OMX_ImageFilterWhiteboard	Filters data to emphasize symbols written on a whiteboard.
OMX_ImageFilterBlackboard	Filters data to emphasize symbols written on a blackboard.
OMX_ImageFilterSepia	Filters data to provide a sepia appearance.
OMX_ImageFilterGrayscale	Filters data to provide a gray scale appearance – Black and White.
OMX_ImageFilterNatural	Filters data to provide a natural appearance.
OMX_ImageFilterVivid	Filters data to provide a vivid appearance.
OMX_ImageFilterWaterColor	Filters data to provide a water color appearance.
OMX_ImageFilterFilm	Filters data to provide a film appearance.

4.2.12 OMX_CONFIG_COLORENHANCEMENTTYPE

Color enhancement is applied to image or video data in YUV formats, where the U and V color components of each pixel are replaced with the specified values. Replacement occurs for each pixel and every frame. This enables a component to add specified color hues to the data. For example, this configuration can be used to convert color image or video data to sepia tone.

OMX_CONFIG_COLORENHANCEMENTTYPE is defined as follows.

```
typedef struct OMX_CONFIG_COLORENHANCEMENTTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bColorEnhancement;
    OMX_U8 nCustomizedU;
    OMX_U8 nCustomizedV;
 } OMX_CONFIG_COLORENHANCEMENTTYPE;
```

4.2.12.1 Parameters

The parameters for OMX_CONFIG_COLORENHANCEMENTTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bColorEnhancement is the Boolean value that enables or disables color enhancement.
- nCustomizedU is a value for replacing the U color component of each pixel. The range of values is 0-255. Practical values are in the range of 16-240.

- nCustomizedV is the value for replacing the V color component of each pixel. The range of values is 0-255. Practical values are in the range of 16-240.

4.2.13 OMX_CONFIG_COLORKEYTYPE

Color keying is used to perform per-pixel selection between two sources when mixing image or video data.

OMX_CONFIG_COLORKEYTYPE is defined as follows.

```
typedef struct OMX_CONFIG_COLORKEYTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nARGBColor;
    OMX_U32 nARGBMask;
} OMX_CONFIG_COLORKEYTYPE;
```

4.2.13.1 Parameters

The parameters for OMX_CONFIG_COLORKEYTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nARGBColor indicates a 32-bit color used for keying, where bits 0-7 are blue, bits 15-8 are green, bits 24-16 are red, and bits 31-24 are for alpha. The 32-bit ARGB color is converted to the RGB color format of the port before performing keying operations.
- nARGBMask indicates a 32-bit logical AND mask, which is converted to the RGB color format of the port before performing keying operations.

4.2.14 OMX_CONFIG_COLORBLENDTYPE

Color blending is used to perform arithmetic operations between two sources when mixing image or video data. If more than one input port (representing a plane) on a component is using this config, it should be used in conjunction with OMX_CONFIG_PLANEBLENDTYPE to specify the Z-order of the different ports via the nDepth field.

OMX_CONFIG_COLORBLENDTYPE is defined as follows.

```
typedef struct OMX_CONFIG_COLORBLENDTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nRGBAlphaConstant;
    OMX_COLORBLENDTYPE eColorBlend;
} OMX_CONFIG_COLORBLENDTYPE;
```

4.2.14.1 Parameters

The parameters for OMX_CONFIG_COLORBLENDTYPE are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `nRGBAlphaConstant` is the 32-bit per color channel constant alpha value for blending when the `eColorBlend` is set to `OMX_ColorBlendAlphaConstant` on an input port. If defined on an output port, the `nRGBAlphaConstant` value is written as the per pixel alpha value in the composed image (if the output format supports per pixel alpha). If `eColorBlend` is `OMX_ColorBlendAlphaPerPixel` is defined, the `nRGBAlphaConstant` value is ignored and the alpha coefficients for the output buffer are taken from the corresponding alpha values of the lowest `nDepth` (=highest value) input plane.
  A value of 0 means fully transparent and a value of 1 (0xFFFFFFFF) means opaque.
- `eColorBlend` is the enumerated value indicating the color blend operation used. `eColorBlend` is only valid when set on ports representing the image source input (highest `nDepth` (=lowest value) plane) or on the composed plane. If set on an output port, assuming the output format supports per pixel alpha, the `nRGBAlphaConstant` value is taken (with `eColorBlend = OMX_ColorBlendAlphaConstant`) or the alpha value of the lowest `nDepth` plane is taken (`eColorBlend = OMX_ColorBlendAlphaPerPixel`), as per pixel alpha value in the composed image. Note in the latter case a) if the input (alpha) format does not equal the composed image (alpha) format, the implicit color space conversion takes care of re-calculating the alpha value, and b) if the input format does not have an alpha value, the per pixel alpha value of the composed plane is set to non-transparent. Table 4-35 details the values that can be selected for color blending.

Table 4-35: Color Blending Values

OMX_COLORBLENDTYPE Enumerated Value	Description
`OMX_ColorBlendNone`	Disables color blending.
`OMX_ColorBlendAlphaConstant`	Blends source and destination using the function (alpha_constant * source) + ((1 – alpha_constant) * destination), where the alpha constant is specified for the entire operation.
`OMX_ColorBlendAlphaPerPixel`	Blends source and destination using the function (alpha * source) + ((1 – alpha) * destination), where the alpha value is per pixel.
`OMX_ColorBlendAlternate`	Alternates between selecting source and destination pixels (i.e., checkerboard of source and destination pixels).

OMX_COLORBLENDTYPE Enumerated Value	Description
OMX_ColorBlendAnd	Combines source and destination pixels using the function (source & destination).
OMX_ColorBlendOr	Combines source and destination pixels using the function (source | destination).
OMX_ColorBlendInvert	Combines source and destination pixels using the function ~(source).

4.2.15 OMX_CONFIG_FRAMESTABTYPE

Frame stabilization reduces motion blur during image capture or video recording. Frame stabilization is most often associated with camera sensor source components, a camera sensor filter, or a digital signal processor (DSP).

The frame stabilization feature compensates for the extremely unsteady nature of cameras on handheld devices such as a cell phone or personal digital assistant (PDA).

OMX_CONFIG_FRAMESTABTYPE is defined as follows.

```
typedef struct OMX_CONFIG_FRAMESTABTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bStab;
} OMX_CONFIG_FRAMESTABTYPE;
```

4.2.15.1 Parameters

The parameters for OMX_CONFIG_FRAMESTABTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bStab is the Boolean value that enables or disables frame stabilization.

4.2.16 OMX_CONFIG_ROTATIONTYPE

Rotation is applied to image or video data on a specified port. Components may support rotation only on right angles such as 0°, 90°, 180°, and 270°, although components may support arbitrary rotation angles. Values are interpreted as clockwise.

OMX_CONFIG_ROTATIONTYPE is defined as follows.

```
typedef struct OMX_CONFIG_ROTATIONTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_S32 nRotation;
} OMX_CONFIG_ROTATIONTYPE;
```

4.2.16.1 Parameters

The parameters for OMX_CONFIG_ROTATIONTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nRotation is an integer value that represents the angle of rotation. Some components may only support rotation on right angles such as 0°, 90°, 180°, and 270°. Positive value of nRotation indicates a clockwise rotation.

4.2.17 OMX_CONFIG_MIRRORTYPE

Mirroring is applied to pixel or image data on a specified port. The data can be mirrored in the horizontal direction, vertical direction, or both horizontal and vertical directions.

OMX_CONFIG_MIRRORTYPE is defined as follows.

```
typedef struct OMX_CONFIG_MIRRORTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_MIRRORTYPE eMirror;
} OMX_CONFIG_MIRRORTYPE;
```

4.2.17.1 Parameters

The parameters for OMX_CONFIG_MIRRORTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- eMirror contains the enumerated values indicating the mirroring applied to image or video data. OMX_MirrorNone is used to disable mirroring or have no mirroring. Table 4-36 identifies the mirroring values.

Table 4-36: Mirror Type Values

OMX_MIRRORTYPE Enumerated Value	**Description**
OMX_MirrorNone	Disables mirroring (i.e., no mirroring).
OMX_MirrorHorizontal	Mirrors pixels in the horizontal direction. Hence, pixel at 0,1 is swapped with pixel W,1 where W is the width of the image.
OMX_MirrorVertical	Mirrors pixels in the vertical direction. Hence, pixel at 1,0 is swapped with pixel 1,H where H is the height of the image.

OMX_MIRRORTYPE Enumerated Value	Description
OMX_MirrorBoth	Mirrors pixels in the horizontal and vertical directions. Hence, pixel at 0, 0 is swapped with pixel W,H where W is the width of the image and H is the height of the image.

4.2.18 OMX_CONFIG_POINTTYPE

A point is used to specify the location of image or video data on a port relative to another source image or video stream.

OMX_CONFIG_POINTTYPE is defined as follows.

```
typedef struct OMX_CONFIG_POINTTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_S32 nX;
    OMX_S32 nY;
} OMX_CONFIG_POINTTYPE;
```

4.2.18.1 Parameters

The parameters for OMX_CONFIG_POINTTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nX is the X-coordinate location in pixels in the horizontal direction with respect to the origin (0,0) located in the top-left corner of the image.
- nY is the Y-coordinate location in pixels in the vertical direction with respect to the origin (0,0) located in the top-left corner of the image.

4.2.19 OMX_CONFIG_RECTTYPE

Rectangles are used with several configuration types to indicate orientation, position, inclusion, or exclusion.

OMX_CONFIG_RECTTYPE is defined as follows.

```
typedef struct OMX_CONFIG_RECTTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_S32 nLeft;
    OMX_S32 nTop;
    OMX_U32 nWidth;
    OMX_U32 nHeight;
} OMX_CONFIG_RECTTYPE;
```

4.2.19.1 Parameters

The parameters for OMX_CONFIG_RECTTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nLeft is the leftmost coordinate of the rectangle.
- nTop is the topmost coordinate of the rectangle.
- nWidth is the width of the rectangle in pixels.
- nHeight is the height of the rectangle in pixels.

nLeft and nTop coordinate values are with respect to the origin (0,0) located in the top-left corner of the image.

4.2.20 OMX_CONFIG_WHITEBALCONTROLTYPE

White balance control is used with camera sensors to adjust the color temperature of the image so that pure white appears as white in the image. This adjustment can be controlled automatically or manually.

OMX_CONFIG_WHITEBALCONTROLTYPE is defined as follows.

```
typedef struct OMX_CONFIG_WHITEBALCONTROLTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_WHITEBALCONTROLTYPE eWhiteBalControl;
} OMX_CONFIG_WHITEBALCONTROLTYPE;
```

4.2.20.1 Parameters

The parameters for OMX_CONFIG_WHITEBALCONTROLTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- eWhiteBalControl is the enumerated valued indicating the type of white balance control used. Table 4-37 details the values that can be selected for white balance control.

Table 4-37: White Balance Control

OMX_WHITEBALCONTROLTYPE Enumerated Value	Description
OMX_WhiteBalControlOff	Disables exposure control.
OMX_WhiteBalControlAuto	Automatic white balance control. The color temperature of the captured image or video stream is adjusted per frame using a white reference from within each frame.

OMX_WHITEBALCONTROLTYPE Enumerated Value	Description
OMX_WhiteBalControlSunLight	Manual white balance control when the sun provides the light source.
OMX_WhiteBalControlCloudy	Manual white balance control when the sun provides the light source through clouds.
OMX_WhiteBalControlShade	Manual white balance control when the light source is the sun and the scene is in the shade.
OMX_WhiteBalControlTungsten	Manual white balance control when the light source is tungsten.
OMX_WhiteBalControlFluorescent	Manual white balance control when the light source is fluorescent.
OMX_WhiteBalControlIncandescent	Manual white balance control when the light source is incandescent.
OMX_WhiteBalControlFlash	Manual white balance control when the light source is a flash.
OMX_WhiteBalControlHorizon	Manual white balance control when the light source is the sun on the horizon.

4.2.21 OMX_CONFIG_EXPOSURECONTROLTYPE

Exposure is used to control the image sensor exposure when capturing images or streaming video.

OMX_CONFIG_EXPOSURECONTROLTYPE is defined as follows.

```
typedef struct OMX_CONFIG_EXPOSURECONTROLTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_EXPOSURECONTROLTYPE eExposureControl;
} OMX_CONFIG_EXPOSURECONTROLTYPE;
```

4.2.21.1 Parameters

The parameters for OMX_CONFIG_EXPOSURECONTROLTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- eExposureControl is an enumerated value that selects the type of exposure used. Table 4-38 details the values that can be selected for exposure.

Table 4-38: Exposure Control

OMX_EXPOSURECONTROLTYPE Enumerated Value	Description
OMX_ExposureControlOff	Disables exposure control

OMX_EXPOSURECONTROLTYPE Enumerated Value	Description
OMX_ExposureControlAuto	Automatic exposure
OMX_ExposureControlNight	Exposure at night
OMX_ExposureControlBackLight	Exposure with backlight illuminating the subject
OMX_ExposureControlSpotLight	Exposure with a spotlight illuminating the subject
OMX_ExposureControlSports	Exposure for sports
OMX_ExposureControlSnow	Exposure for the subject in snow
OMX_ExposureControlBeach	Exposure for the subject at a beach
OMX_ExposureControlLargeAperture	Exposure when using a large aperture on the camera
OMX_ExposureControlSmallApperture	Exposure when using a small aperture on the camera

4.2.22 OMX_CONFIG_CONTRASTTYPE

Contrast controls the relative difference between the pixels. Contrast is applied to image or video data on the specified port.

OMX_CONFIG_CONTRASTTYPE is defined as follows.

```
typedef struct OMX_CONFIG_CONTRASTTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_S32 nContrast;
 } OMX_CONFIG_CONTRASTTYPE;
```

4.2.22.1 Parameters

The parameters for OMX_CONFIG_CONTRASTTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nContrast is the value for contrast. The range of values is -100 to 100. The value 0x0 indicates no contrast change to pixel data.

4.2.23 OMX_CONFIG_BRIGHTNESSTYPE

Brightness controls the luminosity of the pixels in the video or image data. Brightness is applied to the image or video data on the specified port.

OMX_CONFIG_BRIGHTNESSTYPE is defined as follows.

```
typedef struct OMX_CONFIG_BRIGHTNESSTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nBrightness;
} OMX_CONFIG_BRIGHTNESSTYPE;
```

4.2.23.1 Parameters

The parameters for OMX_CONFIG_BRIGHTNESSTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nBrightness is the value for brightness in the range 0% to 100%, where 0% produces all black pixels and 100% produces entirely white.

4.2.24 OMX_CONFIG_BACKLIGHTTYPE

The backlight of a flat panel type of display such as a liquid crystal display (LCD) or a thin film transistor (TFT) panel can be controlled using this configuration setting. The IL client sets the percentage brightness of the backlight and the timeout before the backlight automatically turns off.

OMX_CONFIG_BACKLIGHTTYPE is defined as follows.

```
typedef struct OMX_CONFIG_BACKLIGHTTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nBacklight;
    OMX_U32 nTimeout;
} OMX_CONFIG_BACKLIGHTTYPE;
```

4.2.24.1 Parameters

The parameters for OMX_CONFIG_BACKLIGHTTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nBacklight is a value that represents the backlight brightness. The range of values is 0% to 100%, where 0% is completely off and 100% is full backlight intensity.
- nTimeout is the number of milliseconds before the backlight automatically turns off. A value of 0x0 forces the backlight to remain on.

4.2.25 OMX_CONFIG_GAMMATYPE

Gamma is applied to the image or pixel data on the specified port to correct for the non-linear response to the brightness of pixels on a display relative to the digital value of the pixel. Gamma correction is typically applied when data is captured digitally by a camera source, or when data is shown on a display device such as a panel, CRT, or TV.

OMX_CONFIG_GAMMATYPE is defined as follows.

```
typedef struct OMX_CONFIG_GAMMATYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_S32 nGamma;
} OMX_CONFIG_GAMMATYPE;
```

4.2.25.1 Parameters

The parameters for OMX_CONFIG_GAMMATYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nGamma is the display gamma expressed in Q16 format (usually in the 2.0 to 4.0 range). The value 0 is not allowed. The details of how gamma correction is done are implementation-specific.

  In general, an exponential relationship between the input and output pixel intensities is assumed (i.e. Vout = Vin^nGamma) and the gamma correction component is assumed to apply an inverse transfer function (i.e. Vgamma = Vin^(1/nGamma)). It is also assumed that the same nGamma value applies to all three color channels.

4.2.26 OMX_CONFIG_SATURATIONTYPE

Saturation is applied to image or pixel data on the specified port to control the hue intensity.

OMX_CONFIG_SATURATIONTYPE is defined as follows.

```
typedef struct OMX_CONFIG_SATURATIONTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_S32 nSaturation;
} OMX_CONFIG_SATURATIONTYPE;
```

4.2.26.1 Parameters

The parameters for OMX_CONFIG_SATURATIONTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nSaturation is the value for saturation. The range of values is -100 to 100. The value 0x0 indicates no saturation change to pixel data. A value of -100 produces all black pixels, and a value of 100 produces all white pixels.

4.2.27 OMX_CONFIG_LIGHTNESSTYPE

Lightness is applied to image or pixel data on the specified port to control the non-linear response to the brightness of pixels.

OMX_CONFIG_LIGHTNESSTYPE is defined as follows.

```
typedef struct OMX_CONFIG_LIGHTNESSTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_S32 nLightness;
} OMX_CONFIG_LIGHTNESSTYPE;
```

4.2.27.1 Parameters

The parameters for OMX_CONFIG_LIGHTNESSTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nLightness is the value for lightness. The range of values is -100 to 100. The value 0x0 indicates no lightness change to pixel data. A value of -100 produces all black pixels, and a value of 100 produces all white pixels.

4.2.28 OMX_CONFIG_PLANEBLENDTYPE

Plane blending is used to blend pixels from multiple sources into a single destination. The plane depth is specified such that planes with lower numbers are on top of planes with higher numbers. The blending of two planes with the same depth is undefined.

OMX_CONFIG_PLANEBLENDTYPE is defined as follows.

```
typedef struct OMX_CONFIG_PLANEBLENDTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nDepth;
    OMX_U32 nAlpha;
} OMX_CONFIG_PLANEBLENDTYPE;
```

4.2.28.1 Parameters

The parameters for OMX_CONFIG_PLANEBLENDTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nDepth is the depth of the plane for the port. Lower values indicate higher planes, and higher values indicate lower planes. By default, the depth value is the same as the value of nPortIndex. The nDepth is only valid when set on an input port and ignored when applied to an output port.
- nAlpha indicates the alpha value used when blending planes, if the blending operation uses global alpha. When defined on an input port, the default blending operation is (source_alpha * source_color) + ((1 – source_alpha) * destination_color)), where the source is the plane associated with the config and the destination is the blended result of all lower planes. If OMX_CONFIG_COLORBLENDTYPE is defined on the output port, the associated

eColorBlend variable is used to determine the blending equation. For information on blending operations, see section 4.2.14. If defined on an output port, the nAlpha value is written as the per pixel alpha value in the end image (if the output format supports per pixel alpha), after performing the regular alpha calculations from the input ports if defined in combination.

4.2.29 OMX_CONFIG_TRANSITIONEFFECTTYPE

A component may support producing output image or video frames based on two input frames, where the sequence of the output frames forms a transition from one input frame to the next.

OMX_CONFIG_TRANSITIONEFFECTTYPE is defined as follows.

```
typedef struct OMX_CONFIG_TRANSITIONEFFECTTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_TRANSITIONEFFECTTYPE eEffect;
} OMX_CONFIG_TRANSITIONEFFECTTYPE;
```

4.2.29.1 Parameters

The parameters for OMX_CONFIG_TRANSITIONEFFECTTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to .
- eEffect is the enumerated value indicating the transition effect to be used to generate the output frames. details the possible values for transition effects.

Table 4-39: eEffect Values

OMX_TRANSITIONEFFECTTYPE value	Transition Description
OMX_EffectNone	Used to disable or cancel the current transition effect.
OMX_EffectFadeFromBlack	Fades from a solid black frame to the desired input frame.
OMX_EffectFadeToBlack	Fades from the desired input frame to a solid black frame.
OMX_EffectUnspecifiedThroughConstantColor	A vendor specific effect from the first input frame to the second using a constant color frame mid transition.
OMX_EffectDissolve	Dissolves from the first input frame to the second.

OMX_TRANSITIONEFFECTTYPE value	Transition Description
OMX_EffectWipe	Wipes from the first input frame to the second.
OMX_EffectUnspecifiedMixOfTwoScenes	A vendor specific effect from the first input frame to the second. If multiple vendor effects are available, a random one may be chosen.

4.2.30 OMX_CONFIG_DITHERTYPE

Dithering is used when performing color format conversion where the source color format has higher precision than the destination color format. Two standard types of dithering are supported: OMX_DitherOrdered and OMX_DitherErrorDiffusion. OMX_DitherOther provides a means for vendor-specific dithering algorithms.

OMX_CONFIG_DITHERTYPE is defined as follows.

```
typedef struct OMX_CONFIG_DITHERTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_DITHERTYPE eDither;
 } OMX_CONFIG_DITHERTYPE;
```

4.2.30.1 Parameters

The parameters for OMX_CONFIG_DITHERTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- eDither is the type of dithering used when performing color format conversion. Table 4-40 details the values that can be selected for dithering.

Table 4-40: Dithering Values

OMX_DITHERTYPE Enumerated Value	Description
OMX_DitherNone	Disables dithering
OMX_DitherOrdered	Enables ordered dithering
OMX_DitherErrorDiffusion	Enables error diffusion dithering
OMX_DitherOther	Enables a vendor specific dithering algorithm

4.2.31 OMX_CONFIG_EXPOSUREVALUETYPE

Exposure is the amount of light which falls upon the sensor of a digital camera. Shutter speed, sensitivity, and aperture are adjusted to achieve optimal exposure of a scene. Most digital cameras offer a variety of exposure modes, from fully-automatic to semi-automatic to full manual mode.

OMX_CONFIG_EXPOSUREVALUETYPE is defined as follows.

```
typedef struct OMX_CONFIG_EXPOSUREVALUETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_METERINGTYPE eMetering;
    OMX_S32 xEVCompensation;
    OMX_U32 nApertureFNumber;
    OMX_BOOL bAutoAperture;
    OMX_U32 nShutterSpeedMsec;
    OMX_BOOL bAutoShutterSpeed;
    OMX_U32 nSensitivity;
    OMX_BOOL bAutoSensitivity;
 } OMX_CONFIG_EXPOSUREVALUETYPE;
```

4.2.31.1 Parameters

The parameters for OMX_CONFIG_EXPOSUREVALUETYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- eMetering is the metering type to be used. Table 4-41 lists the valid metering modes that can be used.

Table 4-41: Metering Modes

OMX_METERINGTYPE Enumerated Value	**Description**
OMX_MeteringModeAverage	Center weight average metering
OMX_MeteringModeSpot	Spot (partial) metering
OMX_MeteringModeMatrix	Matrix or evaluative metering

- xEVCompensation is the Exposure Value compensation defined in Q16 format.
- nApertureFNumber is the aperture f-stop setting defined in Q16 format. A value of 2 implies a "f/2" setting. This setting is only valid for SetConfig if auto aperture mode is disabled (via bAutoAperture) .
- bAutoAperture is a Boolean value indicating if auto-aperture is to be enabled and applied.
- nShutterSpeedMsec is the shutter speed specified in units of milliseconds. This setting is only valid for SetConfig if auto shutter speed is disabled (via bAutoShutterSpeed).

- bAutoShutterSpeed is a Boolean value indicating if auto shutter speed is to be enabled and applied.
- nSensitivity is the ISO sensitivity setting. A value of 100 implies a "ISO 100" setting. This setting is only valid for SetConfig if auto sensitivity is disabled (via bAutoSensitivity).
- bAutoSensitivity is a Boolean value indicating if auto sensitivity is to be enabled and applied.

4.2.32 OMX_OTHER_EXTRADATATYPE

The OMX_OTHER_EXTRADATATYPE structure is used to describe the additional buffer payload information included within the buffer. A buffer may contain multiple blocks of extra data and thus multiple instances of this structure.

Each additional EXTRADATATYPE structure shall be required to be 32 bit address aligned, and padding bytes may need to be inserted in order to ensure this alignment.

The order of the additional information is not required to be pre-determined since a component is expected to traverse the OMX_OTHER_EXTRADATATYPE structures to determine the additional information of interest.

OMX_OTHER_EXTRADATATYPE is defined as follows.

```
typedef struct OMX_OTHER_EXTRADATATYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_EXTRADATATYPE eType;
    OMX_U32 nDataSize;
    OMX_U8 data[1];
} OMX_OTHER_EXTRADATATYPE;
```

4.2.32.1 Parameters

The parameters for OMX_OTHER_EXTRADATATYPE are defined as follows.

- nSize is the size of the structure including data bytes and any padding necessary to ensure 32bit alignment of the next OMX_OTHER_EXTRADATATYPE structure.
- nPortIndex represents the port that this structure applies to.
- eType identifies the extra data payload type. Table 4-42 details the values that can be selected for extra data payload type.

Table 4-42: Extra Data Payload Type Enumerated values

Enumerated Value	Description
OMX_ExtraDataNone	Indicates that this terminates the list of extra data sections.
OMX_ExtraDataQuantization	Indicates that the data payload contains quantization data.

Enumerated Value	Description
OMX_ExtraDataInterlaceFormat	Specifies the interlaced format packing of the video frame. The data payload contains an the interlace format associated with the payload - the interlace formats are described in Table 4-50. This information shall be included for all interlaced content being emitted by a port.

- nDataSize identifies the size of supporting data in units of bytes. For the OMX_OTHER_EXTRADATATYPE structure that terminates the list of extra data sections, nDataSize will be zero.
- data is an array of one or more bytes of data as indicated by the nDataSize field.

4.2.32.2 Sample code

The following diagram shows the arrangement of extra data sections in a buffer.

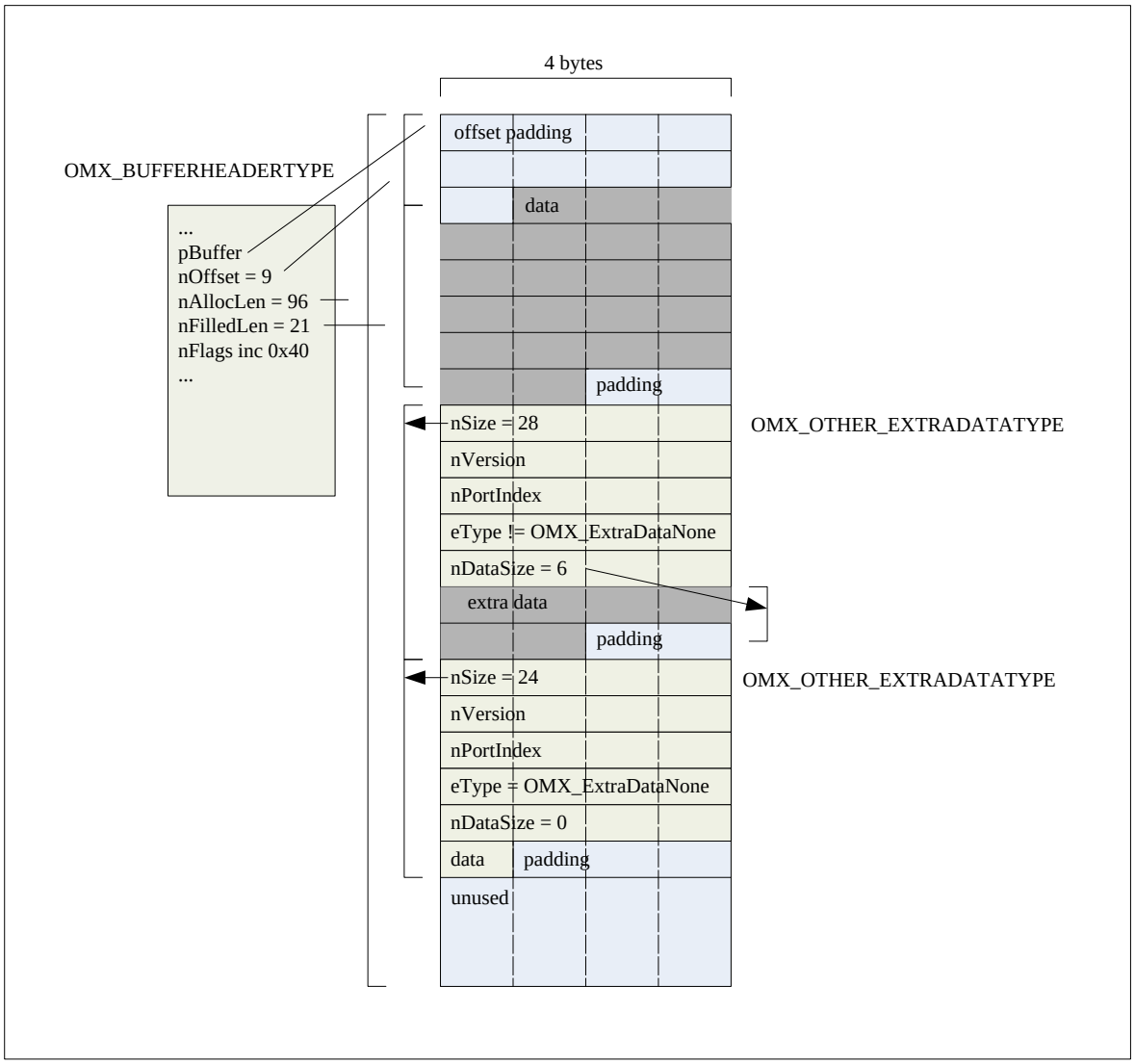


Figure 4-3. Formatting of Extra Buffer Data

The following code sequence shows traversing the list of extra data sections.

```
/* Traverse the list of extra data sections */
  OMX_OTHER_EXTRADATATYPE *pExtra;
  OMX_U8 *pTmp = pBufferHdr->pBuffer + pBufferHdr->nOffset +
pBufferHdr->nFilledLen + 3;

  pExtra = (OMX_OTHER_EXTRADATATYPE *) (((OMX_U32) pTmp) & ~3);

  while(pExtra->eType != OMX_ExtraDataNone)
  {
      pExtra = (OMX_OTHER_EXTRADATATYPE *) (((OMX_U8 *) pExtra) +
pExtra->nSize);
  }
```

4.2.33 OMX_CONFIG_CAPTUREMODETYPE

Capture mode configuration is used to instruct the camera component how it shall behave during the course of capturing: continuous versus frame count limited capturing operations.

OMX_CONFIG_CAPTUREMODETYPE is defined as follows.

```
typedef struct OMX_CONFIG_CAPTUREMODETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bContinuous;
    OMX_BOOL bFrameLimited;
    OMX_U32 nFrameLimit;
 } OMX_CONFIG_CAPTUREMODETYPE;
```

4.2.33.1 Parameters

The parameters for OMX_CONFIG_CAPTUREMODETYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bContinuous is a Boolean used to indicate the frame rate emission. If bContinuous is set to OMX_TRUE, then ignore the port frame rate setting and emit captured frame data as quickly as possible otherwise obey the port's frame rate setting.
- bFrameLimited is a Boolean used to indicate whether capturing shall be terminated after the specified number of frames. If bFrameLimited is set to OMX_TRUE, then frame limited capture is enabled; otherwise the port does not terminate capturing until instructed to do so by the client.
- nFrameLimit is the limit on number of frames emitted during capturing, this parameter is only valid if bFrameLimited is enabled.

4.2.34 OMX_CONFIG_BOOLEANTYPE

The OMX_CONFIG_BOOLEANTYPE structure contains generic Boolean configuration information that may be used to set component level configuration settings rather than port level configuration settings.

OMX_CONFIG_BOOLEANTYPE is defined as follows.

```
typedef struct OMX_CONFIG_BOOLEANTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_BOOL bEnabled;
} OMX_CONFIG_BOOLEANTYPE;
```

4.2.34.1 Parameters

The parameters for OMX_CONFIG_BOOLEANTYPE are defined as follows.

- `bEnabled` is a Boolean used to indicate if a configuration is to be enabled. The configuration setting to be enabled is typically inherent in the name of the configuration or parameter index used with this structure.

For example, the `OMX_IndexAutoPauseAfterCapture` index will use the `OMX_CONFIG_BOOLEANTYPE` structure to enable or disable the auto pause mechanism after a capture request is completed.

4.2.35 OMX_SHARPNESSTYPE

`OMX_SHARPNESSTYPE` is used to apply differing levels of sharpness, alternatively also referred to as blurriness, in order to improve the image quality.

`OMX_SHARPNESSTYPE` is defined as follows.

```
typedef struct OMX_SHARPNESSTYPE{
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_S32 nSharpness;
} OMX_SHARPNESSTYPE;
```

4.2.35.1 Parameters

The parameters for `OMX_SHARPNESSTYPE` are defined as follows.

- `nPortIndex` specifies the component port index.
- `nSharpness` is a signed value identifying the level of sharpness to apply. Increasing positive values applies increassing levels of sharpness while increasig negative values applies increasing levels of blur.

4.2.36 OMX_CONFIG_ZOOMFACTORTYPE

`OMX_CONFIG_ZOOMFACTORTYPE` is used to get and set the zoom factor value.

`OMX_CONFIG_ZOOMFACTORTYPE` is defined as follows.

```
typedef struct OMX_CONFIG_ZOOMFACTORTYPE {
  OMX_U32 nSize;
  OMX_VERSIONTYPE nVersion;
  OMX_U32 nPortIndex;
  OMX_BU32 xZoomFactor;
}OMX_CONFIG_ZOOMFACTORTYPE;
```

4.2.36.1 Parameters

The parameters for `OMX_CONFIG_ZOOMFACTOR` are defined as follows.

- `nPortIndex` specifies the component port index.
- `xZoomFactor` is representing the zoom factor in unsigned Q16 format. Zooming is a method of decreasing (narrowing) the apparent field of view, and the zoom factor tells the actual value of this change.

  Zoom factor is targeted to be used both for optical and digital zoom, so each type of zoom will have its own factor. Optical zoom implies a real zoom lens, while digital zoom is accomplished electronically. In case of digital zoom, camera component first computes the largest field of view constrained by both the port aspect ratio and `OMX_IndexConfigCommonCenterFieldOfView` setting. It then applies centered cropping within this largest field of view, according to the specified digital zoom factor. The resulting cropping window has the same aspect ratio as the port. In case the requested digital zoom factor cannot be applied, the setting fails and IL client is returned error `OMX_ErrorBadParameter`.

  For example,assume the optical zoom is fixed to a certain value. Then, `xZoomFactor` equal to 1.0 for digital zoom means that there is no digital zoom involved and no cropping; if in this case the digital zoom factor is increased to 2 we have a 2X (two times) digital zoom and the apparent field of view will be decreased to half (1/2) of the original in both dimensions of the original scene with same resolution.

4.2.36.2 Functionality

Field of View Centering for Digital Zoom

While for the optical zoom the center of the field of view is always in the center of the observed scene, there is no similar restriction for the digital zoom. For digital zoom, the center of the field of view can be actually any point from the observed scene, or any point from camera sensor provided frames.

The coordinates of the center of the field of view will use a Q16 format representation, relative to the dimensions of the whole field of view with (0,0) being top left, and (1<<16,1<<16) being bottom right.

In case the requested center of view is off the image, the setting fails and IL client is returned error `OMX_ErrorBadParameter.`

4.2.37 OMX_IMAGE_CONFIG_LOCKTYPE

`OMX_IMAGE_CONFIG_LOCKTYPE` is used to lock image settings.

`OMX_IMAGE_CONFIG_LOCKTYPE` is defined as follows.

```
typedef struct OMX_IMAGE_CONFIG_LOCKTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_IMAGE_LOCKTYPE eImageLock;
} OMX_IMAGE_CONFIG_LOCKTYPE;
```

4.2.37.1 Parameters

The parameters for OMX_IMAGE_CONFIG_LOCKTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to .
- eImageLock specifies the locking mode to apply. Table 4-43 details the values that can be selected for image locking.

Table 4-43: eImageLock Values

OMX_IMAGE_LOCKTYPE	Description
OMX_IMAGE_LockOff	Locking of settings shall not be applied.
OMX_IMAGE_LockImmediate	Locking of settings occurs at the end of current search iteration or immediately if there is no outstanding iteration. settings are frozen and no longer automatically updated.
OMX_IMAGE_LockAtCapture	Locking of settings occurs at the beginning of a capture. settings are frozen and no longer automatically updated. For example, freezing the metering settings at the beginning of a multiple image capture.

4.2.37.1 Functionality

When a setting is locked, the component shall refuse any related setting change. The IL client shall first unlock the locked setting.

The IL client is allowed to change from OMX_IMAGE_LockImmediate to OMX_IMAGE_LockAtCapture, but this change implies an implicit pass through OMX_IMAGE_LockOff.

4.2.38 OMX_CONFIG_FOCUSRANGETYPE

OMX_CONFIG_FOCUSRANGETYPE is used to control the range of the focus.

OMX_CONFIG_FOCUSRANGETYPE is defined as follows.

```
typedef struct OMX_CONFIG_FOCUSRANGETYPE {
    OMX_U32 nSize;
```

```
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_FOCUSRANGETYPE eFocusRange;
} OMX_CONFIG_FOCUSRANGETYPE;
```

4.2.38.1 Parameters

The parameters for `OMX_CONFIG_FOCUSRANGETYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to .
- `eFocusRange` specifies focus range mode to apply. Table 4-44 shows the values that can be selected for focus range.

Table 4-44: eFocusRange Values

OMX_FOCUSRANGETYPE	Description
`OMX_FocusRangeAuto`	Allows the focus range to be chosen automatically by the component.
`OMX_FocusRangeHyperfocal`	Sets the focus range to be from half the hyperfocal distance to infinity. Hyperfocal distance is the focus distance with the maximum depth of field. It is connected to the focal point of the lens and represents the range around that point that has acceptable sharpness.
`OMX_FocusRangeNormal`	Sets the focus range from approximately 40 cm to infinity
`OMX_FocusRangeSuperMacro`	Sets the focus range from approximately 4 to 10 cm
`OMX_FocusRangeMacro`	Sets the focus range from approximately 10 to 50 cm
`OMX_FocusRangeInfinity`	Sets the focus range to be at infinity. The differs from OMX_FocusRangeHyperfocal in that the user is interested in objects at infinity, when everything is far away; in this case the maximum possible sharpness is achieved at infinity. In case of OMX_FocusRangeHyperfocal range we achieve maximum possible depth of field.

4.2.39 OMX_IMAGE_CONFIG_FLASHSTATUSTYPE

OMX_IMAGE_CONFIG_FLASHSTATUSTYPE is used to query the flash status.

OMX_IMAGE_CONFIG_FLASHSTATUSTYPE is defined as follows.

```
typedef struct OMX_IMAGE_CONFIG_FLASHSTATUSTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_IMAGE_FLASHSTATUSTYPE eFlashStatus;
} OMX_IMAGE_CONFIG_FLASHSTATUSTYPE ;
```

4.2.39.1 Parameters

The parameters for OMX_IMAGE_CONFIG_FLASHSTATUSTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to .
- eFlashStatus specifies the flash status. Table 4-45 shows the possible values for flash status.

Table 4-45: eFlashStatus Values

OMX_IMAGE_FLASHSTATUSTYPE	Description
OMX_IMAGE_FlashUnknown	Flash status is unknown. Conditions under which a flash may provide this status are: • No clear state can be considered, • Device is in transition from one flash state to another • During firing of the flash since this is a short transition period.
OMX_IMAGE_FlashOff	Flash is off.
OMX_IMAGE_FlashCharging	Flash is charging.
OMX_IMAGE_FlashReady	Flash is ready to be fired.
OMX_IMAGE_FlashNotAvailable	Flash is not available. Flash is not present or overheated.
OMX_IMAGE_FlashInsufficientCharge	Flash cannot be charged due to insufficient battery charge.

4.2.40 OMX_CONFIG_EXTCAPTUREMODETYPE

OMX_CONFIG_EXTCAPTUREMODETYPE is used to configure the capture behavior. It is used for best picture selection by allowing the component to capture and emit a number

of frames prior to and after the capture emission is started. OMX_CONFIG_EXTCAPTUREMODETYPE is used in conjunction with OMX_CONFIG_CAPTUREMODETYPE.

OMX_CONFIG_EXTCAPTUREMODETYPE is defined as follows:

```
typedef struct OMX_CONFIG_EXTCAPTUREMODETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nFrameBefore;
    OMX_BOOL bPrepareCapture;
} OMX_CONFIG_EXTCAPTUREMODETYPE;
```

4.2.40.1 Parameters

The parameters for OMX_CONFIG_EXTCAPTUREMODETYPE are defined as follows:

- nPortIndex represents the port that this structure applies to.
- nFrameBefore specifies how many captured frames shall be buffered up by the component after bPrepareCapture has been set to OMX_TRUE, and while waiting for the capture bit (OMX_IndexConfigCommonPortCapturing) to be set. These frames will be constantly updated with new captures until the capture bit is set by the client. The component will then emit these frames as the first frames in the multiple frame capture mode (defined by bFrameLimited and nFrameLimit in OMX_CONFIG_CAPTUREMODETYPE structure).
- bPrepareCapture specifies if the component enables pre-capturing. The component shall not deliver buffered captured frames until capturing starts.

4.2.40.2 Functionality

To illustrate the functionality, sample sequences for standard multiple image capture and extended multiple capture are depicted in Figure 4-4 and Figure 4-5, respectively. In both cases it is assumed that before the first sequence call (1.0 SetConfig(…)), the camera component and any cooperating components are instantiated and tunneled. Camera parameters are also set, image mode established (bOneShot has value OMX_TRUE), and components are in OMX_StateExecuting state. The OMX_IndexAutoPauseAfterCapture index can be used to enable or disable the auto pause mechanism after the capture request is completed. The auto pause mechanism is disabled by default. For standard multiple image capture nFrameLimit images are captured after capture is triggered. For extended multiple image capture, an additional nFrameBefore images are captured before the capture is triggered, for a total of nFrameLimit+nFrameBefore images.

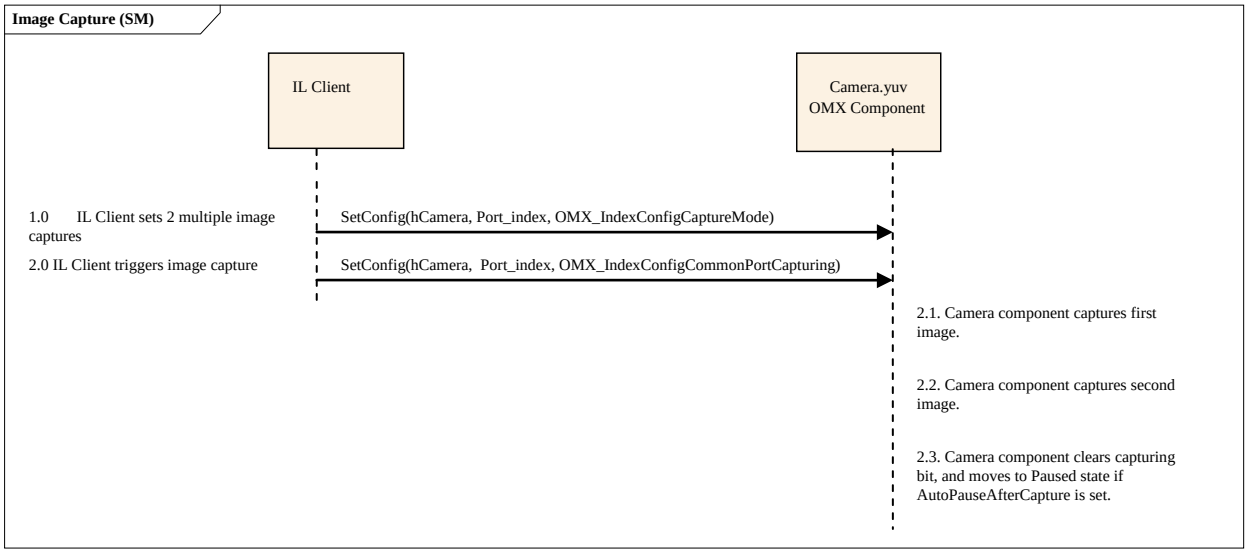


Figure 4-4: Extended Capture - Standard Multiple Image Capture

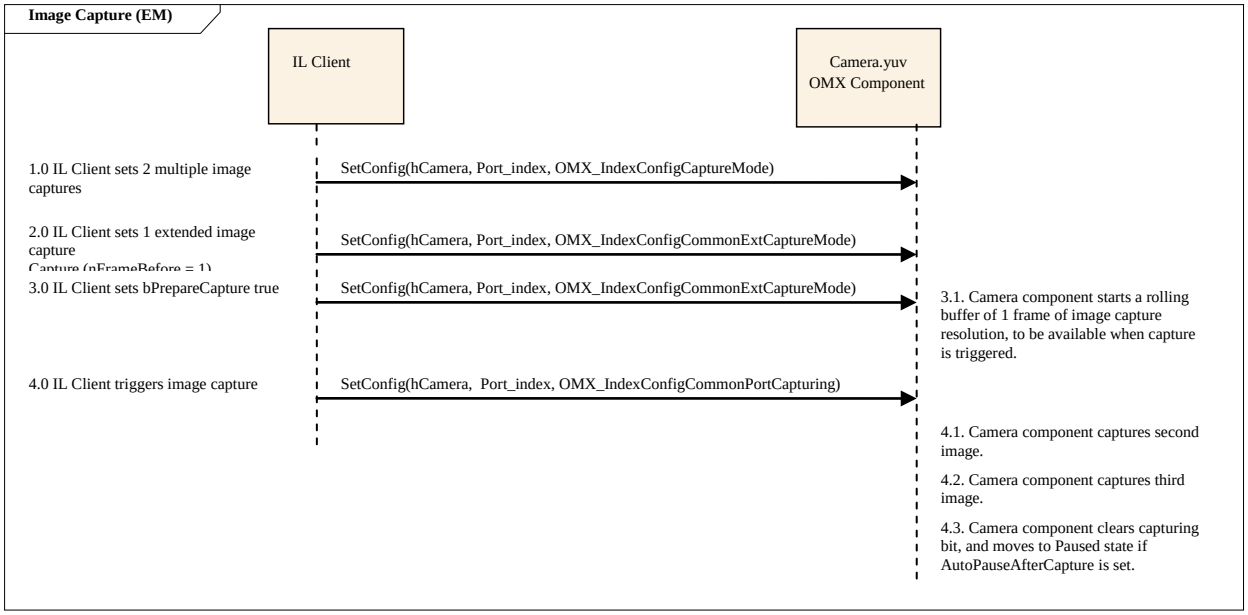


Figure 4-5: Extended Capture - Extended Multiple Image Capture

4.2.41 OMX_CONFIG_NDFILTERCONTROLTYPE

`OMX_CONFIG_NDFILTERCONTROLTYPE` is used to control the ND Filter functionality.

Enabling the ND filter leads to reducing the light received by the sensor, the result being that different aperture/shutter speed combinations for the same target total exposure become available.

OMX_CONFIG_NDFILTERCONTROLTYPE is defined as follows.

```
typedef struct OMX_CONFIG_NDFILTERCONTROLTYPE{
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_NDFILTERCONTROLTYPE eNDFilterControl;
} OMX_CONFIG_NDFILTERCONTROLTYPE;
```

4.2.41.1 Parameters

The parameters for OMX_CONFIG_NDFILTERCONTROLTYPE are defined as follows:

- eNDFilterControl specifies ND Filter control setting. Table 4-46 shows the possible values for ND filter settings.

Table 4-46: eNDFilterControl Values

OMX_NDFILTERCONTROLTYPE	Description
OMX_NDFilterOff	ND Filter is off (disabled).
OMX_NDFilterOn	ND Filter is on (enabled).
OMX_NDFilterAuto	ND Filter is on (enabled) and allows automatic control of the filter by the component.

4.2.42 OMX_CONFIG_AFASSISTANTLIGHTTYPE

OMX_CONFIG_AFASSISTANTLIGHTTYPE is used to control the autofocus assistant light.

OMX_CONFIG_AFASSISTANTLIGHTTYPE is defined as follows:

```
typedef struct OMX_CONFIG_AFASSISTANTLIGHTTYPE{
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_AFASSISTANTLIGHTTYPE eAFAssistantLight;
} OMX_CONFIG_AFASSISTANTLIGHTTYPE;
```

4.2.42.1 Parameters

The parameters for OMX_CONFIG_AFASSISTANTLIGHTTYPE are defined as follows:

- eAFAssistantLight specifies assistant light control setting. Table 4-47 shows the possible values for assistant light settings.

Table 4-47: eAFAssistLight Values

OMX_AFASSISTANTLIGHTTYPE	Description
OMX_AFAssistantLightOff	Forces turning off the autofocus assistant light during following autofocus cycles, and not immediately if there is an ongoing autofocus cycle.
OMX_AFAssistantLightOn	Forces turning on the autofocus assistant light during following autofocus cycles, and not immediately if there is an ongoing autofocus cycle.
OMX_AFAssistantLightAuto	Allows automatic control by the component of the autofocus assistant light during following autofocus cycles, and not immediately if there is an ongoing autofocus cycle.

4.2.43 OMX_FROITYPE

OMX_FROITYPE is used to specific regions of interest for focus.

OMX_FROITYPE is defined as follows.

```
typedef struct OMX_FROITYPE {
    OMX_S32 nRectX;
    OMX_S32 nRectY;
    OMX_S32 nRectWidth;
    OMX_S32 nRectHeight;
    OMX_S32 xFocusDistance;
    OMX_FOCUSSTATUSTYPE eFocusStatus;
} OMX_FROITYPE;
```

4.2.43.1 Parameters

The parameters for OMX_FROITYPE are defined as follows.

- nRectX specifies the relative leftmost coordinate of a rectangle representing the region of interest. This coordinate is relative to the dimensions of the whole observed area for which the focusing is operating. All reported focusing areas shall be contained within this reference window (i.e., the reference window is represented as (0, 0, 1<<16, 1<<16)). This value is represented in Q16 format.
- nRectY specifies the relative topmost coordinate of a rectangle representing the region of interest. This coordinate is relative to the dimensions of the whole observed area for which the focusing is operating. All reported focusing areas shall be contained within this reference window (i.e., the reference window is represented as (0, 0, 1<<16, 1<<16)). This value is represented in Q16 format.
- nRectWidth specifies the relative width of a rectangle representing the region of interest. This coordinate is relative to the dimensions of the whole observed area for which the focusing is operating. All reported focusing areas shall be

contained within this reference window (i.e., the reference window is represented as (0, 0, 1<<16, 1<<16)). This value is represented in Q16 format.

- `nRectHeight` specifies the relative height of a rectangle representing the region of interest. This coordinate is relative to the dimensions of the whole observed area for which the focusing is operating. All reported focusing areas shall be contained within this reference window (i.e., the reference window is represented as (0, 0, 1<<16, 1<<16)). This value is represented in Q16 format.
- `xFocusDistance` is the estimated focusing distance in meters. This value is represented in Q16 format. When `xFocusDistance` takes on the maximum Q16 value, it means the distance cannot be reported since AF is at infinity. When `xFocusDistance` is 0, it means the distance is unknown.
- `eFocusStatus` specifies the status of the focus. Table 4-49 details the possible values for focus status.

Table 4-48 : eFocus Status Types

Focus Status	Focus Status Description
`OMX_FocusStatusOff`	Focus request is disabled
`OMX_FocusStatusRequest`	Focus request is currently being processed.
`OMX_FocusStatusReached`	Focus has been reached.
`OMX_FocusStatusUnableToReach`	Focus is unreachable, the maximum is too close to the average noise
`OMX_FocusStatusLost`	Focus has been lost, the main subject has moved in the scene

4.2.44 OMX_CONFIG_FOCUSREGIONSTATUSTYPE

`OMX_CONFIG_FOCUSREGIONSTATUSTYPE` is used to retrieve the status of the focus.

`OMX_CONFIG_FOCUSREGIONSTATUSTYPE` is defined as follows.

```
typedef struct OMX_CONFIG_FOCUSREGIONSTATUSTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_BOOL bFocused;
    OMX_U32 nMaxFAreas;
    OMX_U32 nFAreas;
    OMX_FROITYPE sFROIs[1];
} OMX_CONFIG_FOCUSREGIONSTATUSTYPE;
```

4.2.44.1 Parameters

The parameters for OMX_CONFIG_FOCUSREGIONSTATUSTYPE are defined as follows.

- bFocused is true when any of the focus regions is currently successfully focused.
- nMaxFAreas is the maximum number of focusing areas that are to be reported. When IL client queries this information with the value set to zero, the component updates the value with the maximum supported number of focus regions and no other fields are updated. If the IL client queries this information with this value set to non-zero and within the range of supported values, the component does not update the value and only reports focus regions up to this number; so in this case the parameter is the size of returned array sFROIs. If the IL client calls GetConfig with this value non-zero and outside the range of supported values the IL client is returned error OMX_ErrorBadParameter.
- nFAreas is the actual number of regions used by focusing. By default, this value is 1.
- sFROIs is an array which contains the coordinates of the areas in focus.

4.2.44.2 Usage

If there are multiple focus regions, the indication that focus has been achieved is updated if any of the focus regions is able to achieve focus.

The IL client needs to subscribe to callbacks in order to get events when focus status changes.

4.2.45 OMX_MANUALFOCUSRECTTYPE

OMX_MANUALFOCUSRECTTYPE is used to indicate the manual focus rectangle information.

OMX_MANUALFOCUSRECTTYPE is defined as follows.

```
typedef struct OMX_MANUALFOCUSRECTTYPE {
    OMX_S32 nRectX;
    OMX_S32 nRectY;
    OMX_S32 nRectWidth;
    OMX_S32 nRectHeight;
 } OMX_MANUALFOCUSRECTTYPE;
```

4.2.45.1 Parameters

The parameters for OMX_MANUALFOCUSRECTTYPE are defined as follows.

- nRectX specifies the leftmost coordinate of the rectangle. These coordinates are relative to the resolution of the port. All focusing areas that are specified by the

IL client shall be contained within this reference window. This value is represented in Q16 format.

- `nRectY` specifies the topmost coordinate of the rectangle. All focusing areas that are specified by the IL client shall be contained within this reference window. This value is represented in Q16 format.
- `nRectWidth` specifies the width of the rectangle. All focusing areas that are specified by the IL client shall be contained within this reference window. This value is represented in Q16 format.
- `nRectHeight` specifies the height of the rectangle. All focusing areas that are specified by the IL client shall be contained within this reference window. This value is represented in Q16 format.

4.2.46 OMX_CONFIG_FOCUSREGIONCONTROLTYPE

`OMX_CONFIG_FOCUSREGIONCONTROLTYPE` is used to set and get the control information for focus regions.

`OMX_CONFIG_FOCUSREGIONCONTROLTYPE` is defined as follows.

```
typedef struct OMX_CONFIG_FOCUSREGIONCONTROLTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nFAreas;
    OMX_FOCUSREGIONCONTROLTYPE eFocusRegionsControl;
    OMX_MANUALFOCUSRECTTYPE sManualFRegions[1];
} OMX_CONFIG_FOCUSREGIONCONTROLTYPE;
```

4.2.46.1 Parameters

The parameters for `OMX_CONFIG_FOCUSREGIONCONTROLTYPE` are defined as follows.

- `nFAreas` is the number of regions to be used for focusing. By default, this value is 1.

  In the case of manual focus `OMX_FocusRegionControlManual,` the value of `nFAreas` represents the dimension of the `sManualFRegions` vector. In the non-manual case, the value of `nFAreas` represents the maximum number of focus regions to be used for focusing. When queried in manual focus cases only the required number of regions are reported.

  This value limits the number of areas where focus is attempted and it is reported in the focus status `OMX_CONFIG_FOCUSREGIONSTATUSTYPE`. The maximum supported number of areas can be queried via `OMX_CONFIG_FOCUSREGIONSTATUSTYPE` (Refer to 4.2.44)

- `eFocusRegionsControl` specifies the focusing control type. Table 4-49 shows the possible values for focus control type.

Table 4-49 : eFocusRegionControl Values

OMX_FOCUSREGIONCONTROLTYPE	Description
OMX_FocusRegionControlAuto	Used when focus regions are automatically selected by autofocus (AF) algorithm.
OMX_FocusRegionControlManual	Used when focus regions are manually selected by the IL client.
OMX_FocusRegionControlFacePriority	Used when focus should be attempted to priority face if available, otherwise automatically selected by AF algorithm.
OMX_FocusRegionControlObjectPriority	Used when focus should be attempted to priority object if available, otherwise automatically selected by AF algorithm.

- `sManualFRegions` is an array which contains the relative coordinates to be used for focusing (top left corner coordinates and size).

  This information is provided by the IL client; it is used only when `eFocusRegionsControl` is set to `OMX_FocusRegionControlManual`.

4.2.47 OMX_INTERLACEFORMATTYPE

This structure is used to specify the formatting of interlaced video content.

Components such as video decoders may emit interlaced video content. When displayed on a progressive display there are visible artifacts that can be avoided using a de-interlace filter. To compensate for the visible artifacts, information about the interlace format needs to be made available so that the consumer of the content may be configured appropriately.

OMX_INTERLACEFORMATTYPE is defined as follows

```
typedef struct OMX_INTERLACEFORMATTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nFormat;
    OMX_TICKS nTimeStamp;
} OMX_INTERLACEFORMATTYPE;
```

4.2.47.1 Parameters

The parameters for OMX_INTERLACEFORMATTYPE are defined as follows.

- nFormat specifies a bitmapped value identifying the interlace formats supported by the component port. This format information identifies the temporal relationship between the two fields.

  The available formats are described in

Table 4-50: Interlace Type Values

OMX_INTERLACETYPE Enumerated Value	**Description**
OMX_InterlaceFrameProgressive	The data contains a progressive (non-interlaced) frame. The data is not interlaced, it is progressive scan
OMX_InterlaceInterleaveFrameTopFieldFirst	The data contains an interlaced frame containing interleaved Top and Bottom fields. The frame content in the buffer starts with the Top Field content as the first video line, followed by the Bottom Field video line, and the remaining video lines interleaving between Top and Bottom Field content. The temporal relationship between the fields is that the Top Field occurs earlier than the Bottom Field.

OMX_InterlaceInterleaveFrameBottomFieldFirst	The data contains an interlaced frame containing interleaved Top and Bottom fields. The frame content in the buffer starts with the Top Field content as the first video line, followed by the Bottom Field video line, and the remaining video lines interleaving between Top and Bottom Field content. The temporal relationship between the fields is that the Bottom Field occurs earlier than the Top Field.
OMX_InterlaceFrameTopFieldFirst	The data contains an interlaced frame containing all the Top Field information followed by the Bottom Field information. The temporal relationship between the fields is that the Top Field occurs earlier than the Bottom Field.
OMX_InterlaceFrameBottomFieldFirst	The data contains an interlaced frame containing all the Top Field information followed by the Bottom Field information. The temporal relationship between the fields is that the Bottom Field occurs earlier than the Top Field.
OMX_InterlaceInterleaveFieldTop	The Data contains an interlaced frame containing a single field of interlaced content. The data contains only the Top field information, with the field content occupying every other video line starting from the first line in the buffer.
OMX_InterlaceInterleaveFieldBottom	The Data contains an interlaced frame containing a single field of interlaced content. The data contains only the Bottom field information, with the field content occupying every other video line starting from the second line in the buffer.

Note: In some specifications, Top\Bottom Field is referenced as Upper\Lower or Odd\Even Field.

- nTimeStamp specifies the temporal timestamp information for the second field. The nTimeStamp parameter provided via the OMX_BUFFERHEADERTYPE structure specifies the timestamp information for the first field.

  For example, for OMX_InterlaceInterleaveFrameTopFieldFirst the nTimeStamp parameter provided via OMX_BUFFERHEADERTYPE specifies the timestamp of the Top Field and the nTimeStamp parameter provided via OMX_INTERLACEFORMATTYPE structure specifies the timestamp for the Bottom Field.

  If the temporal timestamp information can not be determined for the second field, the nTimeStamp parameter for OMX_INTERLACEFORMATTYPE structure shall be set the same as the nTimeStamp parameter via the OMX_BUFFERHEADERTYPE structure.

4.2.48 OMX_DEINTERLACETYPE

This structure is used to enable or disable deinterlacing support. By default, deinterlacing support is disabled.

OMX_DEINTERLACETYPE is defined as follows

```
typedef struct  OMX_DEINTERLACETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bEnable;
} OMX_DEINTERLACETYPE;
```

4.2.48.1 Parameters

The parameters for OMX_DEINTERLACETYPE are defined as follows.

- eEnable specifies the requested state of the deinterlacing support.

  By default, deinterlacing is disabled.

4.2.49 OMX_STREAMINTERLACEFORMATTYPE

This structure is used to query if the stream contains interlaced or progressive content.

OMX_STREAMINTERLACEFORMATTYPE is defined as follows

```
typedef struct  OMX_STREAMINTERLACEFORMATTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bInterlaceFormat;
    OMX_U32 nInterlaceFormats;
} OMX_STREAMINTERLACEFORMAT;
```

4.2.49.1 Parameters

The parameters for `OMX_DEINTERLACETYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `bInterlaceFormat` specifies if the stream contains interlace or progressive content – `OMX_TRUE` indicates interlace and `OMX_FALSE` indicates progressive.
- `nInterlaceFormat` specifies a bitmapped value identifying the interlace formats detected within the stream. The available formats are described in .

  It may not always be possible to determine the full extent of the formats within the stream at the commencement of the stream processing. This information may be dynamically populated as the component processes the individual frames within the stream.

4.2.49.2 Post-processing Conditions

A component shall emit an `OMX_EventError` event when it has detected content containing an unsupported format has been supplied to it.

4.2.49.3 Functionality

This section illustrates various call sequence chart examples to configure components for interlaced video content and the notification of events.

Figure 4-6 shows the steps to coordinate the consumption of interlace content between a Video Decoder Component and IV Renderer Component.

Figure 4-7 shows the events arising when the IV Renderer component is consuming an interlace content that it does not support.

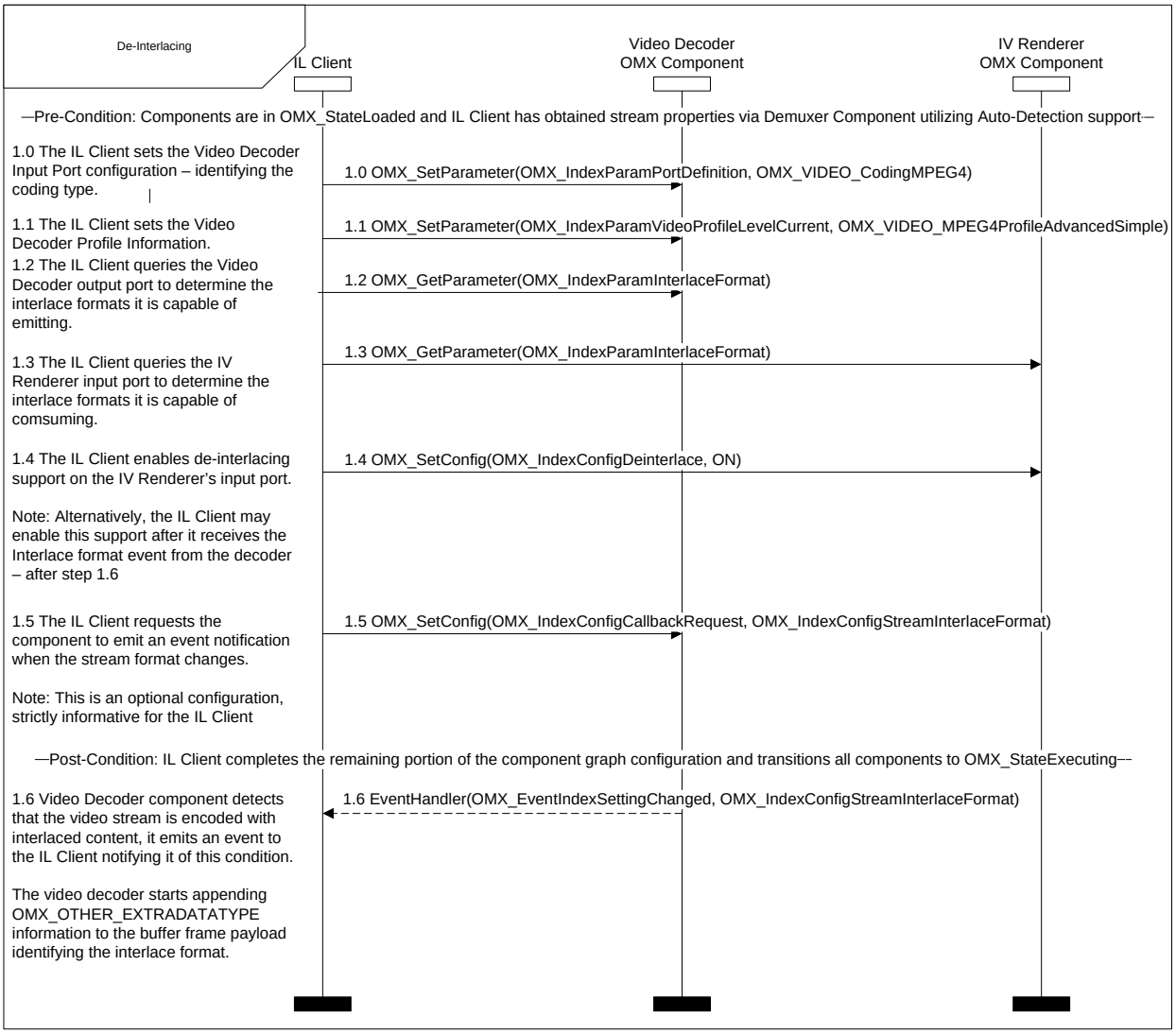


Figure 4-6: De-Interlacing Configuration Setup

The sequence starts with a Pre-Condition that the IL client has loaded the components, and retrieved the stream information from the Demuxer component.

The IL client informs the Video Decoder input port that it will be consuming a MPEG4 Advanced Simple Profile stream – Step 1.0 and 1.1

The IL client queries the Video Decoder output port to determine the possible interlace formats it may emit, this information is based on the Video Decoder's input port stream configuration – Step 1.2.

The IL client queries the IV Renderer input port to determine the possible interlace formats it can consume – Step 1.3.

The IL client enables de-interlacing support on the IV Renderer input port – Step 1.4.

Note: The IL client may wish to delay this configuration until after Step 1.6 when the Video Decoder component confirms that it will be emitting interlaced content.

The IL client requests the component to emit an event notification if the stream format changes from the Progressive (default mode) to Interlace – Step 1.5. This is an optional configuration, strictly informative for the IL client.

The IL client completes the remaining portion of the component graph configuration and transitions all the components to `OMX_StateExecuting`.

At this point, the Video Decoder starts consuming the MPEG4 stream, it determines that the video stream contains interlaced content. The video decoder emits an event to the IL client informing it of this condition and it starts appending `OMX_OTHER_EXTRADATATYPE` information to the video frames identifying the interlace coding format – Step 1.6.

The Video Decoder output continues to emit frames.

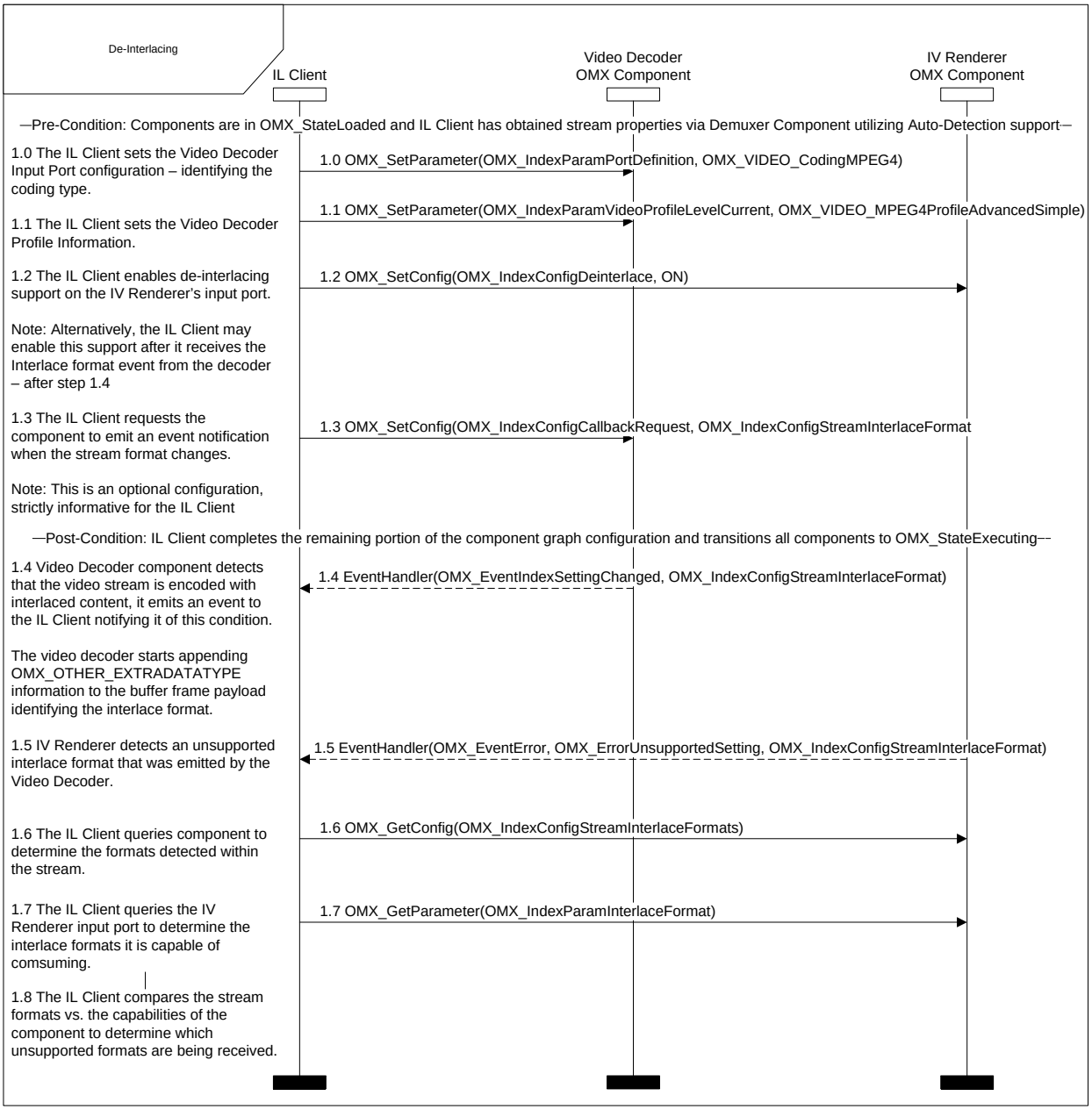


Figure 4-7: Unsupported Interlace Format Detection

The sequence starts with a Pre-Condition that the IL client has loaded the components, and retrieved the stream information from the Demuxer component.

The IL client informs the Video Decoder input port that it will be consuming a MPEG4 Advanced Simple Profile stream – Step 1.0 and 1.1

The IL clients neglects to query both the Video Decoder output and IV Renderer input port to determine their support formats as identified in Steps 1.2 and 1.3, additionally it does not limited the type of formats to be emitted by the Video Decoder output port as identified in Step 1.4.

The outcome at this stage is that the video decoder will be emitting all possible formats that it is capable of supporting, which may be more than the formats supported by the IV Renderer input port.

The IL client enables de-interlacing support on the IV Renderer input port – Step 1.2.

Note: The IL client may wish to delay this configuration until after Step 1.4 when the Video Decoder component confirms that it will be emitting interlaced content.

The IL client requests the component to emit an event notification if the stream format changes from the Progressive (default mode) to Interlace – Step 1.3. This is an optional configuration, strictly informative for the IL client.

The IL client completes the remaining portion of the component graph configuration and transitions all the components to `OMX_StateExecuting`.

At this point, the Video Decoder starts consuming the MPEG4 stream, it determines that the video stream contains interlaced content. The video decoder emits an event to the IL client informing it of this condition and it starts appending `OMX_OTHER_EXTRADATATYPE` information to the video frames identifying the interlace coding format – Step 1.4.

The Video Decoder output continues to emit frames.

The IV Renderer receives the emitted frames from the Video Decoder output port and determines that the embedded interlace content is not supported, it emits an `OMX_EventError` to the IL client informing it of this condition – Step 1.5.

The IV Renderer continues to render the content, some frames may be displayed showing interlacing artifacts due to the unsupported formats.

The IL client queries the IV Renderer to determine the formats it support and the formats it detected within the stream – Steps 1.6 and 1.7.

The IL client compares this information to determine which unsupported format was detected by the IV Renderer and then it takes any appropriate action it deems necessary – Step 1.8.

4.3 Video

This section describes the parameter and configuration details for ports in the video domain. These parameter and configuration details are specified in the OMX_Video.h header.

4.3.1 Video Use Case Examples

depicts one possible set of components as well as the tunneling of ports for these components to implement a H.263 video encoding scheme. This use case encodes raw video into H.263 format and writes it to a file while previewing the captured video on a display.

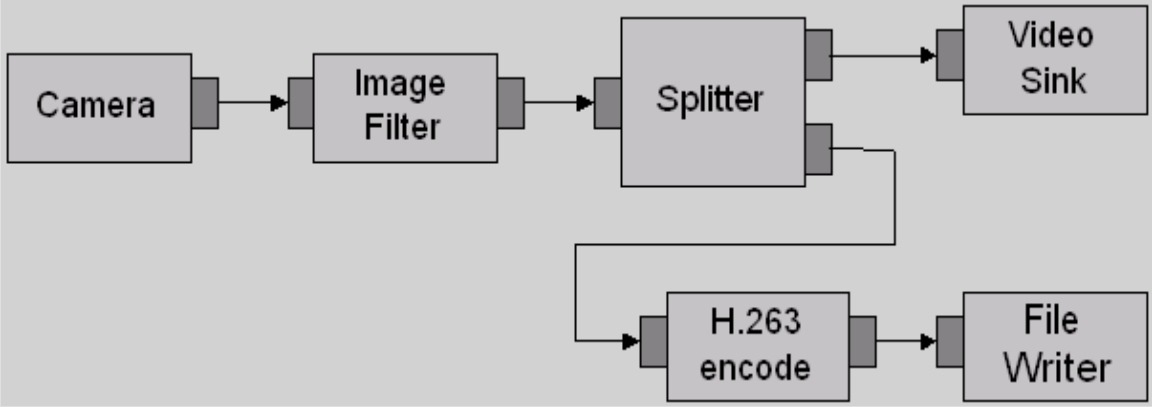


Figure 4-8. H.263 Video Encode Use Case

shows six components, namely the camera, the image filter, the splitter, the H.263 video encoder, the file writer, and the video sink.

shows a more complex use case, which is video conferencing. This use case supports simultaneous encoding and decoding of video streams. To simplify the use case, the corresponding audio components are not included.

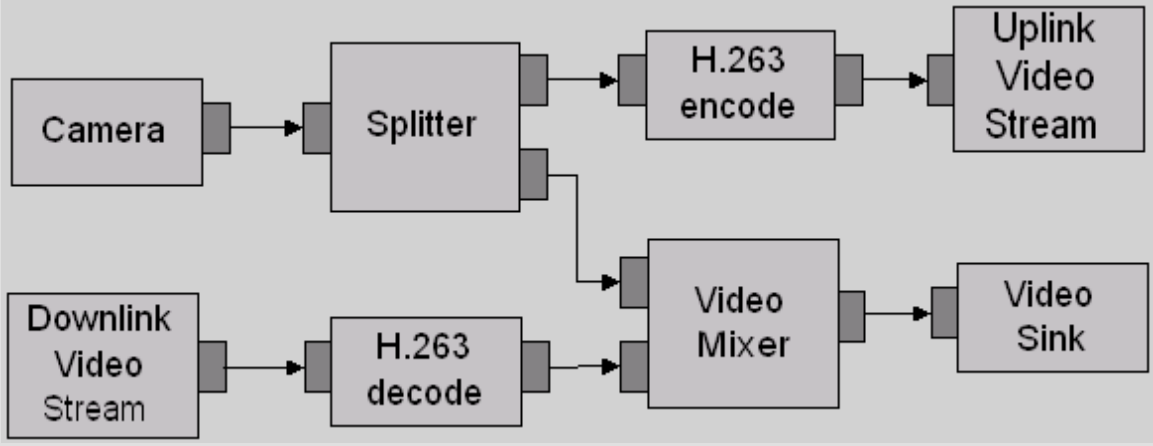


Figure 4-9. Video Conferencing Use Case

Raw video is encoded to H.263 format and then transmitted via a video uplink to the far-side conferencing participant. At the same time, a H.263 video stream is received from the far-side participant via a video downlink and decoded to raw video format before being mixed into a pre-determined presentation layout via the video mixer such that both the local participant's video and far-side participant's video are displayed via the local video sink.

4.3.2 General Enumerations

The OMX_VIDEO_CODINGTYPE enumeration defines the video coding types supported.. If OMX_VIDEO_CodingUnused is selected, then the coding selection shall be done in a vendor-specific way. Table 4-51 shows the OpenMAX IL-supported video compression formats.

Table 4-51: Supported Video Compression Formats

Field Name	Coding Type Descriptions	References to Standards
OMX_VIDEO_CodingUnused	No coding applied. Use eColorFormat	Not available
OMX_VIDEO_CodingAutoDetect	Auto-detection by the OpenMAX IL component	Not available
OMX_VIDEO_CodingMPEG2	MPEG-2, also known as H.262 video format	MPEG2
OMX_VIDEO_CodingH263	ITU H.263 video format	H263
OMX_VIDEO_CodingMPEG4	MPEG-4 video format	MPEG4
OMX_VIDEO_CodingWMV	All versions of the Windows Media video format	WMV
OMX_VIDEO_CodingRV	All versions of the RealVideo® format	RV
OMX_VIDEO_CodingAVC	ITU H.264/AVC video format	H264
OMX_VIDEO_CodingMJPEG	Motion JPEG video format	MJPEG
OMX_VIDEO_CodingVC1	VC-1 format (SMPTE 421M)	VC-1
OMX_VIDEO_CodingVP8	VP8 Video Bitstream	VP8

The OMX_VIDEO_PICTURETYPE enumeration defines the video picture types supported. Table 4-52 describes the supported video picture types.

Table 4-52: Supported Video Picture Types

Field Name	Picture Type Descriptions
OMX_VIDEO_PictureTypeI	General I-frame type
OMX_VIDEO_PictureTypeP	General P-frame type
OMX_VIDEO_PictureTypeB	General B-frame type
OMX_VIDEO_PictureTypeSI	H.263 SI-frame type
OMX_VIDEO_PictureTypeSP	H.263 SP-frame type
OMX_VIDEO_PictureTypeEI	H.264 EI-frame type
OMX_VIDEO_PictureTypeEP	H.264 EP-frame type
OMX_VIDEO_PictureTypeS	MPEG-4 S-frame type

4.3.3 Parameter and Configuration Indices

The header OMX_Index.h contains the enumeration OMX_INDEXTYPE, which contains all of the standard index values used with the OpenMAX IL core functions OMX_GetParameter, OMX_SetParameter, OMX_GetConfig, and OMX_SetConfig.

The index values that relate to video are described in this section. For example, OMX_IndexParamVideoPortFormat index is used with OMX_GetParameter and OMX_SetParameter to access the OMX_VIDEO_PARAM_PORTFORMATTYPE. Table 4-53 identifies the video indices.

Table 4-53: Video Indices

OpenMAX IL Indices (OMX_Index.h)	Corresponding OpenMAX IL Video Structures (OMX_Video.h)
OMX_IndexParamPortDefinition	OMX_PARAM_PORTDEFINITIONTYPE with OMX_VIDEO_PORTDEFINITIONTYPE
OMX_IndexParamVideoPortFormat	OMX_VIDEO_PARAM_PORTFORMATTYPE
OMX_IndexParamVideoQuantization	OMX_VIDEO_PARAM_QUANTIZATIONTYPE
OMX_IndexParamVideoFastUpdate	OMX_VIDEO_PARAM_VIDEOFASTUPDATETYPE
OMX_IndexParamVideoBitrate	OMX_VIDEO_PARAM_BITRATETYPE
OMX_IndexParamVideoMotionVector	OMX_VIDEO_PARAM_MOTIONVECTORTYPE
OMX_IndexParamVideoIntraRefresh	OMX_VIDEO_PARAM_INTRAREFRESHTYPE
OMX_IndexConfigVideoIntraRefresh	OMX_VIDEO_PARAM_INTRAREFRESHTYPE
OMX_IndexParamVideoErrorCorrection	OMX_VIDEO_PARAM_ERRORCORRECTIONTYPE
OMX_IndexParamVideoVBSMC	OMX_VIDEO_PARAM_VBSMCTYPE
OMX_IndexParamVideoMpeg2	OMX_VIDEO_PARAM_MPEG2TYPE
OMX_IndexParamVideoMpeg4	OMX_VIDEO_PARAM_MPEG4TYPE
OMX_IndexParamVideoWmv	OMX_VIDEO_PARAM_WMVTYPE
OMX_IndexParamVideoRv	OMX_VIDEO_PARAM_RVTYPE

OpenMAX IL Indices (OMX_Index.h)	Corresponding OpenMAX IL Video Structures (OMX_Video.h)
OMX_IndexParamVideoAvc	OMX_VIDEO_PARAM_AVCTYPE
OMX_IndexParamVideoH263	OMX_VIDEO_PARAM_H263TYPE
OMX_IndexParamVideoVp8	OMX_VIDEO_PARAM_VP8TYPE
OMX_IndexConfigVideoVp8ReferenceFrame	OMX_VIDEO_VP8REFERENCEFRAMETYPE
OMX_IndexConfigVideoVp8ReferenceFrameType	OMX_VIDEO_VP8REFERENCEFRAMEINFOTYPE
OMX_IndexParamVideoProfileLevelQuerySupported	OMX_VIDEO_PARAM_PROFILELEVELTYPE
OMX_IndexParamVideoProfileLevelCurrent	OMX_VIDEO_PARAM_PROFILELEVELTYPE
OMX_IndexConfigVideoBitrate	OMX_VIDEO_CONFIG_BITRATETYPE
OMX_IndexConfigVideoFramerate	OMX_CONFIG_FRAMERATETYPE
OMX_IndexConfigVideoIntraVOPRefresh	OMX_CONFIG_INTRAREFRESHVOPTYPE
OMX_IndexConfigVideoIntraMBRefresh	OMX_CONFIG_MACROBLOCKERRORMAPTYPE
OMX_IndexConfigVideoMBErrorReporting	OMX_CONFIG_MBERRORREPORTINGTYPE
OMX_IndexParamVideoMacroblocksPerFrame	OMX_PARAM_MACROBLOCKSTYPE
OMX_IndexConfigVideoMacroBlockErrorMap	OMX_CONFIG_MACROBLOCKERRORMAPTYPE
OMX_IndexParamVideoSliceFMO	OMX_VIDEO_PARAM_AVCSLICEFMO
OMX_IndexConfigVideoAVCIntraPeriod	OMX_VIDEO_CONFIG_AVCINTRAPERIOD
OMX_IndexConfigVideoNalSize	OMX_VIDEO_CONFIG_NALSIZE
OMX_IndexParamNalStreamFormatSupported	OMX_NALSTREAMFORMATTYPE
OMX_IndexParamNalStreamFormat	OMX_NALSTREAMFORMATTYPE
OMX_IndexParamNalStreamFormatSelect	OMX_NALSTREAMFORMATTYPE
OMX_IndexParamVideoVC1	OMX_VIDEO_PARAM_VC1TYPE
OMX_IndexConfigVideoIntraPeriod	OMX_VIDEO_INTRAPERIODTYPE

4.3.4 OMX_VIDEO_PORTDEFINITIONTYPE

The PortDefinition structure defines all of the parameters necessary for the compliant component to set up an input or an output video path. If additional information is needed to define the parameters of the port such as frame rate and bit rate, additional structures shall be sent. For example, to change the bit rate, send the OMX_VIDEO_PARAM_BITRATETYPE structure to supply the extra parameters for the port. The number of video paths for input and output will vary by the type of the video component.

The OMX_VIDEO_PORTDEFINITIONTYPE structure can query the current definition of a video port or set the definition of a video port for a component. The

OMX_VIDEO_PORTDEFINITIONTYPE structure is included as part of the OMX_PARAM_PORTDEFINITIONTYPE structure, it is accessed via the OMX_GetParameter function or the OMX_GetParameter function using the OMX_IndexParamPortDefinition index.

OMX_VIDEO_PORTDEFINITIONTYPE is defined as follows.

```
typedef struct OMX_VIDEO_PORTDEFINITIONTYPE {
    OMX_NATIVE_DEVICETYPE pNativeRender;
    OMX_U32 nFrameWidth;
    OMX_U32 nFrameHeight;
    OMX_S32 nStride;
    OMX_U32 nSliceHeight;
    OMX_U32 nBitrate;
    OMX_U32 xFramerate;
    OMX_BOOL bFlagErrorConcealment;
    OMX_VIDEO_CODINGTYPE eCompressionFormat;
    OMX_COLOR_FORMATTYPE eColorFormat;
    OMX_NATIVE_WINDOWTYPE pNativeWindow;
} OMX_VIDEO_PORTDEFINITIONTYPE;
```

4.3.4.1 Parameters

The parameters for OMX_VIDEO_PORTDEFINITIONTYPE are defined as follows.

- pNativeRender is a platform specific reference for a render object. When the port is on a display sink component, this field is interpreted as a platform specific native display object when non-NULL. If NULL, the component uses the pNativeWindow field.
- nFrameWidth is the width of the data in pixels. If the value is 0x0 for an input port, the component will automatically detect and configure the width. For output ports, the width will be detected during OMX_SetupTunnel.
- nFrameHeight is the height of the data in pixels. If the value is 0x0 for an input port, the component will automatically detect and configure the height. For output ports, the height will be detected during OMX_SetupTunnel.
- nStride is a read-write field indicating the number of bytes per span of an image, where nStride is the amount added to go from span N to span N+1. A negative value for nStride indicates that the data is stored bottom-to-top instead of top-to-bottom. If the value is set to 0, the component automatically computes the value.

  The nStride default shall be determined by the component. There are cases however when the default value for nStride does not match the stride requirements of a used buffer, or that of a tunneled port. The IL client is allowed to overwrite this default value.
- nSliceHeight is a read-write field containing the slice height parameter used when processing uncompressed image data. Buffers received on the port shall contain integer multiples of slices. For more information on the minimum buffer

payload for uncompressed data, see section 4.2.2. If the value is set to 0, the component automatically computes the value.

The `nSliceHeight` default shall be determined by the component. There are cases however when the default value for `nSliceHeight` does not match the stride requirements of a used buffer, or that of a tunneled port. The IL client is allowed to overwrite this default value.

- `nBitrate` is the bit rate in bits per second of the frame to be used on the port if the data is compressed. The value 0x0 is used if the bit rate is unknown, variable or is not needed.
- `xFramerate` is the frame rate is in frames per second. This value is represented in Q16 format. The value 0x0 is used to indicate the frame rate is unknown, variable, or is not needed.
- `bFlagErrorConcealment` is a Boolean value that enables or disables error concealment if it is supported by the port.
- `eCompressionFormat` is the compression format used on the port. If the coding is being used to specify the ENCODE type, then additional work shall be done to configure the exact flavor of the compression to be used. For decode cases where the user application cannot differentiate between MPEG-4 and H.264 bit streams, the codec is responsible for the compression format. When `OMX_VIDEO_CodingUnused` is specified, the `eColorFormat` field is valid. For possible coding types, see Table 4-51.
- `eColorFormat` is the color format of the data for the port. This field is invalid unless the `eCompressionFormat` is `OMX_VIDEO_CodingUnused`. For more information on color format types, see Table 4-35.
- `pNativeWindow` is a platform specific reference for a windows object when being processed as part of a video sink component, otherwise this field is 0.

4.3.4.2 Functionality

When the IL client sets the `nFrameWidth` and `nFrameHeight` of the port for the first time, it can provide a value of 0 for `nStride` and `nSliceHeight`. Upon receiving the `OMX_SetParameter` call, the component computes updated values based on the settings of `nFrameWidth` and `nFrameHeight`. The IL client may retrieve the updated values via `OMX_GetParameter`.

If the IL client wishes to update and override these values, it may do so via OMX_SetParameter provided that the new value(s) are not less than the newly updated values. If the new values cannot be accommodated by the component, the component shall return `OMX_ErrorBadParameter`. This allows the OMX IL client to immediately be informed of the incompatibility.

By setting first a non-zero value for `nStride` (resp. nSliceHeight) and a zero value for `nSliceHeight` (resp. nStride), the IL client is able to benefit from the computation of

another default value for nSliceHeight (resp. nStride). This may be relevant in cases when an override of nStride changes the default nSliceHeight.

Components shall validate nStride and nSliceHeight:

- When it is commanded from OMX_StateLoaded to OMX_StateIdle or during a port enable request. An OMX_EventError event with OMX_ErrorPortsNotCompatible shall be emitted if the port validation fails.
- During a tunnel setup call. OMX_SetupTunnel shall return OMX_ErrorPortsNotCompatible if the port validation fails.

4.3.5 OMX_VIDEO_PARAM_PORTFORMATTYPE

OMX_VIDEO_PARAM_PORTFORMATTYPE is the structure for the port format parameter. It enumerates the various data input/output formats supported by the port.

OMX_VIDEO_PARAM_PORTFORMATTYPE can be used with both OMX_GetParameter and OMX_SetParameter. In the OMX_GetParameter case, the caller specifies all fields and the OMX_GetParameter call returns the value of eFormat. The value of nIndex is the range 0 to N-1, where N is the number of formats supported by the port. There is no need for the port to report N, as the caller can determine N by enumerating all the formats supported by the port. Each port shall support at least one format. If there are no more formats, OMX_GetParameter returns OMX_ErrorNoMore (i.e., nIndex is supplied where the value is N or greater). Ports supply formats in order of preference, which means that higher preference formats are provided with lower values of nIndex.

On OMX_SetParameter, the field in nIndex is ignored. If the format is supported, it is set as the format of the port, and the default values for the format are programmed into the port definition type as a side effect. This allows the caller to query the default values for the format without having to know them in advance.

OMX_VIDEO_PARAM_PORTFORMATTYPE is defined as follows.

```
typedef struct OMX_VIDEO_PARAM_PORTFORMATTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nIndex;
    OMX_VIDEO_CODINGTYPE eCompressionFormat;
    OMX_COLOR_FORMATTYPE eColorFormat;
    OMX_U32 xFramerate;
} OMX_VIDEO_PARAM_PORTFORMATTYPE;
```

4.3.5.1 Parameters

The parameters for OMX_VIDEO_PARAM_PORTFORMATTYPE are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `nIndex` indicates the enumeration index for the format from 0x0 to N-1.
- `eCompressionFormat` is the compression format used on the port. If the coding is being used to specify the ENCODE type, then additional work shall be done to configure the exact flavor of the compression to be used. For decode cases where the user application cannot differentiate between MPEG-4 and H.264 bit streams, the codec is responsible for the compression format. When `OMX_VIDEO_CodingUnused` is specified, the `eColorFormat` field is valid. For possible coding types, see Table 4-51.
- `eColorFormat` is the color format of the data for the port. This field is invalid unless the `eCompressionFormat` is `OMX_VIDEO_CodingUnused`. For more information on color format types, see Table 4-32: Uncompressed Data Formats
- `xFramerate` indicates the desired full frame rate is frames per second. This value is represented in Q16 format

4.3.6 OMX_VIDEO_PARAM_QUANTIZATIONTYPE

Quantization controls the compression used during the discrete cosine transform (DCT) step of video encoding. This generic structure is shared between several video standards. The structure allows independent settings of quantization factors for I, P, and B video frames. The structure is not applicable to variable bit rate encoding or constant rate encoding. Not all video standards support independent settings of quantization factors for different frame types.

`OMX_VIDEO_PARAM_QUANTIZATIONTYPE` is defined as follows.

```
typedef struct OMX_VIDEO_PARAM_QUANTIZATIONTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nQpI;
    OMX_U32 nQpP;
    OMX_U32 nQpB;
} OMX_VIDEO_PARAM_QUANTIZATIONTYPE;
```

4.3.6.1 Parameters

The parameters for `OMX_VIDEO_PARAM_QUANTIZATIONTYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `nQpI` is the quantization parameter for I frames.
- `nQpP` is the quantization parameter for P frames.
- `nQpB` is the quantization parameter for bi-directional (B) frames).

4.3.6.2 Dependencies

This parameter is only applicable to certain video encoders, which include MPEG-2 and MPEG-4.

4.3.7 OMX_VIDEO_PARAM_VIDEOFASTUPDATETYPE

Video fast update is a shared parameter between multiple video encoding standards (for example, H.261 and H.263) that specifies fast update parameters for the video encoder.

`OMX_VIDEO_PARAM_VIDEOFASTUPDATETYPE` is defined as follows.

```
typedef struct OMX_VIDEO_PARAM_VIDEOFASTUPDATETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bEnableVFU;
    OMX_U32 nFirstGOB;
    OMX_U32 nFirstMB;
    OMX_U32 nNumMBs;
} OMX_VIDEO_PARAM_VIDEOFASTUPDATETYPE;
```

4.3.7.1 Parameters

The parameters for `OMX_VIDEO_PARAM_VIDEOFASTUPDATETYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `bEnableVFU` is a Boolean value that enables or disables video fast update.
- `nFirstGOB` contains the number of the first row of macroblocks
- `nFirstMB` is the location of the first macroblock row relative to the first group of blocks (GOB).
- `nNumMBs` The number of macroblocks to be refreshed from the `nFirstGOB` and `nFirstMB`.

4.3.7.2 Dependencies

This parameter is only applicable to certain video encoders, such as H.261 and H.263.

4.3.8 OMX_VIDEO_PARAM_BITRATETYPE

Video encode bit rate control for variable bit rate video encoders is shared between multiple video encode standards, and is specified before starting video encoding.

`OMX_VIDEO_PARAM_BITRATETYPE` is defined as follows.

```
typedef struct OMX_VIDEO_PARAM_BITRATETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
```

```
    OMX_VIDEO_CONTROLRATETYPE eControlRate;
    OMX_U32 nTargetBitrate;
} OMX_VIDEO_PARAM_BITRATETYPE;
```

4.3.8.1 Parameters

The parameters for OMX_VIDEO_PARAM_BITRATETYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- eControlRate is an enumerated value that sets the bit rate control. If enabled, the type of bit rate control is specified as constant, variable, constant with frame skipping, or variable with frame skipping. Table 4-54 enumerates the possible video bit rate control types for OMX_VIDEO_CONTROLRATETYPE.

Table 4-54: Supported Video Bit Rate Control Types

Field Name	Bit Rate Control Descriptions
OMX_Video_ControlRateDisable	Disable – in this mode the encoder will ignore nTargetBitrate setting and use the appropriate Qp (nQpI, nQpP, nQpB) values for encoding
OMX_Video_ControlRateVariable	Variable bit rate
OMX_Video_ControlRateConstant	Constant bit rate – the encoder can modify the Qp values to meet the nTargetBitrate target
OMX_Video_ControlRateVariableSkipFrames	Variable bit rate with frame skipping
OMX_Video_ControlRateConstantSkipFrames	Constant bit rate with frame skipping – the encoder cannot modify the Qp values to meet the nTargetBitrate target. Instead, the encoder can drop frames to achieve nTargetBitrate

- nTargetBitrate is the target bit rate for video encoding in units of bits per second.

4.3.8.2 Dependencies

This parameter is only applicable to certain video encoders. For some video encode standards, the bit rate is specified as part of the standard and is not programmable (i.e., value can only be queried).

4.3.9 OMX_VIDEO_PARAM_MOTIONVECTORTYPE

The motion vector parameters used during video encoding are programmable for certain video standards. These parameters can be shared between multiple video standards algorithms, although certain fields only pertain to particular video standards.

OMX_VIDEO_PARAM_MOTIONVECTORTYPE is defined as follows.

```
typedef struct OMX_VIDEO_PARAM_MOTIONVECTORTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_VIDEO_MOTIONVECTORTYPE eAccuracy;
    OMX_BOOL bUnrestrictedMVs;
    OMX_BOOL bFourMV;
    OMX_S32 sXSearchRange;
    OMX_S32 sYSearchRange;
} OMX_VIDEO_PARAM_MOTIONVECTORTYPE;
```

4.3.9.1 Parameters

The parameters for OMX_VIDEO_PARAM_MOTIONVECTORTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- eAccuracy is an enumerated value that specifies the pixel accuracy of the motion vector search during video encode. Accuracy is 1, 1/2, 1/4, or 1/8 pixel. The eAccuracy setting indicates that all larger value motion vector search ranges are also used (i.e., a value of 1/4 indicates motion vectors are also searched on 1 and 1/2 intervals). Table 4-55 enumerates the possible video motion vector types for OMX_VIDEO_MOTIONVECTORTYPE.

Table 4-55: Supported Video Motion Vector Types

Field Name	Motion Vector Descriptions
OMX_Video_MotionVectorPixel	Full pixel motion vectors
OMX_Video_MotionVectorHalfPel	Half pixel motion vectors
OMX_Video_MotionVectorQuarterPel	Quarter pixel motion vectors
OMX_Video_MotionVectorEighthPel	Eighth pixel motion vectors

- bUnrestrictedMVs is a Boolean value that enables unrestricted motion vectors.
- bFourMV is a Boolean value enables using four motion vectors.
- sXSearchRange is the search range of the X motion vector in pixels for video encoders where this is programmable. For example, a search range of 4 indicates a ±4 search area both horizontally and vertically.
- sYSearchRange is the search range of the Y motion vector in pixels for video encoders where this is programmable. For example, a search range of 4 indicates a ±4 search area both horizontally and vertically.

4.3.9.2 Dependencies

This parameter is only applicable to certain video encoders, which include MPEG2 and MPEG4.

4.3.10 OMX_VIDEO_PARAM_INTRAREFRESHTYPE

OMX_VIDEO_PARAM_INTRAREFRESHTYPE contains common parameters for controlling the intra-refresh rate for macroblocks during video encoding. Refresh causes macroblocks of a video stream to be regularly encoded as reference macroblocks. This enables a video decoder to eventually reconstruct a good video image from multiple frames when data is lost or corrupted without receiving a new intra-coded frame.

OMX_VIDEO_PARAM_INTRAREFRESHTYPE is defined as follows.

```
typedef struct OMX_VIDEO_PARAM_INTRAREFRESHTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_VIDEO_INTRAREFRESHTYPE eRefreshMode;
    OMX_U32 nAirMBs;
    OMX_U32 nAirRef;
    OMX_U32 nCirMBs;
} OMX_VIDEO_PARAM_INTRAREFRESHTYPE;
```

4.3.10.1 Parameters

The parameters for OMX_VIDEO_PARAM_INTRAREFRESHTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- eRefreshMode is the enumeration for the type of intra-refresh mode. Table 4-56 shows the possible values for OMX_VIDEO_INTRAREFRESHTYPE.

Table 4-56: Supported Video Intra-Refresh Types

Field Name	Intra-Refresh Descriptions
OMX_VIDEO_IntraRefreshCyclic	Cyclic intra-refresh
OMX_VIDEO_IntraRefreshAdaptive	Adaptive intra-refresh
OMX_VIDEO_IntraRefreshBoth	Cyclic and Adaptive intra-refresh

- nAirMBs is the minimum number of macroblocks to refresh in a frame when adaptive intra-refresh (AIR) is enabled.
- nAirRef is the number of times a motion marked macroblock has to be intra-coded.
- nCirMBs is the number of consecutive macroblocks to be coded as intra when cyclic intra-refresh (CIR) is enabled.

4.3.10.2 Dependencies

This parameter is only applicable to certain video encoders, which includes MPEG4.

4.3.11 OMX_VIDEO_PARAM_ERRORCORRECTIONTYPE

OMX_VIDEO_PARAM_ERRORCORRECTIONTYPE contains common video encoding standard parameters for handling error correction during video encoding.

OMX_VIDEO_PARAM_ERRORCORRECTIONTYPE is defined as follows.

```
typedef struct OMX_VIDEO_PARAM_ERRORCORRECTIONTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bEnableHEC;
    OMX_BOOL bEnableResync;
    OMX_U32  nResynchMarkerSpacing;
    OMX_BOOL bEnableDataPartitioning;
    OMX_BOOL bEnableRVLC;
} OMX_VIDEO_PARAM_ERRORCORRECTIONTYPE;
```

4.3.11.1 Parameters

The parameters for OMX_VIDEO_PARAM_ERRORCORRECTIONTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bEnableHEC is a Boolean value that enables or disables header extension codes.
- bEnableResync is a Boolean value that enables or disables resynchronization markers.
- nResynchMarkerSpacing is the resynchronization marker interval in bits applied to the stream.
- bEnableDataPartitioning is a Boolean value that enables or disables data partitioning.
- bEnableRVLC is a Boolean value that enables or disables reversible variable-length coding.

4.3.11.2 Dependencies

This parameter is only applicable to certain video encoders, which includes MPEG4.

4.3.12 OMX_VIDEO_PARAM_VBSMCTYPE

OMX_VIDEO_PARAM_VBSMCTYPE contains common video encoding standard parameters for selecting variable block size motion compensation during video encoding.

OMX_VIDEO_PARAM_VBSMCTYPE is defined as follows.

```
typedef struct OMX_VIDEO_PARAM_VBSMCTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
```

```
    OMX_BOOL b16x16;
    OMX_BOOL b16x8;
    OMX_BOOL b8x16;
    OMX_BOOL b8x8;
    OMX_BOOL b8x4;
    OMX_BOOL b4x8;
    OMX_BOOL b4x4;
} OMX_VIDEO_PARAM_VBSMCTYPE;
```

4.3.12.1 Parameters

The parameters for `OMX_VIDEO_PARAM_VBSMCTYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `b16x16` is a Boolean value that enables or disables inter-block search in a 16 by 16 region of pixels
- `b16x8` is a Boolean value that enables or disables inter-block search in a 16 by 8 region of pixels
- `b8x16` is a Boolean value that enables or disables inter-block search in a 8 by 16 region of pixels
- `b8x8` is a Boolean value that enables or disables inter-block search in a 8 by 8 region of pixels
- `b8x4` is a Boolean value that enables or disables inter-block search in a 8 by 4 region of pixels
- `b4x8` is a Boolean value that enables or disables inter-block search in a 4 by 8 region of pixels
- `b4x4` is a Boolean value that enables or disables inter-block search in a 4 by 4 region of pixels

4.3.12.2 Dependencies

This parameter is only applicable to certain video encoders, which include MPEG4 and other derivations of MPEG4.

4.3.13 OMX_VIDEO_PARAM_H263TYPE

H.263 is a video standard defined by the ITU. Parameters for this video standard are controlled using the `OMX_VIDEO_PARAM_H263TYPE` structure.

`OMX_VIDEO_PARAM_H263TYPE` is defined as follows.

```
typedef struct OMX_VIDEO_PARAM_H263TYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nPFrames;
    OMX_U32 nBFrames;
```

```
    OMX_VIDEO_H263PROFILETYPE eProfile;
    OMX_VIDEO_H263LEVELTYPE eLevel;
    OMX_BOOL bPLUSPTYPEAllowed;
    OMX_U32 nAllowedPictureTypes;
    OMX_BOOL bForceRoundingTypeToZero;
    OMX_U32 nPictureHeaderRepetition;
    OMX_U32 nGOBHeaderInterval;
} OMX_VIDEO_PARAM_H263TYPE;
```

4.3.13.1 Parameters

The parameters for `OMX_VIDEO_PARAM_H263TYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `nPFrames` is the number of P frames between I frames.
- `nBFrames` is the number of B frames between I frames.
- `eProfile` is the profile type supported for encoding and decoding H.263 content. Table 4-57 shows the possible H.263 video profile types for `OMX_VIDEO_H263PROFILETYPE`.

Table 4-57: Supported H.263 Profile Types

Field Name	H.263 Profile Descriptions
`OMX_VIDEO_H263ProfileUnknown`	Unknown, unused or not required profile setting.
`OMX_VIDEO_H263ProfileBaseline`	H.263 Baseline Profile: H.263 (V1), no optional modes
`OMX_VIDEO_H263ProfileH320Coding`	H.263 Coding Efficiency (H.320) Backward Compatibility Profile: H.263+ (V2), includes annexes I, J, L.4, and T
`OMX_VIDEO_H263ProfileBackward Compatible`	H.263 BackwardCompatible: Backward Compatibility Profile: H.263 (V1), includes annex F
`OMX_VIDEO_H263ProfileISWV2`	H.263 Interactive Streaming Wireless Profile: H.263+ (V2), includes annexes I, J, K, and T
`OMX_VIDEO_H263ProfileISWV3`	H.263 Interactive Streaming Wireless Profile: H.263++ (V3), includes profile 3 and annexes V and W.6.3.8
`OMX_VIDEO_H263ProfileHigh Compression`	H.263 Conversational High Compression Profile: H.263++ (V3), includes profiles 1 and 2 and annexes D and U
`OMX_VIDEO_H263ProfileInternet`	H.263 Conversational Internet Profile: H.263++ (V3), includes profile 5 and annex K
`OMX_VIDEO_H263ProfileInterlace`	H.263 Conversational Interlace Profile: H.263++ (V3), includes profile 5 and annex W.6.3.11

Field Name	H.263 Profile Descriptions
OMX_VIDEO_H263ProfileHighLatency	H.263 High Latency Profile: H.263++ (V3), includes profile 6 and annexes O.1 and P.5

- eLevel is the maximum processing level that an encoder or decoder supports for a particular profile. Table 4-58 shows the possible H.263 video level types.

Table 4-58: Supported H.263 Level Types

Field Name	H.263 Level Descriptions
OMX_VIDEO_H263LevelUnknown	Unknown, unused or not required setting.
OMX_VIDEO_H263Level10	H.263 level 10
OMX_VIDEO_H263Level20	H.263 level 20
OMX_VIDEO_H263Level30	H.263 level 30
OMX_VIDEO_H263Level40	H.263 level 40
OMX_VIDEO_H263Level45	H.263 level 45
OMX_VIDEO_H263Level50	H.263 level 50
OMX_VIDEO_H263Level60	H.263 level 60
OMX_VIDEO_H263Level70	H.263 level 70

- bPLUSPTYPEAllowed is a Boolean value that enables or disables indication of whether PLUSPTYPE (specified in the 1998 version of H.263) is allowed. This applies to custom picture sizes or clock frequencies.
- nAllowedPictureTypes determines whether picture types are allowed in the bit stream. For more information on picture types, see Table 4-52.
- bForceRoundingTypeToZero determines whether the value of the RTYPE bit (bit 6 of MPPTYPE) is not constrained. Change the value of the RTYPE bit for each reference picture in error-free communication.
- nPictureHeaderRepetition is the frequency of picture header repetition.
- nGOBHeaderInterval is the interval of non-empty GOB headers in units of GOBs. A value of zero for this parameter indicates that all GOB headers will be empty.

4.3.13.2 Dependencies

This parameter is only applicable when the port is configured for H.263.

4.3.14 OMX_VIDEO_PARAM_MPEG2TYPE

OMX_VIDEO_PARAM_MPEG2TYPE contains MPEG2 video parameters for controlling MPEG2 video encode.

OMX_VIDEO_PARAM_MPEG2TYPE is defined as follows.

```
typedef struct OMX_VIDEO_PARAM_MPEG2TYPE {
```

```
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nPFrames;
    OMX_U32 nBFrames;
    OMX_VIDEO_MPEG2PROFILETYPE eProfile;
    OMX_VIDEO_MPEG2LEVELTYPE eLevel;
} OMX_VIDEO_PARAM_MPEG2TYPE;
```

4.3.14.1 Parameters

The parameters for OMX_VIDEO_PARAM_MPEG2TYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nPFrames is the number of P frames between I frames.
- nBFrames is the number of B frames between I frames.
- eProfile is the maximum processing level that an encoder or decoder supports for a particular profile. Table 4-59 shows the possible MPEG-2 video profile types in OMX_VIDEO_MPEG2PROFILETYPE.

Table 4-59: Supported MPEG-2 Profile Types

Field Name	MPEG-2 Profile Descriptions
OMX_VIDEO_MPEG2ProfileUnknown	Unknown, unused or not required profile setting.
OMX_VIDEO_MPEG2ProfileSimple	Simple profile
OMX_VIDEO_MPEG2ProfileMain	Main profile
OMX_VIDEO_MPEG2Profile422	4:2:2 profile
OMX_VIDEO_MPEG2ProfileSNR	SNR profile
OMX_VIDEO_MPEG2ProfileSpatial	Spatial profile
OMX_VIDEO_MPEG2ProfileHigh	High profile

- eLevel is the maximum processing level that an MPEG-2 encoder or decoder supports for a particular profile. Table 4-60 shows the possible MPEG-2 video level types in OMX_VIDEO_MPEG2LEVELTYPE.

Table 4-60: Supported MPEG-2 Level Types

Field Name	MPEG-2 Level Descriptions
OMX_VIDEO_MPEG2LevelUnknown	Unknown, unused or not required setting.
OMX_VIDEO_MPEG2LevelLL	Low level
OMX_VIDEO_MPEG2LevelML	Main level
OMX_VIDEO_MPEG2LevelH14	High 1440 level
OMX_VIDEO_MPEG2LevelHL	High level

4.3.14.2 Dependencies

This parameter is only applicable when the port is configured for MPEG-2.

4.3.15 OMX_VIDEO_PARAM_MPEG4TYPE

OMX_VIDEO_PARAM_MPEG4TYPE contains the MPEG-4 video parameters for controlling MPEG-4 video encoding and decoding.

OMX_VIDEO_PARAM_MPEG4TYPE is defined as follows.

```
typedef struct OMX_VIDEO_PARAM_MPEG4TYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nSliceHeaderSpacing;
    OMX_BOOL bSVH;
    OMX_BOOL bGov;
    OMX_U32 nPFrames;
    OMX_U32 nBFrames;
    OMX_U32 nIDCVLCThreshold;
    OMX_BOOL bACPred;
    OMX_U32 nMaxPacketSize;
    OMX_U32 nTimeIncRes;
    OMX_VIDEO_MPEG4PROFILETYPE eProfile;
    OMX_VIDEO_MPEG4LEVELTYPE eLevel;
    OMX_U32 nAllowedPictureTypes;
    OMX_U32 nHeaderExtension;
    OMX_BOOL bReversibleVLC;
} OMX_VIDEO_PARAM_MPEG4TYPE;
```

4.3.15.1 Parameters

The parameters for OMX_VIDEO_PARAM_MPEG4TYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nSliceHeaderSpacing is the number of macroblocks in a slice (H263+ Annex K). This value shall be zero if not used.
- bSVH is a Boolean value that enables or disables short header mode.

- `bGov` is a Boolean value that enables or disables group of VOP (GOV), where VOP is the abbreviation for video object planes.
- `nPFrames` is the number of P frames between I frames.
- `nBFrames` is the number of B frames between I frames.
- `nIDCVLCThreshold` is the value of the intra-DC variable-length coding (VLC) threshold.
- `bACPred` is the Boolean value that enables or disables AC prediction.
- `nMaxPacketSize` is the maximum size of the packet in bytes.
- `nTimeIncRes` is the VOP time increment resolution for MPEG-4. This value is interpreted as described in the MPEG-4 standard.
- `eProfile` is the profile used for MPEG-4 encoding or decoding. Table 4-61 shows the possible MPEG-4 video profile types in `OMX_VIDEO_MPEG4PROFILETYPE`.

Table 4-61: Supported MPEG-4 Profile Types

Field Name	MPEG-4 Profile Descriptions
`OMX_VIDEO_MPEG4ProfileUnknown`	Unknown, unused or not required profile setting.
`OMX_VIDEO_MPEG4ProfileSimple`	MPEG-4 Simple Profile, Levels 1-3
`OMX_VIDEO_MPEG4ProfileSimpleScalable`	MPEG-4 Simple Scalable Profile, Levels 1-2
`OMX_VIDEO_MPEG4ProfileCore`	MPEG-4 Core Profile, Levels 1-2
`OMX_VIDEO_MPEG4ProfileMain`	MPEG-4 Main Profile, Levels 2-4
`OMX_VIDEO_MPEG4ProfileNbit`	MPEG-4 N-bit Profile, Level 2
`OMX_VIDEO_MPEG4ProfileScalableTexture`	MPEG-4 Scalable Texture Profile, Level 1
`OMX_VIDEO_MPEG4ProfileSimpleFace`	MPEG-4 Simple Face Animation Profile, Levels 1-2
`OMX_VIDEO_MPEG4ProfileSimpleFBA`	MPEG-4 Simple Face and Body Animation (FBA) Profile, , Levels 1-2
`OMX_VIDEO_MPEG4ProfileBasicAnimated`	MPEG-4 Basic Animated Texture Profile, Levels 1-2
`OMX_VIDEO_MPEG4ProfileHybrid`	MPEG-4 Hybrid Profile, Levels 1-2
`OMX_VIDEO_MPEG4ProfileAdvancedRealTime`	MPEG-4 Advanced Real Time Simple Profiles, Levels 1-4
`OMX_VIDEO_MPEG4ProfileCoreScalable`	MPEG-4 Core Scalable Profile, Levels 1-3
`OMX_VIDEO_MPEG4ProfileAdvancedCoding`	MPEG-4 Advanced Coding Efficiency Profile, Levels 1-4

Field Name	MPEG-4 Profile Descriptions
OMX_VIDEO_MPEG4ProfileAdvancedCore	MPEG-4 Advanced Core Profile, Levels 1-2
OMX_VIDEO_MPEG4ProfileAdvancedScalable	MPEG-4 Advanced Scalable Texture, Levels 2-3
OMX_VIDEO_MPEG4ProfileAdvancedSimple	MPEG-4 Advanced Simple Profile

- eLevel is the maximum processing level that an encoder or decoder supports for a particular MPEG-4 profile. Table 4-62 shows the possible MPEG-4 video level types in OMX_VIDEO_MPEG4LEVELTYPE.

Table 4-62: Supported MPEG-4 Level Types

Field Name	MPEG-4 Level Descriptions
OMX_VIDEO_MPEG4LevelUnknown	Unknown, unused or not required setting.
OMX_VIDEO_MPEG4Level0	Level 0
OMX_VIDEO_MPEG4Level0b	Level 0b
OMX_VIDEO_MPEG4Level1	Level 1
OMX_VIDEO_MPEG4Level2	Level 2
OMX_VIDEO_MPEG4Level3	Level 3
OMX_VIDEO_MPEG4Level4	Level 4
OMX_VIDEO_MPEG4Level4a	Level 4a
OMX_VIDEO_MPEG4Level5	Level 5

- nAllowedPictureTypes identifies the picture types allowed in the bit stream. For more information on picture types, see Table 4-52: Supported Video Picture Types.
- nHeaderExtension specifies the number of consecutive video packets between header extension codes (conversely, insert a header extension code every nHeaderExtension number of packets).
- bReversibleVLC is a Boolean value that enables or disables the use of reversible variable-length coding

4.3.15.2 Dependencies

This parameter is only applicable when the port is configured for MPEG-4.

4.3.16 OMX_VIDEO_PARAM_WMVTYPE

OMX_VIDEO_PARAM_WMVTYPE contains common standard video decoder parameters that control Windows Media formats, including WMV7, WMV8, and WMV9.

OMX_VIDEO_PARAM_WMVTYPE is defined as follows.

```
typedef struct OMX_VIDEO_PARAM_WMVTYPE {
```

```
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_VIDEO_WMVFORMATTYPE eFormat;
} OMX_VIDEO_PARAM_WMVTYPE;
```

4.3.16.1 Parameters

The parameters for OMX_VIDEO_PARAM_WMVTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- eFormat is the enumerated format of the data stream. Table 4-63 shows the possible Windows Media video format types for OMX_VIDEO_WMVFORMATTYPE.

Table 4-63: Supported Windows Media Video Format Types

Field Name	Windows Media Video Format Descriptions
OMX_VIDEO_WMVFormatUnknown	Unknown, unused or not required setting.
OMX_VIDEO_WMVFormatUnused	Format unused or unknown
OMX_VIDEO_WMVFormat7	Windows Media video format 7
OMX_VIDEO_WMVFormat8	Windows Media video format 8
OMX_VIDEO_WMVFormat9	Windows Media video format 9

4.3.16.2 Dependencies

This parameter is only applicable when the port is configured for Windows Media video.

4.3.17 OMX_VIDEO_PARAM_RVTYPE

OMX_VIDEO_PARAM_RVTYPE contains common standard video decoder parameters that control RealVideo formats, including RealVideo 8 and RealVideo 9.

OMX_VIDEO_PARAM_RVTYPE is defined as follows.

```
typedef struct OMX_VIDEO_PARAM_RVTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_VIDEO_RVFORMATTYPE eFormat;
    OMX_U16 nBitsPerPixel;
    OMX_U16 nPaddedWidth;
    OMX_U16 nPaddedHeight;
    OMX_U32 nFrameRate;
    OMX_U32 nBitstreamFlags;
```

```
    OMX_U32 nBitstreamVersion;
    OMX_U32 nMaxEncodeFrameSize;
    OMX_BOOL bEnablePostFilter;
    OMX_BOOL bEnableTemporalInterpolation;
    OMX_BOOL bEnableLatencyMode;
} OMX_VIDEO_PARAM_RVTYPE;
```

4.3.17.1 Parameters

The parameters for OMX_VIDEO_PARAM_RVTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- eFormat is the video format. Table 4-64 shows the possible RealVideo video format types in OMX_VIDEO_RVFORMATTYPE.

Table 4-64: Supported RealVideo Format Types

Field Name	RV Format Descriptions
OMX_VIDEO_RVFormatUnknown	Unknown, unused or not required setting.
OMX_VIDEO_RVFormatUnused	Format unused or unknown
OMX_VIDEO_RVFormat8	RealVideo 8 format
OMX_VIDEO_RVFormat9	RealVideo 9 format
OMX_VIDEO_RVFormatG2	RealVideo G2 format

- nBitsPerPixel is the number of bits per pixel coded in the frame.
- nPaddedWidth is the padded width in pixels of a video frame.
- nPaddedWidth is the padded width in pixels of a video frame.
- nFrameRate is the rate of the video in frames per second as a 32-bit fixed point value in which the upper 16 bits are the integer part and the lower 16 bits are the fractional part.
- nBitstreamFlags is a 32 bit integer containing flags which provide internal information about the bitstream to the codec. These will be interpreted differently depending on the bitstream format and version.
- nBitstreamVersion is a 32 bit integer containing the bitstream version.
- nMaxEncodeFrameSize is the size in bytes of the largest encoded frame (defined only for OMX_VIDEO_RVFormat9).
- bEnablePostFilter is a Boolean value that enables or disables the post filter.
- bEnableTemporalInterpolation a Boolean value that enables or disables the temporal interpolation.

- `bEnableLatencyMode` is a Boolean value that enables or disables the decoder from displaying a decoded frame until it has detected that no enhancement layer frames or dependent B frames will be coming. This detection usually occurs when a subsequent non-B frame is encountered.

4.3.17.2 Dependencies

This parameter is only applicable when the port is configured for RealVideo.

4.3.18 OMX_VIDEO_PARAM_AVCTYPE

MPEG4 P10 Advanced Video Coding (AVC) is commonly referred to as H.264 which is a video standard defined by the Joint Video Team (JVT). Parameters for this video standard are controlled using the `OMX_VIDEO_PARAM_AVCTYPE` structure.

`OMX_VIDEO_PARAM_AVCTYPE` is defined as follows.

```
typedef struct OMX_VIDEO_PARAM_AVCTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nSliceHeaderSpacing;
    OMX_U32 nPFrames;
    OMX_U32 nBFrames;
    OMX_BOOL bUseHadamard;
    OMX_U32 nRefFrames;
    OMX_U32 nRefIdx10ActiveMinus1;
    OMX_U32 nRefIdx11ActiveMinus1;
    OMX_BOOL bEnableUEP;
    OMX_BOOL bEnableFMO;
    OMX_BOOL bEnableASO;
    OMX_BOOL bEnableRS;
    OMX_VIDEO_AVCPROFILETYPE eProfile;
    OMX_VIDEO_AVCLEVELTYPE eLevel;
    OMX_U32 nAllowedPictureTypes;
    OMX_BOOL bFrameMBsOnly;
    OMX_BOOL bMBAFF;
    OMX_BOOL bEntropyCodingCABAC;
    OMX_BOOL bWeightedPPrediction;
    OMX_U32 nWeightedBipredicitonMode;
    OMX_BOOL bconstIpred ;
    OMX_BOOL bDirect8x8Inference;
    OMX_BOOL bDirectSpatialTemporal;
    OMX_U32 nCabacInitIdc;
    OMX_VIDEO_AVCLOOPFILTERTYPE eLoopFilterMode;

} OMX_VIDEO_PARAM_AVCTYPE;
```

4.3.18.1 Parameters

The parameters for `OMX_VIDEO_PARAM_AVCTYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.

- `nSliceHeaderSpacing` is the number of macroblocks in a slice. This value is set to 0x0 when not used.
- `nPFrames` is the number of P frames between I frames.
- `nBFrames` is the number of B frames between I frames.
- `bUseHadamard` is a Boolean value that enables or disables the Hadamard transform.
- `nRefFrames` is the number of reference frames in the range 1 to 16 that are used for inter-motion search.
- `nRefIdx10ActiveMinus1` is the picture parameter set reference frame index, which is the index into the reference frame buffer of the trailing frames list. This value supports B frames.
- `nRefIdx11ActiveMinus1` is the picture parameter set reference frame index, which is the index into the reference frame buffer of the forward frames list. This value supports B frames.
- `bEnableUEP` is a Boolean value that enables or disables unequal error protection. This parameter is only applicable if data partitioning is enabled.
- `bEnableFMO` is a Boolean value that enables or disables flexible macroblock ordering.
- `bEnableASO` is a Boolean value that enables or disables for arbitrary slice ordering.
- `bEnableRS` is a Boolean value enables or disables sending redundant slices.
- `eProfile` is the profile used for the types of AVC encoding or decoding that are supported. Table 4-65 shows the possible AVC video profile types in `OMX_VIDEO_AVCPROFILETYPE`.

Table 4-65: Supported AVC Profile Types

Field Name	AVC Profile Descriptions
`OMX_VIDEO_AVCProfileUnknown`	Unknown, unused or not required profile setting.
`OMX_VIDEO_AVCProfileBaseline`	Baseline profile
`OMX_VIDEO_AVCProfileMain`	Main profile
`OMX_VIDEO_AVCProfileExtended`	Extended profile
`OMX_VIDEO_AVCProfileHigh`	High profile
`OMX_VIDEO_AVCProfileHigh10`	High 10 profile
`OMX_VIDEO_AVCProfileHigh422`	High 4:2:2 profile
`OMX_VIDEO_AVCProfileHigh444`	High 4:4:4 profile

- `eLevel` is the maximum processing level that an AVC encoder or decoder supports for a particular profile. Table 4-66 shows the possible AVC video level types in `OMX_VIDEO_AVCLEVELTYPE`.

Table 4-66: Supported AVC Level Types

Field Name	AVC Level Descriptions
`OMX_VIDEO_AVCLevelUnknown`	Unknown, unused or not required setting.
`OMX_VIDEO_AVCLevel1`	AVC level 1
`OMX_VIDEO_AVCLevel1b`	AVC level 1b
`OMX_VIDEO_AVCLevel11`	AVC level 1.1
`OMX_VIDEO_AVCLevel12`	AVC level 1.2
`OMX_VIDEO_AVCLevel13`	AVC level 1.3
`OMX_VIDEO_AVCLevel2`	AVC level 2
`OMX_VIDEO_AVCLevel21`	AVC level 2.1
`OMX_VIDEO_AVCLevel22`	AVC level 2.2
`OMX_VIDEO_AVCLevel3`	AVC level 3
`OMX_VIDEO_AVCLevel31`	AVC level 3.1
`OMX_VIDEO_AVCLevel32`	AVC level 3.2
`OMX_VIDEO_AVCLevel4`	AVC level 4
`OMX_VIDEO_AVCLevel41`	AVC level 14.1
`OMX_VIDEO_AVCLevel42`	AVC level 4.2
`OMX_VIDEO_AVCLevel5`	AVC level 5
`OMX_VIDEO_AVCLevel51`	AVC level 5.1

- `nAllowedPictureTypes` identifies the allowed picture types in the bit stream.
- `bFrameMBsOnly` is a Boolean value indicating that every coded picture of the coded video sequence is a coded frame containing only frame macroblocks.
- `bMBAFF` is a Boolean value that enables or disables macroblock adaptive frame and field (MBAFF) support within a picture.
- `bEntropyCodingCABAC` is a Boolean value that enables or disables the entropy decoding method.
- `bWeightedPPrediction` is a Boolean value that enables or disables weighted prediction applied to P and SP slices.
- `nWeightedBipredicitonMode` is the default weighted prediction applied to B slices.
- `bconstIpred` is a Boolean value that enables or disables intra-prediction.

- `bDirect8x8Inference` specifies the method used in the derivation process for luma motion vectors for `B_Skip`, `B_Direct_16x16`, and `B_Direct_8x8` as specified in subclause 8.4.1.2 of the AVC spec.
- `bDirectSpatialTemporal` is a flag that indicates the spatial or temporal direct mode used in B-slice coding, which is related to `bDirect8x8Inference` . Spatial direct mode is the default.
- `nCabacInitIdx` is the index used to initialize Context-based Adaptive Binary Arithmetic Coding (CABAC) contexts.
- `eLoopFilterMode` enables or disables the AVC loop filter. Table 4-67 shows the possible AVC video coding loop filter types in `OMX_VIDEO_AVCLOOPFILTERTYPE`.

Table 4-67: Supported AVC Loop Filter Types

Field Name	AVC Loop Filter Level Descriptions
`OMX_VIDEO_AVCLoopFilterEnable`	Enables AVC loop filter
`OMX_VIDEO_AVCLoopFilterDisable`	Disables AVC loop filter
`OMX_VIDEO_AVCLoopFilterDisable SliceBoundary`	Disables AVC loop filter on slice boundary

4.3.18.2 Dependencies

This parameter is only applicable when the port is configured for AVC.

4.3.19 OMX_VIDEO_PARAM_VP8TYPE

`OMX_VIDEO_PARAM_VP8TYPE` contains the VP8 video parameters for controlling VP8 video encoding and decoding.

`OMX_VIDEO_PARAM_VP8TYPE` is defined as follows.

```
typedef struct OMX_VIDEO_PARAM_VP8TYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_VIDEO_VP8PROFILETYPE eProfile;
    OMX_VIDEO_VP8LEVELTYPE eLevel;
    OMX_U32 nDCTPartitions;
    OMX_BOOL bErrorResilientMode;
} OMX_VIDEO_PARAM_VP8TYPE;
```

4.3.19.1 Parameters

The parameters for `OMX_VIDEO_PARAM_VP8TYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `eProfile` is the profile used for the types of VP8 encoding or decoding that are supported.

Table 4-68 shows the possible VP8 video profile types in OMX_VIDEO_VP8PROFILETYPE.

Table 4-68: Supported VP8 Profile Types

Field Name	VP8 Profile Descriptions
OMX_VIDEO_VP8ProfileUnknown	Unknown, unused or not required profile setting.
OMX_VIDEO_VP8ProfileMain	VP8 Main profile

- eLevel is the level used for VP8 encoding or decoding.

  Table 4-69 shows the possible VP8 video level types in OMX_VIDEO_VP8LEVELTYPE.

Table 4-69: Supported VP8 Level Types

Field Name	VP8 Level Descriptions
OMX_VIDEO_VP8LevelUnknown	Unknown, unused or not required setting
OMX_VIDEO_VP8Level_Version0	VP8 Level "Version 0"
OMX_VIDEO_VP8Level_Version1	VP8 Level "Version 1"
OMX_VIDEO_VP8Level_Version2	VP8 Level "Version 2"
OMX_VIDEO_VP8Level_Version3	VP8 Level "Version 3"

In VP8 certain decoding tools are enabled or disabled based on the eLevel and higher level means less decoding complexity.

Table 4-70 shows which decoding tools are enabled or disabled.

Table 4-70: Decoding Tools Usage Based on VP8 level

Level	Reconstruction Filter	Loop Filter
"Version 0"	Bicubic	Normal
"Version 1"	Bilinear	Simple
"Version 2"	Bilinear	None
"Version 3"	None	None

- nDCTPartitions specifies the number of DCT coefficient data partitions within a compressed frame. Using more than 1 partition may allow more effective multi-threaded decoding.

  Table 4-71 shows the possible values for nDCTPartitions.

Table 4-71: nDCTPartitions Values

Value	Description
0	1 DCT residual partition
1	2 DCT residual partitions
2	4 DCT residual partitions
3	8 DCT residual partitions

- `bErrorResilientMode` is a Boolean value used to indicate if error resilient mode is enabled. This mode prevents cumulative probability updates and is used in video telephony.

4.3.19.2 Dependencies

This parameter is only applicable when the port is configured for VP8.

4.3.20 OMX_VIDEO_VP8REFERENCEFRAMETYPE

`OMX_VIDEO_VP8REFERENCEFRAMETYPE` structure is used to configure the type of reference frames to be used while video encoding is in progress.

VP8 uses two encoding concepts:

1) Frame coding type.

There are only two types of frames in VP8, intraframes (key frames, I-frames) and interframes (prediction frames, P-frames). Frame coding type is controlled with `OMX_CONFIG_INTRAREFRESHVOPTYPE` structure.

2) Reference frame buffers.

VP8 uses three reference frame buffers called immediately previous frame, golden frame and alternate frame to predict blocks in an interframe. Every key frame is automatically golden frame and alternate frame. Optionally any interframe may replace the most recent golden frame and/or alternate frame.

`OMX_VIDEO_VP8REFERENCEFRAMETYPE` is defined as follows.

```
typedef struct OMX_VIDEO_VP8REFERENCEFRAMETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL nPreviousFrameRefresh;
    OMX_BOOL bGoldenFrameRefresh;
    OMX_BOOL bAlternateFrameRefresh;
    OMX_BOOL bUsePreviousFrame;
    OMX_BOOL bUseGoldenFrame;
    OMX_BOOL bUseAlternateFrame;
} OMX_VIDEO_VP8REFERENCEFRAMETYPE ;
```

4.3.20.1 Parameters

The parameters for OMX_VIDEO_VP8REFERENCEFRAMETYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bPreviousFrameRefresh is a Boolean value used to indicate if the next frame is used to refresh (update) the immediately previous frame.
- bGoldenFrameRefresh is a Boolean value used to indicate if the next frame is to be encoded as a golden reference frame.
- bAlternateFrameRefresh is a Boolean value used to indicate if the next frame is to be encoded as an alternate reference frame.
- bUsePreviousFrame is a Boolean value used to indicate if the immediately previous frame should be used for prediction.
- bUseGoldenFrame is a Boolean value used to indicate if the golden reference frame should be used for prediction.
- bUseAlternateFrame is a Boolean value used to indicate if the alternate reference frame should be used for prediction.

Table 4-72: Possible Ways to Refresh Reference Frames

bPrevious FrameRefresh	bGolden FrameRefresh	bAlternate FrameRefresh	Effect on Coding
OMX_FALSE	OMX_FALSE	OMX_FALSE	Droppable frame. Usable for temporal scalability, as no future frames will use this frame as a reference. Frame is droppable if bErrorResilientMode = OMX_TRUE.
OMX_FALSE	OMX_FALSE	OMX_TRUE	Alternate reference frame is updated by this frame.
OMX_FALSE	OMX_TRUE	OMX_FALSE	Golden reference frame is updated by this frame.
OMX_FALSE	OMX_TRUE	OMX_TRUE	Alternate reference frame and golden reference frame are updated by this frame.
OMX_TRUE	OMX_FALSE	OMX_FALSE	Immediately previous frame is updated by this frame.
OMX_TRUE	OMX_FALSE	OMX_TRUE	Immediately previous frame and alternate reference frame are updated by this frame.
OMX_TRUE	OMX_TRUE	OMX_FALSE	Immediately previous frame and golden reference frame are updated by this frame.
OMX_TRUE	OMX_TRUE	OMX_TRUE	Immediately previous frame, golden reference frame, and alternate reference frame are updated by this frame.

4.3.21 OMX_VIDEO_VP8REFERENCEFRAMEINFOTYPE

OMX_VIDEO_VP8REFERENCEFRAMEINFOTYPE structure is used to report the VP8 reference frame type while video decoding is in progress.

OMX_VIDEO_VP8REFERENCEFRAMEINFOTYPE is defined as follows.

```
typedef struct OMX_VIDEO_VP8REFERENCEFRAMEINFOTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bIsIntraFrame;
    OMX_BOOL bIsGoldenOrAlternateFrame;
} OMX_VIDEO_VP8REFERENCEFRAMEINFOTYPE ;
```

4.3.21.1 Parameters

The parameters for OMX_VIDEO_VP8REFERENCEFRAMEINFOTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bIsIntraFrame is a Boolean value used to indicate if the frame is an Intra frame.
- bIsGoldenOrAlternateFrame is a Boolean value used to indicate if the frame is a golden frame or an alternate frame.

4.3.21.2 Dependencies

The parameter may only be used to query the reference frame type at any time that the component is in the OMX_StateExecuting state.

4.3.22 OMX_VIDEO_CONFIG_BITRATETYPE

The video encoder's bit rate setting may be updated while the video encoder is actively encoding, the OMX_VIDEO_CONFIG_BITRATETYPE structure contains the parameters for updating the video bit rate.

OMX_VIDEO_CONFIG_BITRATETYPE is defined as follows.

```
typedef struct OMX_VIDEO_CONFIG_BITRATETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nEncodeBitrate;
} OMX_VIDEO_CONFIG_BITRATETYPE;
```

4.3.22.1 Parameters

The parameters for OMX_VIDEO_CONFIG_BITRATETYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.

- nEncodeBitrate is the target bit rate for the video encoding in units of bits per second.

4.3.23 OMX_CONFIG_FRAMERATETYPE

The video encoder's frame rate setting may be updated while the video encoder is actively encoding, the OMX_CONFIG_FRAMERATETYPE structure contains the parameters for updating the video frame rate.

OMX_CONFIG_FRAMERATETYPE is defined as follows.

```
typedef struct OMX_CONFIG_FRAMERATETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 xEncodeFramerate;
} OMX_CONFIG_FRAMERATETYPE;
```

4.3.23.1 Parameters

The parameters for OMX_CONFIG_FRAMERATETYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- xEncodeFramerate is the frame rate for the video encoding in units of frames per second. This value is represented in Q16 format

4.3.24 OMX_CONFIG_INTRAREFRESHVOPTYPE

The OMX_CONFIG_INTRAREFRESHVOPTYPE structure is used to force the next video frame to be encoded as an I-VOP.

OMX_CONFIG_INTRAREFRESHVOPTYPE is defined as follows.

```
typedef struct OMX_CONFIG_INTRAREFRESHVOPTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL IntraRefreshVOP;
} OMX_CONFIG_INTRAREFRESHVOPTYPE;
```

4.3.24.1 Parameters

The parameters for OMX_CONFIG_INTRAREFRESHVOPTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- IntraRefreshVOP is a Boolean value used to indicate if the next frame is to be encoded as an I VOP.

4.3.25 OMX_CONFIG_MACROBLOCKERRORMAPTYPE

The OMX_CONFIG_MACROBLOCKERRORMAPTYPE structure is used to force some of all of the macroblocks within the next video frame to be encoded as Intra macroblocks.

Typically the map of the macroblocks requested to be refreshed as intra macroblocks correlates to macroblock decoding errors encountered during a video telephony use case on the remote device.

OMX_CONFIG_MACROBLOCKERRORMAPTYPE is defined as follows.

```
typedef struct OMX_CONFIG_MACROBLOCKERRORMAPTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nErrMapSize;
    OMX_U8 ErrMap[1];
} OMX_CONFIG_MACROBLOCKERRORMAPTYPE;
```

4.3.25.1 Parameters

The parameters for OMX_CONFIG_MACROBLOCKERRORMAPTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nErrMapSize is the size of the macroblock map containing the refresh information, this parameter is specified in units of bytes.
- ErrMap contains the map of the macroblocks within the frame that are to be refreshed as intra macroblocks. The array contains one or more bytes as indicated by the nErrMapSize field

  The format of the macroblock map is a bit mapped string of values that corresponds to each macroblock within the video frame, when the bit value is set it indicates that the corresponding macroblock is to be refreshed as an intra macroblock.

  As an example, a video frame having a resolution of 176x144 contains 99 macroblocks thus the macroblock map will contain 99 bit mapped values identifying each and every macroblock within the frame (the nErrMapSize parameter will contain a size of 13 – rounded up to the nearest byte boundary). Bit 0 of the macroblock map refers to macroblock 0 within the video frame, bit 1 refers to macroblock 1 and so on.

  The error map information is cumulative between frames; it is to be cleared:

  - Upon each OMX_GetConfig request.
  - Each time an Intra Frame is detected. The error map information is to include any macroblock errors found within the Intra frame.

4.3.25.2 Dependencies

The parameter may only be used to get the macroblock error map information using `OMX_GetConfig` at any time that the component is in the `OMX_StateExecuting` state.

4.3.25.3 Error Conditions

On processing the `OMX_CONFIG_MACROBLOCKERRORMAPTYPE` structure, the following error conditions can occur:

- `OMX_ErrorMbErrorsInFrame` when macroblock errors are found within a frame.

  When macroblock errors are encountered during the processing, the component will issue an `OMX_EventError` event with the value `OMX_ErrorMbErrorsInFrame` notifying the IL client of this occurrence.

4.3.26 OMX_PARAM_MACROBLOCKSTYPE

The `OMX_PARAM_MACROBLOCKSTYPE` structure is used to report the number of macroblocks available within the current video stream's frame.

`OMX_PARAM_MACROBLOCKSTYPE` is defined as follows.

```
typedef struct OMX_PARAM_MACROBLOCKSTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nMacroblocks;
} OMX_PARAM_MACROBLOCKSTYPE;
```

4.3.26.1 Parameters

The parameters for `OMX_PARAM_MACROBLOCKSTYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `nMacroblocks` is the number of macroblocks available within the video frame.

4.3.26.2 Dependencies

The parameter may only be used to query the number of macroblocks within the video frame using `OMX_GetParameter` at any time that the component is in the `OMX_StateExecuting` state.

4.3.27 OMX_CONFIG_MBERRORREPORTINGTYPE

The `OMX_CONFIG_MBERRORREPORTINGTYPE` structure is used to enable or disable the macroblock error reporting support.

The macroblock error map information is queried from the video decoder with OMX_GetConfig using OMX_IndexConfigVideoMacroBlockErrorMap and the OMX_CONFIG_MACROBLOCKERRORMAPTYPE structure.

OMX_CONFIG_MBERRORREPORTINGTYPE is defined as follows.

```
typedef struct OMX_CONFIG_MBERRORREPORTINGTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bEnabled;
} OMX_CONFIG_MBERRORREPORTINGTYPE;
```

4.3.27.1 Parameters

The parameters for OMX_CONFIG_MBERRORREPORTINGTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bEnabled is a Boolean value indicating to enable to disable the macroblock error reporting support.

4.3.28 OMX_VIDEO_PARAM_PROFILELEVELTYPE

The OMX_VIDEO_PARAM_PROFILELEVELTYPE structure is used to query the video encoders and decoders for their supported profiles and associated levels when used with the OMX_IndexParamVideoProfileLevelQuerySupported.

In addition the structure may also be used to query or set the profile and level of the video stream that is currently being processed, this is achieved using OMX_IndexParamVideoProfileLevelCurrent.

The codec information retrieved is dependent on the current coding format specified as per the port definition. The caller is required to type cast eCodecType, eProfile and eLevel parameters to the proper data enumeration types prior to interpreting the parameter information. The type casting is to be based on the eCompressionFormat parameter defined by either OMX_VIDEO_PORTDEFINITIONTYPE or OMX_VIDEO_PARAM_PORTFORMATTYPE.

Some of the structure parameters may not be applicable or used for some of the coding types, refer to Table 4-73: Profile and Level Type Casting to understand the parameter usage versus coding type.

OMX_VIDEO_PARAM_PROFILELEVELTYPE is defined as follows.

```
typedef struct OMX_VIDEO_PARAM_PROFILELEVELTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 eProfile;
    OMX_U32 eLevel;
    OMX_U32 nIndex;
    OMX_U32 eCodecType;
```

```
} OMX_VIDEO_PARAM_PROFILELEVELTYPE;
```

4.3.28.1 Parameters

The parameters for `OMX_VIDEO_PARAM_PROFILELEVELTYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `eProfile` is the profile setting as associated with the `eCompressionFormat` parameter.

  The caller is required to type cast this parameter to the proper data enumeration types prior to interpreting the parameter information – refer to Table 4-73: Profile and Level Type Casting for the casting parameters.

Table 4-73: Profile and Level Type Casting

Coding Type	Codec Type	Profile Type	Level Type
OMX_VIDEO_CodingMPEG2	Not Applicable	OMX_VIDEO_MPEG2PROFILETYPE	OMX_VIDEO_MPEG2LEVELTYPE
OMX_VIDEO_CodingH263	Not Applicable	OMX_VIDEO_H263PROFILETYPE	OMX_VIDEO_H263LEVELTYPE
OMX_VIDEO_CodingMPEG4	Not Applicable	OMX_VIDEO_MPEG4PROFILETYPE	OMX_VIDEO_MPEG4LEVELTYPE
OMX_VIDEO_CodingWMV	OMX_VIDEO_WMVFORMATTYPE	OMX_VIDEO_WMVPROFILETYPE	OMX_VIDEO_WMVLEVELTYPE
OMX_VIDEO_CodingRV	Not Applicable	OMX_VIDEO_RVFORMATTYPE	Not Applicable
OMX_VIDEO_CodingAVC	Not Applicable	OMX_VIDEO_AVCPROFILETYPE	OMX_VIDEO_AVCLEVELTYPE
OMX_VIDEO_CodingVC1	Not Applicable	OMX_VIDEO_VC1PROFILETYPE	OMX_VIDEO_VC1LEVELTYPE
OMX_VIDEO_CodingVP8	Not Applicable	OMV_VIDEO_VP8PROFILETYPE	OMX_VIDEO_VP8LEVELTYPE

- `eLevel` is the profile level setting as associated with the `eCompressionFormat` and `eProfile` parameters.

  The caller is required to type cast this parameter to the proper data enumeration types prior to interpreting the parameter information – refer to Table 4-73: Profile and Level Type Casting for the casting parameters.

  For OMX_VIDEO_CodingWMV coding type the profile and level definitions are:

Table 4-74: WMV Profile Types

OMX_VIDEO_WMVPROFILETYPE Types	WMV Profile Descriptions
OMX_VIDEO_WMVProfileUnknown	Unknown, unused or not required setting.
OMX_VIDEO_WMVProfileSimple	WMV Simple Profile
OMX_VIDEO_WMVProfileMain	WMV Main Profile
OMX_VIDEO_WMVProfileAdvanced	WMV Advanced Profile

Table 4-75: WMV Level Types

OMX_VIDEO_WMVLEVELTYPE Types	WMV Level Descriptions
OMX_VIDEO_WMVLevelUnknown	Unknown, unused or not required setting.
OMX_VIDEO_WMVLevelLow	WMV Low level
OMX_VIDEO_WMVLevelMedium	WMV Medium Level
OMX_VIDEO_WMVLevelHigh	WMV High Level
OMX_VIDEO_WMVLevelL0	WMV L0 Level
OMX_VIDEO_WMVLevelL1	WMV L1 Level
OMX_VIDEO_WMVLevelL2	WMV L2 Level
OMX_VIDEO_WMVLevelL3	WMV L3 Level
OMX_VIDEO_WMVLevelL4	WMV L4 Level

- nIndex is used to enumerate the supported profiles. The caller specifies all fields and the OMX_GetParameter call returns the value of the supported profile and level. The value of nIndex goes from 0 to N-1, where N is the number of profiles supported by the port. The port does not need to report N as the caller can determine N by enumerating all the formats supported by the port. Each port shall support at least one profile. If there are no more profiles, OMX_GetParameter returns OMX_ErrorNoMore.

Table 4-76: ProfileLevel Call Details

Action	Index	Description
Query for supported profiles and levels	OMX_IndexParamVideoProfileLevelQuerySupported	Multiple calls with increasing values of nIndex will enumerate the supported profiles until OMX_ErrorNoMore is returned. With each successful call, a supported profile will be identified with the maximum supported associated level setting.

Action	Index	Description
Query the profile and level for the current stream	OMX_IndexParamVideoProfileLevelCurrent	eCompressionFormat, eProfile and eLevel will return the current stream's information. The nIndex parameter is an ignored parameter.
Configure the encoder to use a specific profile and level for the current stream	OMX_IndexParamVideoProfileLevelCurrent	eCompressionFormat, eProfile and eLevel will contain the requested settings to be used as part of the encoding. The nIndex parameter is an ignored parameter.

- eCodecType is the format setting as associated with the eCompressionFormat parameter.

  The caller is required to type cast this parameter to the proper data enumeration types prior to interpreting the parameter information – refer to Table 4-73: Profile and Level Type Casting**Error! Reference source not found.** for the casting parameters.

4.3.28.2 Dependencies

The parameter using the index OMX_IndexParamVideoProfileLevelCurrent may be queried using OMX_GetParameter or set using OMX_SetParameter at any time that the component is initialized.

The IL client shall ignore any parameter identified in Table 4-60 as "Not Applicable" as any parameter specified in the table as "Not Applicable" has no associated information for the specified Coding Type.

4.3.29 OMX_VIDEO_PARAM_AVCSLICEFMO

The OMX_VIDEO_PARAM_AVCSLICEFMO structure is used to enable and configure the Flexible Macroblock Ordering (FMO) slice modes within the AVC video encoder.

OMX_VIDEO_PARAM_AVCSLICEFMO is defined as follows.

```
typedef struct OMX_VIDEO_PARAM_AVCSLICEFMO {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U8 nNumSliceGroups;
```

```
    OMX_U8 nSliceGroupMapType;
    OMX_VIDEO_AVCSLICEMODETYPE eSliceMode;
} OMX_VIDEO_PARAM_AVCSLICEFMO;
```

4.3.29.1 Parameters

The parameters for `OMX_VIDEO_PARAM_AVCSLICEFMO` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `nNumSliceGroups` specifies the number of slice groups that can be supported in the encode session. This parameter is enabled when FMO mode is enabled, refer to `OMX_VIDEO_PARAM_AVCTYPE` for enabling FMO mode support.

  The setting information for this parameter is directly related to the functionality as specified within the ITU H.264/AVC specification and is dependent on the video profile currently in use.

  The currently defined parameter range settings are listed in Table 4-77.

Table 4-77: AVC Parameter Range Settings

Video Profile	Range
`OMX_VIDEO_AVCProfileBaseline`	0 to 7
`OMX_VIDEO_AVCProfileMain`	0
`OMX_VIDEO_AVCProfileExtended`	0 to 7
`OMX_VIDEO_AVCProfileHigh`	0
`OMX_VIDEO_AVCProfileHigh10`	0
`OMX_VIDEO_AVCProfileHigh422`	0
`OMX_VIDEO_AVCProfileHigh444`	0

- `nSliceGroupMapType` specifies the type of slice groupings that is to be used during encoding.

  The setting information for this parameter is directly related to the functionality as specified within the ITU H.264/AVC specification.

  The currently defined parameter settings are:

Table 4-78: Slice Group Map Type Values

Slice Group Map Value	Description
0	Indicates interleaves slices.
1	Indicates a dispersed macroblock allocation
2	Indicates to explicitly assign a slice group to each macroblock in raster scan order
3	Indicates one or more “foreground” slice groups and a “leftover” slice group
4	Indicates changing slice groups.

Slice Group Map Value	Description
5	Indicates changing slice groups.
6	Indicates changing slice groups.

- `eSliceMode` specifies the type of slice that is to be used for encoding the frame.

Table 4-79: Slice Mode Type Casting

`Slice Mode`	AVC Slice Mode Description
`OMX_VIDEO_SLICEMODE_AVCDefault`	Normal frame encoding, one slice per frame
`OMX_VIDEO_SLICEMODE_AVCMBSlice`	NAL mode based on number of macroblocks per slice
`OMX_VIDEO_SLICEMODE_AVCByteSlice`	NAL Mode based on number of bytes per slice.

4.3.30 OMX_VIDEO_CONFIG_AVCINTRAPERIOD

The `OMX_VIDEO_CONFIG_AVCINTRAPERIOD` structure is used to enable and configure the IDR and Intra periodicity for the AVC encoder during an encoding session.

`OMX_VIDEO_CONFIG_AVCINTRAPERIOD` is defined as follows.

```
typedef struct OMX_VIDEO_CONFIG_AVCINTRAPERIOD {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nIDRPeriod;
    OMX_U32 nPFrames;
} OMX_VIDEO_CONFIG_AVCINTRAPERIOD;
```

4.3.30.1 Parameters

The parameters for `OMX_VIDEO_CONFIG_AVCINTRAPERIOD` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `nIDRPeriod` defines the periodicity of IDR occurrence. This specifies coding a frame as IDR after a specific number of intra frames. The periodicity of the intra frame coding is specified by `nPFrames`. If `nIDRPeriod` is set to 0, only the first frame of the encode session is an IDR frame.
- `nPFrames` specifies coding of a frame as Intra (non-inclusive of the first frame) after every `nPFrames` of Inter frames.

4.3.31 OMX_VIDEO_CONFIG_NALSIZE

The `OMX_VIDEO_CONFIG_NALSIZE` structure is used to specify the size of a NAL unit for the AVC encoder during an encoding session.

`OMX_VIDEO_CONFIG_NALSIZE` is defined as follows.

```
typedef struct OMX_VIDEO_CONFIG_NALSIZE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nNaluBytes;
} OMX_VIDEO_CONFIG_NALSIZE;
```

4.3.31.1 Parameters

The parameters for OMX_VIDEO_CONFIG_NALSIZE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nNaluBytes specifies the number of bytes of data to be contained in the current NAL Units.

4.3.32 OMX_NALSTREAMFORMATTYPE

The OMX_NALSTREAMFORMATTYPE structure is used to specify the NAL unit format and its associated size.

OMX_NALSTREAMFORMATTYPE is defined as follows.

```
typedef struct OMX_NALSTREAMFORMATTYPE{
    OMX_U32  nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32  nPortIndex;
    OMX_NALUFORMATSTYPE  eNaluFormat;
} OMX_NALSTREAMFORMATTYPE;
```

4.3.32.1 Parameters

The parameters for OMX_NALSTREAMFORMATTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- eNaluFormat indicates the format of the NALU. Refer to Table 4-80 for a listing of the various formats. This parameter contains bit-mapped values as defined by Table 4-80.

  The default mode of operation shall be OMX_NaluFormatStartCodes.

NALU Format	Description
OMX_NaluFormatStartCodes	NALUs separated by Start Codes (ITU-T H.264\ISO 14496-10 Annex B)
OMX_NaluFormatOneNaluPerBuffer	One NALU per buffer. Multiple NALUs in the same buffer are forbidden
OMX_NaluFormatOneByteInterleaveLength	NALU separated by 1-byte interleaved length fields
OMX_NaluFormatTwoByteInterleaveLength	NALU separated by 2-byte interleaved length fields

OMX_NaluFormatFourByteInterleaveLength	NALU separated by 4-byte interleaved length fields

Table 4-80: NALU Formats

Payload Packaging Options for cases when Start Codes are not in use:

A buffer containing a single NAL unit appears as:

<NAL Size X bytes><NAL unit>

A buffer containing multiple NALs unit appears as:

<NAL Size X bytes><NAL unit><NAL Size X bytes><NAL unit>

<NAL Size X Bytes> is the number of bytes indicating the size of the NAL unit payload

In case of "OMX_NaluFormatOneNaluPerBuffer" NALU format, only one full or partial <NAL unit> payload data without start codes or NAL sizes is present in the buffer.

4.3.32.2 Functionality

In order for an OpenMAX IL client to properly configure a component graph to consume the stream, it needs to be able to query the components to determine:

- The stream packaging format (Source Component – component emitting the NALU payload)
- The stream formats supported by the component consuming the NALU payload.

The determination of the NALU formatting shall be queried via the source components that will be emitting the stream content, for example Demuxer Components. These components will only have access to this formatting information when it has been given the opportunity to parse the source content, typically achieved when in OMX_StateExecuting state. Utilizing the auto-detection support, the IL client will be able to query this information after the component issues the OMX_EventPortFormatDetected event.

The IL client shall use OMX_GetParameter(OMX_IndexParamNalStreamFormat) on the source component's output port to query the native NALU packaging format within the embedded stream.

The IL client shall use `OMX_GetParameter(OMX_IndexParamNalStreamFormatSupported)` on the source component's output port to query the NALU packaging formats supported – the nNaluFormat parameter shall return all the formats supported or'ed together.

The IL client shall use `OMX_GetParameter(OMX_IndexParamNalStreamFormatSupported)` on the consumer component's input port to query the NALU packaging format supported – the nNaluFormat parameter shall return all the formats supported or'ed together.

In the case where a consumer component's input port does not support the NAL stream format selection, the responsibility of formatting the stream payload reverts to the source component. A source component shall support the ability to emit the NALU payload in either configurable option, however it is not mandated that the component shall support the ability to package multiple NALUs within a single buffer – although this is highly recommended.

Note: Configuring a source component to format the NALU payload in a format that is non-native to the stream's embedded format may incur a performance penalty.

The IL client shall use `OMX_SetParameter(OMX_IndexParamNalStreamFormatSelect)` on a source component's output port to configure it to the appropriate setting.

In the case where a consumer component's input port is capable of supporting the native NALU packaging format within the embedded stream but differs from the default OMX_NaluFormatStartCodes mode, the IL client may configure the consumer component's input port instead of the source component's output port to consume the stream. The IL client shall use `OMX_SetParameter(OMX_IndexParamNalStreamFormatSelect)` on a consumer component's input port to configure it to the appropriate setting.

4.3.28.3 Call Sequence Examples

This section provides various examples that may be encountered.

Figure 4-10 shows the case when a Video Decoder supports the NALU configuration support within the embedded stream.

Figure 4-11 shows the case when a Video Decoder does not support the NALU configuration within the stream, the IL client configures the Demuxer to emit a format supported by the Video Decoder (e.g. NALU using Start Code - ITU-T H.264\ISO 14496-10 Annex B Specification).

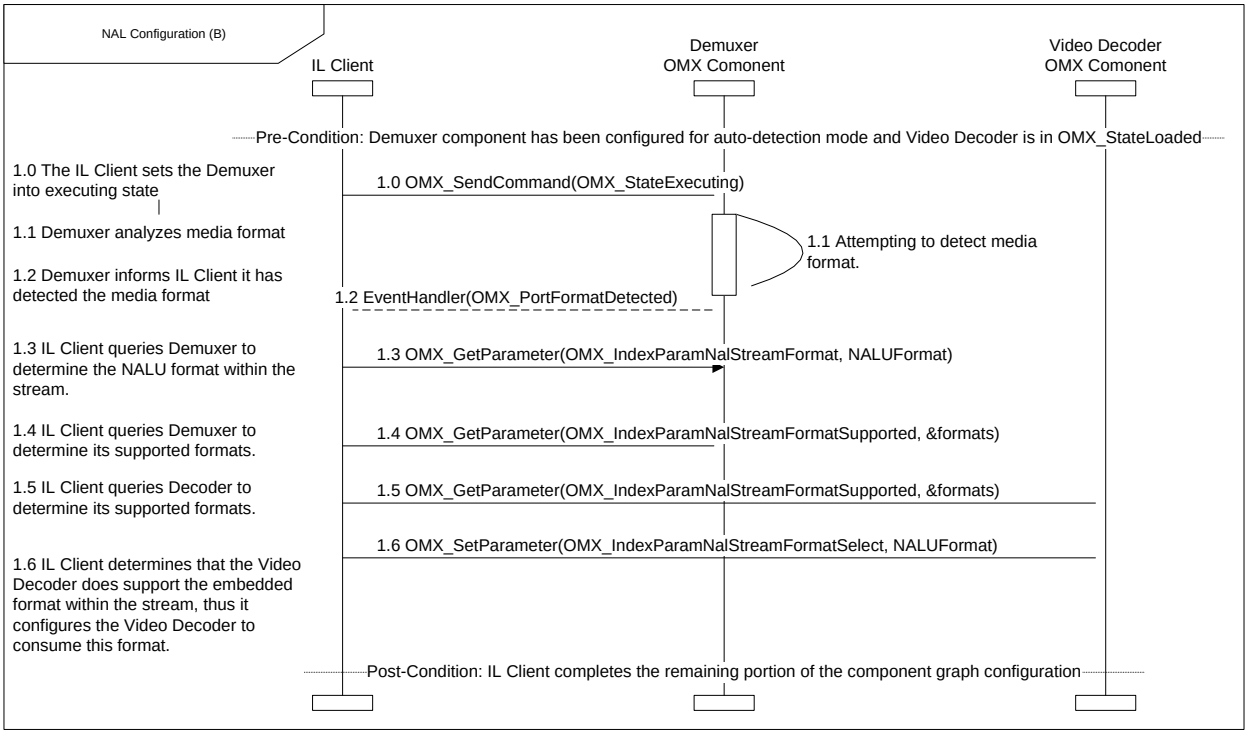


Figure 4-10: NALU Formatting Supported By Video Decoder

The sequence starts with a Pre-Condition that the IL client has configured the output port formats (e.g. OMX_IndexParamVideoPortFormat) of the Demuxer to auto detect.

The IL client commands the Demuxer component to transition into executing state – Step 1.0.

The Demuxer reads and parses the media content until it is able to detect the media container format – Step 1.1.

The Demuxer component detects the media format and notifies the IL client via an event callback – Step 1.2.

At this point, the Demuxer component is capable of determining the native NALU stream formatting within the embedded container. The IL client queries this information from the Demuxer – Step 1.3.

IL client queries the Demuxer to determine its supported formats (not a required step, shown for completeness) – Step 1.4.

IL client queries the Video Decoder to determine its supported formats – Step 1.5.

The IL client determines that the Video Decoder is capable of supporting the format within the stream. The IL client configures the Video Decoder to consume this format – Step 1.6.

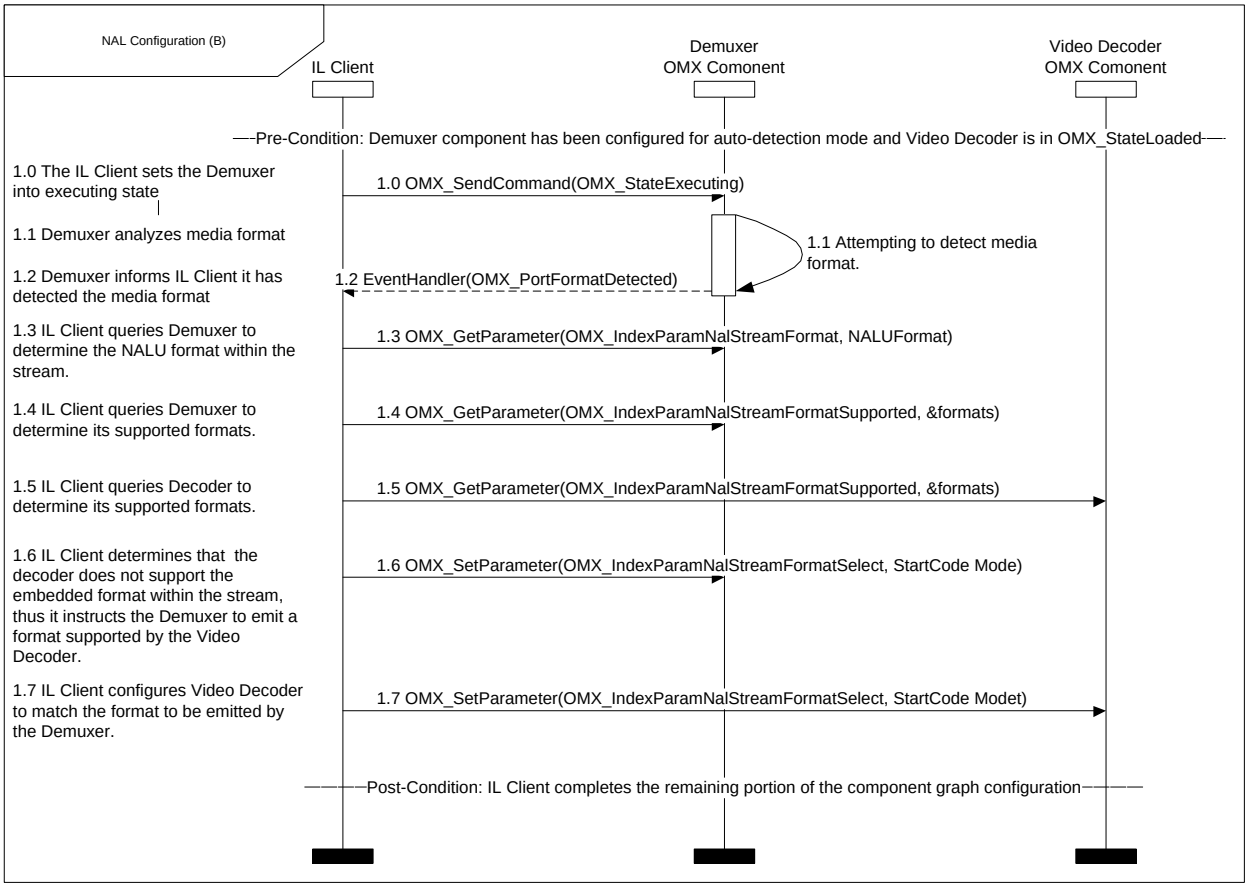


Figure 4-11: NALU Formatting Not Supported By Video Decoder

The sequence starts with a Pre-Condition that the IL client has configured the output port formats (e.g. OMX_IndexParamVideoPortFormat) of the Demuxer to auto detect.

The IL client commands the Demuxer component to transition into executing state – Step 1.0.

The Demuxer reads and parses the media content until it is able to detect the media container format – Step 1.1.

The Demuxer component detects the media format and notifies the IL client via an event callback – Step 1.2.

At this point, the Demuxer component is capable of determining the native NALU stream formatting within the embedded container. The IL client queries this information from the Demuxer – Step 1.3.

IL client queries the Demuxer to determine its supported formats (not a required step, shown for completeness) – Step 1.4.

IL client queries the Video Decoder to determine its supported formats – Step 1.5.

The IL client determines that the Video Decoder does not support the format within the stream. The IL client configures the Demuxer to emit a format supported by the Video Decoder – Step 1.6.

The IL client configures the Video Decoder to consume the format to be emitted by the Demuxer – Step 1.7.

4.3.33 OMX_VIDEO_PARAM_VC1TYPE

`OMX_VIDEO_PARAM_VC1TYPE` contains common standard video codec parameters that are used with VC-1 format including VC-1 Simple profile, VC-1 Main Profile and VC-1 Advanced profile.

`OMX_VIDEO_PARAM_VC1TYPE` is defined as follows.

```
typedef struct OMX_VIDEO_PARAM_VC1TYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_VIDEO_VC1PROFILETYPE eProfile;
    OMX_VIDEO_VC1LEVELTYPE eLevel;
} OMX_VIDEO_PARAM_VC1TYPE;
```

4.3.33.1 Parameters

The parameters for `OMX_VIDEO_PARAM_VC1TYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `eProfile` is the enumerated value representing the profile of the VC-1 data stream. Table 4-81 shows the possible VC-1 video profile types for `OMX_VIDEO_VC1PROFILETYPE`.

Table 4-81 : Supported VC-1 Profile Types

Field Name	VC-1 Profile Descriptions
`OMX_VIDEO_VC1ProfileUnknown`	Unknown, unused or not required setting.
`OMX_VIDEO_VC1ProfileUnused`	Profile unused or unknown
`OMX_VIDEO_VC1ProfileSimple`	Simple Profile
`OMX_VIDEO_VC1ProfileMain`	Main Profile
`OMX_VIDEO_VC1ProfileAdvanced`	Advanced Profile

- `eLevel` is the processing level of a particular VC-1 profile. Table 4-82 shows the possible VC-1 video level types in `OMX_VIDEO_VC1LEVELTYPE`.

Table 4-82 : Supported VC-1 Level Types

Field Name	VC-1 Level Descriptions
`OMX_VIDEO_VC1LevelUnknown`	Unknown, unused or not required setting.
`OMX_VIDEO_VC1LevelUnused`	Level unused or unknown
`OMX_VIDEO_VC1LevelLow`	Simple or Main Profile - Low Level
`OMX_VIDEO_VC1LevelMedium`	Simple or Main Profile - Medium Level
`OMX_VIDEO_VC1LevelHigh`	Main Profile - High Level
`OMX_VIDEO_VC1Level0`	Level 0 - Advanced Profile
`OMX_VIDEO_VC1Level1`	Level 1 – Advanced Profile
`OMX_VIDEO_VC1Level2`	Level 2 – Advanced Profile
`OMX_VIDEO_VC1Level3`	Level 3 – Advanced Profile
`OMX_VIDEO_VC1Level4`	Level 4 – Advanced Profile

4.3.33.2 VC-1 Codec Configuration Data

4.3.33.2.1 VC-1 Codec Configuration Data and Frame Layer Data for Main and Simple Profile

In case of VC-1 Simple and Main profile, the codec configuration data passed to the VC-1 decoder component shall be in the format specified by Table 265 of Annex L.2. of VC-1 specification, where STRUCT_A, STRUCT_B and STRUCT_C of Table 265 are defined by Tables 260, 261, 263 of Annex J of VC-1 specification respectively. In case of VC-1 Simple and Main profile, in addition to codec configuration data, valid data in OMX buffer payloads (i.e. data pointed to by "`pBuffer+nOffset`" parameters of OMX buffer headers) provided to VC-1 decoder component shall contain compressed frame data corresponding to only FRAMEDATA portion of frame layer data of Table 266 of Annex L.3. of VC-1 specification.

4.3.33.3 VC-1 Advanced Profile Start Codes

In case of VC-1 Advanced Profile, start codes are a part of the VC-1 bitstream, as defined in Annex G of VC-1 specification. As a default, the source component or IL client shall provide VC-1 start codes to the VC-1 decoder component and shall not attempt to remove them from the bitstream, and the VC-1 decoder component shall support VC-1 Advanced Profile start codes. The source component or IL client may negotiate with the VC-1 decoder component a different format of VC-1 Advanced Profile bitstream and use such a format if both sides agree to it. However, as a minimum, both the VC-1 decoder

component and the source component or IL client shall support the default VC-1 Advanced Profile bitstream format with embedded start codes.

4.3.33.4 Handling of Multiple VC-1 Streams

In the case where ASF content contains multiple VC-1 streams, each stream may require its own codec configuration. Also, in the case of multiple VC-1 streams, the source component or IL client may send data buffers that contain data from one stream, then switch to sending data buffers that contain data from a different stream to the VC-1 decoder component.

In case of such stream-switching, the source component or IL client shall ensure that the VC-1 decoder component receives the codec configuration data for the appropriate stream in-band, such that codec configuration data for the new stream immediately precedes the bitstream data for that stream (e.g. the source component or IL client ensure this by sending a buffer containing the codec configuration data for the new stream prior to sending any data buffers that contain the new bitstream data). This enables the VC-1 decoder component to reconfigure the decoder and correctly start processing data that belongs to the appropriate stream. Note that a VC-1 decoder component is not expected to handle multiple streams in parallel, since data that belongs to different streams is provided sequentially to the VC-1 decoder component (i.e. stream 1 codec configuration data, followed by stream 1 frame data, followed by stream 2 codec configuration data, followed by stream 2 frame data, etc.).

4.3.33.5 Dependencies

This parameter is only applicable when the port is configured for VC-1 video.

4.3.34 OMX_VIDEO_INTRAPERIODTYPE

Intra-frame period is a parameter that may need to be updated during a video encoding session.

`OMX_VIDEO_INTRAPERIODTYPE` is a structure used to control this behavior.

`OMX_VIDEO_INTRAPERIODTYPE` is defined as follows.

```
typedef struct OMX_VIDEO_INTRAPERIODTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nIDRPeriod;
    OMX_S32 nPFrames;
    OMX_S32 nBFrames;
} OMX_VIDEO_INTRAPERIODTYPE;
```

4.3.34.1 Parameters

The parameters for `OMX_VIDEO_INTRAPERIODTYPE` are defined as follows:

- `nPortIndex` represents the port that this structure applies to.
- `nIDRPeriod` is a value that defines the periodicity of IDR occurrence. This specifies coding a frame as IDR after a specific number of intra frames. The periodicity of intra-frame coding is specified by the `nPFrames` and `nBFrames`. If `nIDRPeriod` is set to 0, only the first frame of the encode session is an IDR frame. This field is used only for codecs that support IDR period, such as AVC.
- `nPFrames` is a value that specifies the number of P frames between each I Frame. The value less than 0 in this field shall be considered as "do-not-care" value.
- `nBFrames` is a value that specifies the number of B frames between each I Frame. Not all codec-profile types support configuring the presence of B Frames. This setting would be ignored for such codecs/profiles. The value less than 0 in this field shall be considered as "do-not-care" value.

4.4 Image

This section describes the parameter and configuration details for components and ports in the image domain. These parameter and configuration details are specified in the OMX_Image.h header file.

4.4.1 Image Use Case Example

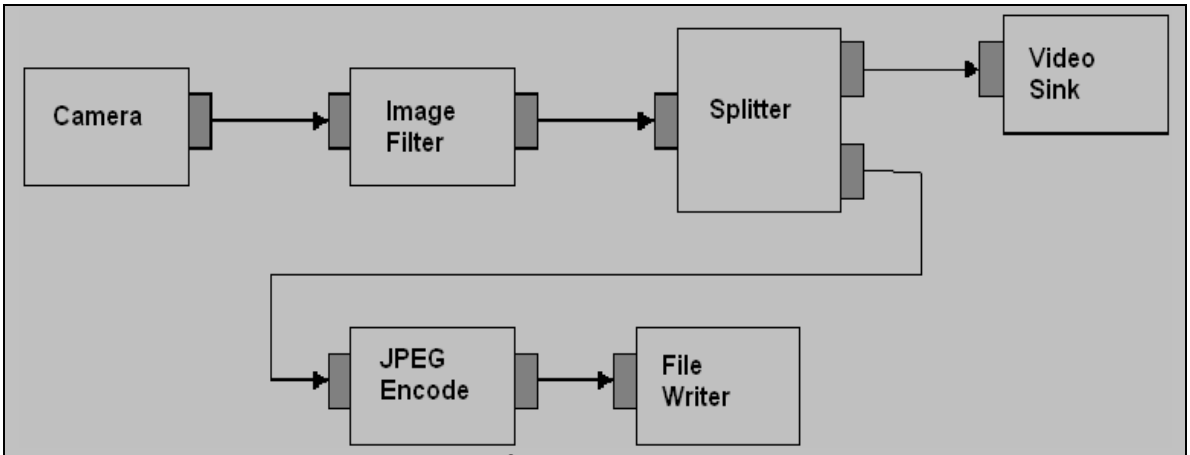


Figure 4-12 depicts one possible set of tunneled components and associated ports to implement a JPEG encoder with pre- and post-processing. This use case encodes an image to a file while allowing a preview of the captured image via a display.

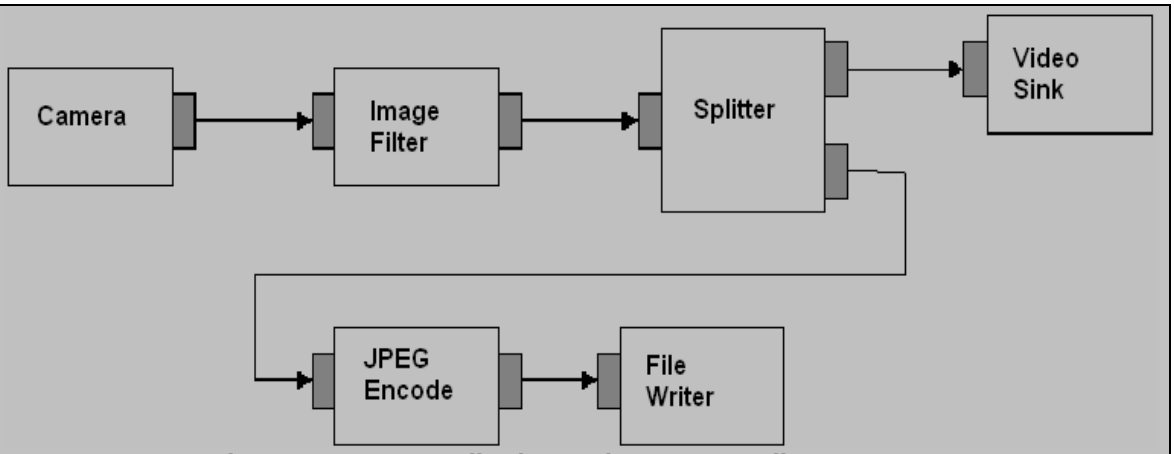


Figure 4-12. Image Filtering and JPEG Encoding Use Case

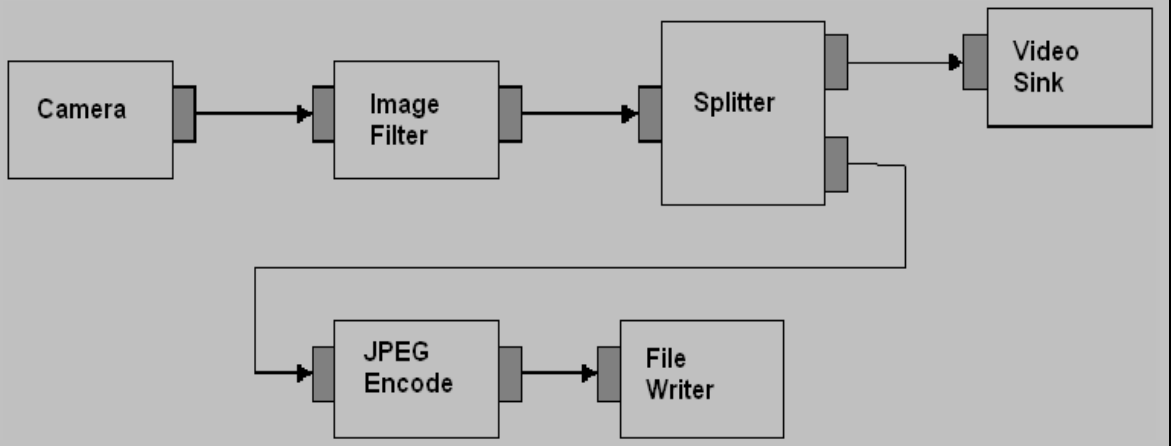


Figure 4-12 shows six components, namely the camera, the image filter, the splitter, the JPEG encoder, the file writer, and the image sink.

4.4.2 Parameter and Configuration Indices

The header OMX_Index.h contains the enumeration `OMX_INDEXTYPE`, which contains all standard index values used core functions `OMX_GetParameter`, `OMX_SetParameter`, `OMX_GetConfig`, and `OMX_SetConfig`. shows the index values that relate to imaging.

Table 4-83: Image Indices

OpenMAX IL Indices (OMX_Index.h)	Corresponding OpenMAX IL Image Structures (OMX_Image.h)
`OMX_IndexParamPortDefinition`	`OMX_PARAM_PORTDEFINITIONTYPE` with `OMX_IMAGE_PORTDEFINITIONTYPE`
`OMX_IndexParamImagePortFormat`	`OMX_IMAGE_PARAM_PORTFORMATTYPE`
`OMX_IndexParamImageInit`	`OMX_PORT_PARAM_TYPE`
`OMX_IndexParamFlashControl`	`OMX_IMAGE_PARAM_FLASHCONTROLTYPE`
`OMX_IndexConfigFlashControl`	`OMX_IMAGE_PARAM_FLASHCONTROLTYPE`
`OMX_IndexConfigFocusControl`	`OMX_IMAGE_CONFIG_FOCUSCONTROLTYPE`

OpenMAX IL Indices (OMX_Index.h)	Corresponding OpenMAX IL Image Structures (OMX_Image.h)
OMX_IndexParamQFactor	OMX_IMAGE_PARAM_QFACTORTYPE
OMX_IndexParamQuantizationTable	OMX_IMAGE_PARAM_QUANTIZATIONTABLETYPE
OMX_IndexParamHuffmanTable	OMX_IMAGE_PARAM_HUFFMANTTABLETYPE
OMX_IndexConfigFlickerRejection	OMX_CONFIG_FLICKERREJECTIONTYPE
OMX_IndexConfigImageHistogram	OMX_IMAGE_HISTOGRAMTYPE
OMX_IndexConfigImageHistogramData	OMX_IMAGE_HISTOGRAMDATATYPE
OMX_IndexConfigImageHistogramInfo	OMX_IMAGE_HISTOGRAMINFOTYPE
OMX_IndexConfigFileFormat	OMX_CONFIG_FILEFORMATTYPE
OMX_IndexConfigImageCaptureStarted	OMX_PARAM_U32TYPE
OMX_IndexConfigImageCaptureEnded	OMX_PARAM_U32TYPE

For example, OMX_IndexParamImagePortFormat index is used with OMX_GetParameter and OMX_SetParameter to access OMX_IMAGE_PARAM_PORTFORMATTYPE.

4.4.3 OMX_IMAGE_PORTDEFINITIONTYPE

OMX_IMAGE_PORTDEFINITIONTYPE is the data structure that is used to define an image path. The number of image paths for input and output will vary by the type of the image component:

- Input (also known as source) has zero inputs and one output.
- Splitter has one input and two or more outputs.
- Processing element has one input and one output.
- Mixer has two or more inputs and one output.
- Output (also known as sink) has one input and zero outputs.

The OMX_IMAGE_PORTDEFINITIONTYPE structure can query the current definition of an image port or set the definition of an image port for a component. The OMX_IMAGE_PORTDEFINITIONTYPE structure is included as part of the OMX_PARAM_PORTDEFINITIONTYPE structure, it is accessed via the OMX_GetParameter function or the OMX_GetParameter function using the OMX_IndexParamPortDefinition index.

OMX_IMAGE_PORTDEFINITIONTYPE is defined as follows.

```
typedef struct OMX_IMAGE_PORTDEFINITIONTYPE {
    OMX_NATIVE_DEVICETYPE pNativeRender;
    OMX_U32 nFrameWidth;
    OMX_U32 nFrameHeight;
    OMX_S32 nStride;
    OMX_U32 nSliceHeight;
    OMX_BOOL bFlagErrorConcealment;
    OMX_IMAGE_CODINGTYPE eCompressionFormat;
```

```
    OMX_COLOR_FORMATTYPE eColorFormat;
    OMX_NATIVE_WINDOWTYPE pNativeWindow;
} OMX_IMAGE_PORTDEFINITIONTYPE;
```

4.4.3.1 Parameters

The parameters for `OMX_IMAGE_PORTDEFINITIONTYPE` are defined as follows.

- `pNativeRender` is the read-only platform specific reference for a display synchronization; otherwise this field is 0. This parameter is ignored on `OMX_SetParameter` calls.
- `nFrameWidth` is the width of frame to be used on the port if uncompressed format is used. Use 0 for unknown, no preference, or variable.
- `nFrameHeight` is the height of the frame to be used on the port if uncompressed format is used. Use 0 for unknown, no preference, or variable.
- `nStride` is a field containing the number of bytes per span of an image, which indicates the number of bytes to get from span N to span N+1. A negative value for `nStride` indicates the data is stored bottom-to-top instead of top-to-bottom.

  Normally the stride parameter is determined by the component, there are cases however when the stride parameter may need to be updated based on external buffer stride requirements.

  An example of such a case includes when IL clients submit buffers to the component for processing, the IL client may have differing stride requirements from the component port.

  By allowing the flexibility for the stride to be modified, the component and IL client may negotiate a common stride setting to suit each other needs and in turn possibly improve the performance of processing the buffer.
- `nSliceHeight` is a read-only field containing the slice height parameter used when processing uncompressed image data. Buffers received on the port shall contain integer multiples of slices. For more information on minimum buffer payload for uncompressed data, see section 4.2.2.
- `bFlagErrorConcealment` is a flag indicating that the OpenMAX IL component supports error concealment. This flag is returned by a component upon invoking `OMX_GetParameter`; it is ignored on `OMX_SetParameter` calls.
- `eCompressionFormat` is the enumeration describing the compression format used on the port. When `OMX_IMAGE_CodingUnused` is specified, the `eColorFormat` field is valid. Table 4-84 shows the supported image compression formats.

Table 4-84: Supported Image Compression Formats

Field Name	Compression Format Description	Reference to Standard
OMX_IMAGE_CodingUnused	No coding applied, use eColorFormat	Not available
OMX_IMAGE_CodingAutoDetect	Auto detection by the OpenMAX IL component	Not available
OMX_IMAGE_CodingJPEG	JPEG/JFIF image format	JPEG
OMX_IMAGE_CodingJPEG2K	JPEG 2000 image format	JPEG2K
OMX_IMAGE_CodingEXIF	EXIF image format	EXIF
OMX_IMAGE_CodingTIFF	TIFF image format	TIFF
OMX_IMAGE_CodingGIF	Graphics image format	GIF
OMX_IMAGE_CodingPNG	PNG image format	PNG
OMX_IMAGE_CodingLZW	LZW image format	LZW
OMX_IMAGE_CodingBMP	Windows Bitmap format	BMP
OMX_IMAGE_CodingWEBP	WebP image format	WebP

- eColorFormat is the decompressed color format used for the port. This field is valid only when the eCompressionFormat field is set to OMX_IMAGE_CodingUnused.
- pNativeWindow is a platform specific reference for a windows object when being processed within as part of a video sink component, otherwise this field is 0 and ignored.

4.4.4 OMX_IMAGE_PARAM_PORTFORMATTYPE

OMX_IMAGE_PARAM_PORTFORMATTYPE is used to enumerate the various data input/output format supported by the port.

OMX_IMAGE_PARAM_PORTFORMATTYPE is defined as follows.

```
typedef struct OMX_IMAGE_PARAM_PORTFORMATTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nIndex;
    OMX_IMAGE_CODINGTYPE eCompressionFormat;
    OMX_COLOR_FORMATTYPE eColorFormat;
} OMX_IMAGE_PARAM_PORTFORMATTYPE;
```

4.4.4.1 Parameters

The parameters for OMX_IMAGE_PARAM_PORTFORMATTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nIndex indicates the enumeration index for the format from 0x0 to N-1.

- `eCompressionFormat` is an enumeration describing the compression format used on the port. When `OMX_IMAGE_CodingUnused` is specified, the `eColorFormat` field is valid. For enumerations regarding `OMX_IMAGE_CODINGTYPE`, see Table 4-84.
- `eColorFormat` is the decompressed color format used for the port. This field is valid only when the `eCompressionFormat` field is set to `OMX_IMAGE_CodingUnused`. For enumerations on `OMX_COLOR_FORMATTYPE`, see section 4.2.

4.4.5 OMX_IMAGE_PARAM_FLASHCONTROLTYPE

The `OMX_IMAGE_PARAM_FLASHCONTROLTYPE` structure defines the mode of operation for flash control and configuration.

`OMX_IMAGE_PARAM_FLASHCONTROLTYPE` is defined as follows.

```
typedef struct OMX_IMAGE_PARAM_FLASHCONTROLTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_IMAGE_FLASHCONTROLTYPE eFlashControl;
} OMX_IMAGE_PARAM_FLASHCONTROLTYPE;
```

4.4.5.1 Parameters

The parameters for `OMX_IMAGE_PARAM_FLASHCONTROLTYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `eFlashControl` is an enumeration for the flash control modes. Table 4-85 shows the supported image flash controls.

Table 4-85: Supported Image Flash Controls

Field Name	Flash Control Description
`OMX_IMAGE_FlashControlOn`	Strobe at every shot
`OMX_IMAGE_FlashControlOff`	Strobe off
`OMX_IMAGE_FlashControlAuto`	Strobe according to environment light
`OMX_IMAGE_FlashControlRedEyeReduction`	Pre-shot strobes
`OMX_IMAGE_FlashControlFillin`	Flash for background/ foreground effect
`OMX_IMAGE_FlashControlTorch`	Flash is always on

4.4.6 OMX_IMAGE_CONFIG_FOCUSCONTROLTYPE

`OMX_IMAGE_CONFIG_FOCUSCONTROLTYPE` controls the focus mode and range.

OMX_IMAGE_CONFIG_FOCUSCONTROLTYPE is defined as follows.

```
typedef struct OMX_IMAGE_CONFIG_FOCUSCONTROLTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_IMAGE_FOCUSCONTROLTYPE eFocusControl;
    OMX_U32 nFocusSteps;
    OMX_U32 nFocusStepIndex;
} OMX_IMAGE_CONFIG_FOCUSCONTROLTYPE;
```

4.4.6.1 Parameters

The parameters for OMX_IMAGE_CONFIG_FOCUSCONTROLTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- eFocusControl is an enumeration that specifies the image focus controls. Table 4-86 shows the supported image focus controls.

Table 4-86: Supported Image Focus Controls

Field Name	Focus Control Description
OMX_IMAGE_FocusControlOn	Focus control On Focus adjustments are being performed manually by the user. Focus status determination is performed by the component.
OMX_IMAGE_FocusControlOff	Focus control off Focus adjustments are being performed manually by the user. Focus status determination is performed manually (visually inspection via viewfinder) by the user.
OMX_IMAGE_FocusControlAuto	Auto focus control on Focus adjustments are being performed automatically and continuously by the component until a capture request is issued. Focus status determination is performed by the component.

Field Name	Focus Control Description
OMX_IMAGE_FocusControlAutoLock	Auto focus control with lock support on Focus adjustment is locked to the current focus adjustment setting. Focus status determination is performed by the component. The focus status request for this mode continually reflects the focus status upon receiving this lock focus request.

- nFocusSteps is a value that specifies the number of steps that the focus can take on. The range is 0 mm to infinity.
- nFocusStepIndex defines the current position of the focus.

4.4.7 OMX_IMAGE_PARAM_QFACTORTYPE

OMX_IMAGE_PARAM_QFACTORTYPE determines the quality factor for JPEG compression, which controls the tradeoff between image quality and size. Q Factor provides a simpler means of controlling the JPEG compression quality than directly programming quantization tables for chroma and luma.

OMX_IMAGE_PARAM_QFACTORTYPE is defined as follows.

```
typedef struct OMX_IMAGE_PARAM_QFACTORTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nQFactor;
} OMX_IMAGE_PARAM_QFACTORTYPE;
```

4.4.7.1 Parameters

The parameters for OMX_IMAGE_PARAM_QFACTORTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nQFactor is a compression quality factor value in the range 1–100. A factor of 1 produces the smallest, worst quality images, and a factor of 100 produces the largest, best quality images. A typical default is 75 for small, good quality images.

4.4.8 OMX_IMAGE_PARAM_QUANTIZATIONTABLETYPE

OMX_IMAGE_PARAM_QUANTIZATIONTABLETYPE provides JPEG quantization tables, which are used to determine DCT compression for YUV data. OMX_IMAGE_PARAM_QUANTIZATIONTABLETYPE is an alternative to specifying Q factor, providing exact control of compression.

OMX_IMAGE_PARAM_QUANTIZATIONTABLETYPE is defined as follows.

```
typedef struct OMX_IMAGE_PARAM_QUANTIZATIONTABLETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_IMAGE_QUANTIZATIONTABLETYPE eQuantizationTable;
    OMX_U8 nQuantizationMatrix[64];
} OMX_IMAGE_PARAM_QUANTIZATIONTABLETYPE;
```

4.4.8.1 Parameters

The parameters for `OMX_IMAGE_PARAM_QUANTIZATIONTABLETYPE` are defined as follows.

- `nPortIndex` represents the port that this structure applies to.
- `eQuantizationTable` is an enumeration for the quantization table type, which defines luma or chroma table types. Table 4-87 shows the supported image quantization table types.

Table 4-87: Supported Image Quantization Table Types

Field Name	Quantization Table Description
`OMX_IMAGE_QuantizationTableLuma`	Quantization table for the luma coefficients
`OMX_IMAGE_QuantizationTableChroma`	Quantization table for both the Cb and Cr chroma coefficients
`OMX_IMAGE_QuantizationTableChromaCb`	Quantization table for Cb chroma coefficients only
`OMX_IMAGE_QuantizationTableChromaCr`	Quantization table for Cr chroma coefficients only

- `nQuantizationMatrix` is the JPEG quantization table of coefficients stored in increasing columns and then by rows of data (i.e., row 1,… row 8). Quantization values are in the range 0–255 and are stored in linear order (i.e., the component will zigzag the quantization table data internally if required).

4.4.8.2 Error Conditions

On processing the `OMX_IMAGE_PARAM_QUANTIZATIONTABLETYPE` structure, the following error conditions can occur:

- `OMX_ErrorSeperateTablesUsed` when `OMX_GetParameter` function is called using `OMX_IMAGE_QuantizationTableChroma` and separate quantization tables are used for the Chroma (Cb and Cr) coefficients.

This error indicates that separate `OMX_GetParameter` function calls need to be issued using `OMX_IMAGE_QuantizationTableChromaCb` and `OMX_IMAGE_QuantizationTableChromaCb` to query for the separate chroma coefficient quantization tables.

4.4.9 OMX_IMAGE_PARAM_HUFFMANTTABLETYPE

The OMX_IMAGE_PARAM_HUFFMANTTABLETYPE structure is used to set the Huffman variable code length type used for JPEG.

OMX_IMAGE_PARAM_HUFFMANTTABLETYPE is defined as follows.

```
typedef struct OMX_IMAGE_PARAM_HUFFMANTTABLETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_IMAGE_HUFFMANTABLETYPE eHuffmanTable;
    OMX_U8 nNumberOfHuffmanCodeOfLength[16];
    OMX_U8 nHuffmanTable[256];
}OMX_IMAGE_PARAM_HUFFMANTTABLETYPE;
```

4.4.9.1 Parameters

The parameters for OMX_IMAGE_PARAM_HUFFMANTTABLETYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- eHuffmanTable is an enumeration for the Huffman table types. Table 4-88 shows the supported Huffman table types.

Table 4-88: Supported Huffman Table Types

Field Name	Huffman Table Description
OMX_IMAGE_HuffmanTableAC	Huffman table to be applied to Luma and Chroma AC coefficients
OMX_IMAGE_HuffmanTableDC	Huffman table to be applied to Luma and Chroma DC coefficients
OMX_IMAGE_HuffmanTableACLuma	Huffman table to be applied to Luma AC coefficients only
OMX_IMAGE_HuffmanTableACChroma	Huffman table to be applied to Chroma AC coefficients only
OMX_IMAGE_HuffmanTableDCLuma	Huffman table to be applied to Luma DC coefficients only
OMX_IMAGE_HuffmanTableDCChroma	Huffman table to be applied to Chroma DC coefficients only

- nNumberOfHuffmanCodeOfLength is a value in the range of 0–16 that represents the number of Huffman codes of each possible length.
- nHuffmanTable is a value in the range of 0–255. The table sizes used for AC and DC Huffman tables are 16 and 162.

4.4.9.2 Error Conditions

On processing the OMX_IMAGE_PARAM_HUFFMANTTABLETYPE structure, the following error conditions can occur:

- OMX_ErrorSeperateTablesUsed when the OMX_GetParameter function is called using OMX_IMAGE_HuffmanTableAC or OMX_IMAGE_HuffmanTableDC and separate Huffman tables are used for the Luma and Chroma coefficients.

This error indicates that separate OMX_GetParameter function calls need to be issued using OMX_IMAGE_HuffmanTableACLuma and OMX_IMAGE_HuffmanTableACChroma to obtain the AC coefficient information and separate OMX_GetParameter function calls need to be issued using OMX_IMAGE_HuffmanTableDCLuma and OMX_IMAGE_HuffmanTableDCChroma to obtain the DC coefficient information.

4.4.10 OMX_CONFIG_FLICKERREJECTIONTYPE

OMX_CONFIG_FLICKERREJECTIONTYPE is used to specify the flicker rejection mode, generally used to account for the flicker effect noticeable under electric lighting.

OMX_CONFIG_FLICKERREJECTIONTYPE is defined as follows.

```
typedef struct OMX_CONFIG_FLICKERREJECTIONTYPE{
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_FLICKERREJECTIONTYPE eFlickerRejection;
} OMX_CONFIG_FLICKERREJECTIONTYPE;
```

4.4.10.1 Parameters

The parameters for OMX_CONFIG_FLICKERREJECTIONTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- eFlickerRejection is an enumerated value indicating the flicker rejection mode.

Table 4-89: eFlickerRejection Values

OMX_FLICKERREJECTIONTYPE value	Mode Description
OMX_FlickerRejectionOff	No flicker rejection
OMX_FlickerRejectionAuto	Automatic flicker rejection
OMX_FlickerRejection50	Flicker rejection at 50Hz
OMX_FlickerRejection60	Flicker rejection at 60Hz

4.4.11 OMX_IMAGE_HISTOGRAMTYPE

The image histogram is measured on the data image input and gives the number of pixels for each tonal value. The result is delivered with the OMX_IMAGE_HISTOGRAMDATATYPE structure. It is possible that not all histogram types are supported by a camera, or even that a camera might not support all histogram types depending on what its port settings are. This support information is retrieved with the OMX_IMAGE_HISTOGRAMINFOTYPE structure.

Histogram measurements can be controlled by the following data structure.

```
typedef struct OMX_IMAGE_HISTOGRAMTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nBins;
    OMX_HISTOGRAMTYPE eHistType;
} OMX_IMAGE_HISTOGRAMTYPE;
```

4.4.11.1 Parameters

The parameters for OMX_IMAGE_HISTOGRAMTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nBins specifies the number of histogram bins. When queried with set to zero, the response gives the maximum number of bins allowed.
- eHistType is an enumeration specifying the histogram type, as shown in Table 4-90. This parameter is also used to enable and disable histogram generation.

Table 4-90: Histogram control type Values

OMX_HISTOGRAMTYPE value	Histogram Description
OMX_Histogram_Off	Histogram is disabled.
OMX_Histogram_RGB	RGB Color (R, G, B) histogram mode (and implicitly enables histogram if previously off).
OMX_Histogram_Luma	Luma (Y) histogram mode (and implicitly enables histogram if previously off).
OMX_Histogram_Chroma	Chroma (Cb, Cr) histogram (and implicitly enables histogram if previously off).

4.4.12 OMX_IMAGE_HISTOGRAMDATATYPE

The histogram estimation data is described with the following structure.

```
typedef struct OMX_IMAGE_HISTOGRAMDATATYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
```

```
    OMX_U32 nPortIndex;
    OMX_HISTOGRAMTYPE eHistType;
    OMX_U32 nBins;
    OMX_U8 data[1];
} OMX_IMAGE_HISTOGRAMDATATYPE;
```

4.4.12.1 Parameters

The parameters for `OMX_IMAGE_HISTOGRAMDATATYPE` are defined as follows.

- `nSize` is the size of the structure including the length of data field containing the histogram data.
- `nPortIndex` represents the port that this structure applies to.
- `eHistType` is the read-only value specifying the histogram type, as shown in Table 4-90. Multiple histogram components may be generated depending on the type specified.
- `nBins` is the read-only value containing the number of bins the histogram allocates for each component.
- `data[1]` first byte of the histogram data. Histogram data is recorded as number of pixels per bin. For `eHistType` that specify multiple histogram components the size of the `data` will be number of histogram components times the number of bins and the data for each component will be grouped together. For example, `OMX_Histogram_RGB` has three components, thus the total number of bins recorded in `data` will be 3x `nBins` with the entire R histogram followed by entire G histogram followed by entire B histogram. The number of bits per bin may vary by histogram type and by how many bins are specified. This information can be retrieved with the `OMX_IMAGE_HISTOGRAMINFOTYPE` structure.

4.4.13 OMX_IMAGE_HISTOGRAMINFOTYPE

This structure is used to retrieve the histogram types supported by the component. It will also report the maximum number of bins for the histogram type as well as the number of bits used to represent the histogram data per bin.

```
typedef struct OMX_IMAGE_HISTOGRAMINFOTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_HISTOGRAMTYPE eHistType;
    OMX_U32 nBins;
    OMX_U16 nBitsPerBin;
} OMX_IMAGE_HISTOGRAMINFOTYPE;
```

4.4.13.1 Parameters

The parameters for OMX_IMAGE_HISTOGRAMDATATYPE are defined as follows.

- nSize is the size of the structure including the length of data field containing the histogram data.
- nPortIndex represents the port that this structure applies to.
- eHistType is the histogram type being queried whether it is supported or not. The types of histograms available are listed in Table 4-90.
- nBins is the read-only value containing the maximum number of bins available for the specified eHistType. If the histogram type is not supported then nBins will be 0.
- nBitsPerValue is the number of bits per bin.

4.4.14 OMX_CONFIG_FILEFORMATTYPE

OMX_CONFIG_FILEFORMATTYPE is used to specify how the component shall interpret the URI, for example if it is formatted in DCF form and whether it will auto-increment the file name index.

```
typedef struct OMX_CONFIG_FILEFORMATTYPE{
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_FILEFORMATTYPE eFileFormat;
} OMX_CONFIG_FILEFORMATTYPE;
```

4.4.14.1 Parameters

The parameters for OMX_CONFIG_FILEFORMATTYPE are defined as follows.

- eFileFormat is file format type to let the component know how to interpret the given URI and whether any characters are to be incremented.

4-91: FileFormat Types

OMX_FILEFORMATTYPE value	Mode Description
OMX_FileFormatNone	No file format naming convention is followed.

OMX_FileFormatDCF	DCF file naming convention. The provided URI should be DCF formatted with the first four characters of the file name any alphanumeric characters and the final four a file number greater than 0 to a maximum of 9999. For example, "file:///DCIM/100abcde/ab4_1234.jpg" where "ab4_" is the alphanumeric portion and"1234" is the file number. The component shall increment the file number part of the name until it reaches the maximum. The component shall return OMX_ErrorNoMore if the index has reached the maximum possible value.

4.4.15 IMAGE CAPTURE START-END NOTIFICATIONS

This functionality allows IL client to be notified of the exact moments when image capturing is active.

In case of camera devices with mechanical shutter, this corresponds to the opening and closing of the shutter.

In case of devices with rolling shutter, it corresponds to the moments of exposure starting and ending.

This support can utilized to better synchronize the actual moments of image capture with notifications provided by IL client to the user that image capture is active.

OMX_IndexConfigImageCaptureStarted allows the IL client to be notified of when an image to be captured by the component is exposed (capture process actually starts). OMX_IndexConfigImageCaptureEnded allows the IL client to be notified of when an image to be captured by the camera component has ended being exposed (capture process actually ends). These notifications allows for synchronization with other functionality in the system , e.g. playback of camera capture sounds or graphics.

OMX_IndexConfigImageCaptureStarted and OMX_IndexConfigImageCaptureEnded are both associated with the OMX_PARAM_U32TYPE structure. The unsigned 32-bit value in the structure is used to count the number image capture start and end occurrences. This counter is initialized with 0.

For example, to enable notification of the exact moment of image capture start, the IL client shall subscribe to callbacks using OMX_IndexConfigCallbackRequest

with `OMX_CONFIG_CALLBACKREQUESTTYPE::nIndex= OMX_IndexConfigImageCaptureStarted.`

When an image exposure is initiated, the camera component issues the `OMX_EventIndexSettingChanged` event and the IL client knows that an image is being exposed. It is not expected that the IL client will need to query the value of the counter. If the IL client intends to use the counter, it is expected that it also resets its value at appropriate moments.

Similar usage is for notifications on the exact moment of image capture end.

4.5 "Other" Domain

This section describes the concepts, structures, and configurations for the domain designated as "other" and moniker distinguishing it from the audio, video and image domains. The OMX_Other.h header specifies the parameters and configurations in detail.

Presently the other domain formalizes only a "time" data format and its associated operation though other data formats may be formalized in the future. The time data format exists to facilitate synchronization. To provide context to the definition of the time data format, the following section explains OpenMAX IL's synchronization mechanisms.

4.5.1 Parameters and Config Indexes

The header OMX_Index.h contains the enumeration `OMX_INDEXTYPE`, which contains all of the standard index values used with the functions `OMX_GetParameter`, `OMX_SetParameter`, `OMX_GetConfig`, and `OMX_SetConfig`. Table 4-92 describes the index values that relate to Other Domain.

Table 4-92: Index Values for Other Domain

Index	Description
`OMX_IndexConfigTimeScale`	Used with `OMX GetConfig` and `OMX_SetConfig` to access a `OMX_TIME_CONFIG_SCALETYPE` structure denoting the scale of the media clock.
`OMX_IndexConfigTimeClockState`	Used with `OMX_GetConfig` and `OMX_SetConfig` to access a `OMX_TIME_CONFIG_CLOCKSTATETYPE` structure denoting the state of the media clock.
`OMX_IndexConfigTimeCurrentMediaTime`	Used with `OMX_GetConfig` to query a `OMX_TIME_CONFIG_TIMESTAMPTYPE` structure denoting the current media time.
`OMX_IndexConfigTimeCurrentWallTime`	Used with `OMX_GetConfig` to query a `OMX_TIME_CONFIG_TIMESTAMPTYPE` structure denoting the current wall clock time.
`OMX_IndexConfigTimeMediaTimeRequest`	Used with `OMX_SetConfig` to request a clock component operation using a `OMX_TIME_CONFIG_MEDIATIMEREQUESTTYPE` structure.

Index	Description
OMX_IndexConfigTimeClientStartTime	Used with OMX_SetConfig to set the start time of the given client stream using the OMX_TIME_CONFIG_TIMESTAMPTYPE structure.
OMX_IndexConfigTimePosition	Used with OMX_GetConfig and OMX_SetConfig to access a OMX_TIME_CONFIG_SCALETYPE structure denoting the current position in time.
OMX_IndexConfigTimeSeekMode	Used with OMX_GetConfig and OMX_SetConfig to access a OMX_TIME_CONFIG_SCALETYPE structure denoting the current seek mode.
OMX_IndexConfigTimeCurrentReference	Used with OMX_SetConfig to set the OMX_TIME_CONFIG_TIMESTAMPTYPE structure denoting the current reference clock time.
OMX_IndexConfigTimeActiveRefClockUpdate	Used with OMX_SetConfig to set the OMX_TIME_CONFIG_ACTIVEREFCLOCKUPDATETYPE structure denoting the role of reference clock provider to clock clients.
OMX_IndexConfigTimeUpdate	Used with OMX_SetConfig to pass updates from the clock to its clients using the OMX_TIME_MEDIATIMETYPE structure.

4.5.2 OMX_TIME_CONFIG_SEEKMODETYPE

A component's seek mode defines the semantics it follows when an IL client requests a change in position (via the OMX_IndexConfigTimePosition configuration).

OMX_TIME_CONFIG_SEEKMODETYPE is defined as follows.

```
typedef struct OMX_TIME_CONFIG_SEEKMODETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_TIME_SEEKMODETYPE eType;
} OMX_TIME_CONFIG_SEEKMODETYPE;
```

4.5.2.1 Parameters

The parameters for OMX_TIME_CONFIG_SEEKMODETYPE are defined as follows.

- eType is seek mode and must be a value from the OMX_TIME_SEEKMODETYPE enumeration

Table 4-93: Seek Modes Defined by OMX_TIME_SEEKMODETYPE

Field Name	Description
OMX_TIME_SeekModeFast	Prefer seeking to an approximation of the requested seek position over the actual seek position if it results in a faster seek.
OMX_TIME_SeekModeAccurate	Prefer seeking to the actual seek position over an approximation of the requested seek position even if it results in a slower seek.

4.5.3 OMX_TIME_CONFIG_TIMESTAMPTYPE

A timestamp represents a position in time relative to some clock. The OMX_IndexConfigTimeCurrentWallTime, OMX_IndexConfigTimeCurrentMediaTime, and OMX_IndexConfigTimeCurrentReference configurations leverage this structure.

OMX_TIME_CONFIG_TIMESTAMPTYPE is defined as follows.

```
typedef struct OMX_TIME_CONFIG_TIMESTAMPTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_TICKS nTimestamp;
} OMX_TIME_CONFIG_TIMESTAMPTYPE;
```

4.5.3.1 Parameters

The parameters for OMX_TIME_CONFIG_TIMESTAMPTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- nTimestampType holds the actual timestamp value.

4.5.4 OMX_TIME_CONFIG_MEDIATIMEREQUESTTYPE

The media time request represents a request for notification at the media time specified.

OMX_TIME_CONFIG_MEDIATIMEREQUESTTYPE is defined as follows.

```
typedef struct OMX_TIME_CONFIG_MEDIATIMEREQUESTTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_PTR pClientPrivate;
    OMX_TICKS nMediaTimestamp;
    OMX_TICKS nOffset;
} OMX_TIME_CONFIG_MEDIATIMEREQUESTTYPE;
```

4.5.4.1 Parameters

The parameters for OMX_TIME_CONFIG_MEDIATIMEREQUESTTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- pClientPrivate client private data to disambiguate this media time from others.
- nMediaTimestamp media time requested.
- nOffset amount of wall clock time by which this request should be fulfilled early.

4.5.5 OMX_TIME_MEDIATIMETYPE

The media time structure is sent to a port either to fulfill a media time request or when the clock state or scale has changed. This structure is either used with the index OMX_IndexConfigTimeUpdate and a call to OMX_SetConfig if the port is tunneled, or written into the payload of a buffer and sent via the OMX_FillBufferDone callback to the client.

OMX_TIME_MEDIATIMETYPE is defined as follows.

```
typedef struct OMX_TIME_MEDIATIMETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nClientPrivate;
    OMX_TIME_UPDATETYPE eUpdateType;
    OMX_TICKS nMediaTimestamp;
    OMX_TICKS nOffset;
    OMX_TICKS nWallTimeAtMediaTime;
    OMX_S32 xScale;
    OMX_TIME_CLOCKSTATE eState;
} OMX_TIME_MEDIATIMETYPE;
```

4.5.5.1 Parameters

The parameters for OMX_TIME_MEDIATIMETYPE are defined as follows.

- pClientPrivate clock client private data to disambiguate this media time from others. If the eUpdateType field indicates this is scale or state change notification, the pClientPrivate field shall be zero.
- eUpdateType designates reason for the this update was sent and must be a value from the OMX_TIME_UPDATETYPE enumeration

Table 4-94: Media Time Update Types Defined by OMX_TIME_UPDATETYPE

Field Name	Description
OMX_TIME_UpdateRequestFulfillment	Update is the fulfillment of a media time request.
OMX_TIME_UpdateScaleChanged	Update to indicate the clock scale has changed.
OMX_TIME_UpdateClockStateChanged	Update to indicate the clock state has changed.

- nMediaTimeStamp denotes the media time requested (if this is a request fulfillment).
- nOffset designates amount of wall clock time by which this request was actually fulfilled early (if this is a request fulfillment).
- nWallTimeAtMediaTime denotes the wall time corresponding to nMediaTimeStamp (if this is a request fulfillment).
- xScale designates the current media time scale in Q16 format when the structure was completed.
- eState designates the clock state when the structure was completed, and must be a value from the OMX_TIME_CLOCKSTATE enumeration

Table 4-95: Clock States Defined by OMX_TIME_CLOCKSTATE

Field Name	Description
OMX_TIME_ClockStateRunning	Clock is running.
OMX_TIME_ClockStateWaitingForStartTime	Clock is waiting until the prescribed clients emit their start time.
OMX_TIME_ClockStateStopped	Clock is stopped.

4.5.6 OMX_TIME_CONFIG_SCALETYPE

The clock scale config represents the current clock scale. It allows the IL client to query and set the clock scale.

OMX_TIME_CONFIG_SCALETYPE is defined as follows.

```
typedef struct OMX_TIME_CONFIG_SCALETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_S32 xScale;
} OMX_TIME_CONFIG_SCALETYPE;
```

4.5.6.1 Parameters

The parameters for OMX_TIME_CONFIG_SCALETYPE are defined as follows.

- xScale the scale of the media time in Q16 format.

4.5.7 OMX_TIME_CONFIG_CLOCKSTATETYPE

The clock state config represents the current state of the media clock. It allows the IL client to set and query the clock state.

OMX_TIME_CONFIG_CLOCKSTATETYPE is defined as follows.

```
 typedef struct OMX_TIME_CONFIG_CLOCKSTATETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_TIME_CLOCKSTATE eState;
    OMX_TICKS nStartTime;
    OMX_TICKS nOffset;
    OMX_U32 nWaitMask;
} OMX_TIME_CONFIG_CLOCKSTATETYPE;
```

4.5.7.1 Parameters

The parameters for OMX_TIME_CONFIG_CLOCKSTATETYPE are defined as follows.

- eState denotes the state of the media clock and must be a value in the OMX_TIME_CLOCKSTATE enumeration.
- nStartTime designates the media time the media clock is initialized to.
- nOffset designates the time to offset the media time by.
- nOffset specifies a mask of OMX_CLOCKPORT values designating which ports, if any, to wait on.

Table 4-96: Possible Clock Port Values

Field Name	Value
OMX_CLOCKPORT0	0x00000001
OMX_CLOCKPORT1	0x00000002
OMX_CLOCKPORT2	0x00000004
OMX_CLOCKPORT3	0x00000008
OMX_CLOCKPORT4	0x00000010
OMX_CLOCKPORT5	0x00000020
OMX_CLOCKPORT6	0x00000040
OMX_CLOCKPORT7	0x00000080

4.6 Common or Domain Independent

This section describes the parameter and configuration details for controls that are either domain independent or applicable for all domains.

4.6.1 Parameter and Configuration Indices

The header OMX_Index.h contains the enumeration `OMX_INDEXTYPE`, which contains all standard index values used core functions `OMX_GetParameter`, `OMX_SetParameter`, `OMX_GetConfig`, and `OMX_SetConfig`. Table 4-97 shows the index values that relate to domain independent support.

Table 4-97: Domain Independent Indices

OpenMAX IL Indices (OMX_Index.h)	Corresponding OpenMAX IL Structures
`OMX_IndexConfigCommitMode`	`OMX_CONFIG_COMMITMODETYPE`
`OMX_IndexConfigCommit`	`OMX_CONFIG_COMMITTYPE`
`OMX_IndexConfigCallbackRequest`	`OMX_CONFIG_CALLBACKREQUESTTYPE`
`OMX_IndexParamMediaContainer`	`OMX_MEDIACONTAINER_INFOTYPE`
`OMX_IndexParamReadOnlyBuffers`	`OMX_PORTBOOLEANTYPE` Used for setting and querying the use of ready-only buffers, i.e., buffers marked with OMX_BUFFERFLAG_READONLY. By querying this index on an output port, the IL client can determine whether the output port will produce read-only buffers. By enabling this setting on an input port, the IL client can notify the component on its intention to send read-only buffers to the port. Setting this parameter on an output port is not supported.

For example, `OMX_IndexConfigCommitMode` index is used with `OMX_GetConfig` and `OMX_SetConfig` to access `OMX_CONFIG_COMMITMODETYPE`.

4.6.2 OMX_CONFIG_COMMITMODETYPE

The `OMX_CONFIG_COMMITMODETYPE` structure is used for configuring the component to operate in a deferred or immediate mode.

In deferred mode, all settings (`OMX_SetConfig()` calls) are cached and not applied until the IL client issues an explicit commit request (`OMX_IndexConfigCommit`).

Upon receiving the commit request, the component shall apply all the cached setting atomically.

In immediate mode, the component does not cache any settings and applies the settings immediately. Immediate mode shall be the default mode of operation.

In the case, a component is requested to transition from deferred to immediate mode while holding onto cached settings (`OMX_IndexConfigCommit` has not been issued), the component shall discard all the cached setting requests and complete the transition to immediate mode without any changes to its configuration settings.

At the time of commit, all cached settings are to be applied simultaneously and atomically; the order of the individual `OMX_SetConfig()` calls shall not affect the end result. However, if the IL client issues the same setting request multiple times before commit, the last setting request shall override the earlier settings.

If the IL client calls `OMX_GetConfig()` on a setting that is currently cached, the result shall reflect the current active value and not the setting request currently cached.

Requesting the component to transition to deferred mode while the component is currently configured in deferred mode shall result in an error (`OMX_ErrorInvalidMode`).

`OMX_CONFIG_COMMITMODETYPE` is defined as follows.

```
typedef struct OMX_CONFIG_COMMITMODETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_BOOL bDeferred;
} OMX_CONFIG_COMMITMODETYPE;
```

4.6.2.1 Parameters

The parameters for `OMX_CONFIG_COMMITMODETYPE` are defined as follows.

- `eDeferred` specifies the mode of operation. `OMX_TRUE` shall indicate deferred mode and `OMX_FALSE` shall indicate immediate mode.

  A component shall default to immediate mode (`OMX_FALSE`).

4.6.2.2 Error Conditions

On processing the `OMX_CONFIG_COMMITMODETYPE` structure, the following error conditions can occur:

- `OMX_ErrorInvalidMode` when the component is being requested to transition to deferred mode while currently configured for deferred mode.

4.6.3 OMX_CONFIG_COMMITTYPE

The OMX_CONFIG_COMMITMODETYPE structure is used to commit previously cached settings.

This functionality is only valid if the component is configured for deferred mode of operation, otherwise an error shall be issued.

OMX_CONFIG_COMMITTYPE is defined as follows.

```
typedef struct OMX_CONFIG_COMMITTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
} OMX_CONFIG_COMMITTYPE;
```

4.6.3.1 Parameters

None

4.6.3.2 Functionality

If an OMX_SetConfig() fails (returns an error code) while a component is in the deferred mode, the next commit request shall fail with OMX_ErrorBadParameter. For these cases when OMX_SetConfig() fails, it is recommended that an IL client immediately issues a commit in order to flush the cached settings.

It is also possible that separate configs have interdependencies, e.g., a particular setting of one config may restrict the valid value range of another setting. The component is not required to immediately validate the current OMX_SetConfig() settings, it is however required to perform this validation at the time of commitment. In such cases the individual OMX_SetConfig() calls can fail with an error code only if the component will not support a particular setting in any situation. If all the individual OMX_SetConfig() calls have succeeded, but the component determines the combination invalid, it shall signal the IL client by returning OMX_ErrorBadParameter from the commit OMX_SetConfig() call.

When a commit fails, all settings cached in the component shall be discarded.

4.6.3.3 Error Conditions

On processing the OMX_CONFIG_COMMITTYPE structure, the following error conditions can occur:

- OMX_ErrorInvalidMode when the component is configured for immediate mode of operation (refer to OMX_CONFIG_COMMITMODETYPE).
- OMX_ErrorBadParameter when a OMX_SetConfig() cache settings call failed

- OMX_ErrorBadParameter when the component determines the combinational cached settings are invalid.

4.6.4 OMX_CONFIG_CALLBACKREQUESTTYPE

The OMX_CONFIG_CALLBACKREQUESTTYPE structure is used to signal setting changes associated with a parameter or config index (OMX_INDEXTYPE).

The notification is associated with the OMX_EventIndexSettingChanged event, the event callback includes the parameter or config index that is associated with the setting change. When receiving the event, the IL client shall use OMX_GetParameter() or OMX_GetConfig() as appropriate to retrieve the new value of the parameter or config.

The callback settings are fully independent of any other settings applied to the component, including component state.

OMX_CONFIG_CALLBACKREQUESTTYPE is defined as follows.

```
typedef struct OMX_CONFIG_CALLBACKREQUESTTYPE{
    OMX_U32  nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32  nPortIndex;
    OMX_INDEXTYPE nIndex;
    OMX_BOOL bEnable;
} OMX_CONFIG_CALLBACKREQUESTTYPE;
```

4.6.4.1 Parameters

The parameters for OMX_CONFIG_CALLBACKREQUESTTYPE are defined as follows.

- nPortIndex is the value containing the index of the port (can be OMX_ALL).
- nIndex specifies the OMX_INDEXTYPE index to be associated with the event notification.
- bEnabled is a Boolean field that indicates if event notification for the nIndex shall be enabled.

  By default event notifications associated with this functionality are disabled.

4.6.5 OMX_MEDIACONTAINER_INFOTYPE

The OMX_MEDIACONTAINER_INFOTYPE structure identifies the media container format.

OMX_MEDIACONTAINER_INFOTYPE is defined as follows.

```
typedef struct OMX_MEDIACONTAINER_INFOTYPE{
    OMX_U32  nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_MEDIACONTAINER_FORMATTYPE eFmtType;
```

```
} OMX_MEDIACONTAINER_INFOTYPE;
```

4.6.5.1 Parameters

The parameters for `OMX_MEDIACONTAINER_INFOTYPE` are defined as follows.

- `eFmtType` specifies the media container format type.

Table 4-98 : Media Container Formats

OMX_MEDIACONTAINER_FORMATTYPE Enumerated Value	Description
OMX_FORMAT_RAW	No Format
OMX_FORMAT_MP4	Mpeg4 File
OMX_FORMAT_3GP	3GP File
OMX_FORMAT_3G2	3G2 File
OMX_FORMAT_AMC	AMC file
OMX_FORMAT_SKM	SKM file
OMX_FORMAT_K3G	K3G file
OMX_FORMAT_VOB	VOB file
OMX_FORMAT_AVI	AVI File
OMX_FORMAT_ASF	ASF File
OMX_FORMAT_RM	Real Media
OMX_FORMAT_MPEG_ES	Mpeg2 ES
OMX_FORMAT_DIVX	Divx file
OMX_FORMAT_MPEG_TS	Mpeg2 TS
OMX_FORMAT_QT	Quicktime
OMX_FORMAT_M4A	M4A file
OMX_FORMAT_MP3	Mp3 file
OMX_FORMAT_WAVE	Wave file
OMX_FORMAT_XMF	XMF file
OMX_FORMAT_AMR	AMR file
OMX_FORMAT_AAC	AAC file
OMX_FORMAT_EVRC	EVRC file
OMX_FORMAT_QCP	QCP file
OMX_FORMAT_SMF	SMF file
OMX_FORMAT_OGG	OGG file
OMX_FORMAT_BMP	BMP file
OMX_FORMAT_JPG	JPG file
OMX_FORMAT_JPG2000	JPG2000 file
OMX_FORMAT_MKV	MKV file
OMX_FORMAT_FLV	FLV file

OMX_FORMAT_M4V	M4V file
OMX_FORMAT_F4V	F4V file
OMX_FORMAT_WEBM	WebM file
OMX_FORMAT_WEBP	WebP file

4.6.6 OMX_CONFIG_PORTBOOLEANTYPE

OMX_CONFIG_PORTBOOLEANTYPE is used to specify a port specific Boolean property.

OMX_CONFIG_PORTBOOLEANTYPE is defined as follows.

```
typedef struct OMX_CONFIG_PORTBOOLEANTYPE{
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bEnabled;
} OMX_CONFIG_PORTBOOLEANTYPE;
```

4.6.6.1 Parameters

The parameters for OMX_CONFIG_PORTBOOLEANTYPE are defined as follows.

- nPortIndex represents the port that this structure applies to.
- bEnabled is a Boolean field used to enable a port specific functionality.

5 OpenMAX IL Component Extension APIs

5.1 Description of the Extension Process

An OpenMAX IL component may support any setting defined in the OpenMAX IL specification. Vendors can add to the list of parameters and configurations not included in the standard header files. These additions are referred to as *extensions*.

Any extensions approved by Khronos are considered OpenMAX IL extensions. Any extensions not approved by Khronos are vendor-defined extensions.

OpenMAX IL extensions are defined in a predefined set of extension header files, namely:

- OMX_CoreExt.h: OpenMAX IL core extension API
- OMX_ComponentExt.h: OpenMAX IL component extension API
- OMX_AudioExt.h: OpenMAX IL audio domain extension data structures
- OMX_IVCommonExt.h: OpenMAX IL extension structures common to image and video domains
- OMX_VideoExt.h: OpenMAX IL video domain extension data structures
- OMX_ImageExt.h: OpenMAX IL image domain extension data structures
- OMX_OtherExt.h : OpenMAX IL other domain extension data structures (includes A/V synchronization extensions)
- OMX_IndexExt.h: Index of all OpenMAX IL extension data structures

Any vendor that develops OpenMAX IL components may add to the list of standard indexes a collection of one or more custom parameters or configuration indexes. Each vendor-specific index shall have a value greater than the value of `OMX_IndexVendorStartUnused` and less than the value of `OMX_IndexMax` - 1. Each OpenMAX IL extension index has a value greater than the value of `OMX_IndexKhronosExtension` and less than the value of `OMX_IndexVendorStartUnused – 1`.

Each extension parameter or configuration index may apply to one of the four existing domains, namely audio, video, image, and “other”. It may also apply to a parameter or configuration that does not belong to any known domain.

A vendor-specific extension index to a parameter or configuration may be defined by a string and be reported in the component description documentation. The IL client may obtain the index related to this property using the component function `OMX_GetExtensionIndex`. This function provides a numeric index from a string that names the custom index. The function is specific to a component, so a component handle shall be passed to the function. The function is described in section 3.2.2.12.

The numeric index can be used with the functions `OMX_GetParameter` and `OMX_SetParameter` if the index regards a parameter or with the functions `OMX_GetConfig` and `OMX_SetConfig` if the index is a configuration index. The nature of the parameter or configuration value should be documented in the extension section of the component documentation. Khronos, or its designee, will maintain a publicly-accessible registry of OpenMAX IL extensions. These extensions are baselined to a version of an OpenMAX IL specification and may be promoted to a subsequent release of the OpenMAX IL specification.

5.1.1 GetExtensionIndex

The `OMX_GetExtensionIndex` method will translate a vendor-specific configuration or parameter string into an OpenMAX IL structure index. There is no requirement for the component to support this command for the indexes already found in the `OMX_INDEXTYPE` enumeration or in the anonymous enumeration in OMX_IndexExt.h, thus reducing a component's memory footprint. The component may support vendor-supplied extension indexes not found in the `OMX_INDEXTYPE` enumeration. This is a blocking call. The component should return from this call within five milliseconds.

The parameters for the `OMX_GetExtensionIndex` method are defined as follows.

Parameter	Description
hComponent [in]	The handle of the component to be accessed. This component handle is returned by the call to the `GetHandle` function.
cParameterName [in]	The string that the component will translate into a 32-bit index. `OMX_STRING` shall be less than 128 characters long including the trailing null byte.
pIndexType [out]	A pointer to `OMX_INDEXTYPE` that receives the index value.

5.1.1.1 Prerequisites for This Method

This macro has no prerequisites.

5.1.1.2 Method Implementation

The following code defines the method implementation.

```
    OMX_ERRORTYPE (*GetExtensionIndex)(
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_STRING cParameterName,
            OMX_OUT OMX_INDEXTYPE* pIndexType);
```

5.1.2 Custom Data Structures

Each index refers to a structure or a memory area that stores the data for the parameter or configuration. The vendor shall provide a data container that is a vendor-specific structure within a vendor-specific header file. Khronos shall provide a data container that

is an OpenMAX IL extensions structure within one of the OpenMAX IL extension header files. The header file is to be included by the component that implements the extension feature, and by the IL client that uses the extension feature.

If the data container is simply a pointer to a memory area, the IL client shall know how to manage the data. Each extension parameter shall be described in the component description document and follows the convention of standard OpenMAX IL data structures.

Each vendor-specific feature shall be documented in the component specifications, which describe the relationship between the string that defines a property, which is used with the `GetExtensionIndex` function, and the related data structure that corresponds to the index returned from `GetExtensionIndex` for the string.

5.1.3 Enumerations

OpenMAX IL enumeration types, as specified in the standard OpenMAX IL header files, may be extended using anonymous enum declarations in the OpenMAX IL extension or vendor-specific header files.

Each OpenMAX IL extension enumeration has a value greater than `OMX_<enum>KhronosExtensions` and smaller than `OMX_<enum>VendorStartUnused – 1`. Each Vendor specific extension enumeration has a value greater than `OMX_<enum>VendorStartUnused` and smaller than `OMX_<enum>Max`.

It may be necessary to cast the anonymous enum values to the standard OpenMAX IL enumeration types explicitly to avoid compilation errors.

5.1.4 Promoting extensions to specification

Extensions may be promoted to the OpenMAX-IL specification in subsequent releases of the OpenMAX-IL interface.

After promotion, the standard OpenMAX-IL header shall include a new standard enumeration value, as well as the extended enumeration value that remains in the OpenMAX IL extension file. It may be that both enumeration values point to the same feature.

5.2 Examples of Using Extension Querying API

This section shows sample code for extension APIs.

5.2.1 Sample Code Showing Calling Sequence

This following code sample shows how to use a vendor-specific parameter.

```
   /* Get the vendor-specific mp3 faster
      decoding feature settings */
   OMX_U32 eIndexParamFasterDecomp;
```

```
OMX_CUSTOM_AUDIO_STRUCTURE oFasterDecompParams;

InitializeAudioStructure(&oFasterDecompParams);

OMX_GetExtensionIndex(hMp3DecoderComp,
      "OMX.CompanyXYZ.index.param.fasterdecomp",
      &eIndexParamFasterDecomp);
OMX_GetParameter(hMp3DecoderComp, eIndexParamFasterDecomp,
      &oFasterDecompParams);
```

In this example, a special parameter of an MP3 decoder is presented. The index `eIndexParamFasterDecomp` is retrieved, and the related data structure is stored in the `oFasterDecompParams` structure by the `GetParameter` function.

6 Synchronization

This section specifies synchronization functionality including seeking and clock component behavior.

6.1 Seeking Component

A component may be designated as a *seeking component* if it can change and report on its position in the data stream that it is processing. For instance, an IL client may command a seeking source component that retrieves an audio/video stream from a repository (for example, a local or remote file) to begin emitting data from a different location in the audio/video stream. Furthermore, an IL client may query the position that the source is currently emitting.

6.1.1 Seeking Configurations

A seeking component shall support the following configurations:

- `OMX_IndexConfigTimePosition`, which passes `OMX_TIME_CONFIG_TIMESTAMPTYPE` as a parameter. `OMX_GetConfig` returns the timestamp of the data that the component is currently emitting. `OMX_SetConfig` commands the component to seek the given timestamp.
- `OMX_IndexConfigTimeSeekMode`, which defines the manner in which the seek component performs the seek. Table 6-1 shows the seek modes.

Table 6-1: Seek Modes

Seek Mode	Interpretation
`OMX_TIME_SeekModeFast`	Prefers seeking an approximation of the requested seek position over the actual seek position if it results in a faster seek.
`OMX_TIME_SeekModeAccurate`	Prefers seeking to the requested seek position over an approximation of the requested seek position even if it results in a slower seek.

An arbitrary seek in a stream may request a target position whose data depends on data that precedes it. For example, consider the case where an IL client requests seeking an interframe in a video stream. Some amount of data prior to the target interframe shall be decoded to reconstruct the target frame starting with the first intraframe preceding the target. If fast mode is set, the seeking component may use the intraframe as an approximation of the target and start displaying frames immediately at that intraframe. If accurate mode is set, the seeking component decodes frames starting with the intraframe but does not display frames until the target position.

6.1.2 Seeking Buffer Flags

A seeking component communicates the role of certain buffers in the context of seeking to its downstream components via special buffer flags. A buffer flag corresponds to the first new logical data unit in a buffer, which is the first unit with its starting boundary occurring in the buffer.

The special buffer flags of note are as follows.

- `OMX_BUFFERFLAG_DECODEONLY`: The seeking component sets this flag on a buffer if the buffer shall be decoded but not displayed. In the example above, if the seeking component is in accurate mode, it would set this flag on all frames preceding the target interframe. A decoder component decodes but does not propagate downstream a buffer marked decode only. A component that renders data shall ignore any buffer with this flag set.
- `OMX_BUFFERFLAG_STARTTIME`: The seeking component sets this flag on the buffer that carries the starting timestamp of the data stream. In the example above, the seeking component would set this flag on the intraframe (i.e., the approximation) when in fast seek mode and on the interframe (i.e., the original target) when in accurate seek mode. When a clock component client receives a buffer with this flag set, it performs an `OMX_SetConfig` call with `OMX_IndexConfigTimeClientStartTime` on the clock component that is sending the buffer's timestamp. The transmission of the start time informs the clock component that the client's stream is ready for presentation and the timestamp of the first data to be presented.

6.1.3 Seek Event Sequence

To implement a seek on a chain of components, an IL client shall perform the following operations in order:

1. Pause the component through the use of `OMX_SendCommand` requesting a state transition to `OMX_StatePause`.
2. Stop the clock component's media clock through the use of `OMX_SetConfig` on `OMX_TIME_CONFIG_CLOCKSTATETYPE` requesting a transition to `OMX_TIME_ClockStateStopped`.
3. Seek to the desired location through the use of `OMX_SetConfig` on `OMX_IndexConfigTimePosition` requesting the desired timestamp.
4. Flush all components.
5. Start the clock component's media clock through the use of `OMX_SetConfig` on `OMX_TIME_CONFIG_CLOCKSTATETYPE` requesting a transition to either `OMX_TIME_ClockStateRunning` or `OMX_TIME_ClockStateWaitingForStartTime`.
6. Un-pause the component through the use of `OMX_SendCommand` requesting a state transition to `OMX_StateExecuting`.

If the IL client requests a transition to `OMX_TIME_ClockStateRunning`, the clock component immediately starts the media clock using the designated start time. This is a simpler transition than going to `OMX_TIME_ClockStateWaitingForStartTime` but may compromise synchronization at the start of playback after a seek operation since it ignores the start times of the individual media streams.

If the IL client requests a transition to `OMX_TIME_ClockStateWaitingForStartTime`, it designates which clock component clients to wait for. The clock component then waits for these clients to send their start times via the `OMX_IndexConfigTimeClientStartTime` configuration. Once all required clients have responded, the clock component starts the media clock using the earliest client start time. This approach ensures the following:

- All clients are ready to render data, eliminating any initial drift between streams.
- The media clock start time reflects the clocks of all clients and any adjustment made by the seeking component.

6.2 Clock Component

OpenMAX IL defines a special component denoted the *clock component* to facilitate the smooth and synchronized delivery or capture of audio and video streams as well as rate control. The clock component takes a reference clock as input, from which it derives a media clock. The clock component shares the media time with the clients with whom it is connected via clock ports (one clock port per client). The clock component also exposes a mechanism for controlling the media clock and makes clients aware of the rate control events via their clock ports.

6.2.1 Timestamps

All timestamps and durations are expressed as `OMX_TICKS` values as shown in the following structure.

```
typedef struct OMX_TICKS
      {
         OMX_U32 nLowPart;
         OMX_U32 nHighPart;
} OMX_TICKS;
```

This structure shall be interpreted as a signed 64-bit value representing microseconds. This representation accommodates the following:

- Positive and negative time values. Examples of negative time values include pre-roll timestamp and time deltas.
- High-resolution timestamps (e.g., MPEG2 presentation timestamps based on a 90 kHz clock) and allow more accurate and synchronized delivery (e.g., individual audio samples delivered at 192 kHz).
- A large dynamic range of approximately plus or minus 26 million days; 32-bit resolution provides a range of only about plus or minus 35 minutes.

Implementations with limited precision may convert the signed 64-bit value to a signed 32-bit value internally but risk loss of precision.

6.2.2 Media Clock

The clock component maintains a media clock that tracks the current position in the media stream. The instantaneous media time is represented as the timestamp, relative to the start of the stream, of the data being delivered or captured at that instant (e.g., the current audio sample). Consequently, media time increases (corresponding to playing or fast forwarding), decreases (corresponding to rewinding), or holds at some constant (corresponding to pausing) according to the rate control applied to the media clock.

The clock component can be queried for the current media clock time using `OMX_GetConfig` with the read-only index `OMX_IndexConfigTimeCurrentMediaTime` and structure `OMX_TIME_CONFIG_TIMESTAMPTYPE`. The current media clock time is written into the nTimestamp field. This index must be used with the nPortIndex field as `OMX_ALL`, since the media clock is not specific to any port.

6.2.2.1 Media Clock Scale

The clock component maintains the media time's current scale factor, which corresponds directly to the rate control applied on it. The scale is a Q16 value relative to a 1X forward advancement of the media clock. Thus, scale ranges map to modes of playback, as shown in Figure 6-1.

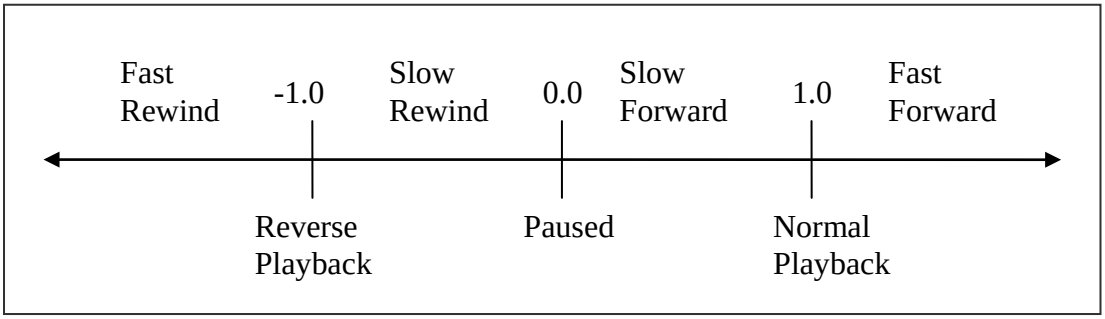


Figure 6-1. Mapping Time Scale Factors to Trick Modes

The IL client queries and sets the media clock's scale via the `OMX_IndexConfigTimeScale` configuration, passing the following structure:

```
typedef struct OMX_TIME_CONFIG_SCALETYPE {
OMX_U32 nSize;
            OMX_VERSIONTYPE nVersion;
            OMX_S32 xScale;
} OMX_TIME_CONFIG_SCALETYPE;
```

The clock component's client components are notified of changes in scale via their clock ports (see Clock Ports section for details).

6.2.2.2 Client Start Time

When a client is sent a start time (i.e., the timestamp of a buffer marked with the OMX_BUFFERFLAG_STARTTIME flag), it sends the start time to the clock component via OMX_SetConfig on OMX_IndexConfigTimeClientStartTime. This action communicates to the clock component the following information about the client's data stream:

- The stream is ready.
- The starting timestamp of the stream, either at startup or after a seek.

The clock component maintains a start time for every client component via a set of OMX_TIME_CONFIG_TIMESTAMPTYPE structures. When transitioned to OMX_TIME_ClockStateWaitingForStartTime, the clock component waits on all start times prescribed by the transition. This ensures proper synchronization at the beginning of playback.

6.2.2.3 Media Clock State

The following structure represents the state of the clock component's media clock:

```
typedef struct OMX_TIME_CONFIG_CLOCKSTATETYPE {
      OMX_U32 nSize;
      OMX_VERSIONTYPE nVersion;
      OMX_TIME_CLOCKSTATE eState;
      OMX_TICKS nStartTime;
      OMX_TICKS nOffset;
      OMX_U32 nWaitMask;
} OMX_TIME_CONFIG_CLOCKSTATETYPE;
```

The nStartTime field specifies the media time when the clock was started or will be started.

The nWaitMask field is a bit mask specifying the client components that the clock component will wait on in the OMX_TIME_ClockStateWaitingForStartTime state. Bit masks are defined as OMX_CLOCKPORT0 through OMX_CLOCKPORT7.

The nOffset field specifies the time by which to offset the media time. The clock component factors this value into the calculation of media time, effectively adding the offset to the media time reported to its clients. For example, a nOffset value of –x implies a pre-roll of duration x.

The eState field contains one of the possible clock state values shown in Table 6-2:

Table 6-2: Clock State Values

OMX_TIME_CLOCKSTATE Value	Interpretation
OMX_TIME_ClockStateRunning	The media clock is running.
OMX_TIME_ClockStateWaitingForStartTime	The media clock is waiting to run until all designated clients emit their start time.
OMX_TIME_ClockStateStopped	The media clock is stopped.

An OMX_GetConfig execution using index OMX_IndexConfigTimeClockState and structure OMX_TIME_CONFIG_CLOCKSTATETYPE queries the current clock state.

An OMX_SetConfig execution using index OMX_IndexConfigTimeClockState and structure OMX_TIME_CONFIG_CLOCKSTATETYPE commands the clock component to transition to the given state, effectively providing the IL client a mechanism for starting and stopping the media clock. Figure 6-2 shows the clock state transitions.

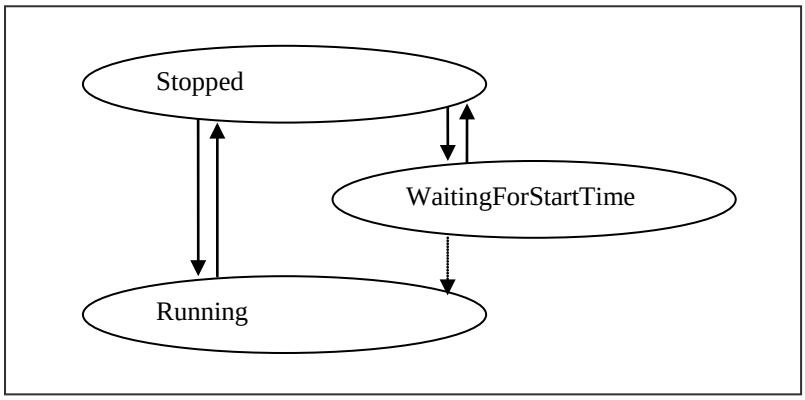


Figure 6-2. Clock State Transitions

Upon receiving OMX_SetConfig from the IL client that requests a transition to the given state, the clock component will do the following:

- OMX_TIME_ClockStateStopped: Immediately stop the media clock, clear all pending media time requests, clear and all client start times, and transition to the stopped state. This transition is valid from all other states.
- OMX_TIME_ClockStateRunning: Immediately start the media clock using the given start time and offset, and transition to the running state. This transition is valid from all other states.
- OMX_TIME_ClockStateWaitingForStartTime: Transition immediately to the waiting state, wait for all clients specified in nWaitMask to report their start time, start the media clock using the minimum of all client start times and transition to OMX_TIME_ClockStateRunning. This transition is only valid from the OMX_TIME_ClockStateStopped state.

6.2.3 Wall Clock

The clock component maintains its own free running wall clock. It uses the wall clock to extrapolate media time values from the periodic updates from the reference clock. An IL client may query the current wall time via the OMX_IndexConfigTimeCurrentWallTime configuration.

6.2.4 Reference Clocks

Clock component is ignorant of the reference clock provider. It is the responsibility of the IL client to select and inform the reference clock provider of its reference provider role.

6.2.4.1 Reference clock provider

IL client sets the reference clock provider via `OMX_SetConfig` using `OMX_IndexConfigTimeActiveRefClockUpdate` configuration and the following structure:

```
typedef struct OMX_TIME_CONFIG_ACTIVEREFCLOCKUPDATETYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_BOOL bEnableRefClockUpdates;
    OMX_TICKS nRefTimeUpdateInterval;
} OMX_TIME_CONFIG_ACTIVEREFCLOCKUPDATETYPE;
```

- `bEnableRefClockUpdates` field indicates that a clock client is a reference clock provider or not. A value of `OMX_TRUE` means that the clock client is responsible for providing reference clock updates to clock. `OMX_FALSE` as value indicates the clock client to stop reporting the reference time updates to clock.
- `nRefTimeUpdateInterval` field indicates the report frequency of reference time updates to clock in microseconds. Zero value indicates the clock clients to use its default value. This field is valid only when `bEnableRefClockUpdates` is set to `OMX_TRUE`.

All Clock clients by default shall not provide any reference time updates unless instructed by the IL client. It is recommended for IL client to make sure no two clock clients report reference time updates at same time. When IL client intends to switch the reference clock provider, it is recommended to disable the current reference clock provider before enabling the new provider.

A Clock client that is not capable of being a reference clock provider shall return `OMX_ErrorUnsupportedSetting` when it is instructed. IL client can decide to dismantle the graph or continue with the graph where clock component will work on wall clock.

In general, any time audio is rendered or captured, the IL client should prefer the audio reference clock. Otherwise, the IL client should prefer the video reference.

6.2.4.2 Reference Clock updates

The clock component can accept reference updates from clock clients. Each reference clock tracks the media time at its associated component (i.e., the timestamp of the data currently being processed at that component) and provides periodic references to the clock component via `OMX_SetConfig` using `OMX_IndexConfigTimeCurrentReference` and passing the `OMX_TIME_CONFIG_TIMESTAMPTYPE` structure

When the clock component receives a reference, it updates its internally maintained media time with the reference. This action synchronizes the clock component with the client that is providing the reference clock.

6.2.4.3 Media Time Updates

A clock component sends its clients media time updates, which can be either the fulfillment of a request, or a scale or state change notification, over its clock port. The structure used is the `OMX_TIME_MEDIATIMETYPE` structure. This can be signaled using the `OMX_SetConfig` call, with the index `OMX_IndexConfigTimeUpdate`, or written into the payload of a buffer.

The first method, using `OMX_SetConfig`, is mandatory when the clock port is connected to another component using a tunnel. In this case, when the tunnel is created using `OMX_SetupTunnel`, the clock ports shall advertise `nBufferCountActual` to be zero, since no buffers are required on this port. When these ports are required to allocate buffers, then since the buffer count is zero these ports shall be automatically populated.

If the port on the clock component is not tunneled, so directly connected to the client, then the default non-zero value of `nBufferCountActual` shall be used, so that buffers are allocated on the port, and used to send `OMX_TIME_MEDATIMETYPE` structures.

```
typedef struct OMX_TIME_MEDIATIMETYPE {
      OMX_U32 nSize;
      OMX_VERSIONTYPE nVersion;
      OMX_U32 nClientPrivate;
      OMX_TIME_UPDATETYPE eUpdateType;
      OMX_TICKS nMediaTimestamp;
      OMX_TICKS nOffset;
      OMX_TICKS nWallTimeAtMediaTime;
      OMX_S32 xScale;
      OMX_TIME_CLOCKSTATE eState;
} OMX_TIME_MEDIATIMETYPE;
```

- If the `eUpdateType` field indicates this is a request fulfillment message, the `nClientPrivate` field contains the value of `pClientPrivate` from the `OMX_TIME_CONFIG_MEDIATIMEREQUESTTYPE` structure used to signal the request that this message is fulfilling. If the `eUpdateType` field indicates this is scale or state change notification, the `nClientPrivate` field will be zero.
- `eUpdateType` indicates the reason for the update and as one of the values shown in Table 6-3:

Table 6-3: Update Types

`OMX_TIME_UPDATETYPE` Value	**Interpretation**
`OMX_TIME_UpdateRequestFulfillment`	Fulfillment of a media time request.
`OMX_TIME_UpdateScaleChanged`	Notification of a scale change.
`OMX_TIME_UpdateClockStateChanged`	Notification of a clock state change.

- The `nMediaTimestamp` field specifies the target media timestamp (if this is a request fulfillment).

- The nOffset field specifies the distance in wall time between the current time and the target time (if this is a request fulfillment).
- The nWallTimeAtMediaTime field specifies the wall time corresponding to the target media timestamp (if this is a request fulfillment).
- The xScale field contains the scale of the media clock when the structure was completed.
- The eState field contains the clock state of the media clock when the structure was completed.

6.2.4.4 Media Time Request

A client requests the transmission of a particular timestamp via OMX_SetConfig on its clock port using the OMX_IndexConfigTimeMediaTimeRequest configuration. The following structure encapsulates a request:

```
typedef struct OMX_TIME_CONFIG_MEDIATIMEREQUESTTYPE {
      OMX_U32 nSize;
      OMX_VERSIONTYPE nVersion;
      OMX_U32 nPortIndex;
      OMX_PTR pClientPrivate;
      OMX_TICKS nMediaTimestamp;
      OMX_TICKS nOffset;
} OMX_TIME_CONFIG_MEDIATIMEREQUESTTYPE;
```

The client's request includes a timestamp, which is usually associated with some operation (e.g., the presentation of a frame) that the client shall execute at that time. Conceptually, the clock component fulfills the request when the media time matches the timestamp specified.

In practice, the client component may need the request fulfilled slightly earlier than the timestamp specified. In this case, the client specifies the earlier time need of the fulfillment via the nOffset field. nOffset specifies the desired difference between the wall time when the timestamp actually occurs and the wall time when the request is to be fulfilled. (The nOffset value should represent a relatively small interval, on the order of a few milliseconds.) Note that, due to the way scale modifies the progression of media time, a client cannot simply subtract the offset from the timestamp requested.

The request also includes a pointer to any private data that the client wants to associate with it (e.g., a pointer to the frame to deliver at the given timestamp).

6.2.4.5 Media Time Request Fulfillment

When fulfilling a request, the OMX_TIME_MEDIATIMETYPE structure contains the requested media time, the wall time that corresponds to that media time, and the offset in wall time between when the media time will actually occur and when the request was actually fulfilled.

Since some clock component implementations may have difficulty fulfilling the request at exactly the time specified, the fulfillment may occur slightly earlier, leading to a

fulfillment offset larger than the one requested. The clock component shall fulfill the request as close to the requested time as possible without being late. Figure 6-3 shows the timeline for the request and fulfillment of a media time update.

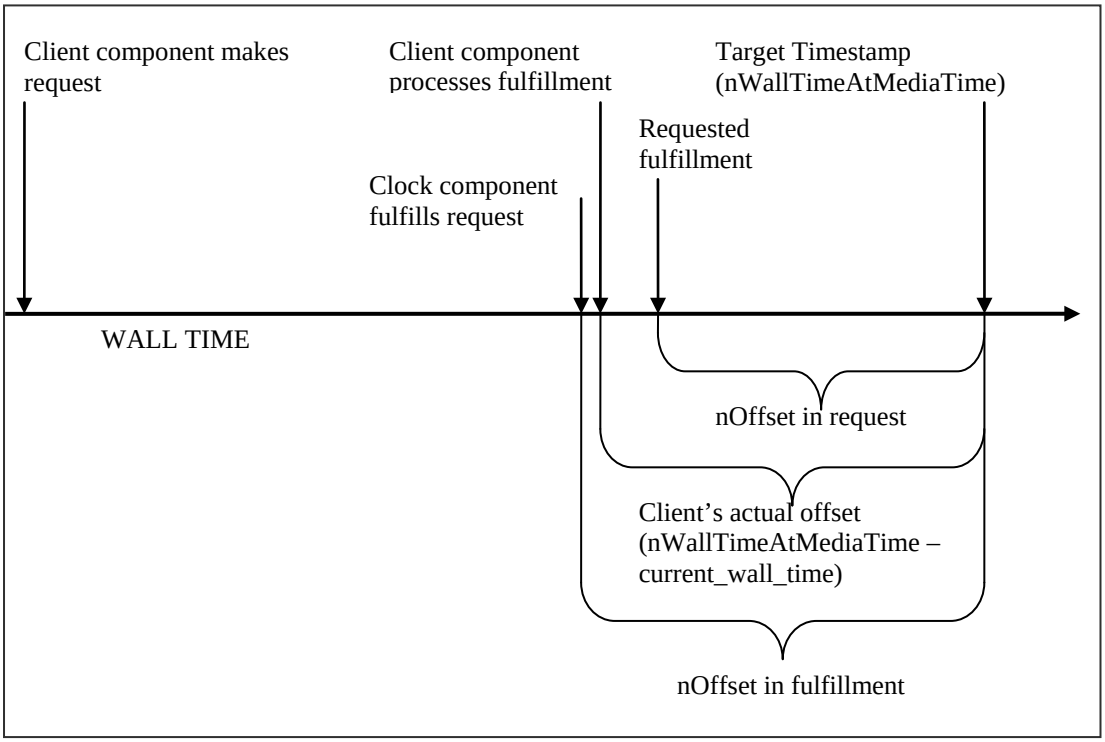


Figure 6-3. Timeline for Request and Fulfillment of Media Time Update

When a client receives the fulfillment of a request, it may time any associated operation (e.g., frame delivery) more precisely by waiting any of the remaining interval until the timestamp itself. The client may estimate the interval until the timestamp actually occurs by using `nOffset` directly, although this does not account for any delay between when the clock component fulfilled the request and when the client began processing the fulfillment. A client may obtain a more accurate estimate for this interval by taking the difference between `nWallTimeAtMediaTime` and the clock component's current wall time, which is obtained via `OMX_GetConfig` on `OMX_IndexConfigTimeCurrentWallTime`.

This interval should be small enough for the client to use its own wall clock to implement the wait. The effect of any scale change during the interval or any drift between the clock component's wall clock and the client's wall clocks should be negligible for so short a duration.

6.2.4.6 Scale Change Notifications

A `eUpdateType` value of `OMX_TIME_UpdateScaleChanged` identifies a media time update as a scale change notification.

The clock component alerts its clients to scale changes via media time updates for optimization and data correction. For instance, during fast forward, a video component

might skip intra frames and an audio component might scale and pitch correct its samples or drop them entirely. Nevertheless, components should never alter the presentation timestamp associated with a media sample. Time scaling is always applied to the media time, not the media samples.

A component that provides a reference clock shall watch for scale changes and behave accordingly. In particular, it shall:

- Cease all data delivery and its reference clock when the scale is zero (i.e., paused).
- Resume data delivery and its reference clock when the scale changes to non-zero (i.e., unpaused).

The `xScale` field contains the new scale. The `nMediaTimestamp` and `nWallTimeAtMediaTime` fields contain the media and wall time, respectively, when the scale change occurred. `nOffset` should reflect the difference, if any, between the wall time of the scale change and the wall time of the transmission of the corresponding media time update.

6.2.4.7 Clock State Change Notifications

A `eUpdateType` value of `OMX_TIME_UpdateClockStateChanged` identifies a media time update as a scale change notification.

The clock component alerts its clients to clock state transitions via media time updates so that they may take any action appropriate in that clock state. In particular:

- Any rendering component shall cease data delivery when the media clock transitions into the stopped state.
- Any client providing a reference clock shall use a media time request to time the resumption of data delivery and, hence, its reference clock when the media clock transitions into the running state

The `eState` field contains the new clock state. The `nMediaTimestamp` and `nWallTimeAtMediaTime` fields contain the media and wall time, respectively, when the clock change occurred. `nOffset` should reflect the difference, if any, between the wall time of the state change and the wall time of the transmission of the corresponding media time update.

6.2.5 Rendering Delay

Clock should have to accommodate for the rendering delay of clock clients before starting the media clock to provide a proper audio / video synchronization.

Additionally Clock shall maintain per client rendering delay to use as offset for all media time requests. Offsets in each media time requests shall be used in addition to the rendering delay of that clock client. Most use cases would not require offset per media time request.

Clock can query the rendering delay of the clients using `OMX_IndexConfigTimeRenderingDelay` configuration and the following structure:

```
typedef struct OMX_TIME_CONFIG_RENDERINGDELAYTYPE {
            OMX_U32 nSize;
            OMX_VERSIONTYPE nVersion;
            OMX_U32 nPortIndex;
            OMX_TICKS nRenderingDelay;
} OMX_TIME_CONFIG_RENDERINGDELAYTYPE;
```

- `nPortIndex` represents the port that this structure applies to.
- `nRenderingDelay` field indicates rendering delay in microseconds for the clock client on the `nPortIndex` port.

Clock can default the value to zero for clients that do not support the `OMX_IndexConfigTimeRenderingDelay` config. Clock component can query this config from clients either during the state transition to `OMX_TIME_ClockStateRunning` or `OMX_TIME_ClockStateWaitingForStartTime`.

Renderers can use this mechanism to propagate any change in rendering delay to the clock. In case of video domain, video renderer should inform the scheduler which in turn SHALL update the rendering delay to the clock. Clock on receipt of a new rendering delay shall update all outstanding requests for that client based on the new value. The rendering delay value shall be updated by `OMX_SetConfig` with the index param `OMX_IndexConfigTimeRenderingDelay`.

Other special clock client components like demuxers, etc which is not involved in rendering operation directly can either opt not to support the config or return to zero as rendering delay.

6.2.5.1 Example

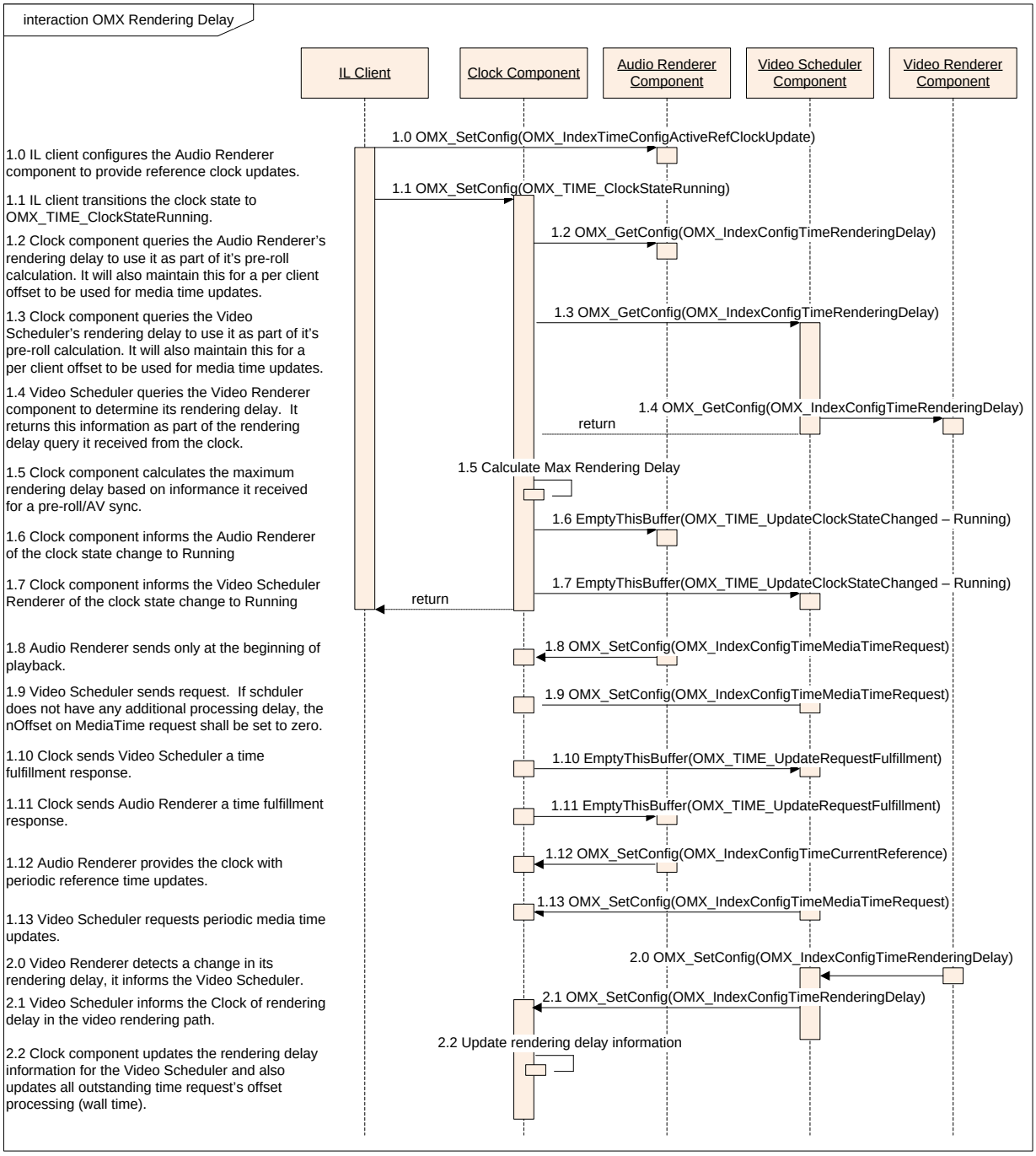


Figure 6-4: OpenMAX Rendering Delay

6.2.6 Clock Component Implementation

The clock component is responsible for implementing the semantics described in this section. Specifically the clock component should implement the following:

- Queries of its wall or media clock
- Queries of or changes to its media clock's state or scale

- Queries of or changes to its active reference clock
- Client notification of scale changes
- Fulfillment of media time requests
- Updates from the reference clocks

This following discussion describes aspects of these obligations that are not implicit in the preceding description of clock component semantics.

6.2.6.1 Deriving Media Time

The clock component derives the media time from the reference clock and the wall clock. When the reference clock sends the clock component a time reference, R_{now}, the clock component queries the wall clock for its current value, W_{now}.

The rendering delay has to be adjusted into the media time calculation. If an IL client specified an offset when it started the clock component (e.g., to implement a pre-roll), then the clock component calculates the ClockOffset in the following way

MaxRenderingDelay is the maximum of delay across all clients.

ClientOffset is the value provided by IL client

```
If( ClientOffset < 0 )
        ClockOffset = MIN(ClientOffset, -MaxRenderingDelay)
Else
        ClockOffset = ClientOffset – MaxRenderingDelay
```

With negative ClockOffset (used for preroll & Rendering delay)

When clock starts, (Note that wall clock starts running, but not media clock)

$R_{base} = 0$

$W_{base} = W_{now}$ + ClockOffset

Reference time updates

The clock component stores the ultimate reference/wall time pair, representing the base of extrapolation, for later use as $<R_{base}, W_{base}>$ where:

$R_b = R_{now}$

$W_b = W_{now}$

Media time is calculated using the following:

```
If( MediaClock started )
        Mnow = Rbase + scale * (Wnow – Wbase)
Else
```

```
// Initialized Mnow value
```

6.2.6.2 Scale Changes

Upon invocation of a scale factor, *Scale*, the clock component first establishes a new base of extrapolation by querying the current media time, M_{now}, and the current wall time, W_{now}:

$$R_{base} = M_{now}$$

$$W_{base} = W_{now}$$

The clock component then notifies all client components of the new scale via a media time update. It fills in the fields of the corresponding `OMX_TIME_MEDIATIMETYPE` structure as follows:

- `nClientPrivate` = NULL
- `nMediaTimestamp` = M_{now}
- `nWallTimeAtMediaTime` = W_{now}
- `xScale` = *Scale*

6.2.6.3 Fulfilling Media Time Requests

A clock component's approach to servicing media time requests is implementation specific. Certain operating system constructs (e.g., timers) may be useful in avoiding the expense of the spin locks associated with comparing requested times with the current media time. Nevertheless, clock component implementers should be wary of any skew between the clock component and the clock used by the operating system constructs that compromise the timely, accurate fulfillment of requests.

The clock component shall account for any offset specified by the request. Assume a requested timestamp of $M_{request,}$ an offset $Offset_{request}$, and a scale factor of *Scale*. Instead of comparing against $M_{request}$, the clock component should compare against the following:

$$M_{request} - (Offset_{request} * Scale)$$

Furthermore, the comparison between requested times and media time differ between forward playback, backward, and paused playback. Specifically, the comparisons shown in Table 6-4 should be used according to scale:

Table 6-4: Media Time Request Scale

Scale	Fulfill request when
> 0.0 (forward playback)	$M_{now} >= (M_{request} - (Offset_{request} * Scale))$
< 0.0 (backward playback)	$M_{now} <= (M_{request} - (Offset_{request} * Scale))$
0.0 (paused)	Never

6.2.7 Audio-Video File Playback Example Use Case

As an example, examine the playback of a file containing synchronized audio and video as illustrated in Figure 6-5. This example assumes that each audio or video frame has a presentation timestamp associated with it. In this construction, a file reader/de-multiplexing component feeds compressed audio and video streams to a pair of decoders. The decoders send uncompressed data to an audio renderer and video scheduler. The audio renderer delivers data to the hardware and the video scheduler will send the data to the video renderer which will send the data to the hardware.

The audio renderer and video scheduler coordinate with the clock component to implement smooth synchronized audio-video delivery. The audio renderer, video scheduler and file demuxer are clients of the clock component (connected on their respective clock ports) so they may watch for scale changes. The video scheduler also uses the clock component to time delivery of video frames via media time requests.

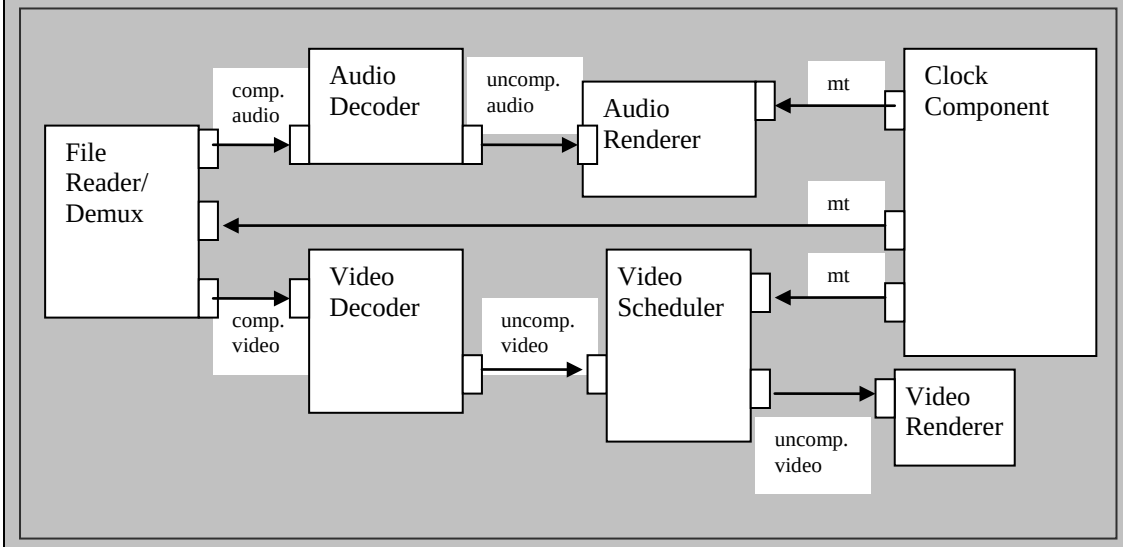


Figure 6-5. Example Use Case of Audio-Video File Playback

The audio renderer and video scheduler act as the audio and video reference clocks, each sending their reference times to the clock component as they deliver data.

In this example, the IL client uses the audio renderer as the reference clock at any time audio data is being delivered during normal playback. Thus, the IL client does not need to use the clock component to coordinate the delivery of audio data. It simply feeds new data to the audio device whenever it can, provided that the current scale allows it. When the audio device is presenting an audio buffer, the audio renderer emits the timestamp of that buffer as a reference.

The video scheduler, however, shall coordinate with the clock component when delivering video frames. For each frame that the video scheduler will deliver the frame to the video renderer at a particular timestamp, the following occurs:

1. The video scheduler submits a media time request, referencing the frame data in the private pointer and specifying fulfillment slightly earlier that the timestamp.

2. The clock component fulfills the request when it becomes current via a media time update to the video scheduler that references the original timestamp and includes the private pointer.
3. The video scheduler receives the media time update, de-references the private pointer to obtain the frame data, and delivers the frame to the video renderer. The video scheduler uses an implementation-specific mechanism to wait the remainder of the time until the timestamp before delivery (e.g., schedules a hardware flip with the video driver).

The IL client controls the clock component via specialized configurations to start and stop the media clock. To implement trick modes, the IL client sets the scale factor configuration. When the clock component applies the scale to the calculation of media time, it sends a media time update with the scale change to all of its clients.

The client components react to that scale change appropriately. When the scale is 0 (i.e., the media clock is paused), the audio renderer silences audio and ceases sending data. Furthermore, in this example, the file demuxer might elect to ignore input during non-1X playback.

If audio is effectively silenced during trick modes, the IL client may switch the active reference clock from the audio reference to the video reference.

Finally, the IL client may query the current media time from the clock component to, for instance, update the user interface such as through a progress bar.

7 Container Parsing

This section describes container parsing including access to available streams and metadata.

7.1 Parameter and Configuration Indexes

The header `OMX_Index.h` contains the enumeration `OMX_INDEXTYPE`, which contains all of the standard index values used with the functions `OMX_GetParameter`, `OMX_SetParameter`, `OMX_GetConfig`, and `OMX_SetConfig`. Table 7-1 describes the index values that relate to file parsing.

Table 7-1: Index Values for File Parsing

Index	Description
`OMX_IndexParamNumAvailableStreams`	Specifies the number of alternative streams available on a given output port. The corresponding structure is `OMX_PARAM_U32TYPE`.
`OMX_IndexParamActiveStream`	Specifies the active stream (among those available) on a given output port. The corresponding structure is `OMX_PARAM_U32TYPE`.
`OMX_IndexParamMetadataKeyFilter`	Specifies whether a key (or all keys) are enabled or disabled with respect to the metadata filter. An enabled key is in the filter and metadata with this key is retained for future potential querying. The corresponding structure is `OMX_PARAM_METADATAFILTERTYPE`.
`OMX_IndexConfigMetadataItemCount`	Specifies number of metadata items associated with a resource contained within a media file at a specific scope. The corresponding structure is `OMX_CONFIG_METADATAITEMCOUNTTYPE`.
`OMX_IndexConfigMetadataItem`	Specifies the contents of the metadata item indicated by the given index or key. The corresponding structure is `OMX_CONFIG_METADATAITEMTYPE`.

Index	Description
OMX_IndexConfigContainerNodeCount	Specifies the number of child nodes a given node contains. The corresponding structure is OMX_CONFIG_CONTAINERNODECOUNTTYPE.
OMX_IndexConfigCounterNodeID	Specifies the node id of specific node. The corresponding structure is OMX_CONFIG_CONTAINERNODEIDTYPE.
OMX_IndexParamMetadataFilterType	Specifies the filters to be applied for the meta data accesses

7.2 Format Detection

A particular container parser implementation supports a finite set of container formats, yet the component might not definitively determine support for a particular datastream until it attempts to parse the datastream. Therefore OpenMAX IL introduces the following mechanisms for a parser to communicate its ability or inability to recognize the format of a given datastream:

- The OMX_ErrorFormatNotDetected error. A component sends the client this error (in the form of an OMX_EventError event passed via the EventHandler callback) when it cannot parse or determine the format of the given datastream.
- The OMX_EventPortFormatDetected event. A component sends the client this event (via the EventHandler callback) when it has successfully recognized a format and determined that it can support it.

The IL client may use these mechanisms (perhaps in conjunction with autodetect ports) to determine whether a given parser is appropriate for a given datastream.

7.3 Port Streams

When parsing a datastream a component may discover multiple alternative streams suitable for emission as output on a given output port. For instance, when parsing a video stream muxed with synchronized audio, a parser component may discover the container datastream includes several alternative languages represented as different audio streams each a candidate for output out the same audio output port.

A port exposes the set of candidate streams as a “port stream”. If a port supports port streams (e.g. a parser output port), discovering the port streams is part of that port’s autodetect process. When the autodetect is completed (i.e. the component issues a OMX_EventPortSettingsChanged event) such a port is ready to service queries and writes on the following configs:

- The `OMX_IndexParamNumAvailableStreams` config. This read only parameter denotes the number of streams available on the port.
- The `OMX_IndexParamActiveStream` config. This read/write parameter denotes the currently selected stream for the port.

The port populates its settings according to the currently selected stream. An IL client may use thus use the `OMX_IndexParamActiveStream` parameter to both browse the settings associated with each available streams and to ultimately select the final stream for playback.

This may be performed by the IL client in the following way:

1. Instantiate the component and set any relevant configs/parameters (e.g. identifying the target content)
2. Set all output ports where the IL client desires stream discovery to autodetect and put the component into the `OMX_StateExecuting` state.
3. Wait until the port generates an `OMX_EventPortSettingsChanged` event. This event indicates it has parser enough data to have discovered the alternative streams.
4. Query the number of available streams for that port via `OMX_IndexParamNumAvailableStreams`. For each possible stream set that stream as active via `OMX_IndexParamActiveStream`. This will cause the port to populate its settings according to the active stream. The IL client may then discover the properties of the stream by reading the appropriate port parameters.
5. After reading the properties of each stream, the IL client may select the one it desires via `OMX_IndexParamActiveStream`.

7.4 Metadata Extraction

OpenMAX IL supports retrieving metadata items captured by a component. A metadata item is defined as a key/value pair, where both key and value are buffers formatted using specified character sets. OpenMAX IL enables an IL client to perform the following operations with regards to metadata:

- Specify an client-defined set of keys to filters which metadata items will be captured by the component
- Scope a metadata query to seek particular elements of the content, inclusive of the entire content
- Determine the number of distinct metadata items available at any given scope
- Retrieve all metadata items by iterating through all metadata items by available at any given scope by index
- Retrieve a metadata value for a specific metadata key

7.4.1.1 Key/Value Query

OpenMAX IL supports the querying of key/value pair data captured by a component that parses metadata via a set of component configs. The purpose of these configs is to enable an IL client to determine how many metadata items are present at a given scope, iterate through the items by index to retrieve the key/value data and query values for specific keys.

7.4.1.2 Node Traversal

OpenMAX IL supports the traversal of metadata nodes captured by a component that parses metadata via a set of component configs.

The purpose of these configs is to define a mechanism for obtaining a set of specifiers which can be used to uniquely scope metadata searches to atomic elements, or ‘nodes’, of data within a media file. Each node has a component-defined 'node ID' that the component can use to uniquely locate the node within the media file. Note that a node ID should be considered an opaque ID, therefore it need not have any intrinsic value or meaning; it need only be a value that the component can use to uniquely set the scope of a metadata search.

All media files contain exactly one 'root node’ whose node ID always has value OMX_ALL; this represents the 'top-level' metadata associated with the media file. The root node is the only node without a parent node. All other nodes have exactly one parent.

In general, the node traversal configs uses the term ‘node’ is used to represent a node for which one wants to know the ID value, and the term ‘parent node’ is used to represent the parent of one or more nodes for which one wants to know the ID value(s).

7.4.1.3 Key Filtering

OpenMAX IL supports the filtering of metadata captured by a component that parses metadata via the `OMX_IndexParamMetadataKeyFilter` parameter. This parameter allows the client to add or remove keys from the filter before the component begins processing the data. A component will retain all metadata associated with keys in the filter (so the IL client may query them later) and may safely ignore all keys not in the filter.

7.4.1.4 Specifying Language/Country

The concepts of Language and Country for a metadata item exist in some but not all file format metadata schemes. Where they do exist, most formats have only Language (including ID3v2), whereas others combine Language and Country together into a single, compound specifier. Only 3GPP has a standard metadata key that uses a Country specifier but no Language (in ‘locl’ metadata items).

Because of the relatively rare usage of these features, at the API level we combine Language and Country into a single compound Language-Country specifier, where Language comes first and Country is optional, as per the HTTP specification (RFC 2068). This approach accommodates all use cases; for example, “en” indicates English language

content for all countries, “en-US” indicates English language content for the US, “en-UK” indicates English language content for the UK, etc.

Individual requirements for Language and Country follow.

7.4.1.4.1 Language Codes

When accessing the value of a metadata item for which a language is specified, the client shall be given the language specifier. When creating a metadata item for which a language may be specified, or when changing its value, the client shall be able to indicate the language used in the supplied value. This is necessary because some file formats allow some metadata items to include a language specifier (this is usually limited to text, though not necessarily; for example, images and sounds can also be in a particular language). In some cases, there may be multiple, alternative versions of the same metadata item in different languages, and in these cases the language specifier allows the client application to select and present just the most appropriate version.

Public standards for Language specifiers include RFC 1766 / ISO 639.

7.4.1.4.2 Country Codes

Similar to the Language requirement: When accessing the value of a metadata item for which a Country (geographic location) is specified, the client shall be given the Country specifier. When creating a metadata item for which a Country may be specified, or when changing its value, the client shall be able to indicate the Country to which the supplied value applies.

Public standards for Country specifiers include ISO 3166.

7.5 Types and Structures

7.5.1 OMX_PARAM_U32TYPE

Parameters represented by unsigned 32 bit values (e.g. `OMX_IndexParamActiveStream`) use the `OMX_PARAM_U32TYPE` which is defined as follows:

```
typedef struct OMX_PARAM_U32TYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nU32;
} OMX_PARAM_U32TYPE;
```

7.5.2 OMX_METADATACHARSETTYPE

The `OMX_METADATACHARSETTYPE` enumeration defines the range of possible character sets (e.g. where a particular character is used to represent a metadata key).

Table 7-2: Supported Metadata Characterset Types

Value Name	Character Set Description
OMX_MetadataCharsetUnknown	Unknown character encoding
OMX_MetadataCharsetASCII	ASCII
OMX_MetadataCharsetBinary	Binary
OMX_MetadataCharsetCodePage1252	Microsoft Code Page 1252
OMX_MetadataCharsetUTF8	Unicode UTF-8
OMX_MetadataCharsetJavaConformantUTF8	Unicode UTF-8 (Java Conformant)
OMX_MetadataCharsetUTF7	Unicode UTF7
OMX_MetadataCharsetImapUTF7	Unicode UTF-7 per IETF RFC 2060
OMX_MetadataCharsetUTF16LE	Unicode UTF-16 (Little Endian)
OMX_MetadataCharsetUTF16BE	Unicode UTF-16 (Big Endian)
OMX_MetadataCharsetGB12345	GB 12345 (Chinese)
OMX_MetadataCharsetHZGB2312	HZ GB 2312 (Chinese)
OMX_MetadataCharsetGB2312	GB 2312 (Chinese)
OMX_MetadataCharsetGB18030	GB 18030 (Chinese)
OMX_MetadataCharsetGBK	GBK (CP936) (Chinese)
OMX_MetadataCharsetBig5	Big 5 (Chinese)
OMX_MetadataCharsetISO88591	ISO-8859-1 (Latin1 – West European languages)
OMX_MetadataCharsetISO88592	ISO-8859-2 (Latin2 – East European)
OMX_MetadataCharsetISO88593	ISO-8859-3 (Latin3 – South European)
OMX_MetadataCharsetISO88594	ISO-8859-4 (Latin4 – North European)
OMX_MetadataCharsetISO88595	ISO-8859-5 (Cyrillic)
OMX_MetadataCharsetISO88596	ISO-8859-6 (Arabic)
OMX_MetadataCharsetISO88597	ISO-8859-7 (Greek)
OMX_MetadataCharsetISO88598	ISO-8859-8 (Hebrew)
OMX_MetadataCharsetISO88599	ISO-8859-9 (Latin5 - Turkish)
OMX_MetadataCharsetISO885910	ISO-8859-10 (Latin6 – Nordic)
OMX_MetadataCharsetISO885913	ISO-8859-13 (Latin7 – Baltic Rim)
OMX_MetadataCharsetISO885914	ISO-8859-14 (Latin8 - Celtic)
OMX_MetadataCharsetISO885915	ISO-8859-15 (Latin9 – updates to Latin1)
OMX_MetadataCharsetShiftJIS	Shift-JIS (Japanese)
OMX_MetadataCharsetISO2022JP	ISO-2022-JP (Japanese)
OMX_MetadataCharsetISO2022JP1	ISO-2022-JP-1 (Japanese)
OMX_MetadataCharsetISOEUCJP	ISO EUC-JP (Japanese)
OMX_MetadataCharsetSMS7Bit	SMS 7-bit

7.5.3 OMX_METADATASCOPETYPE

The OMX_METADATASCOPETYPE structure is used to identify the type of the metadata search scope that is being specified. A scope type value is used in conjunction with a scope specifier value to identify the type of said specifier.

Table 7-3: Supported Metadata ScopeTypes

Value Name	Client usage	Component action
OMX_MetadataScopeAllLevels	Search entire piece of content—scope specifier is ignored	Search entire piece of content for matching metadata.
OMX_MetadataScopeTopLevel	Limit search scope to root level—scope specifier is ignored	Search only at the content's root level for matching metadata. Root level is defined as the only container level with no logical parent.
OMX_MetadataScopePortLevel	Limit search scope to port level—scope specifier is the port index for an output port	Search for matches only among those metadata items associated with the media resource being emitted from the indicated port. If multiple streams can be emitted from the indicated port, the component will only search for matching metadata associated with the currently active stream, as determined using the port streams mechanism.
OMX_MetadataScopeNodeLevel	Limit search scope to container file node level—scope specifier is a node ID.	Search for matches only among those metadata items explicitly associated with the specified container node and exclusive of sub-nodes of the specified container node.

7.5.4 OMX_CONFIG_METADATAITEMCOUNTTYPE

The IL client uses the OMX_IndexConfigMetadataItemCount and the OMX_CONFIG_METADATAITEMCOUNTTYPE structure to query a component for the number of metadata items associated with a resource contained within a media file at a specific scope.

OMX_CONFIG_METADATAITEMCOUNTTYPE is defined as follows.

```
typedef struct OMX_CONFIG_METADATAITEMCOUNTTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_METADATASCOPETYPE eScopeMode;
    OMX_U32 nScopeSpecifier;
```

```
    OMX_U32 nMetadataItemCount;
} OMX_CONFIG_METADATAITEMCOUNTTYPE;
```

7.5.4.1 Parameter Definitions

The parameters for OMX_CONFIG_METADATAITEMCOUNTTYPE are defined as follows.

- eScopeMode defines the type of scope being specified. See Section 10—Implementing Buffer Sharing for usage.
- nScopeSpecifier is the value of the scope specifier. See Section 10—Implementing Buffer Sharing for usage.
- nMetadataItemCount is the number of metadata items found at the scope being queried.

7.5.4.2 Dependencies

The OMX_CONFIG_METADATAITEMCOUNTTYPE structure may be queried at any time as generally allowed when calling OMX_GetConfig. However, it is possible the count of metadata items at a given scope may change as the data being processed by the component changes.

7.5.4.3 Functionality

The OMX_CONFIG_METADATAITEMCOUNTTYPE structure identifies the number of metadata items in a particular scope.

7.5.4.4 OMX_METADATASEARCHMODETYPE

The OMX_METADATASEARCHMODETYPE enumeration lists the types of queries that can be performed using the OMX_CONFIG_METADATAITEMTYPE structure.

As such the search mode specifies the usage of the other fields (input and output) of this configuration structure.

Table 7-4: Supported Metadata Search Types

Value Name	Client usage	Component action
OMX_MetadataSearchValue SizeByIndex	Get metadata value size by index nMetadataItemIndex = valid index for the given scope	nValueMaxSize = number of bytes needed to hold value of the found metadata item (No actual Key or Value data are returned, only the size.)

Value Name	Client usage	Component action
OMX_MetadataSearchItemByIndex	Get metadata key and value by index nMetadataItemIndex = valid index for the given scope nValueMaxSize = size in bytes of nValue buffer. nValue = empty buffer at least nValueMaxSize bytes long (Key buffer has fixed size.)	eKeyCharset = charset of key data in nKey nKeySizeUsed = number of bytes used in nKey nKey = buffer containing key data from the found metadata item eValueCharset = charset of value data in nValue nValueSizeUsed = number of bytes used in nValue nValue = buffer containing value data from the found metadata item
OMX_MetadataSearchNextItemByKey	Get value of first, nth, or next metadata item matching a given key nMetadataItemIndex = Valid index for the given scope. To obtain the Nth occurrence of the key, set to N - 1. To obtain the first occurrence of the key, set to OMX_ALL. eKeyCharset = charset of key data in nData nKeySizeUsed = number of bytes used in nKey nKey = buffer containing the key data to match nValueMaxSize = size in bytes of allocated by client to receive value data nValue = empty buffer at least nValueSize bytes long	nMetadataItemIndex = index of matching/found metadata item eValueCharset = charset of value data in nValue nValueSizeUsed = number of bytes used in nValue nValue = buffer containing value data from the found metadata item

7.5.5 OMX_CONFIG_METADATAITEMTYPE

The IL client uses the OMX_IndexConfigMetadataItem and the OMX_CONFIG_METADATAITEMTYPE structure to query a component for one metadata item. It can be used to retrieve a metadata item either by index or by key, or to get the size of a metadata item by index.

```
typedef struct OMX_CONFIG_METADATAITEMTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_METADATASCOPETYPE eScopeMode;
    OMX_U32 nScopeSpecifier;
    OMX_U32 nMetadataItemIndex;
    OMX_METADATASEARCHMODETYPE eSearchMode;
    OMX_METADATACHARSETTYPE eKeyCharset;
    OMX_U8 nKeySizeUsed;
    OMX_U8 nKey[128];
    OMX_METADATACHARSETTYPE eValueCharset;
    OMX_U8 sLanguageCountry[128];
    OMX_U32 nValueMaxSize;
    OMX_U32 nValueSizeUsed;
    OMX_U8 nValue[1];
} OMX_CONFIG_METADATAITEMTYPE;
```

7.5.5.1 Parameter Definitions

The parameters for OMX_CONFIG_METADATAITEMTYPE are defined as follows.

- eScopeMode defines the type of scope being specified.
- nScopeSpecifier is the value of the scope specifier.
- nMetadataItemIndex is the index of the metadata item being queried.
- eSearchMode is the type of query being performed.
- eKeyCharset is the OMX_METADATACHARSETTYPE of the key data within nKey.
- nKeySizeUsed is number of bytes within nKey that are populated with key data.
- nKey is the buffer of key data.
- eValueCharset is the OMX_METADATACHARSETTYPE of the value data within nValue.
- sLanguageCountry is the combined language and country specifier.
- nValueMaxSize is the size in bytes of the nValue buffer. ***Note:*** when nValueMaxSize is an input parameter and is a value less than the size of the metadata value, an OMX_ErrorInsufficientResources error will be returned and no output parameters will be populated.
- nValueSizeUsed is the number of bytes within nValue that are populated with value data.
- nValue is the buffer of value data.

7.5.5.2 Dependencies

The OMX_CONFIG_METADATAITEMTYPE structure may be queried at any time as generally allowed when calling OMX_GetConfig. However, it can be possible that the metadata item being sought may not yet be accessible if the corresponding portion of content has not yet been processed by the component.

7.5.5.3 Functionality

The OMX_CONFIG_METADATAITEMTYPE structure identifies a particular metadata item in a particular scope. The type of query performed by OMX_GetParameter is defined by the eSearchMode field. Refer to Section 7.5.4.4 above for details.

7.5.6 OMX_PARAM_METADATAFILTERTYPE

The IL client uses the OMX_IndexParamMetadataFilterType and OMX_PARAM_METADATAFILTERTYPE parameter structure to specify the inclusion or exclusion of a particular key, or of all keys using a given character set, in a component's filter of metadata keys. An IL client leverages writes to this parameter to enable or disable a particular key or key character set, which effectively includes or excludes that key or key character set from the set of metadata retained by the component for querying later. An IL client may also leverage reads of this parameter to query the for the inclusion/exclusion of keys from this filter. Metadata items may also be optionally filtered for Language/Country code in combination with a particular key or key character set.

```
typedef struct OMX_PARAM_METADATAFILTERTYPE
{
      OMX_U32 nSize;
      OMX_VERSIONTYPE nVersion;
      OMX_BOOL bAllKeys;
      OMX_METADATACHARSETTYPE eKeyCharset;
      OMX_U32 nKeySizeUsed;
      OMX_U8   nKey [128];
      OMX_U32 nLanguageCountrySizeUsed;
      OMX_U8  nLanguageCountry[ 128 ];
      OMX_BOOL bEnabled;
} OMX_PARAM_METADATAFILTERTYPE;
```

7.5.6.1 Parameter Definitions

The parameters for OMX_PARAM_METADATAFILTERTYPE are defined as follows.

- nVersion is the version of the structure.
- nSize is the size of the structure in bytes. This value shall be specified when this structure is used as either an input to or output from a function.
- bAllKeys

If this field is false, then only the particular specified key is included in the filter, and the filter matches metadata items with the indicated language/country code (if present). None of the other fields are ignored.

If this field is true and `nKeySizeUsed` is zero and `eKeyCharset` is `MetadataCharsetUnknown`, then this structure refers to all possible keys in all possible eKeyCharsets, and matches metadata items with the indicated language/country codes (if present). The nKey field is ignored.

If this field is true and `nKeySizeUsed` is zero and `eKeyCharset` is not `MetadataCharsetUnknown`, then this structure refers to all possible keys in the specified `eKeyCharset`, and matches metadata items with the indicated language/country code (if present). The `nKey` field is ignored.

- `eKeyCharset` – If `nKeySizeUsed` in not zero, then this must be used to indicate the `OMX_METADATACHARSETTYPE` of the key data within `nKey`. If `nKeySizeUsed` is zero, then all keys with this character set will be added to the filter; the value `MetadataCharsetUnknown` will match all key character sets.
- `nKeySizeUsed` is number of bytes within `nKey` that are populated with key data. If zero, there is no key associated with this metadata filter item (just an `eKeyCharset` and/or language/country code). If this is not zero, then the `eKeyCharset` must indicate the encoding of the key data in `nKey`.
- `nKey` is the buffer of key data.
- `nLanguageCountrySizeUsed` is the number of bytes within `nLanguageCountry` that are populated with Language / Country code data. If zero, there is no Language/Country code associated with this metadata filter item (just a key).
- `nLanguageCountry` is the buffer of Language/Country code data.
- `bEnabled` if true then key is part of filter (e.g. retained for query later). If false then key is not part of filter is the buffer of key data.

7.5.6.2 Dependencies

The `OMX_PARAM_METADATAFILTERTYPE` structure may be queried at any time. The structure may be set using `OMX_SetParameter` only when the component is in the `OMX_StateLoaded` state.

7.5.6.3 Functionality

The `OMX_PARAM_METADATAFILTERTYPE` structure identifies whether a particular metadata key or language/country code (or all metadata keys) are in the metadata filter (that is, they are retained by the parser for potential querying later). An IL client may thus leverage this structure and the `OMX_IndexParamMetadataKeyFilter` parameter to set or get filter settings.

Table 7-5: Meta Data Key Access Use Cases

Use case	Function	bAllKeys	eKeyCharset nKeySizeUsed nKey, nLanguageCountrySizeUsed, nLanguageCountry	bEnabled
Add a key and/or language/country code to the filter	SetParameter	OMX_FALSE	Specifies particular key (and its encoding) being added to filter, with optional language/country code	OMX_TRUE
Add all keys to the filter (also matches language/country code, if any); if eKeyCharset is a known encoding, then only keys with that encoding are included in the filter	SetParameter	OMX_TRUE	Required: eKeyCharsetOptional: nLanguageCountrySizeUsed, nLanguageCountry. Others are not applicable/ignored	OMX_TRUE
Remove a key and/or language/country code from the filter	SetParameter	OMX_FALSE	Specifies particular key (and its encoding)being removed from filter, with optional language/country code	OMX_FALSE
Remove all keys from the filter (also matches language/country code, if any) ; if eKeyCharset is a known encoding, only keys with that encoding are included in the filter	SetParameter	OMX_TRUE	Required: eKeyCharset, Optional: nLanguageCountrySizeUsed, nLanguageCountry. Others are not applicable/ignored	OMX_FALSE
Query whether a key and/or language/country code is part of the filter	GetParameter	Not applicable/ignored	Specifies particular key (and its encoding) being queried, with optional language/country code	Output field filled in by GetParameter

7.5.6.4 Post-processing Conditions

The changes specified to the component's metadata filter (i.e. the enabling or disabling of keys) are applied upon the return of a `OMX_SetParameter` call when used with the `OMX_CONFIG_METADATAITEMTYPE` structure. The component retains only the cumulative set of keys specified as enabled in the filter.

7.5.7 OMX_CONFIG_CONTAINERNODECOUNTTYPE

The IL client uses the `OMX_IndexConfigContainerNodeCount` and the `OMX_CONFIG_CONTAINERNODECOUNTTYPE` structure to query a parent node for the number of nodes it contains.

```
typedef struct OMX_CONFIG_CONTAINERNODECOUNTTYPE
{
      OMX_U32 nSize;
      OMX_VERSIONTYPE nVersion;
      OMX_BOOL bAllKeys;
      OMX_U32 nParentNodeID;
      OMX_U32 nNumNodes;
} OMX_CONFIG_CONTAINERNODECOUNTTYPE;
```

7.5.7.1 Parameter Definitions

The parameters for `OMX_CONFIG_CONTAINERNODECOUNTTYPE` are defined as follows.

- `nParentNodeID` is the node ID for the node being queried. To specify the media file's root node, use the value OMX_ALL
- `nNumNodes` is the number of nodes contained by the indicated parent node.

7.5.7.2 Dependencies

The `OMX_CONFIG_CONTAINERNODECOUNTTYPE` structure may be queried at any time as generally allowed when calling `OMX_GetConfig`. However, it is possible that the count of nodes returned by this query may change if the component is actively processing data.

7.5.7.3 Functionality

The `OMX_CONFIG_CONTAINERNODECOUNTTYPE` structure identifies the node count on given a node ID.

7.5.8 OMX_CONFIG_CONTAINERNODEIDTYPE

The IL client uses the `OMX_IndexConfigCounterNodeID` and the `OMX_CONFIG_CONTAINERNODEIDTYPE` structure to obtain information about a specific node.

```
typedef struct OMX_CONFIG_CONTAINERNODEIDTYPE
{
      OMX_U32 nSize;
      OMX_VERSIONTYPE nVersion;
      OMX_BOOL bAllKeys;
      OMX_U32 nParentNodeID;
      OMX_U32 nNodeIndex;
      OMX_U32 nNodeID;
      OMX_U8 cNodeName[128];
      OMX_BOOL bIsLeafType;
} OMX_CONFIG_CONTAINERNODEIDTYPE;
```

7.5.8.1 Parameter Definitions

The parameters for `OMX_CONFIG_CONTAINERNODEIDTYPE` are defined as follows.

- `nParentNodeID` is the node ID for the node being queried. To specify the media file's root node, use the value OMX_ALL
- `nNodeIndex` is the index of this node.
- `nNodeID` is the node ID for this node.
- `cNodeName` name of this node.
- `bIsLeafType` indicates whether this node may be a parent to other nodes. If the component does not know whether this node is a parent or not, the component will return OMX_FALSE.

7.5.8.2 Dependencies

The `OMX_CONFIG_CONTAINERNODEIDTYPE` structure may be queried at any time as generally allowed when calling `OMX_GetConfig`. However, it is possible that if the underlying data has changed the node being sought may no longer be accessible.

7.5.8.3 Functionality

The `OMX_CONFIG_CONTAINERNODEIDTYPE` structure identifies the properties of the node which is the specified child of the specified parent node.

8 Mandatory Component Parameters

8.1 Component Role

A component implementation may support one or more roles. We define a role as the behavior of component acting according to a particular standard or vendor-specific component definition. The name of the component definition identifies the role.

For example a given component implementation named "OMX.CompanyXYZ.MyAudioDecoder" might support the following roles:

audio_decoder.mp3

audio_decoder.aac

audio_decoder.amr

When the audio_decoder.mp3 role has been set on this component implementation it obeys the definition of the audio_decoder.mp3 standard component. It shall, for example, expose the defined audio input and output ports and support the mandated configs and parameters on those ports. Upon setting the role, the component shall populate the default values on all parameters, configs and internal state, according to the role.

Via the mechanisms defined the below, the core extracts information about which roles are supported by which component implementation and, using this information, provides two convenient functions for the IL client to query about such support. Furthermore, a component implementation allows an IL client to set the role which defines its behavior.

8.1.1 ComponentRoleEnum

The `ComponentRoleEnum` component function allows the IL core to query a component for all the roles it supports. This function allows the IL core to service `OMX_ComponentOfRoleEnum` and `OMX_RoleOfComponentEnum` calls. An efficient IL core will likely cache the role information it extracts from components (e.g. at installation) to avoid instantiating a component during `OMX_ComponentOfRoleEnum` and `OMX_RoleOfComponentEnum` calls.

`ComponentRoleEnum` enumerates (one role at a time) the component roles that a component supports.

```
OMX_ERRORTYPE (*ComponentRoleEnum)(
            OMX_IN OMX_HANDLETYPE hComponent,
            OMX_OUT OMX_U8* cRole,
            OMX_IN U32 nIndex);
```

Parameters include:

- `hComponent` : The handle of the component that executes the call
- `cRole`: The name of the specified role. The role name string has a limit of 128 bytes (including '\0').

- `nIndex`: The index of the role being queried.

8.1.2 OMX_PARAM_COMPONENTROLETYPE

The `OMX_PARAM_COMPONENTROLETYPE` structure enables the IL client to set the role of the component via the OMX_IndexParamStandardComponentRoleIndex. The result of a query on this parameter is undefined.

Setting this parameter on a component shall result in the component populating all the component data structures with default values according to the specified role. This includes defaults on all mandatory component and port parameters, and any other additional component structures.

All components shall support setting this parameter with the role name "default". When this value is used, the component shall set all parameters, all configs and internal state as they are, when the component is first instantiated. The "default" role shall not be reported when the component is interrogated for supported roles.

```
typedef struct OMX_PARAM_COMPONENTROLETYPE {
   OMX_U32 nSize;
   OMX_VERSIONTYPE nVersion;
   OMX_U8 cRole[OMX_MAX_STRINGNAME_SIZE];
}OMX_PARAM_COMPONENTROLETYPE;
```

Parameters include:

- `cRole`: name of the role (i.e. name of the component definition).

  `OMX_MAX_STRINGNAME_SIZE` is defined to have a value of 128.

8.1.3 OMX_RoleOfComponentEnum

The function that enables the IL client to query all the roles fulfilled by a given a component.

```
OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_RoleOfComponentEnum (
      OMX_OUT OMX_STRING role,
      OMX_IN OMX_STRING compName,
      OMX_IN OMX_32  nIndex);
```

Parameters include:

- `role`: This is the role being returned. The caller shall provide a pointer to a valid string of at least 128 characters. If the function returns `OMX_ErrorNone` then this string is filled in with the appropriate role of the component.
- `compName`: This is the name of the component being queried about.
- `nIndex`: A number containing the enumeration index for the role of the component. Multiple calls to `OMX_RoleOfComponentEnum` with increasing values of nIndex will enumerate through the roles of the component until `OMX_ErrorNoMore` is returned.

8.1.4 OMX_ComponentOfRoleEnum

The OMX_ComponentOfRoleEnum function that enables the IL client to query the names of all installed components that support a given role.

```
OMX_ERRORTYPE OMX_ComponentOfRoleEnum (
      OMX_OUT OMX_STRING compName,
      OMX_IN OMX_STRING role,
      OMX_IN OMX_U32 nIdex);
```

Parameters include:

- compName: The name of the component being returned. The caller shall provide a pointer to a valid string of at least 128 characters. If the function returns OMX_ErrorNone then this string is filled in with the name of the appropriate component.
- role: This name of the role being queried about.
- nIndex: A number containing the enumeration index for the component implementing this role. Multiple calls to OMX_ComponentOfRoleEnum with increasing values of nIndex will enumerate through the components implementing this role until OMX_ErrorNoMore is returned.

8.2 Mandatory Port Parameters

Across all standard components, OpenMAX IL 1.2 mandates support for certain parameters. Specifically:

- All standard components shall support the following parameters:
    - OMX_IndexParamPortDefinition
    - OMX_IndexParamCompBufferSupplier
    - OMX_IndexParamAudioInit
    - OMX_IndexParamImageInit
    - OMX_IndexParamVideoInit
    - OMX_IndexParamOtherInit
    - OMX_IndexParamStandardComponentRole
- All *audio* ports on a standard component shall support the following parameters:
    - OMX_IndexParamAudioPortFormat
- All *video* ports on a standard component shall support the following parameters
    - OMX_IndexParamVideoPortFormat
- All *image* ports on a standard component shall support the following parameters:
    - OMX_IndexParamImagePortFormat
- All *other* ports on a standard component shall support the following parameters:

- OMX_IndexParamOtherPortFormat

9 Standard Components

In the interest of facilitating strict component portability, OpenMAX IL defines a set of standard components. Each standard component definition associates specific interface criteria and functionality to the named standard component. To the extent these definitions are adhered to by clients and components, this allows one IL client to operate seamlessly with component implementations from multiple vendors and allows one component to operate seamlessly across multiple IL clients.

This section defines the set of OpenMAX IL standard components including:

- The hierarchy of standard component definitions.
- The mechanism for exposing standard components to an IL client.
- The definition of all standard classes and standard components.

9.1 Hierarchy of Standard Component Definition

OpenMAX IL establishes two constructs for the hierarchical definition of the set of standard components:

- Standard component class: a category of standard components that share the same ports and high level functionality.
- Standard component: an instance of a standard component class that has the same ports and high level functionality as the class but that specifies the supported formats, parameters, and configs on those ports as well as the specific functionality of the component.

Thus OpenMAX IL divides the set of all standard components into classes of similar components, formally defining the characteristics of each class in terms of the ports it exposes and its overall function. Within each class, OpenMAX IL identifies specific standard components, formally defining the formats, parameters, and config operations supported on each port as well the specific type of functionality the individual component supports.

For instance, OpenMAX IL defines an audio_decoder class that represents all components that receive encoded audio on a single audio input port and emit decoded audio on single audio output. Furthermore, the audio_decoder class contains a standard component definition for each audio format: audio_decoder.aac, audio_decoder.amr, audio_decoder.amr, etc.

The difference in functionality between components in the previous example is the specific format of audio decoding implemented. However, the differences between components in a single class may also be distinguished in terms of their specific functionality. Each component in the audio_processing class, for example, operates on the same format (i.e. pcm audio) but implements different effects, e.g. `audio_processing.pcm.stereo_widening_loudspeakers`.

Thus, generally speaking, a component class defines a category of functionality and each component in that class implements one specific type of functionality within that category.

9.1.1 Standard Component Class Definition

The definition of a standard component class consists of:

- *Name:* The name of the standard component class.
- *Description:* Description of high level functionality.
- The set of ports exposed including the following information for each port:
    - *Index:* the index of the port.
    - *Domain:* the port's domain (audio, video, image, or other).
    - *Direction:* the ports direction (input or output).
    - *Description:* a description of the port's functionality relative to the component.

9.1.2 Standard Components Definition

The definition of a standard component consists of:

- *Name*: The name of the standard component.
- *Description:* Description of the specific functionality implemented by the component.
- For each port:
    - *Index:* The index of the described port.
    - *Description:* Description of the functionality implemented by the port relative to the component.
    - *Parameters and Configs:* A list of supported OpenMAX IL parameters and configs including the following information for each.
        - *Index:* The index value of the parameter or config used from the `OMX_INDEXTYPE` enumeration.
        - *Access:* The read/write access of the parameter/config which is a any combination of the following:
            - *Read*: IL client is querying a component value via `GetParameter` or `GetConfig`. The component will fill in the appropriate fields of the structure passed.
            - *Write*: IL client is setting a component value via `SetParameter` or `SetConfig`. The IL client will fill in the appropriate fields of the structure passed.

- *Description:* Description of the parameter or config's function relative to the port.

9.2 Notation Used

The standard component definitions use certain conventions in their notation. Specifically:

- "APB" denotes the audio port base which is defined to be the `nStartPortNumber` value returned on a query of the `OMX_IndexParamAudioInit` param.
- "IPB" denotes the image audio port base which is defined to be the `nStartPortNumber` value returned on a query of the `OMX_IndexParamImageInit` param.
- "VPB" denotes the video port base which is defined to be the `nStartPortNumber` value returned on a query of the `OMX_IndexParamVideoInit` param.
- "OPB" denotes the other port base which is defined to be the `nStartPortNumber` value returned on a query of the `OMX_IndexParamOtherInit` param.

Furthermore, when a field of a parameter or config is specified all the listed values in the 'Description' column shall be supported and the *italicized* value shall be the default. A component that supports multiple standard component roles shall populate its fields with default settings according to the current role.

All parameter and config settings specified indicate the minimum settings that the components shall support to be categorized as a standard components.

9.3 Video and Image Order of Operations

As part of the Video and Image domain, features have been defined that will apply data transform operations to data payloads. These data transforms consist of cropping, rotation, mirroring and scaling.

Depending on the ordering of the transforms applied to the data payload varying results will be produced. In order for the IL client to deterministically achieve a desired output among standard components that support such operations, the order of the these transforms applied to the data payload on a per port basis shall be as follows:

1. Cropping
2. Rotation
3. Mirroring
4. Scaling

This order is to be applied by components that support all or a subset of transforms.

For example:

- If a port within standard component A supports all four transforms then the order will be cropping followed by rotation followed by mirroring followed by scaling
- If a port within standard component B supports just three of the transforms – cropping, rotation and scaling – then the order will be cropping followed by rotation followed by scaling

Implementations of standard components supporting these transforms are not required to internally implement these transforms as outlined, rather the standard component implementations need to apply the operations to the payload in the logical order outlined such that a deterministic output is achieved.

This ordering of operations provides consistency for the IL client between different standard component implementations. It does not dictate the implementation of those components.

9.4 Standard Audio Components

9.4.1 Audio Decoder Class

Name	audio_decoder			
Description	Decodes the given compressed audio stream into an uncompressed audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts encoded audio.
	APB+1	audio	output	Emits decoded audio.

Sample rate conversions, downmix and upmix support are not mandated. If these features are not supported, the component shall implement the slaving behavior as described in section (ref to slaving behavior section).

9.4.1.1 AAC Decoder Component

Name	audio_decoder.aac			
Description	Decodes the given compressed audio stream into an uncompressed audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts encoded audio.
	APB+1	audio	output	Emits decoded audio.

Port Index	APB+0		
Description	Accepts encoded audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingAAC
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingAAC

Port Index	APB+0		
	OMX_IndexParamAudioAac	r/w	nChannels = *2 (stereo)* 1 (mono) nSampleRate = 8000 11025 12000 16000 22050 24000 32000 44100 *48000* nBitRate = up to 288Kbps per channel eAACProfile = *OMX_AUDIO_AACObjectLC* OMX_AUDIO_AACObjectHE OMX_AUDIO_AACObjectHE_PS eAACStreamFormat = *OMX_AUDIO_AACStreamFormatMP2ADTS* OMX_AUDIO_AACStreamFormatMP4ADTS OMX_AUDIO_AACStreamFormatADIF OMX_AUDIO_AACStreamFormatRAW (headerless) eChannelMode = *OMX_AUDIO_ChannelModeStereo* OMX_AUDIO_ChannelModeMono

Port Index	APB+1		
Description	Emits decoded audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamPortDefinition	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingPCM
	OMX_IndexParamAudioPortFormat	r/w	eEncoding = OMX_AUDIO_CodingPCM

Port Index	APB+1		
	OMX_IndexParamAudioPcm	r/w	nChannels = *2 (stereo)* 1 (mono) eNumData = *OMX_NumericalDataSigned* nSampleRate = 8000 11025 12000 16000 22050 24000 32000 44100 *48000* ePCMMode = *OMX_AUDIO_PCMModeLinear* nBitPerSample = *16*

9.4.1.2 AMR-NB Decoder Component

Name	audio_decoder.amrnb			
Description	Decodes the given compressed audio stream into an uncompressed audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts encoded audio.
	APB+1	audio	output	Emits decoded audio.

Port Index	APB+0		
Description	Accepts encoded audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamPortDefinition	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingAMR
	OMX_IndexParamAudioPortFormat	r/w	eEncoding = OMX_AUDIO_CodingAMR

Port Index	APB+0		
	`OMX_IndexParamAudioAmr`	r/w	nChannels = *1* nBitRate = 4750 5150 5900 6700 7400 7950 10200 12200 OMX_AUDIO_PARAM_AMRTYPE:: OMX_AUDIO_AMRBANDMODETYPE = OMX_AUDIO_AMRBandModeNB0 OMX_AUDIO_AMRBandModeNB1 OMX_AUDIO_AMRBandModeNB2 OMX_AUDIO_AMRBandModeNB3 OMX_AUDIO_AMRBandModeNB4 OMX_AUDIO_AMRBandModeNB5 OMX_AUDIO_AMRBandModeNB6 OMX_AUDIO_AMRBandModeNB7 eAMRDTXMode = OMX_AUDIO_AMRDTXModeOnAuto eAMRFrameFormat = OMX_AUDIO_AMRFrameFormatIF2 OMX_AUDIO_AMRFrameFormatFSF

Port Index	APB+1		
Description	Emits decoded audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingPCM

Port Index	APB+1		
	OMX_IndexParamAudioPcm	r/w	nChannels = *1 (mono)* eNumData = *OMX_NumericalDataSigned* nSampleRate = 8000 ePCMMode = *OMX_AUDIO_PCMModeLinear* nBitPerSample = *16*

9.4.1.3 AMR-WB Decoder Component

Name	audio_decoder.amrwb			
Description	Decodes the given compressed audio stream into an uncompressed audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts encoded audio.
	APB+1	audio	output	Emits decoded audio.

Port Index	APB+0		
Description	Accepts encoded audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamPortDefinition	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingAMR
	OMX_IndexParamAudioPortFormat	r/w	eEncoding = OMX_AUDIO_CodingAMR

Port Index	APB+0		
	`OMX_IndexParamAudioAmr`	r/w	nChannels = *1* nBitRate = 6600 8850 12650 14250 15850 18250 19850 23050 23850 OMX_AUDIO_PARAM_AMRTYPE:: OMX_AUDIO_AMRBANDMODETYPE = OMX_AUDIO_AMRBandModeWB0 OMX_AUDIO_AMRBandModeWB1 OMX_AUDIO_AMRBandModeWB2 OMX_AUDIO_AMRBandModeWB3 OMX_AUDIO_AMRBandModeWB4 OMX_AUDIO_AMRBandModeWB5 OMX_AUDIO_AMRBandModeWB6 OMX_AUDIO_AMRBandModeWB7 OMX_AUDIO_AMRBandModeWB8 eAMRDTXMode = OMX_AUDIO_AMRDTXModeOnAuto eAMRFrameFormat = OMX_AUDIO_AMRFrameFormatIF2 OMX_AUDIO_AMRFrameFormatFSF

Port Index	APB+1		
Description	Emits decoded audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingPCM

Port Index	APB+1		
	OMX_IndexParamAudioPcm	r/w	nChannels = *1 (mono)* eNumData = *OMX_NumericalDataSigned* nSampleRate = 16000 ePCMMode = *OMX_AUDIO_PCMModeLinear* nBitPerSample = *16*

9.4.1.4 AMR-WB+ Decoder Component

Name	audio_decoder.amrwb+			
Description	Decodes the given compressed audio stream into an uncompressed audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	Input	Accepts encoded audio.
	APB+1	audio	output	Emits decoded audio.

Port Index	APB+0		
Description	Accepts encoded audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamPortDefinition	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingAMR
	OMX_IndexParamAudioPortFormat	r/w	eEncoding = OMX_AUDIO_CodingAMR

Port Index	APB+0		
	OMX_IndexParamAudioAmr	r/w	nChannels = *2 or 1* nBitRate = between 5200 bps and 48000 bps OMX_AUDIO_PARAM_AMRTYPE::OMX_AUDIO_AMRBANDMODETYPE = Between OMX_AUDIO_AMRBandModeWBP0 and OMX_AUDIO_AMRBandModeWBP47 eAMRDTXMode = OMX_AUDIO_AMRDTXModeOnAuto eAMRFrameFormat = OMX_AUDIO_AMRFrameFormatWBPlusTIF OMX_AUDIO_AMRFrameFormatWBPlusFSF eAMRISFIndex = between (OMX_AUDIO_AMRISFIndex0 and OMX_AUDIO_AMRISFIndex13) or OMX_AUDIO_AMRISFIndexUnknown nSampleRate = 48000 44100 32000 24000 22050 16000 11025 8000

Port Index	APB+1		
Description	Emits decoded audio.		
Required	**Index**	**Access**	**Description**

Port Index	APB+1		
Parameters/ Configs	`OMX_IndexParamPortDefinition`	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPcm`	r/w	nChannels = *2 (stereo)* *1 (mono)* eNumData = *OMX_NumericalDataSigned* nSampleRate = 48000 44100 32000 24000 22050 16000 11025 8000 ePCMMode = *OMX_AUDIO_PCMModeLinear* nBitPerSample = *16*

9.4.1.5 MP3 Decoder Component

Name	audio_decoder.mp3			
Description	Decodes the given compressed audio stream into an uncompressed audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts encoded audio.
	APB+1	audio	output	Emits decoded audio.

Port Index	APB+0		
Description	Accepts encoded audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingMP3
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingMP3

Port Index	APB+0		
	`OMX_IndexParamAudioMp3`	r/w	nChannels = *2 (stereo)* 1 (mono) nSampleRate = 32000 44100 *48000* nBitRate = 80000 to 320000 eChannelMode = *OMX_AUDIO_ChannelModeStereo* OMX_AUDIO_ChannelModeJointStereo OMX_AUDIO_ChannelModeDual OMX_AUDIO_ChannelModeMono eFormat = *OMX_AUDIO_MP3StreamFormatMP1Layer3*

Port Index	APB+1		
Description	Emits decoded audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingPCM

Port Index	APB+1		
	OMX_IndexParamAudioPcm	r/w	nChannels = *2 (stereo)* 1 (mono) eNumData = *OMX_NumericalDataSigned* nSampleRate = 32000 44100 *48000* ePCMMode = *OMX_AUDIO_PCMModeLinear* nBitPerSample = *16*

9.4.1.6 Real Audio Decoder Component

Name	audio_decoder.ra			
Description	Decodes the given compressed audio stream into an uncompressed audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts encoded audio.
	APB+1	audio	output	Emits decoded audio.

Port Index	APB+0		
Description	Accepts encoded audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamPortDefinition	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingRA
	OMX_IndexParamAudioPortFormat	r/w	eEncoding = OMX_AUDIO_CodingRA

Port Index	APB+0		
	OMX_IndexParamAudioRa	r/w	nChannels = *2 (stereo)* 1 (mono) nBitRate = 8000 to 96000 bps nSamplingRate = 8000, 11025, 22050 *44100* nSample PerFrame = 256, 512, 1024 eFormat = *OMX_AUDIO_RA10_CODEC*

Port Index	APB+1		
Description	Emits decoded audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamPortDefinition	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingPCM
	OMX_IndexParamAudioPortFormat	r/w	eEncoding = OMX_AUDIO_CodingPCM
	OMX_IndexParamAudioPcm	r/w	nChannels = *2 (stereo)* 1 (mono) eNumData = *OMX_NumericalDataSigned* nSampleRate = 8000 11025 22050 *44100* ePCMMode = *OMX_AUDIO_PCMModeLinear* nBitPerSample = *16*

9.4.1.7 WMA Decoder Component

Name	audio_decoder.wma

Name	audio_decoder.wma			
Description	Decodes the given compressed audio stream into an uncompressed audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts encoded audio.
	APB+1	audio	output	Emits decoded audio.

Port Index	APB+0		
Description	Accepts encoded audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingWMA
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingWMA
	`OMX_IndexParamAudioWma`	r/w	nChannels = *2 (stereo)* 1 (mono) nBitRate = 5000 to 385000 bps eFormat = *OMX_AUDIO_WMAFormat9* OMX_AUDIO_WMAFormat8 OMX_AUDIO_WMAFormat7 nSamplingRate = 8000 11025 12000 16000 22050 24000 32000 44100 *48000*

Port Index	APB+1		
Description	Emits decoded audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingPCM

Port Index	APB+1		
	OMX_IndexParamAudioPcm	r/w	nChannels = *2 (stereo)* 1 (mono) eNumData = *OMX_NumericalDataSigned* nSampleRate = 8000 11025 12000 16000 22050 24000 32000 44100 *48000* ePCMMode = *OMX_AUDIO_PCMModeLinear* nBitPerSample = *16*

9.4.2 Audio Encoder Class

Name	audio_encoder			
Description	Encodes the giiven audio stream into a compressed audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts audio for encoding.
	APB+1	audio	output	Emits encoded audio.

Sample rate conversions, downmix and upmix support are not mandated. If these features are not supported, the component shall implement the slaving behavior as described in section (ref to slaving behavior section).

9.4.2.1 AAC Encoder Component

Name	audio_encoder.aac			
Description	Encodes the giiven audio stream into a compressed audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts audio for encoding.
	APB+1	audio	output	Emits encoded audio.

Port Index	APB+0		
Description	Accepts audio for encoding.		
Required	**Index**	**Access**	**Description**

Port Index	APB+0		
Parameters/ Configs	`OMX_IndexParamPortDefinition`	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPcm`	r/w	Specify/query the sampling rate and number of channels. nChannels = *2 (Stereo)* 1 (Mono) eNumData = *OMX_NumericalDataSigned* bInterleaved = *OMX_TRUE* nBitPerSample = *16* nSamplingRate = 8000 11025 12000 16000 22050 24000 32000 44100 *48000* ePCMMode = *OMX_AUDIO_PCMModeLinear*

Port Index	APB+1		
Description	Emits encoded audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingAAC
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingAAC

Port Index	APB+1		
	OMX_IndexParamAudioAac	r/w	nChannels = *2 (stereo)* 1 (mono) nSampleRate = 8000 11025 12000 16000 22050 24000 32000 44100 *48000* nBitRate = at least 288Kbps per channel nAudioBandWidth = *0* nFrameLength = *0* eAACProfile = *OMX_AUDIO_AACObjectLC* OMX_AUDIO_AACObjectHE OMX_AUDIO_AACObjectHE_PS eAACStreamFormat = *OMX_AUDIO_AACStreamFormatMP2ADTS* OMX_AUDIO_AACStreamFormatMP4ADTS OMX_AUDIO_AACStreamFormatADIF OMX_AUDIO_AACStreamFormatRAW (headerless) eChannelMode = *OMX_AUDIO_ChannelModeStereo* OMX_AUDIO_ChannelModeMono

9.4.2.2 AMR-NB Encoder Component

Name	audio_encoder.amrnb			
Description	Encodes the giiven audio stream into a compressed audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts audio for encoding.
	APB+1	audio	output	Emits encoded audio.

Port Index	APB+0		
Description	Accepts audio for encoding.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPcm`	r/w	Specify/query the sampling rate and number of channels. nChannels = 1 (Mono) eNumData = *OMX_NumericalDataSigned* bInterleaved = *OMX_TRUE* nBitPerSample = *16* nSamplingRate = 8000 ePCMMode = *OMX_AUDIO_PCMModeLinear*

Port Index	APB+1		
Description	Emits encoded audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingAMR
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingAMR

Port Index	APB+1		
	`OMX_IndexParamAudioAmr`	r/w	nChannels = *1* nBitRate = 4750 5150 5900 6700 7400 7950 10200 12200 OMX_AUDIO_PARAM_AMRTYPE:: OMX_AUDIO_AMRBANDMODETYPE = OMX_AUDIO_AMRBandModeNB0 OMX_AUDIO_AMRBandModeNB1 OMX_AUDIO_AMRBandModeNB2 OMX_AUDIO_AMRBandModeNB3 OMX_AUDIO_AMRBandModeNB4 OMX_AUDIO_AMRBandModeNB5 OMX_AUDIO_AMRBandModeNB6 OMX_AUDIO_AMRBandModeNB7 eAMRDTXMode = OMX_AUDIO_AMRDTXModeOff OMX_AUDIO_AMRDTXModeOnVAD1 OMX_AUDIO_AMRDTXModeOnVAD2 eAMRFrameFormat = OMX_AUDIO_AMRFrameFormatIF2 OMX_AUDIO_AMRFrameFormatFSF

Port Index	APB+1		
	OMX_IndexConfigAudioAmrMode	r/w	nBitRate = 4750 5150 5900 6700 7400 7950 10200 12200 OMX_AUDIO_AMRMODETYPE::OMX_AUDIO_AMRBANDMODETYPE = OMX_AUDIO_AMRBandModeNB0 OMX_AUDIO_AMRBandModeNB1 OMX_AUDIO_AMRBandModeNB2 OMX_AUDIO_AMRBandModeNB3 OMX_AUDIO_AMRBandModeNB4 OMX_AUDIO_AMRBandModeNB5 OMX_AUDIO_AMRBandModeNB6 OMX_AUDIO_AMRBandModeNB7

9.4.2.3 AMR-WB Encoder Component

Name	audio_encoder.amrwb			
Description	Encodes the giiven audio stream into a compressed audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts audio for encoding.
	APB+1	audio	output	Emits encoded audio.

Port Index	APB+0		
Description	Accepts audio for encoding.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamPortDefinition	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingPCM
	OMX_IndexParamAudioPortFormat	r/w	eEncoding = OMX_AUDIO_CodingPCM

Port Index	APB+0		
	`OMX_IndexParamAudioPcm`	r/w	Specify/query the sampling rate and number of channels. nChannels = 1 (Mono) eNumData = *OMX_NumericalDataSigned* bInterleaved = *OMX_TRUE* nBitPerSample = *16* nSamplingRate = 16000 ePCMMode = *OMX_AUDIO_PCMModeLinear*

Port Index	APB+1		
Description	Emits encoded audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingAMR
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingAMR

Port Index	APB+1		
	`OMX_IndexParamAudioAmr`	r/w	nChannels = *1* nBitRate = 6600 8850 12650 14250 15850 18250 19850 23050 23850 OMX_AUDIO_PARAM_AMRTYPE:: OMX_AUDIO_AMRBANDMODETYPE = OMX_AUDIO_AMRBandModeWB0 OMX_AUDIO_AMRBandModeWB1 OMX_AUDIO_AMRBandModeWB2 OMX_AUDIO_AMRBandModeWB3 OMX_AUDIO_AMRBandModeWB4 OMX_AUDIO_AMRBandModeWB5 OMX_AUDIO_AMRBandModeWB6 OMX_AUDIO_AMRBandModeWB7 OMX_AUDIO_AMRBandModeWB8 eAMRDTXMode = OMX_AUDIO_AMRDTXModeOnAuto eAMRFrameFormat = OMX_AUDIO_AMRFrameFormatIF2 OMX_AUDIO_AMRFrameFormatFSF

Port Index	APB+1		
	OMX_IndexConfigAudioAmrMode	r/w	nBitRate = 6600 8850 12650 14250 15850 18250 19850 23050 23850 OMX_AUDIO_AMRMODETYPE::OMX_AUDIO_AMRBANDMODETYPE = OMX_AUDIO_AMRBandModeWB0 OMX_AUDIO_AMRBandModeWB1 OMX_AUDIO_AMRBandModeWB2 OMX_AUDIO_AMRBandModeWB3 OMX_AUDIO_AMRBandModeWB4 OMX_AUDIO_AMRBandModeWB5 OMX_AUDIO_AMRBandModeWB6 OMX_AUDIO_AMRBandModeWB7 OMX_AUDIO_AMRBandModeWB8

9.4.2.4 AMR-WB+ Encoder Component

Name	audio_encoder.amrwb+			
Description	Encodes the giiven audio stream into a compressed audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts audio for encoding.
	APB+1	audio	output	Emits encoded audio.

Port Index	APB+0		
Description	Accepts audio for encoding.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamPortDefinition	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingPCM
	OMX_IndexParamAudioPortFormat	r/w	eEncoding = OMX_AUDIO_CodingPCM

Port Index	APB+0		
	`OMX_IndexParamAudioPcm`	r/w	Specify/query the sampling rate and number of channels. nChannels = 2 (Stereo) or 1 (Mono) eNumData = *OMX_NumericalDataSigned* bInterleaved = *OMX_TRUE* nBitPerSample = *16* nSamplingRate = 48000 44100 32000 24000 22050 16000 11025 8000 ePCMMode = *OMX_AUDIO_PCMModeLinear*

Port Index	APB+1		
Description	Emits encoded audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingAMR
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingAMR

Port Index	APB+1		
	OMX_IndexParamAudioAmr	r/w	nChannels = *2 or 1* nBitRate = between 5200 bps and 48000 bps OMX_AUDIO_PARAM_AMRTYPE::OMX_AUDIO_AMRBANDMODETYPE = Between (OMX_AUDIO_AMRBandModeWBP0 and OMX_AUDIO_AMRBandModeWBP47) Or OMX_AUDIO_AMRBandModeAuto eAMRDTXMode = OMX_AUDIO_AMRDTXModeOnAuto eAMRISFIndex = between (OMX_AUDIO_AMRISFIndex0 and OMX_AUDIO_AMRISFIndex13) Or OMX_AUDIO_AMRISFIndexAuto eAMRFrameFormat = OMX_AUDIO_AMRFrameFormatWBPlusTIF OMX_AUDIO_AMRFrameFormatWBPlusFSF

9.4.2.5 MP3 Encoder Component

Name	audio_encoder.mp3			
Description	Encodes the giiven audio stream into a compressed audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts audio for encoding.
	APB+1	audio	output	Emits encoded audio.

Port Index	APB+0		
Description	Accepts audio for encoding.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamPortDefinition	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingPCM
	OMX_IndexParamAudioPortFormat	r/w	eEncoding = OMX_AUDIO_CodingPCM

Port Index	APB+0		
	`OMX_IndexParamAudioPcm`	r/w	Specify/query the sampling rate and number of channels. nChannels = *2 (Stereo)* 1 (Mono) eNumData = *OMX_NumericalDataSigned* bInterleaved = *OMX_TRUE* nBitPerSample = *16* nSamplingRate = 32000 44100 *48000* ePCMMode = *OMX_AUDIO_PCMModeLinear*

Port Index	APB+1		
Description	Emits encoded audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingMP3
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingMP3

Port Index	APB+1		
	OMX_IndexParamAudioMp3	r/w	nChannels = *2 (stereo)* 1 (mono) nBitRate = 80000 to 320000 bps nSampleRate = 32000 44100 *48000* nAudioBandWidth = *0* eChannelMode = *OMX_AUDIO_ChannelModeStereo* OMX_AUDIO_ChannelModeJointStereo OMX_AUDIO_ChannelModeDual OMX_AUDIO_ChannelModeMono

9.4.3 Audio Mixer Class

The PCM format endianness is left to be native, meaning it can be either big endian or little endian depending on the underlying hardware. Endianness conversions, if needed, are left outside the standard audio post processing components.

Name	audio_mixer			
Description	Accetps multiple (N) audio streams, mixes them into a single stream, and emits the resulting stream as output.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	output	Emits audio stream resulting from mixing.
	APB+1 to APB+N	audio	input	Accepts audio stream for mixing.

9.4.3.1 PCM Mixer Component

Name	audio_mixer.pcm			
Description	Performs mixing of multiple audio input channels to 1 audio output mixing.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	output	Emits audio stream resulting from mixing.
	APB+1 to APB+N	audio	input	Accepts audio stream for mixing.

Port Index	APB+0

Port Index	APB+0		
Description	Emits audio stream resulting from mixing.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPcm`	r/w	Specify/query the sampling rate and number of channels. nChannels = *2 (Stereo)* 1 (Mono) eNumData = *OMX_NumericalDataSigned* eEndian = « Native » bInterleaved = *OMX_TRUE* nBitPerSample = *16* nSamplingRate = 8000, 11025 12000 16000 22050 24000 32000 44100 *48000* ePCMMode = *OMX_AUDIO_PCMModeLinear* eChannelMapping[0]= *OMX_AUDIO_ChannelLF (stereo)* OMX_AUDIO_ChannelCF (mono) eChannelMapping[1]= *OMX_AUDIO_ChannelRF (stereo)*
	`OMX_IndexConfigAudioVolume`	r/w	bLinear = *OMX_FALSE* sVolume = *Configurable*
	`OMX_IndexConfigAudioMute`	r/w	bMute = *OMX_FALSE* OMX_TRUE

Port Index	APB+1 to APB+N		
Description	Accepts audio for mixing.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the audio port settings. eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPcm`	r/w	Specify/query the sampling rate and number of channels. nChannels = *2 (Stereo)* 1 (Mono) eNumData = *OMX_NumericalDataSigned* eEndian = « Native » bInterleaved = *OMX_TRUE* nBitPerSample = *16* nSamplingRate = 8000, 11025 12000 16000 22050 24000 32000 44100 *48000* ePCMMode = *OMX_AUDIO_PCMModeLinear* eChannelMapping[0]= *OMX_AUDIO_ChannelLF (stereo)* OMX_AUDIO_ChannelCF (mono) eChannelMapping[1]= *OMX_AUDIO_ChannelRF (stereo)*
	`OMX_IndexConfigAudioVolume`	r/w	bLinear = *OMX_FALSE* sVolume = *Configurable*

Port Index	APB+1 to APB+N		
	`OMX_IndexConfigAudioMute`	r/w	bMute = *OMX_FALSE* OMX_TRUE

9.4.4 Audio Reader Class

Name	audio_reader			
Description	Reads an audio filestream and emits contained audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	output	Emits audio stream found in filestream.

9.4.4.1 Binary Audio Reader Class

Name	audio_reader.binary			
Description	Blindly reads any audio filestream (e.g. an MP3 file) irrespective of format and emits contained elementary audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	output	Emits audio stream found in filestream.

9.4.5 Audio Renderer Class

The PCM format endianness is left to be native, meaning it can be either big endian or little endian depending on the underlying hardware. Endianness conversions, if needed, are left outside the standard audio post processing components.

Audio renderers SHALL support providing reference clock.

Name	audio_renderer			
Description	Renders a given audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts audio for presentation.
	OPB+0	other/time	input	Accepts time updates

9.4.5.1 PCM Renderer Component

Name	audio_renderer.pcm			
Description	Renders a given audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts audio for presentation.
	OPB+0	other/time	input	Accepts time updates

Port Index	APB+0		
Description	Accepts audio for rendering.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the audio port settings. eEncoding = *OMX_AUDIO_CodingPCM*
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = *OMX_AUDIO_CodingPCM*

Port Index	APB+0		
	`OMX_IndexParamAudioPcm`	r/w	Specify/query the sampling rate and number of channels. nChannels = *2 (Stereo)* 1 (Mono) eNumData = *OMX_NumericalDataSigned* eEndian = « Native » bInterleaved = *OMX_TRUE* nBitPerSample = *16* nSamplingRate = 8000 11025 12000 16000 22050 24000 32000 44100 *48000* ePCMMode = *OMX_AUDIO_PCMModeLinear* eChannelMapping[0]= *OMX_AUDIO_ChannelLF (stereo)* OMX_AUDIO_ChannelCF (mono) eChannelMapping[1]= *OMX_AUDIO_ChannelRF (stereo)*
	`OMX_IndexConfigAudioVolume`	r/w	bLinear = *OMX_FALSE* sVolume = *Configurable*
	`OMX_IndexConfigAudioMute`	r/w	bMute = *OMX_FALSE* OMX_TRUE

Port Index	OPB+0
Description	Accepts media time updates. Provides mechanism for audio renderer component to query for media time. Audio renderer can provide the audio reference clock to the clock component which facilitates synchronization of other processing (e.g. video rendering) to audio rendering.

Port Index	OPB+0		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexConfigTimeRenderingDelay`	Read	Queries the new rendering delay

Port Index	OPB+0
Description	Accepts media time updates. Provides mechanism for audio renderer component to query for media time. Audio renderer can provide the audio reference clock to the clock component which facilitates synchronization of other processing (e.g. video rendering) to audio rendering..

9.4.6 Audio Writer Class

Name	audio_writer			
Description	Writes given audio stream to an audio filestream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts audio stream to be written to the audio filestream.

9.4.6.1 Binary Audio Writer Class

Name	audio_writer.binary			
Description	Blindly writes given elementary audio stream to an audio filestream (e.g. an MP3 file) irrespective of format.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts audio stream to be written to the audio filestream.

9.4.7 Audio Capturer Class

The PCM format endianness is left to be native, meaning it can be either big endian or little endian depending on the underlying hardware. Endianness conversions, if needed, are left outside the standard audio post processing components.

Audio capture components shall be capable of providing reference clock updates.

Name	audio_capturer			
Description	Emits an audio stream from an audio source.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	output	Emits source's audio stream.
	OPB+0	other/time	input	Receives media time updates/provides access to clock component.

9.4.7.1 PCM Audio Capturer

Name	audio_capturer.pcm			
Description	Emits an audio stream from an audio source.			
Ports	**Index**	**Domain**	**Direction**	**Description**

Name	audio_capturer.pcm			
	APB+0	audio	output	Emits source's audio stream.
	OPB+0	other/time	input	Receives media time updates/provides access to clock component.

Port Index	APB+0		
Description	Accepts audio for rendering.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamAudioPortFormat`	r/w	Specify/query the sampling rate and number of channels. eEncoding = *OMX_AUDIO_CodingPCM*
	`OMX_IndexParamPortDefinition`	r/w	eEncoding = *OMX_AUDIO_CodingPCM*

Port Index	APB+0		
	OMX_IndexParamAudioPcm		Specify/query the sampling rate and number of channels. nChannels = *2 (Stereo)* 1 (Mono) eNumData = *OMX_NumericalDataSigned* eEndian = « Native » bInterleaved = *OMX_TRUE* nBitPerSample = *16* nSamplingRate = 8000 11025 12000 16000 22050 24000 32000 44100 *48000* ePCMMode = *OMX_AUDIO_PCMModeLinear* eChannelMapping[0]= *OMX_AUDIO_ChannelLF (stereo)* OMX_AUDIO_ChannelCF (mono) eChannelMapping[1]= *OMX_AUDIO_ChannelRF (stereo)*
	OMX_IndexConfigAudioVolume	r/w	bLinear = *OMX_FALSE* sVolume = *Configurable*
	OMX_IndexConfigAudioMute	r/w	bMute = *OMX_FALSE* OMX_TRUE

Port Index	OPB+0
Description	Accepts media time updates. Provides mechanism for audio capturer component to query for media time. Audio capturer can provide the audio reference clock to the clock component which facilitates synchronization of other processing (e.g. video capture) to audio capture.

9.4.7.2 Audio Capture Use Case

An IL client using an audio source to capture an audio stream may do so via the following steps:

1. Instantiate the audio source component and any co-operating components
2. Set audio source settings:
3. Set the desired characteristics of the captured audio stream (e.g. sampling rate, channels)
4. Set the gain via the volume/mute controls
5. Establish any necessary tunnels between the audio source component and other components (e.g. an audio encoder tunneling with the capture port).
6. Select the clock component's active reference clock. In a use case with audio capture this is normally the audio clock as provided by the audio capturer.
7. Transition all components to the `OMX_StateIdle` state. Then transition the audio source component to the `OMX_StatePause` state, and transition all other components to the `OMX_StateExecuting` state. Although all other components are ready for capture, the audio source's output port is not yet emitting data.
8. To initiate capture transition the audio source component to the `OMX_StateExecuting` state. If using a clock component start the clock component. The audio source component will begin emitting captured audio of the prescribed characteristics.
9. To terminate capture transition the audio source component to the `OMX_StatePause` state. The audio source component will cease emitting captured audio.

9.4.8 Audio Processor class

Name	audio_processor			
Description	Processes a raw audio stream			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts raw audio.
	APB+1	audio	output	Emits raw audio

9.4.8.1 Properties that apply to all audio processing components

Sample rate conversions, downmix and upmix support is not mandated. If these features are not supported, the component shall implement the slaving behavior as described in section (ref to slaving behavior section).

The PCM format endianness is left to be native, meaning it can be either big endian or little endian depending on the underlying hardware. Endianness conversions, if needed, are left outside the standard audio post processing components.

Port Index	APB+0

Port Index	APB+0		
Description	Accepts raw audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	eDomain = OMX_PortDomainAudio format.eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPcm`	r/w	nChannels = 2 (Stereo) eNumData = OMX_NumericalDataSigned eEndian = <native> bInterleaved = True nBitPerSample = 16 ePCMMode = OMX_AUDIO_PCMModeLinear eChannelMapping = OMX_AUDIO_ChannelLF, OMX_AUDIO_ChannelRF

Port Index	APB+1		
Description	Emits raw audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r	eDomain = OMX_PortDomainAudio format.eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPortFormat`	r	eEncoding = OMX_AUDIO_CodingPCM

Port Index	APB+1		
	OMX_IndexParamAudioPcm	r	nChannels = 2 (Stereo) eNumData = OMX_NumericalDataSigned eEndian = <native> bInterleaved = True nBitPerSample = 16 ePCMMode = OMX_AUDIO_PCMModeLinear eChannelMapping = OMX_AUDIO_ChannelLF, OMX_AUDIO_ChannelRF

9.4.8.2 Stereo widening loudspeakers

Headphone and loudspeaker versions of this standard component are separated to better support multi-components and to allow vendors to implement just one of the two algorithm variations.

In case the implementation supports only one single value for the `nStereoWidening` field of the `OMX_AUDIO_CONFIG_STEREOWIDENINGTYPE` structure, that value shall be 100, and the component shall always return 100 as the value for the field for all `OMX_GetConfig` calls. See Section 4.1.49—`OMX_AUDIO_CONFIG_STEREOWIDENINGTYPE`.

Name	audio_processor.pcm.stereo_widening_loudspeakers			
Description	Adds stereo widening to a raw audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts raw audio.
	APB+1	audio	output	Emits raw audio

Port Index	APB+0		
Description	Accepts raw audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamAudioPcm	r/w	nBitPerSample = *16* nSamplingRate = 16000, 22050, 24000, 32000, 44100, *48000* Hz

Port Index	APB+1		
Description	Emits raw audio.		
Required	**Index**	**Access**	**Description**

Port Index	APB+1		
Parameters/ Configs	OMX_IndexConfigAudioStereoWidening	r/w	bEnable = *False*, True eWideningType = OMX_AUDIO_StereoWideningLoudspeakers
	OMX_IndexParamAudioPcm	r	nBitPerSample = *16* nSamplingRate = 16000, 22050, 24000, 32000, 44100, *48000* Hz

9.4.8.3 Stereo widening headphones

In case the implementation supports only one single value for the nStereoWidening field of the OMX_AUDIO_CONFIG_STEREOWIDENINGTYPE structure, that value shall be 100, and the component shall always return 100 as the value for the field for all OMX_GetConfig calls. See Section 4.1.49—OMX_AUDIO_CONFIG_STEREOWIDENINGTYPE .

Name	audio_processor.pcm.stereo_widening_headphones			
Description	Adds stereo widening to a raw audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts raw audio.
	APB+1	audio	output	Emits raw audio

Port Index	APB+0		
Description	Accepts raw audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamAudioPcm	r/w	nBitPerSample = *16* nSamplingRate = 16000, 22050, 24000, 32000, 44100, *48000* Hz

Port Index	APB+1		
Description	Emits raw audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexConfigAudioStereoWidening	r/w	bEnable = *False*, True eWideningType = OMX_AUDIO_StereoWideningHeadphones
	OMX_IndexParamAudioPcm	r	nBitPerSample = *16* nSamplingRate = 16000, 22050, 24000, 32000, 44100, *48000* Hz

9.4.8.4 Reverberation

Name	audio_processor.pcm.reverberation			
Description	Adds reverberation to a raw audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts raw audio.

Name	audio_processor.pcm.reverberation			
	APB+1	audio	output	Emits raw audio

Port Index	APB+0		
Description	Accepts raw audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamAudioPcm`	r/w	nBitPerSample = *16* nSamplingRate = 44100, *48000* Hz

Port Index	APB+1		
Description	Emits raw audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexConfigAudioReverberation`	r/w	bEnable = *False*, True
	`OMX_IndexParamAudioPcm`	r	nBitPerSample = *16* nSamplingRate = 44100, *48000* Hz

9.4.8.5 Chorus

Name	audio_processor.pcm.chorus			
Description	Adds chorus to a raw audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts raw audio.
	APB+1	audio	output	Emits raw audio

Port Index	APB+0		
Description	Accepts raw audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamAudioPcm`	r/w	nBitPerSample = *16* nSamplingRate = 44100, *48000* Hz

Port Index	APB+1		
Description	Emits raw audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexConfigAudioChorus`	r/w	bEnable = *False*, True
	`OMX_IndexParamAudioPcm`	r	nBitPerSample = *16* nSamplingRate = 44100, *48000* Hz

9.4.8.6 Equalizer

Equalizer band count is encoded into the name for convenience, so that the IL client can choose the preferred equalizer, if multiple exists, without loading the components.

Name	audio_processor.pcm.equalizer			
Description	Does equalization on a raw audio stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts raw audio.
	APB+1	audio	output	Emits raw audio

Port Index	APB+0		
Description	Accepts raw audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamAudioPcm`	r	nBitPerSample = *16* nSamplingRate = 44100, *48000* Hz

Port Index	APB+1		
Description	Emits raw audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexConfigAudioEqualizer`	r/w	bEnable = *False*, True sBandLevel = [-1200, 1200]
	`OMX_IndexParamAudioPcm`	r	nBitPerSample = *16* nSamplingRate = 44100, *48000* Hz
	`OMX_IndexConfigAudioLoudness`	r/w	bLoudness = *False*, True
	`OMX_IndexConfigAudioBass`	r/w	bEnable = *False*, True nBass = [-100, 100]
	`OMX_IndexConfigAudioTreble`	r/w	bEnable = *False*, True nTreble = [-100, 100]

9.4.9 Audio 3D Mixer Class

A 3D Audio Mixer component accepts multiple (N) audio streams, applies positional 3D processing, and mixes the streams into two streams, and emits the two resulting streams as output. The first output stream contains the direct path audio and the second output stream contains the room signal.

The PCM format endianness is *native*, meaning it can be either big endian or little endian depending on the underlying hardware. Endianness conversions, if needed, are left outside the standard component.

Name	audio_3D_mixer			
Description	Accepts multiple (N) audio streams, 3D mixes them into two streams, and emits the two resulting streams as output. The first output contains the direct path audio and the second output contains the room signal.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	output	Emits audio stream resulting from positional 3D mixing.
	APB+1	audio	output	Emits audio stream for room effect purposes.
	APB+2 to APB+(N+1)	audio	input	Accepts audio stream for mixing.

9.4.9.1 PCM 3D Mixer Component

Please note that there are two variants of this standard component role: one for headphone rendering and another one for loudspeaker rendering. The variants only differ in the mandated output type.

Role Name	audio_3D_mixer.pcm.[headphones|loudspeakers]			
Description	Performs 3D mixing of multiple (N) audio input channels to 2 audio output mixes: main mix (direct path audio) and room mix (for reverberator).			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	output	Emits audio stream resulting from positional 3D mixing.
	APB+1	audio	output	Emits audio stream for room effect purposes.
	APB+2 to APB+(N+1)	audio	input	Accepts audio stream for 3D mixing.

Port Index	None		
Description	These required configs affect the whole component		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexConfigCommitMode`	r/w	Set or query the commit mode for the 3D mixer. bDeferred = *OMX_FALSE* OMX_TRUE
	`OMX_IndexConfigCommit`	w	Commits the 3D settings if commit mode is deferred.

Port Index	APB+0		
Description	Emits audio stream resulting from positional 3D mixing. This is the main output containing direct path audio.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Set or query the audio port settings. eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingPCM

	OMX_IndexParamAudioPcm	r/w	Set or query the sampling rate and number of channels. nChannels = *2 (Stereo)* eNumData = *OMX_NumericalDataSigned* eEndian = « Native » bInterleaved = *OMX_TRUE* nBitPerSample = *16* nSamplingRate = 8000, 11025 12000 16000 22050 24000 32000 44100 *48000* ePCMMode = *OMX_AUDIO_PCMModeLinear* eChannelMapping[0]= *OMX_AUDIO_ChannelLF* eChannelMapping[1]= *OMX_AUDIO_ChannelRF*
	OMX_IndexConfigAudioVolume	r/w	bLinear = *OMX_FALSE* sVolume = *Configurable*
	OMX_IndexConfigAudioMute	r/w	bMute = *OMX_FALSE* OMX_TRUE
	OMX_IndexConfigAudio3DOutput	r/w	e3DOutputType = *OMX_AUDIO_3DOutput[Headphones|Loudspeakers]*

Port Index	APB+1		
Description	Emits audio stream for room effect purposes.		
Required	**Index**	**Access**	**Description**

Parameters/ Configs	`OMX_IndexParamPortDefinition`	r/w	Set or query the audio port settings. eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPcm`	r/w	Set or query the sampling rate and number of channels. nChannels = 1 (Mono) **(conditional: only if the corresponding change in the Reverb's input port is made)** *2 (Stereo)* eNumData = *OMX_NumericalDataSigned* eEndian = « Native » bInterleaved = *OMX_TRUE* nBitPerSample = *16* nSamplingRate = 8000, 11025 12000 16000 22050 24000 32000 44100 *48000* ePCMMode = *OMX_AUDIO_PCMModeLinear* eChannelMapping[0]= *OMX_AUDIO_ChannelLF* eChannelMapping[1]= *OMX_AUDIO_ChannelRF*
	`OMX_IndexConfigAudioVolume`	r/w	bLinear = *OMX_FALSE* sVolume = *Configurable*
	`OMX_IndexConfigAudioMute`	r/w	bMute = *OMX_FALSE* OMX_TRUE

Port Index	APB+2 to APB+(N+1)		
Description	Accepts audio for mixing.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Set or query the audio port settings. eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPortFormat`	r/w	eEncoding = OMX_AUDIO_CodingPCM
	`OMX_IndexParamAudioPcm`	r/w	Set or query the sampling rate and number of channels. nChannels = *1 (Mono)* 2 (Stereo) eNumData = *OMX_NumericalDataSigned* eEndian = « Native » bInterleaved = *OMX_TRUE* nBitPerSample = *16* nSamplingRate = 8000 11025 12000 16000 22050 24000 32000 44100 *48000* ePCMMode = *OMX_AUDIO_PCMModeLinear* eChannelMapping[0]= *OMX_AUDIO_ChannelLF* eChannelMapping[1]= *OMX_AUDIO_ChannelRF*

	OMX_IndexParamAudio3DDopplerMode	r/w	Set or query the enabled status of the Doppler effect bEnabled = *OMX_FALSE* OMX_TRUE
	OMX_IndexConfigAudio3DLocation	r/w	Set or query the virtual location of the 3D sound source. nX = the whole S32 range (default = 0) nY = the whole S32 range (default = 0) nZ = the whole S32 range (default = 0)
	OMX_IndexConfigAudio3DDopplerSettings	r/w	Set or query the Doppler settings of the 3D sound source. nSoundSpeed = the whole U32 range (default = 340000) nSourceVelocity = the whole S32 range (default = 0) nListenerVelocity = the whole S32 range (default = 0)
	OMX_IndexConfigAudio3DLevels	r/w	Set or query the direct path and room levels for the 3D sound source. sDirectLevel = [0x80000000, 0] (default = 0) sRoomLevel = [0x80000000, 0] (default = 0)

	`OMX_IndexConfigAudio3DDistanceAttenuation`	r/w	Set or query the distance attenuation behavior for the 3D sound source. sMinDistance = [0, 0x7FFFFFFF] (default = 1000) sMaxDistance = [0, 20x7FFFFFF] (default = 0x7FFFFFFF) sRollOffFactor = [0, 10000] (default = 1000) sRoomRollOffFactor = [0, 10000] (default = 0) eRollOffModel = *OMX_AUDIO_RollOffExponential* OMX_AUDIO_RollOffLinear sMuteAfterMax = *OMX_FALSE* OMX_TRUE
	`OMX_IndexConfigAudio3DDirectivitySettings`	r/w	Set or query the directivity behavior for the 3D sound source. sInnerAngle = [0, 360000] (default = 360000) sOuterAngle = [0, 360000] (default = 360000) sOuterLevel = [0x80000000, 0] (default = 0)
	`OMX_IndexConfigAudio3DDirectivityOrientation`	r/w	Set or query the orientation of the directivity of the 3D sound source. nXFront = the whole S32 range (default = 0) nYFront = the whole S32 range (default = 0) nZFront = the whole S32 range (default = -1000)
	`OMX_IndexConfigAudioMute`	r/w	bMute = *OMX_FALSE* OMX_TRUE

9.5 Standard Image Components

9.5.1 Image Decoder Class

Name	image_decoder			
Description	Decodes the given compressed image data stream into an uncompressed image data stream..			
Ports	**Index**	**Domain**	**Direction**	**Description**
	IPB+0	image	input	Accepts encoded image data.
	IPB+1	image	output	Emits decoded image data.

Upscaling and downscaling support is not mandated. If these features are not supported, the component shall implement the slaving behavior as described in section (ref to slaving behavior section).

9.5.1.1 JPEG Decoder

Name	image_decoder.JPEG			
Description	Decodes the given compressed image data stream into an uncompressed image data stream..			
Ports	**Index**	**Domain**	**Direction**	**Description**
	IPB+0	image	input	Accepts encoded image data.
	IPB+1	image	output	Emits decoded image data.

Port Index	IPB+0		
Description	Accepts encoded image data.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the image port settings. nFrameWidth = *640* nFrameHeight = *480* eCompressionFormat = *OMX_IMAGE_CodingJPEG* eColorFormat = *OMX_COLOR_FormatUnused*
	`OMX_IndexParamImagePortFormat`	r/w	Specify/query the image format. eCompressionFormat = *OMX_IMAGE_CodingJPEG* eColorFormat = *OMX_COLOR_FormatUnused*

Port Index	IPB+0		
	OMX_IndexParamQuantizationTable	r/w	eQuantizationTable= *OMX_IMAGE_QuantizationTableLuma* OMX_IMAGE_QuantizationTableChroma nQuantizationMatrix = *configureable*
	OMX_IndexParamHuffmanTable	r/w	eHuffmanTable = *OMX_IMAGE_HuffmanTableAC* OMX_IMAGE_HuffmanTableDC nNumberOfHuffmanCodeOfLength = *configurable* nHuffmanTable = *configurable*

Port Index	IPB+1		
Description	Emits decoded image data.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamPortDefinition	r/w	Specify/query the image port settings. nFrameWidth = *640* nFrameHeight = *480* eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

Port Index	IPB+1		
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the image format. eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

9.5.1.1 WebP Decoder

Name	Image_decoder.WEBP			
Description	Decodes the given compressed image data stream into an uncompressed image data stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	IPB+0	image	input	Accepts encoded image data.
	IPB+1	image	output	Emits decoded image data.

Port Index	IPB+0		
Description	Accepts encoded image data.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the image port settings. nFrameWidth = *640* nFrameHeight = *480* eCompressionFormat = *OMX_IMAGE_CodingWEBP* eColorFormat = *OMX_COLOR_FormatUnused*
	`OMX_IndexParamImagePortFormat`	r/w	Specify/query the image format. eCompressionFormat = *OMX_IMAGE_CodingWEBP* eColorFormat = *OMX_COLOR_FormatUnused*

Port Index	IPB+1		
Description	Emits decoded image data.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the image port settings. nFrameWidth = *640* nFrameHeight = *480* eCompressionFormat = *OMX_IMAGE_CodingUnused* eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the image format. eCompressionFormat = *OMX_IMAGE_CodingUnused* eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

9.5.2 Image Encoder Class

Name	image_encoder			
Description	Encodes the given image data stream into a compressed format.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	IPB+0	image	input	Accepts image data for encoding.
	IPB+1	image	output	Emits compressed image data.

Upscaling and downscaling support is not mandated. If these features are not supported, the component shall implement the slaving behavior as described in section (ref to slaving behavior section).

9.5.2.1 JPEG Encoder

Name	image_encoder.JPEG			
Description	Encodes the given image data stream into a compressed format.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	IPB+0	image	input	Accepts image data for encoding.
	IPB+1	image	output	Emits compressed image data.

Port Index	IPB+0		
Description	Accepts image data for encoding.		
Required	**Index**	**Access**	**Description**

Port Index	IPB+0		
Parameters/ Configs	`OMX_IndexParamPortDefinition`	r/w	Specify/query the image port settings. nFrameWidth = *640* nFrameHeight = *480* eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the image format. eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

Port Index	IPB+1

Port Index	IPB+1		
Description	Emits compressed image data.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the image port settings. nFrameWidth = *640(same as input)* nFrameHeight = *480(same as input)* eCompressionFormat = *OMX_IMAGE_CodingJPEG* eColorFormat = *OMX_COLOR_FormatUnused*
	`OMX_IndexParamImagePortFormat`	r/w	Specify/query the image format. eCompressionFormat = *OMX_IMAGE_CodingJPEG* eColorFormat = *OMX_COLOR_FormatUnused*
	`OMX_IndexParamQuantizationTable`	r/w	eQuantizationTable= *OMX_IMAGE_QuantizationTableLuma* OMX_IMAGE_QuantizationTableChroma nQuantizationMatrix = *configureable*

9.5.2.2 WebP Encoder

Name	Image_encoder.WEBP			
Description	Encodes the given image data stream into a compressed format.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	IPB+0	image	input	Accepts image data for encoding.
	IPB+1	image	output	Emits compressed image data.

Port Index	IPB+0		
Description	Accepts image data for encoding.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the image port settings. nFrameWidth = *640* nFrameHeight = *480* eCompressionFormat = *OMX_IMAGE_CodingUnused* eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar*

			OMX_COLOR_FormatYUV420Packed Planar *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*
	`OMX_IndexParamImagePortFormat`	r/w	Specify/query the video format. eCompressionFormat = *OMX_IMAGE_CodingUnused* eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

Port Index	IPB+1		
Description	Emits compressed image data.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the image port settings. nFrameWidth = *640 (same as input)* nFrameHeight = *480 (same as input)* eCompressionFormat = *OMX_IMAGE_CodingWEBP* eColorFormat = *OMX_COLOR_FormatUnused*

	OMX_IndexParamVideoPortFormat	r/w	Specify/query the video format. eCompressionFormat = *OMX_IMAGE_CodingWEBP* eColorFormat = *OMX_COLOR_FormatUnused*

9.5.3 Image Reader Class

Name	image_reader			
Description	Read an image filestream and emits the contained image stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	IPB+0	image	output	Emits image stream found in filestream.

9.5.3.1 Binary Image Reader Class

Name	image_reader.binary			
Description	Blindly reads any image filestream (e.g. a JPG file) irrespective of the format and emits contained elementary image stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	IPB+0	image	output	Emits image stream found in filestream.

9.5.4 Image Writer Class

Name	image_writer			
Description	Writes given image stream to an image filestream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	IPB+0	image	input	Accepts image stream to be written to the image filestream.

9.5.4.1 Binary Image Writer Class

Name	image_writer.binary			
Description	Blindly writes given elementary image stream to an image filestream (e.g. a JPG file) irrespective of format.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	IPB+0	image	input	Accepts image stream to be written to the image filestream.

9.6 Standard Video Components

9.6.1 Video Decoder Class

Name	video_decoder			
Description	Decodes the given compressed video stream into an uncompressed video stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Accepts encoded video.
	VPB+1	video	output	Emits decoded video.

Upscaling, downscaling and frame rate conversion support is not mandated. If these features are not supported, the component shall implement the slaving behavior as described in section (ref to slaving behavior section).

Video decoders shall emit frames in display order.

9.6.1.1 H.263 Decoder Component

Name	video_decoder.h263			
Description	Decodes the given compressed video stream into an uncompressed video stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Consumes compressed video content.
	VPB+1	video	output	Produces uncompressed raw video.

Port Index	VPB+0		
Description	Consumes compressed video content.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* nBitRate = *64000* xFrameRate = *15* eCompressionFormat = *OMX_VIDEO_CodingH263* eColorFormat = *OMX_COLOR_FormatUnused*
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the video format. eCompressionFormat = *OMX_VIDEO_CodingH263* eColorFormat = *OMX_COLOR_FormatUnused*
	`OMX_IndexParamVideoH263`	r	eProfile = *OMX_VIDEO_H263ProfileBaseline* eLevel= *OMX_VIDEO_H263Level10* bPLUSPTYPEAllowed = *OMX_FALSE* bForceRoundingTypeToZero = *OMX_TRUE*

Port Index	VPB+0		
	`OMX_IndexParamVideoProfileLevelQuerySupported`	r	Query supported profile/level pair by index.
	`OMX_IndexParamVideoProfileLevelCurrent`	r	Query current profile/level pair.

Port Index	VPB+1		
Description	Produces uncompressed raw video.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

Port Index	VPB+1		
	OMX_IndexParamVideoPortFormat	r/w	Specify/query the video format. eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

9.6.1.2 AVC Decoder Component

Name	video_decoder.avc			
Description	Decodes the given compressed video stream into an uncompressed video stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Consumes compressed video content.
	VPB+1	video	output	Produces uncompressed raw video.

Port Index	VPB+0		
Description	Consumes compressed video content.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamPortDefinition	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* nBitRate = *64000* xFrameRate = *15* eCompressionFormat = *OMX_VIDEO_CodingAVC* eColorFormat = *OMX_COLOR_FormatUnused*

Port Index	VPB+0		
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the video format. eCompressionFormat = *OMX_VIDEO_CodingAVC* eColorFormat = *OMX_COLOR_FormatUnused*
	`OMX_IndexParamVideoAvc`	r	eProfile = *OMX_VIDEO_AVCProfileBaseline* eLevel = *OMX_VIDEO_AVCLevel1*
	`OMX_IndexParamVideoProfileLevelQuerySupported`	r	Query supported profile/level pair by index.
	`OMX_IndexParamVideoProfileLevelCurrent`	r	Query current profile/level pair.

Port Index	VPB+1		
Description	Produces uncompressed raw video.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

Port Index	VPB+1		
	OMX_IndexParamVideoPortFormat	r/w	Specify/query the video format. eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

9.6.1.3 MPEG4 Video Decoder Component

Name	video_decoder.mpeg4			
Description	Decodes the given compressed video stream into an uncompressed video stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Consumes compressed video content.
	VPB+1	video	output	Produces uncompressed raw video.

Port Index	VPB+0		
Description	Consumes compressed video content.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamPortDefinition	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* nBitRate = *64000* xFrameRate = *15* eCompressionFormat = *OMX_VIDEO_CodingMPEG4* eColorFormat = *OMX_COLOR_FormatUnused*

Port Index	VPB+0		
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the video format. eCompressionFormat = *OMX_VIDEO_CodingMPEG4* eColorFormat = *OMX_COLOR_FormatUnused*
	`OMX_IndexParamVideoMpeg4`	r/w	eProfile = *OMX_VIDEO_MPEG4ProfileSimple* eLevel = *OMX_VIDEO_MPEG4Level1*
	`OMX_IndexParamVideoProfileLevelQuerySupported`	r	Query supported profile/level pair by index.
	`OMX_IndexParamVideoProfileLevelCurrent`	r	Query current profile/level pair.

Port Index	VPB+1		
Description	Produces uncompressed raw video.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

Port Index	VPB+1		
	OMX_IndexParamVideoPortFormat	r/w	Specify/query the video format. eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

9.6.1.4 Real Video Decoder Component

Name	video_decoder.rv			
Description	Decodes the given compressed video stream into an uncompressed video stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Consumes compressed video content.
	VPB+1	video	output	Produces uncompressed raw video.

Port Index	VPB+0		
Description	Consumes compressed video content.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamPortDefinition	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* nBitRate = *64000* xFrameRate = *15* eCompressionFormat = *OMX_VIDEO_CodingRV* eColorFormat = *OMX_COLOR_FormatUnused*

Port Index	VPB+0		
	OMX_IndexParamVideoPortFormat	r/w	Specify/query the video format. eCompressionFormat = *OMX_VIDEO_CodingRV* eColorFormat = *OMX_COLOR_FormatUnused*
	OMX_IndexParamVideoRv	r/w	Specify/query Real Video specific parameters. eFormat = OMX_VIDEO_RVFormat8 *OMX_VIDEO_RVFormat9* bEnablePostFilter = OMX_TRUE *OMX_FALSE* bEnableLatencyMode = OMX_TRUE *OMX_FALSE*

Port Index	VPB+1		
Description	Produces uncompressed raw video.		
Required	**Index**	**Access**	**Description**

Port Index	VPB+1		
Parameters/ Configs	`OMX_IndexParamPortDefinition`	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the video format. eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

9.6.1.5 WMV Decoder Component

Name	video_decoder.wmv			
Description	Decodes the given compressed video stream into an uncompressed video stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Consumes compressed video content.
	VPB+1	video	output	Produces uncompressed raw video.

Port Index	VPB+0		
Description	Consumes compressed video content.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* nBitRate = *64000* xFrameRate = *15* eCompressionFormat = *OMX_VIDEO_CodingWMV* eColorFormat = *OMX_COLOR_FormatUnused*
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the video format. eCompressionFormat = *OMX_VIDEO_CodingWMV* eColorFormat = *OMX_COLOR_FormatUnused*
	`OMX_IndexParamVideoWmv`	r/w	Specify/query Real Video specific parameters. eFormat = OMX_VIDEO_WMVFormat7 OMX_VIDEO_WMVFormat8 *OMX_VIDEO_WMVFormat9*

Port Index	VPB+1		
Description	Produces uncompressed raw video.		
Required	**Index**	**Access**	**Description**

Port Index	VPB+1		
Parameters/ Configs	`OMX_IndexParamPortDefinition`	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the video format. eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

9.6.1.6 VC-1 Decoder Component

Name	video_decoder.vc1			
Description	Decodes the given compressed video stream into an uncompressed video stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	Video	input	Consumes compressed video content.
	VPB+1	Video	output	Produces uncompressed raw video.

Port Index	VPB+0		
Description	Consumes compressed video content.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* nBitRate = *64000* `xFrameRate` = *15* `eCompressionFormat` = *OMX_VIDEO_CodingVC1* `eColorFormat` = *OMX_COLOR_FormatUnused*
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the video format. `eCompressionFormat` = *OMX_VIDEO_CodingVC1* `eColorFormat` = *OMX_COLOR_FormatUnused*
	`OMX_IndexParamVideoVC1`	r/w	Specify/query VC-1 specific parameters. eProfile = OMX_VIDEO_VC1ProfileSimple OMX_VIDEO_VC1ProfileMain *OMX_VIDEO_VC1ProfileAdvanced*

Port Index	VPB+1		
Description	Produces uncompressed raw video.		
Required	**Index**	**Access**	**Description**

Port Index	VPB+1		
Parameters/ Configs	`OMX_IndexParamPortDefinition`	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* `eCompressionFormat` = `OMX_VIDEO_CodingUnused` eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the video format. `eCompressionFormat` = `OMX_VIDEO_CodingUnused` eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

9.6.1.1 VP8 Decoder Component

Name	video_decoder.vp8			
Description	Decodes the given compressed video stream into an uncompressed video stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Consumes compressed video content.
	VPB+1	video	output	Produces uncompressed raw video.

Port Index	VPB+0		
Description	Consumes compressed video content.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* nBitRate = *64000* xFrameRate = *15* eCompressionFormat = *`OMX_VIDEO_CodingVP8`* eColorFormat = *`OMX_COLOR_FormatUnused`*
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the video format. eCompressionFormat = *`OMX_VIDEO_CodingVP8`* eColorFormat = *`OMX_COLOR_FormatUnused`*
	`OMX_IndexParamVideoVP8`	r	eProfile = *`OMX_VIDEO_VP8ProfileMain`* eLevel = *`OMX_VIDEO_VP8Level_Version0`*
	`OMX_IndexParamVideoProfileLevelQuerySupported`	r	Query supported profile/level pair by index.
	`OMX_IndexParamVideoProfileLevelCurrent`	r	Query current profile/level pair.

Port Index	VPB+1		
Description	Produces uncompressed raw video.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* eCompressionFormat =

			OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the video format. eCompressionFormat = *OMX_VIDEO_CodingUnused* eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

9.6.2 Video Encoder Class

Name	video_encoder			
Description	Encodes the given uncompressed video stream into a compressed format.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Accepts video for encoding.
	VPB+1	video	output	Emits encoded video.

Upscaling, downscaling and frame rate conversion support is not mandated. If these features are not supported, the component shall implement the slaving behavior as described in section (ref to slaving behavior section).

9.6.2.1 H.263 Encoder Component

Name	video_encoder.h263			
Description	Encodes the given uncompressed video stream into a compressed format.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Consumes the uncompressed raw video content.
	VPB+1	video	output	Produces compressed video.

Port Index	VPB+0		
Description	Consumes compressed video content.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

Port Index	VPB+0		
	OMX_IndexParamVideoPortFormat	r/w	Specify/query the video format. eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

Port Index	VPB+1		
Description	Produces compressed video.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamPortDefinition	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* nBitRate = *64000* xFrameRate = *15* eCompressionFormat = *OMX_VIDEO_CodingH263* eColorFormat = *OMX_COLOR_FormatUnused*
	OMX_IndexParamVideoPortFormat	r/w	Specify/query the video format. eCompressionFormat = *OMX_VIDEO_CodingH263* eColorFormat = *OMX_COLOR_FormatUnused*

Port Index	VPB+1		
	OMX_IndexParamVideoBitrate	r/w	eControlRate = *OMX_Video_ControlRateConstant* OMX_Video_ControlRateDisable OMX_Video_ControlRateVariable nTargetBitrate = *64000*
	OMX_IndexParamVideoErrorCorrection	r/w	bEnableHEC = *OMX_TRUE* bEnableResync = *OMX_TRUE* nResynchMarkerSpacing = *Configureable(0 to 0xFFFFFFFF)*
	OMX_IndexParamVideoH263		eProfile = *OMX_VIDEO_H263ProfileBaseline* eLevel= *OMX_VIDEO_H263Level10* nPFrames = 0 to 0xffffffff bPLUSPTYPEAllowed = *OMX_FALSE* bForceRoundingTypeToZero = *OMX_TRUE* nGOBHeaderInterval = 1 to 9
	OMX_IndexConfigVideoFramerate	r/w	Specify/query target framerate xFrameRate = *15*
	OMX_IndexConfigVideoBitrate	r/w	Specify/query target bitrate nBitRate = *64000*
	OMX_IndexParamVideoProfileLevelQuerySupported	r	Query supported profile/level pair by index.
	OMX_IndexParamVideoProfileLevelCurrent	r/w	Specify/query current profile/level pair.

9.6.2.2 AVC Encoder Component

Name	video_encoder.avc			
Description	Encodes the given uncompressed video stream into a compressed format.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Consumes the uncompressed raw video content.
	VPB+1	video	output	Produces compressed video.

Port Index	VPB+0		
Description	Consumes compressed video content.		
Required	**Index**	**Access**	**Description**

Port Index	VPB+0		
Parameters/ Configs	`OMX_IndexParamPortDefinition`	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the video format. eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

Port Index	VPB+1

Port Index	VPB+1		
Description	Produces compressed video.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* nBitRate = *64000* xFrameRate = *15* eCompressionFormat = *OMX_VIDEO_CodingAVC* eColorFormat = *OMX_COLOR_FormatUnused*
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the video format. eCompressionFormat = *OMX_VIDEO_CodingAVC* eColorFormat = *OMX_COLOR_FormatUnused*
	`OMX_IndexParamVideoBitrate`	r/w	eControlRate = *OMX_Video_ControlRateConstant* OMX_Video_ControlRateDisable OMX_Video_ControlRateVariable nTargetBitrate = *64000*

Port Index	VPB+1		
	OMX_IndexParamVideoAvc	r/w	eProfile = *OMX_VIDEO_AVCProfileBaseline* eLevel = *OMX_VIDEO_AVCLevel1* nSliceHeaderSpacing = *Configureable* nPFrames = 0 to 0xffffffff bUseHadamard = *OMX_TRUE* nRefFrames = *1* bEnableFMO = *OMX_FALSE* bEnableASO = *OMX_FALSE* bWeightedPPrediction= *OMX_FALSE* bconstIpred = *OMX_FALSE*
	OMX_IndexConfigVideoFramerate	r/w	Specify/query target framerate xFrameRate = *15*
	OMX_IndexConfigVideoBitrate	r/w	Specify/query target bitrate nBitRate = *64000*
	OMX_IndexParamVideoProfileLevelQuerySupported	r	Query supported profile/level pair by index.
	OMX_IndexParamVideoProfileLevelCurrent	r/w	Specify/query current profile/level pair.

9.6.2.3 MPEG4 Video Encoder Component

Name	video_encoder.mpeg4			
Description	Encodes the given uncompressed video stream into a compressed format.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Consumes the uncompressed raw video content.
	VPB+1	video	output	Produces compressed video.

Port Index	VPB+0		
Description	Consumes compressed video content.		
Required	**Index**	**Access**	**Description**

Port Index	VPB+0		
Parameters/ Configs	`OMX_IndexParamPortDefinition`	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the video format. eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

Port Index	VPB+1

Port Index	VPB+1		
Description	Produces compressed video.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* nBitRate = *64000* xFrameRate = *15* eCompressionFormat = *OMX_VIDEO_CodingMPEG4* eColorFormat = *OMX_COLOR_FormatUnused*
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the video format. eCompressionFormat = *OMX_VIDEO_CodingMPEG4* eColorFormat = *OMX_COLOR_FormatUnused*
	`OMX_IndexParamVideoBitrate`	r/w	eControlRate = *OMX_Video_ControlRateConstant* OMX_Video_ControlRateDisable OMX_Video_ControlRateVariable nTargetBitrate = *64000*
	`OMX_IndexParamVideoMpeg4`	r/w	eProfile = *OMX_VIDEO_MPEG4ProfileSimple* eLevel = *OMX_VIDEO_MPEG4Level1* nSliceHeaderSpacing = *Configureable* bSVH = *OMX_FALSE* bGov = *Configureable* nPFrames = 0 to 0xffffffff nIDCVLCThreshold = *0* bACPred = *OMX_TRUE* nHeaderExtension = 1 to 99 bReversibleVLC = *OMX_FALSE*

Port Index	VPB+1		
	`OMX_IndexConfigVideoFramerate`	r/w	Specify/query target framerate xFrameRate = *15*
	`OMX_IndexConfigVideoBitrate`	r/w	Specify/query target bitrate nBitRate = *64000*
	`OMX_IndexParamVideoProfileLevelQuerySupported`	r	Query supported profile/level pair by index.
	`OMX_IndexParamVideoProfileLevelCurrent`	r/w	Specify/query current profile/level pair.

9.6.2.4 VP8 Encoder Component

Name	video_encoder.vp8			
Description	Encodes the given uncompressed video stream into a compressed format.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Consumes the uncompressed raw video content.
	VPB+1	video	output	Produces compressed video.

Port Index	VPB+0		
Description	Consumes the uncompressed raw video content.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* eCompressionFormat = *`OMX_VIDEO_CodingUnused`* eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the video format. eCompressionFormat = *OMX_VIDEO_CodingUnused*

			eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

Port Index	VPB+1		
Description	Produces compressed video.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the video port settings. nFrameWidth = *176* nFrameHeight = *144* nBitRate = *64000* xFrameRate = *15* eCompressionFormat = *OMX_VIDEO_CodingVP8* eColorFormat = *OMX_COLOR_FormatUnused*
	`OMX_IndexParamVideoPortFormat`	r/w	Specify/query the video format. eCompressionFormat = *OMX_VIDEO_CodingVP8* eColorFormat = *OMX_COLOR_FormatUnused*
	`OMX_IndexParamVideoBitrate`	r/w	eControlRate = *OMX_Video_ControlRateConstant* *OMX_Video_ControlRateDisable* *OMX_Video_ControlRateVariable* nTargetBitrate = *64000*
	`OMX_IndexParamVideoVP8`	r/w	eProfile = *OMX_VIDEO_VP8ProfileMain* eLevel = *OMX_VIDEO_VP8Level_Version0* nDCTPartitions = *0 to 3*

			bErrorResilientMode = *OMX_FALSE* *OMX_TRUE*
	`OMX_IndexConfigVideoFramerate`	r/w	Specify/query target framerate xFrameRate = *15*
	`OMX_IndexConfigVideoBitrate`	r/w	Specify/query target bitrate nBitRate = *64000*
	`OMX_IndexParamVideoProfileLevelQuerySupported`	r	Query supported profile/level pair by index.
	`OMX_IndexParamVideoProfileLevelCurrent`	r/w	Specify/query current profile/level pair.

9.6.3 Video Reader Class

Name	video_reader			
Description	Reads a video filestream and emits the contained video stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	output	Emits video stream found in filestream.

9.6.3.1 Binary Video Reader Component

Name	video_reader.binary			
Description	Blindly reads any video filestream (e.g. a M4V file) irrespective of format and emits contained elementary video stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	output	Emits video stream found in filestream.

9.6.4 Video Scheduler Class

Scheduler by definition SHALL support being slave in A/V synchronization. Scheduler may not be required to provide reference clock updates.

Name	video_scheduler			
Description	Times the delivery of video frames according to their timestamps.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Accepts video.
	VPB+1	video	output	Emits timed video.
	OPB+0	other/time	input	Accepts time updates.

9.6.4.1 Video Scheduler Component

Name	video_scheduler.binary			
Description	Times the delivery of video frames according to their timestamps.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Accepts video.
	VPB+1	video	output	Emits timed video.
	OPB+0	other/time	input	Accepts time updates.

Port Index	OPB+0		
Description	Accepts media time updates to facilitate accurate emission of a frame at the timestamp for the frame (i.e. in the buffer header). Also provides mechanism for video scheduler to query for media time.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexConfigTimeRenderingDelay`	Read /Write	Queries / specifies the new rendering delay

9.6.5 Video Writer Class

Name	video_writer			
Description	Writes given video stream to a video filestream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Accepts video stream to be written to the video filestream.

9.6.5.1 Binary Video Writer Class

Name	video_writer.binary			
Description	Blindly writes given elementary video stream to a video filestream (e.g. an M4V file) irrespective of format.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Accepts video stream to be written to the video filestream.

9.7 Other Standard Components

9.7.1 Camera Class

Video capture components are not mandated to provide reference clock updates.

Although capturing port bits are defined separately for image and video capture they are independent from the bOneShot boolean used to differentiate between image and video modes. By doing so, it becomes possible to capture images in video mode, or to get camera frames of different resolution than preview frames in image mode. If a camera component does not support such advanced features, it returns OMX_ErrorUnsupportedSetting error to IL client if he tries to trigger a capture in a different camera mode.

In case of video capture - image capture can be triggered concurrently on IPB+0, and in case of still image capture - camera frames of different characteristics than preview/viewfinder frames on VPB+0 can be concurrently available on VPB+1.

Name	camera

Name	camera			
Description	Emits preview/viewfinder video and captured video according to settings.			
Ports	Index	Domain	Direction	Description
	VPB+0	video	output	Emits preview/viewfinder video.
	VPB+1	video	output	Emits captured video.
	IPB+0	image	output	Emits captured images.
	OPB+0	other/time	input	Receives media time update/provides access to clock component.

9.7.1.1 YUV Camera Component

Name	camera.yuv			
Description	Emits preview/viewfinder video and captured video according to settings.			
Ports	Index	Domain	Direction	Description
	VPB+0	video	output	Emits preview/viewfinder video.
	VPB+1	video	output	Emits captured video.
	IPB+0	image	output	Emits captures images.
	OPB+0	other/time	input	Receives media time update/provides access to clock component.

Port Index	OMX_ALL		
Description	Properties that apply to all ports.		
Required Parameters/ Configs	Index	Access	Description
	`OMX_IndexParamCommonSensorMode`	r/w	Specifies the sensor mode. The camera resolution should not be changed by the IL client. So the camera may change the sensor resolution as needed to satisfy the resolution specified on the output ports.
	`OMX_IndexConfigCommonWhiteBalance`	r/w	Specifies white balance
	`OMX_IndexConfigCommonDigitalZoom`	r/w	Specifies digital zoom
	`OMX_IndexConfigCommonExposureValue`	r/w	Specifies exposure value compensation
	`OMX_IndexAutoPauseAfterCapture`	r/w	Specifies whether the camera will automatically transition to `OMX_StatePause` after the Capturing boolean is cleared (e.g. to facilitate a frozen viewfinder).

Port Index	VPB+0		
Description	Emits preview/viewfinder video when the camera component is executing.		
Required	Index	Access	Description

Port Index	VPB+0		
Parameters/ Configs	`OMX_IndexParamPortDefinition`	r/w	Specifies preview's resolution. nFrameWidth = *320* nFrameHeight = *240* eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*
	`OMX_IndexParamVideoPortFormat`	r/w	eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

Port Index	VPB+1		
Description	Emits captured video.		
Formats	OMX_VIDEO_CodingUnused		
Required	**Index**	**Access**	**Description**

Port Index	VPB+1		
Parameters/ Configs	`OMX_IndexParamPortDefinition`	r/w	Specifies emitted video's resolution. nFrameWidth = *640* nFrameHeight = *480* eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*
	`OMX_IndexParamVideoPortFormat`	r/w	eCompressionFormat = OMX_VIDEO_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*
	`OMX_IndexConfigCommonPortCapturing`	r/w	Capturing port bit that controls video capture. `OMX_CONFIG_PORTBOOLEANTYPE` Port specific capture boolean.

Port Index	IPB+0		
Description	Emits captured images.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specifies image's resolution. nFrameWidth = 640 nFrameHeight = *480* eCompressionFormat = OMX_IMAGE_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*
	`OMX_IndexParamImagePortFormat`	r/w	eCompressionFormat = OMX_IMAGE_CodingUnused eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

Port Index	IPB+0		
	OMX_IndexConfigCommonPortCapturing	r/w	Capturing port bit that controls image capture. OMX_CONFIG_PORTBOOLEANTYPE Port specific capture boolean.

Port Index	OPB+0
Description	Accepts media time updates. Provides mechanism for camera component to query for media time. Camera component can detect drift between camera clock and media clock (which may use the audio capturer as a master) and correct timestamps on outgoing frames to compensate. In use case where two cameras are used (e.g. one pointed at user and one pointed away) this provides a consistent media time for timestamps across switches between cameras during capture.

9.7.1.2 Video Capture Use Case

An IL client using a camera to capture a video stream may do so via the following steps:

1. Instantiate the camera component and any co-operating components.
2. Set camera parameters:
    a. Set capture port resolution and frame-rate according to desired values of captured stream
    b. Set viewfinder port resolution and frame-rate (e.g. according to desired values of preview window)
    c. Clear the one shot bit of the sensor mode to indicate that the camera should emit a stream of multiple frames, i.e. a video stream. The IL client should leave the sensor resolution at the default allowing the camera to pick a sensor resolution appropriate to the resolution settings of the viewfinder and capture ports.
    d. Set other camera settings (e.g. exposure value compensation, white balance, zoom, etc).
    e. Set or clear auto pause after capture accordingly. If auto pause is set the component will pause and the viewfinder will freeze after a capture.
3. Establish any necessary tunnels between the camera component and other components (e.g. a display component tunneling with the viewfinder port or a video encoder tunneling with the capture port).
4. Select the clock component's active reference clock. If the camera is used in concert with an audio capturer the audio clock will be the active reference clock (i.e. be the master clock) to facilitate synchronized audio/video capture. Otherwise the video clock provided by the camera will be the active reference clock.

5. Transition all components to the `OMX_StateIdle` state and then to the `OMX_StateExecuting` state. The viewfinder port should now be actively emitting preview frames.
6. To initiate video capture set the capturing bit. The capture port will emit captured frames at the frame rate specified. If using a clock component start the clock component. Timestamps applied to video frames will follow the media time to facilitate consistent timestamp authoring between audio and video capture. The viewfinder will continue to emit frames.
7. To terminate video capture clear the capturing bit. The capture port will cease the emission of frames. If set to auto pause the component will pause and the viewfinder will cease the emission of frames. This effectively freezes any associated preview window to the last frame emitted which should be identical to the last frame emitted by the capture port. If auto pause is clear then the viewfinder continues emitting preview frames.
8. If the component is paused and the viewfinder is frozen after a capture then the IL client manually unfreezes the viewfinder by transitioning the component to `OMX_StateExecuting` when appropriate (e.g. after the captured video has been stored by the application).

Note that this sequence of calls can also be used to implement a sequence of consecutive image captures. In the case of a sequence of stills the IL client simply sets the frame rate on the capture port to accommodate the desired interim between captured stills, uses a JPEG encoder instead of an MPEG encoder, and terminates the capture after the desired number of stills have been captured.

9.7.1.3 Still Image Capture

An IL client using a camera to capture an image may do so via the following steps:

1. Instantiate the camera component and any co-operating components
2. Set camera parameters:

    a. Set capture port resolution according to desired values of captured image.
    b. Set viewfinder port resolution and frame-rate (e.g. according to desired values of preview window).
    c. Set the one shot bit of the sensor mode to indicate that the camera should emit a single frame, i.e. an image frame. The IL client should leave the sensor resolution at the default allowing the camera to pick a sensor resolution appropriate to the resolution settings of the viewfinder and capture ports.
    d. Set other camera settings (e.g. exposure value compensation, white balance, zoom, etc).
    e. Set or clear auto pause after capture accordingly. If auto pause is set the component will pause and the viewfinder will freeze after a capture.

3. Establish any necessary tunnels between the camera component and other components (e.g. a display component tunneling with the viewfinder port or an image encoder tunneling with the capture port).
4. Transition all components to the `OMX_StateIdle` state and then to the `OMX_StateExecuting` state. The viewfinder port should now be actively emitting preview frames and the capture port is not transmitting any frames, it is paused.
5. With the viewfinder port enabled, the IL client now has the opportunity to perform any zoom and focus related actions.
6. To signal image capture set the capturing bit. The capture port will emit a single captured frame and then the component will immediately clear the capturing bit. If set to auto pause after capture the component will transition itself to the `OMX_StatePause` state and the viewfinder will cease the emission of frames. This effectively freezes any associated preview window to the captured image frame. If auto pause is clear then the viewfinder continues emitting preview frames.
7. If the component is paused and the viewfinder is frozen after a capture then the IL client manually unfreezes the viewfinder by transitioning the component to `OMX_StateExecuting` when appropriate (e.g. after the captured image has been stored by the application).

9.7.2 Clock Class

Name	Clock			
Description	Implements the OpenMAX IL clock component (add reference to existing section in spec describing the clock component), the component may expose support for 1 to N ports.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	OPB+0 to (OPB+N-1)	other/time	output	Emits time updates.

9.7.2.1 Clock Component

Name	clock.binary
Description	Implements the OpenMAX IL clock component.

Port Index	OPB+0 to (OPB+N-1)		
Description	Emits time updates.		
Formats	OMX_OTHER_FormatTime		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexConfigTimeScale`	Read, write	Query or set current scale applied to the media time.

Port Index	OPB+0 to (OPB+N-1)		
	OMX_IndexConfigTimeClockState	Read, write	Query or set current clock state.
	OMX_IndexConfigTimeActiveRefClock	Read, write	Query or set the active reference clock.
	OMX_IndexConfigTimeCurrentMediaTime	Read	Query current media time.
	OMX_IndexConfigTimeCurrentWallTime	Read	Query current wall clock time.
	OMX_IndexConfigTimeCurrentReference	Write	Set the instantaneous reference clock value.
	OMX_IndexConfigTimeMediaTimeRequest	Write	Make a media time request.
	OMX_IndexConfigTimeClientStartTime	Write	Set the start time of a client stream.
	OMX_IndexConfigTimeRenderingDelay	Write	Set the new rendering delay for the clock client

9.7.3 Container Demuxer Class

Name	container_demuxer			
Description	Parses a container filestream, demuxes its elementary streams, and emits them as independent video, image and audio streams.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	output	Emits demuxed audio stream.
	VPB+0	video	output	Emits demuxed video stream.
	OPB+0	other/time	input	Receives media time updates/provides access to clock component.

Port Index	OMX_ALL		
Description	Properties that apply to all ports.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexConfigTimePosition	r/w	Specifies the position in the container format content.
	OMX_IndexConfigTimeSeekMode	r/w	Specifies the manner in which a seek will be carried out (quickly or precisely).
	OMX_IndexParamContentURI	r/w	Specify/query the current target content.

Port Index	APB+0		
Description	Emits demuxed audio stream.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamPortDefinition	r/w	Specify/query the characteristics of the audio stream.

Port Index	APB+0		
	OMX_IndexParamNumAvailableStreams	r	Query the number of available audio streams for this port given current content.
	OMX_IndexParamActiveStream	r/w	Specify/query the active audio stream by index where indices are numbered from 0 to the number of available streams.

Port Index	VPB+0		
Description	Emits demuxed video stream.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamPortDefinition	r/w	Specify/query the characteristics of the video stream.
	OMX_IndexParamNumAvailableStreams	r	Query the number of available video streams for this port given current content.
	OMX_IndexParamActiveStream	r/w	Specify/query the active video stream by index where indices are numbered from 0 to the number of available streams.

Port Index	OPB+0
Description	Accepts media time updates. Provides mechanism for component to query for media time. The demuxer obeys changes in the media time to implement trick modes. For instance a negative media time scale factor indicates rewind which implies the demuxer shall retrieve data in reverse order.

9.7.3.1 Playback Use Case

An IL client using a container parser to playback content may do so via the following steps:

1. Instantiate the container demuxer component.
2. Set any relevant container demuxer settings:
3. Specify the target content
4. set all outputs to autodetect
5. Execute the component until all each port generates an `OMX_EventPortSettingsChanged` event. For each port that generates this event:

    a. Query the number of available streams for that port and examine the properties of each available stream by making each active and reading the port parameters.

b. Make the desired stream active.

6. Instantiate the set of co-operating components appropriate to the format settings of the parser's output ports.
7. Establish any necessary tunnels between the container parser and component and other components (e.g. an audio decoder tunneling with the audio port or a video decoder tunneling with the video port).
8. Select the clock component's active reference clock. In a use case with audio this is normally the audio clock as provided by the audio renderer.
9. Transition all components to the `OMX_StateIdle` state then the `OMX_StateExecuting` state. If using a clock component start the clock component. The container demuxer will emit the relevant elementary streams facilitating playback.
10. To change the playback rate (i.e. facilitate trick modes) change the media clock scale factor to the appropriate value (e.g. 2.0 implies 2x forward playback and -1.0 implies 1x reverse playback). The clock component will inform the container demuxer of the scale change and the demuxer will retrieve and emit data in a manner appropriate the scale (e.g. in reverse for negative scales or skipping interframes in extreme fast forward).
11. To seek to a particular location the IL client sets the position on the container demux.

9.7.3.2 3GP Demuxer Component

The standard 3GP demuxer component shall support Release 6 of the 3GP format including basic profile (all other profiles are optional).

9.7.3.3 ASF Demuxer Component

The standard ASF demuxer component shall support ASF version 1.2, Revision 1.20.03 (dated December 2004)

9.7.3.4 Real Demuxer Component

The standard Real Demuxer shall support parsing of the Real container format.

9.7.4 Container Muxer Class

Name	container_muxer			
Description	Given independent video, image, and audio streams muxes them into a container filestream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	APB+0	audio	input	Accepts audio stream for muxing.
	VPB+0	video	input	Accepts video stream for muxing.

Port Index	OMX_ALL		
Description	Properties that apply to all ports.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamContentURI`	r/w	Specify/query the current target content.

9.7.4.1 3GP Muxer Component

The standard 3GP muxer component shall support Release 6 of the 3GP format including basic profile (all other profiles are optional).

9.7.5 Image/Video Processor Class

Name	iv_processor			
Description	Performs some processing on a raw image/video stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Accepts video for processing.
	VPB+1	video	output	Emits processed video.

Upscaling, downscaling and frame rate conversion support is not mandated. If these features are not supported, the component shall implement the slaving behavior as described in section (ref to slaving behavior section).

9.7.5.1 YUV Image/Video Processor

Name	iv_processor.yuv			
Description	Performs some processing on a raw image/video stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Accepts video for processing.
	VPB+1	video	output	Emits processed video.

Port Index	VPB+0		
Description	Accepts video for processing.		
Required	**Index**	**Access**	**Description**

Port Index	VPB+0		
Parameters/ Configs	`OMX_IndexParamPortDefinition`	r/w	Specify/query the characteristics of the video stream. nFrameWidth = *640* nFrameHeight = *480* eCompressionFormat = *OMX_VIDEO_CodingUnused* eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*
	`OMX_IndexParamVideoPortFormat`	r/w	eCompressionFormat = *OMX_VIDEO_CodingUnused* eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar* *r*

Port Index	VPB+1

Port Index	VPB+1		
Description	Emits processed video.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the characteristics of the video stream. nFrameWidth = *640* nFrameHeight = *480* (output width and height imply scale if different then input width and height) eCompressionFormat = *OMX_VIDEO_CodingUnused* eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

Port Index	VPB+1		
	OMX_IndexParamVideoPortFormat	r/w	eCompressionFormat = *OMX_VIDEO_CodingUnused* eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*
	OMX_IndexConfigCommonRotate	r/w	Specify/query rotation. Rotation is always performed prior to mirror. nRotation = *0* 90 (-270) 180 (-180) 270 (-90)
	OMX_IndexConfigCommonMirror	r/w	eMirror = *OMX_MirrorNone* OMX_MirrorVertical OMX_MirrorHorizontal OMX_MirrorBoth
	OMX_IndexConfigCommonInputCrop	r/w	Cropping shall be specified within frame boundaries: 0<= nLeft <= frame width -1 0<= nTop <= frame height -1 0<= nWidth <= frame width 0<= nHeight <= frame height Cropping is 16-byte aligned.

9.7.6 Image/Video Renderer Class

Name	iv_renderer			
Description	Displays a given raw image/video stream.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Accepts video for display.

Common to all renderers:

Port Index	VPB+0		
Description	Accepts video rendering.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexConfigCommonRotate`	r/w	Specify/query rotation. Rotation is always performed prior to mirror. nRotation = *0* 90 (-270) 180 (-180) 270 (-90)
	`OMX_IndexConfigCommonMirror`	r/w	eMirror = *OMX_MirrorNone* OMX_MirrorVertical OMX_MirrorHorizontal OMX_MirrorBoth
	`OMX_IndexConfigCommonScale`	r/w	xWidth = downscale factors of 2 xHeight = downscale factors of 2
	`OMX_IndexConfigCommonInputCrop`	r/w	Cropping shall be specified within frame boundaries: 0<= nLeft <= frame width -1 0<= nTop <= frame height -1 0<= nWidth <= frame width 0<= nHeight <= frame height Cropping is 16-byte aligned.
	`OMX_IndexCofigTimeRenderingDelay`	r	Queries the rendering delay

9.7.6.1 YUV Overlay Image/Video Renderer

Name	iv_renderer.yuv.overlay			
Description	Displays a given raw yuv image/video stream using overlays.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Accepts video for display.

Port Index	VPB+0		
Description	Accepts video rendering.		
Required	**Index**	**Access**	**Description**

Port Index	VPB+0		
Parameters/ Configs	`OMX_IndexParamPortDefinition`	r/w	Specify/query the characteristics of the video stream. nFrameWidth = *176* nFrameHeight = 220 eCompressionFormat = *OMX_VIDEO_CodingUnused* eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*
	`OMX_IndexParamVideoPortFormat`	r/w	eCompressionFormat = *OMX_VIDEO_CodingUnused* eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

9.7.6.2 YUV BLTter Image/Video Renderer

Name	iv_renderer.yuv.blter			
Description	Displays a given raw yuv image/video stream via bitBLTs.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Accepts video for display.

Port Index	VPB+0		
Description	Accepts video rendering.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the characteristics of the video stream. nFrameWidth = *176* nFrameHeight = 220 eCompressionFormat = *OMX_VIDEO_CodingUnused* eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

Port Index	VPB+0		
	OMX_IndexParamVideoPortFormat	r/w	eCompressionFormat = *OMX_VIDEO_CodingUnused* eColorFormat = At least one of the following: *OMX_COLOR_FormatYUV420Planar* *OMX_COLOR_FormatYUV420PackedPlanar* *OMX_COLOR_FormatYUV420SemiPlanar* *OMX_COLOR_FormatYUV420PackedSemiPlanar* *OMX_COLOR_FormatYVU420Planar* *OMX_COLOR_FormatYVU420PackedPlanar* *OMX_COLOR_FormatYVU420SemiPlanar* *OMX_COLOR_FormatYVU420PackedSemiPlanar*

9.7.6.3 RGB Overlay Image/Video Renderer

Name	iv_renderer.rgb.overlay			
Description	Displays a given raw rgb image/video stream using overlays.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Accepts video for display.

Port Index	VPB+0		
Description	Accepts video rendering.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	OMX_IndexParamPortDefinition	r/w	Specify/query the characteristics of the video stream. nFrameWidth = *176* nFrameHeight = 220 eCompressionFormat = *OMX_VIDEO_CodingUnused* eColorFormat = *OMX_COLOR_Format16bitRGB565*
	OMX_IndexParamVideoPortFormat	r/w	eCompressionFormat = *OMX_VIDEO_CodingUnused* eColorFormat = *OMX_COLOR_Format16bitRGB565*

9.7.6.4 RGB BLTter Image/Video Renderer

Name	iv_renderer.rgb.blter			
Description	Displays a given raw rgb image/video stream vis bitBLTS.			
Ports	**Index**	**Domain**	**Direction**	**Description**
	VPB+0	video	input	Accepts video for display.

Port Index	VPB+0		
Description	Accepts video rendering.		
Required Parameters/ Configs	**Index**	**Access**	**Description**
	`OMX_IndexParamPortDefinition`	r/w	Specify/query the characteristics of the video stream. nFrameWidth = *176* nFrameHeight = 220 eCompressionFormat = *OMX_VIDEO_CodingUnused* eColorFormat = *OMX_COLOR_Format16bitRGB565*
	`OMX_IndexParamVideoPortFormat`	r/w	eCompressionFormat = *OMX_VIDEO_CodingUnused* eColorFormat = *OMX_COLOR_Format16bitRGB565*

10 Implementing Buffer Sharing

Buffer sharing is implemented on a tunnel within a component and is transparent to other components. The non-supplier port is unaware whether the supplier's component allocated the buffers itself or re-used buffers from another of its ports. Furthermore, the supplier is unaware of whether the non-supplier's component will re-use the buffers that the supplier provided.

A tunnel between any two ports represents a dependency between those ports. Buffer sharing extends that dependency so that all ports that share the same set of buffers form an implicit dependency chain. Exactly one port in that dependency chain allocates the buffers shared by all of them.

If a component chooses to share buffers, its implementation may fulfill the tunnels requirements by doing the following:

- Provide re-used buffers on some supplier ports.
- Account for the needs of shared ports when communicating buffer requirements on ports.
- Internally pass a buffer from an input port to an output port between an `OMX_EmptyThisBuffer` call and its corresponding `EmptyBufferDone` call.

OpenMAX IL defines external component semantics to be compatible with sharing, although it does not explicitly require that a component support sharing. This section discusses the implementation of those semantics in the context of buffer sharing. If no components are sharing buffers, the implementation reduces to a simpler set of steps and obligations.

10.1.1.1 Component Transition from Loaded to Idle State with Sharing

During the `OMX_SetupTunnel` call, the two ports of a tunnel establish which port (input or output) will act as the buffer supplier. Thus, when a component is commanded to transition from loaded to idle, it is aware of the roles of all its supplier or non-supplier ports.

When commanded to transition from loaded to idle, a component performs the following operations in this order:

1. The component determines what buffering sharing it will implement, if any. The following rules apply:
    a) A component may re-use a buffer only from one of its input ports on one or more of its output ports or from one of its output ports on one of its input ports.
    b) Only a supplier port may re-use the buffers from another port.
    c) A component sharing buffers over multiple output ports requires read-only output port as shown in Figure 10-1.

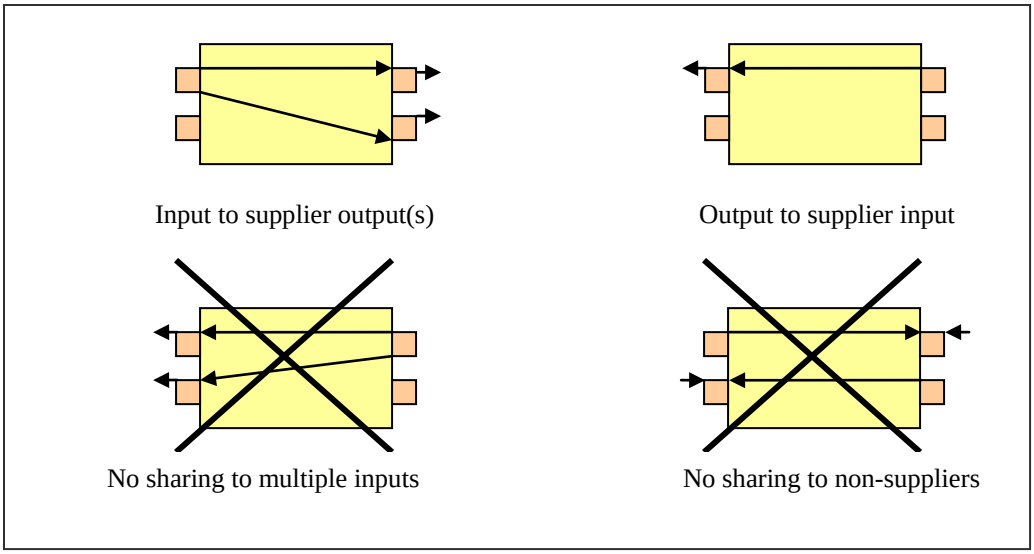


Figure 10-1. Possible Sharing Relationships

2. The component determines which of its supplier ports, if any, are also allocator ports. A supplier port is also an allocator port only if it does not re-use buffers from a non-supplier port on the same component (i.e., is not a sharing port).

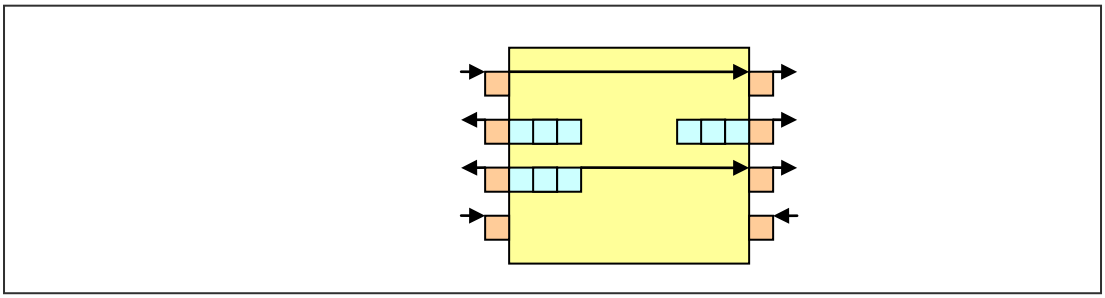


Figure 10-2. Determining Allocators: a supplier port is a port with an arrow pointing away. A non-supplier port is a port with an arrow pointing toward it. An arrow from one port represents a sharing relationship. A port with boxes (buffers) adjacent to it represents an allocator port.

3. The component allocates its buffers for each of its allocator ports as follows:
    a) For each port that re-uses the allocator ports buffer, the allocator port determines the buffer requirements of the sharing port. See obligation A below.
    b) The allocator port determines the buffer requirements of its tunneled port via an `OMX_GetParameter` call. See obligation B below.
    c) The allocator port allocates buffers according to the maximum of its own requirements, the requirements of the tunneled port, and the requirement of all of the sharing ports.
    d) The allocator port informs the tunneled non-supplier port of the actual number of buffers, via an `OMX_SetParameter` call on

OMX_IndexParamPortDefinition by setting the value of nBufferCountActual appropriately. See obligation E below.

e) The allocator port shares its buffers with each sharing port that re-uses its buffers. See obligation D below.

f) For every allocated buffer, the allocator port calls OMX_UseBuffer on its tunneling port after receiving the notification as defined in Section 3.1.3.13. See obligation C below.

A component shall also fulfill the following obligations:

A. For a sharing port to determine its requirements, the sharing port shall first call OMX_GetParameter on its tunneled port to query for requirements and then return the maximum of its own requirements and the requirements of the tunneled ports. This request shall be made after receiving the OMX_PORTSTATUS_ACCEPTUSEBUFFER notification.

B. When a non-supplier port receives an OMX_GetParameter call querying its buffer requirements, the non-supplier port shall first determine the requirements of all ports that re-use its buffers (see obligation A) and then return the maximum of its own requirements and those of its ports.

C. When a non-supplier port receives an OMX_UseBuffer call from its tunneled port, the non-supplier port shall share the buffer with all ports on that component that re-use it.

D. When a port A shares a buffer with a port B on the same component where port B re-uses the buffer of port A, then port B shall call OMX_UseBuffer and pass the buffer on its tunneled port.

E. When a non-supplier port receives a OMX_SetParameter call on OMX_IndexParamPortDefinition from its tunneled port, the non-supplier port shall pass the nBufferCountActual field to any port that re-uses its buffers. Likewise, each supplier port that receives the nBufferCountActual field in this way shall pass the nBufferCountActual to its tunneled port by performing an OMX_SetParameter call on OMX_IndexParamPortDefinition. The actual number of buffers used throughout the dependency chain is propagated in this way.

A component may transition from loaded to idle when all enabled ports have all the buffers they require.

In practice, there could be a direct mapping between the following:

- Steps 1–3 discussed earlier and code in the loaded-to-idle case in the state transition handler
- Obligation A and a subroutine to determine a shared ports buffer requirements
- Obligation B and the OMX_GetParameter implementation
- Obligation C and the OMX_UseBuffer implementation

- Obligation D and a subroutine to share a buffer from one port to another

To clarify why conformity to these steps and obligations leads to proper buffer allocation, consider the example illustrated in Figure 10-3. Note that this example is contrived to exercise every step and obligation outlined above, and is therefore more complex then most real use cases.

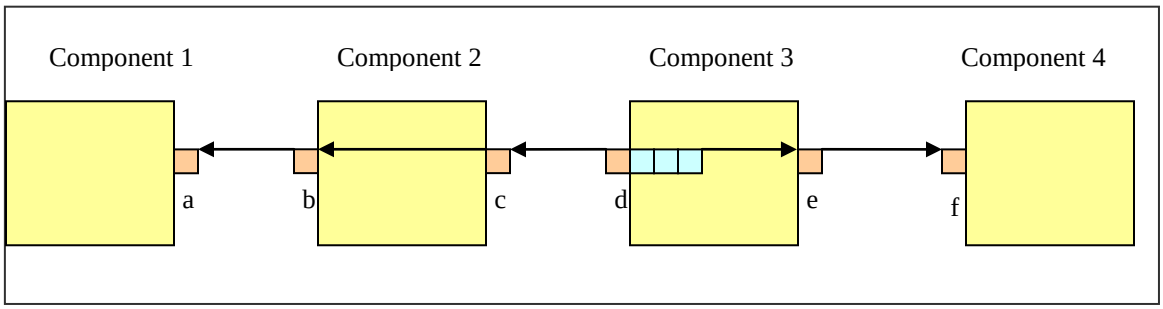


Figure 10-3. Example of Buffer Allocation with Sharing

This discussion focuses only on the transition of component 3 to idle; similar operations occur inside the other components.

When the IL client commands component 3 to transition from loaded to idle, it follows the following prescribed steps:

1. Component 3 notices that it can re-use port d's buffers since port e is a supplier port. Component 3 establishes a sharing relationship from port d to port e.
2. Component 3 decides that since port d is a supplier port that does not re-use buffers, port d shall be an allocator port.
3. Component 3 allocates and distributes port d's buffers:
    a) Since port e will re-use the buffer of port d, component 3 determines the buffer requirements of port e. In accordance with obligation A, port e calls `OMX_GetParameter` on port f to determine its buffer requirements and reports the requirements as the maximum between its own and those of port f.
    b) Port d calls `OMX_GetParameter` on port c to determine its buffer requirements. In accordance with obligation B, port c shall determine the buffer requirements of port b. In accordance with obligation A, port b returns the maximum of its own requirements and the requirement of port a (retrieved via `OMX_GetParameter`) when queried. Port c then returns the maximum of its own requirements and the requirements that port b returns.
    c) Port d allocates buffers according to the maximum of its own requirements and the requirements that ports c and e return. The resulting buffers are effectively allocated according to the maximum requirements of ports a, b, c, d, e, and f, all of which use the buffers of port d.

d) Since port e will re-use the buffers of port d, component 3 shares these buffers with port e. In accordance with obligation D, port e calls `OMX_UseBuffer` on port f for every buffer that is shared.

e) For each buffer allocated, port d calls `OMX_UseBuffer` on port c. In accordance with obligation C, port c shares each buffer with port b. Port b, in turn, obeys obligation D and calls `OMX_UseBuffer` on port a with the buffer.

Since all ports of all components now have their buffers, all components may transition to idle.

10.1.1.2 Protocol for Using a Shared Buffer

When an input port receives a shared buffer via an `OMX_EmptyThisBuffer` call, the input port may re-use that buffer on an output port by obeying the following rules:

- The output port calls `OMX_EmptyThisBuffer` on its tunneling port before the input port sends the corresponding `EmptyBufferDone` call to its tunneling port.
- The input port does not call `EmptyBufferDone` until all output ports on which the buffer is shared (i.e., via `OMX_EmptyThisBuffer` calls) return `EmptyBufferDone`.

11 Appendix A – References

This appendix identifies provides references to documentation on standards and formats presented in this document. The hyperlinks provide access to documents stored on various websites. The references are organized according to the applicable type of media.

11.1 SPEECH

11.1.1 3GPP

AMR-NB	3G TS 26.071 "AMR speech Codec; General Description", Generation Partnership Project (3GPP). And references therein.
AMR-WB	3G TS 26.171 "AMR Wideband Speech Codec; General Description", Generation Partnership Project (3GPP). And references therein.
GSM-EFR	3G TS 46.051 "Enhanced Full Rate (EFR) speech processing functions; General description", Generation Partnership Project (3GPP). And references therein.
GSM-FR	3G TS 46.001 "Full rate speech; Processing functions", Generation Partnership Project (3GPP). And references therein.
GSM-HR	3G TS 46.002 "Half rate speech; Processing functions", Generation Partnership Project (3GPP). And references therein.

11.1.2 3GPP2

SMV	3GPP2-SMV, “Selectable Mode Vocoder (SMV) Service Option for Wideband Spread Spectrum Communication Systems”, 3GPP2 C.S0030-0, 2004.

11.1.3 ARIB

PDC-EFR	RCR-27 EFR, “RCR-27-1: Personal Digital Cellular Telecommunication System,” sec. 5.4, 2003.
PDC-FR	RCR-27 FR, “RCR-27-1: Personal Digital Cellular Telecommunication System,” sec. 5.1, 2003.
PDC-HR	RCR-27 HR, “RCR-27-1: Personal Digital Cellular Telecommunication System,” sec. 5.2, 2003.

11.1.4 ITU

G.711	ITU-G711, “Pulse code modulation (PCM) of voice frequencies “, 1988.
G.723.1	ITU-G.723.1, “Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s”, 1996.
G.726	ITU-G.726, “40, 32, 24, 16 kbit/s adaptive differential pulse code modulation (ADPCM)”, 1990.

G.729	ITU-G.729, “Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear-prediction (CS-ACELP)”’, 1996.

11.1.5 IETF

RFC3267	RFC3267: Real-Time Transport Protocol (RTP) Payload Format and File Storage Format for the Adaptive Multi-Rate (AMR) and Adaptive Multirate Wideband (AMR WB) Audio Codecs.

11.1.6 TIA

EVRC	ANSI/TIA-127-A-2004, “Enhanced Variable Rate Codec Speech Service Option 3 for Wideband Spread Spectrum Digital Systems,” 2004.
QCELP8	ANSI/TIA/EIA-96-C-98, “Speech Service Option Standard for Wideband Spread Spectrum Systems,” 1998.
QCELP13	ANSI/TIA-733-A-2004, “High Rate Speech Service Option 17 for Wideband Spread Spectrum Communications Systems,” 2004.
TDMA-EFR	ANSI/TIA/EIA-136-410-1-2001, “TDMA Cellular PCS - Radio Interface - Enhanced Full-Rate Voice Codec, Addendum 1,” 2001.
TDMA-FR	ANSI/TIA/EIA-136-420-99, “TDMA Cellular PCS, VSELP,” 1999.

11.2 AUDIO

11.2.1 ISO

HE-AAC v1	ISO/IEC JTC1/SC29/WG11 MPEG, International Standard IS 14496-3 “Coding of Audio-Visual Objects—Part 3: Audio, Amendment 1: Bandwidth extension”, November 2003.
HE-AAC v2	ISO/IEC JTC1/SC29/WG11 MPEG, International Standard IS 14496-3 “Coding of Audio-Visual Objects—Part 3: Audio, Amendment 2: Parametric coding for high-quality audio”, August 2004.
MPEG-1 Audio	ISO/IEC JTC1/SC29/WG11 MPEG, International Standard IS 11172-3 "Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s, Part 3: Audio", 1993.
MPEG-2 Audio	ISO/IEC JTC1/SC29/WG11 MPEG, International Standard IS 13818-3 “Information Technology - Generic Coding of Moving Pictures and Associated Audio, Part 3: Audio", 1998.
MPEG-2 AAC	ISO/IEC JTC1/SC29/WG11 MPEG, International Standard IS 13818-7 "Information Technology - Generic Coding of Moving Pictures and Associated Audio, Part 7: MPEG-2 AAC", 2004.
MPEG-4 AAC	ISO/IEC JTC1/SC29/WG11 MPEG, International Standard IS 14496-3 “Coding of Audio-Visual Objects—Part 3: Audio”, 2d Edition, December 2001.

11.2.2 MISC

I3DL2	Interactive 3-D Audio Rendering Guidelines - Level 2.0, Revision 1.0a. Interactive Audio Special Interest Group, September 20, 1999.
SBC	de Bont, F., Groenewegen, M., and Oomen, W., "A High Quality Audio Coding System at 128 kb/s", 98th AES Convention, Feb. 25-28, 1995.
WMA	Windows Media Audio
VORBIS	Vorbis codec
RA	Real Audio 10 Codec
PCM	Pulse-code Modulation
ADPCM	Adaptive Differential PCM
RFC 1766	Tags for the Identification of Languages (http://www.ietf.org/rfc/rfc1766.txt)
ISO 639	Codes for the Representation of Names of Languages (http://www.iso.org/iso/en/StandardsQueryFormHandler.StandardsQueryFormHandler?scope=CATALOGUE&keyword=&isoNumber=639&sortOrder=ISO&title=true&search_type=ISO&search_term=639&languageCode=en)
ISO 3166	Codes for the Representation of Names of Countries and their Subdivisions (http://www.iso.org/iso/en/StandardsQueryFormHandler.StandardsQueryFormHandler?scope=CATALOGUE&keyword=&isoNumber=3166&sortOrder=ISO&title=true&search_type=ISO&search_term=3166&languageCode=en)

11.3 SYNTHETIC AUDIO

11.3.1 MIDI

DLS 1	Downloadable Sounds Level 1 Specification, Version 1.1a, RP-016. MIDI Manufacturers Association, Los Angeles, CA, USA, January 1999.
DLS 2	Downloadable Sounds Level 2 Specification, Version 1.0c, RP-025. MIDI Manufacturers Association, Los Angeles, CA, USA, July 14 1999. Downloadable Sounds Level 2.1 Specification (RP-025/Amd1), MIDI Manufacturers Association, Los Angeles, CA, USA, January 2001.
General MIDI	The Complete MIDI 1.0 Detailed Specification, Document version 96.1, MIDI Manufacturers Association, Los Angeles, CA, USA, 1996 (Contains MIDI 1.0 Detailed Specification, MIDI Time Code, Standard MIDI Files 1.0, General MIDI System Level 1, MIDI Show Control 1.1, and MIDI Machine Control)
General MIDI 2	General MIDI Level 2 Specification (Recommended Practice), v 1.1 (updated), RP-024. MIDI Manufacturers Association, Los Angeles, CA, USA, September 2003.

GM Lite	General MIDI Lite Specification and Guidelines for Use in Mobile Applications, Version 1.0, RP-033. MIDI Manufacturers Association, Los Angeles, CA, USA, October 5, 2001.
Mobile DLS	Mobile DLS Specification, RP-041, MIDI Manufacturers Association, Los Angeles, CA, USA, 2003.
Mobile XMF (XMF type 2)	Mobile XMF Content Format Specification, RP-042. MIDI Manufacturers Association, Los Angeles, CA, USA, September 2004. XMF Meta File Format 2.0, RP-043. MIDI Manufacturers Association, Los Angeles, CA, USA, September 2004.
SP-MIDI	Scalable Polyphony MIDI Specification, Version 1.0, RP-034. MIDI Manufacturers Association, Los Angeles, CA, USA, February 2002 Scalable Polyphony MIDI Device 5-24 Voice Profile for 3GPP, Version 1.0, RP-035. MIDI Manufacturers Association, Los Angeles, CA, USA, February 2002.
XMF type 0 and 1	Type 0 and 1 XMF Files, RP-031. MIDI Manufacturers Association, Los Angeles, CA, USA, 2001. XMF Meta File Format, Version 1.00b, RP-030. MIDI Manufacturers Association, Los Angeles, CA, USA, October 2001. XMF Meta File Format Updates v1.01, RP-039. MIDI Manufacturers Association, Los Angeles, CA, USA, July 2003.

11.4 IMAGE

11.4.1 IETF

RFC804	IETF/RFC 804, "ITU Group 3 encoding: Modified Huffman and Modified Read compression algorithms."
RFC1314	IETF/RFC 1314, "A File Format for the Exchange of Images in the Internet," 1992.
RFC2035	IETF/RFC 2305, "RTP Payload Format for JPEG-compressed Video," 1996.
RFC2083	IETF/RFC 2083, "PNG (Portable Network Graphics) Specification Version 1.0," 1997.
RFC2160	IETF/RFC 2160, "Carrying PostScript in X.400 and MIME," 1998.
RFC2302	IETF/RFC 2302, "Tag Image File Format (TIFF), image/tiff MIME Sub-type Registration," 1998.
RFC2306	IETF/RFC 2306, "Tag Image File Format (TIFF), F Profile for Facsimile," 1998.
RFC3250	IETF/RFC 3250, "Tag Image File Format Fax Extended (TIFF-FX), image/tiff-fx MIME Sub-type Registration," 2002.
RFC3302	IETF/RFC 3302, "Tag Image File Format (TIFF) - image/tiff MIME Sub-type Registration," 2002.
RFC3362	IETF/RFC 3362, "Real-time Facsimile (T.38), image/t38 MIME Sub-type Registration," 2002.
RFC3745	IETF/RFC 3745, "MIME Type Registrations for JPEG 2000 (ISO/IEC 15444)," 2004.

RFC3950 IETF/RFC 3950, "Tag Image File Format Fax Extended (TIFF-FX), image/tiff-fx MIME Sub-type Registration," 2005.

11.4.2 ISO

JPEG v1 ISO/IEC JTC1/SC29/WG1 JPEG, International Standard IS 10918-1, "Digital compression and coding of continuous-tone still images: Requirements and guidelines," 1994.

JPEG v2 ISO/IEC JTC1/SC29/WG1 JPEG, International Standard IS 10918-1/Cor 1, "JPEG patent information update," 2005.

JPEG v3 ISO/IEC JTC1/SC29/WG1 JPEG, International Standard IS 10918-3, "Digital compression and coding of continuous-tone still images: Extensions," 1997.

JPEG v4 ISO/IEC JTC1/SC29/WG1 JPEG, International Standard IS 10918-3/Amd 1, "Provisions to allow registration of new compression types and versions in the SPIFF header," 1999.

JPEG v5 ISO/IEC JTC1/SC29/WG1 JPEG, International Standard IS 10918-4, "Digital compression and coding of continuous-tone still images: Registration of JPEG profiles, SPIFF profiles, SPIFF tags, SPIFF colour spaces, APPnmarkers, SPIFF compression types and Registration Authorities (REGAUT)," 1999.

JPEG v6 ISO/IEC JTC1/SC29/WG1 JPEG, International Standard IS 11544, "Coded representation of picture and audio information, Progressive bi-level image compression," 1993.

JPEG LS v1 ISO/IEC JTC1/SC29/WG1 JPEG LS, International Standard IS 14495-1, "Lossless and near-lossless compression of continuous-tone still images: Baseline," 1999.

JPEG LS v2 ISO/IEC JTC1/SC29/WG1 JPEG LS, International Standard IS 14495-2, "Lossless and near-lossless compression of continuous-tone still images: Extensions," 2003.

JPEG 2000 v1 ISO/IEC JTC1/SC29/WG1 JPEG 2000, International Standard IS 15444-1, "JPEG 2000 image coding system: Core coding system," Ed. 2, 2004.

JPEG 2000 v2 ISO/IEC JTC1/SC29/WG1 JPEG 2000, International Standard IS 15444-2, "JPEG 2000 image coding system: Extensions," Ed. 1, 2004.

JPEG 2000 v3 ISO/IEC JTC1/SC29/WG1 JPEG 2000, International Standard IS 15444-6, "JPEG 2000 image coding system, Part 6: Compound image file format," Ed. 1, 2003.

JPEG 2000 v4 ISO/IEC JTC1/SC29/WG1 JPEG 2000, International Standard IS 15444-12, "JPEG 2000 image coding system, Part 12: ISO base media file format," Ed. 2, 2005.

11.4.3 ITU

T81 ITU-T T.81, "Digital compression and coding of continuous-tone still images, Requirements and guidelines," 1992.

T82 ITU-T T.82, "Coded representation of picture and audio information, Progressive bi-level image compression," 1993.

T84 v1 ITU-T T.84, "Digital compression and coding of continuous-tone still images:

Extensions," 1996.

T84 v2 ITU-T T.84/Amd 1, "Provisions to allow registration of new compression types and versions in the SPIFF header," 1999.

T85 ITU-T T.85, "Application profile for Recommendation T.82, Progressive bi-level image compression (JBIG coding scheme) for facsimile apparatus," 1995.

T86 ITU-T T.86, "Digital compression and coding of continuous-tone still images: Registration of JPEG Profiles, SPIFF Profiles, SPIFF Tags, SPIFF colour Spaces, APPn Markers, SPIFF Compression types and Registration Authorities (REGAUT)," 1998.

T87 ITU-T T.87, "Lossless and near-lossless compression of continuous-tone still images, Baseline," 1998.

T88 v1 ITU-T T.88, "Coded representation of picture and audio information, Lossy/lossless coding of bi-level images," 2000.

T88 v2 ITU-T T.88/Amd 1, "Encoder," 2003.

T88 v3 ITU-T T.88/Amd 2, "Extension of adaptive templates for halftone coding," 2003.

T89 ITU-T T.89, "Application profiles for Recommendation T.88, Lossy/lossless coding of bi-level images (JBIG2) for facsimile," 2001.

11.4.4 JEITA

EXIF JEITA, Japanese Electronics and Information Technology Industries Association, "EXIF (Exchangeable Image File Format) 2.2", 2002.

11.4.5 MIPI

CSI MIPI Camera WG, "CSI 2.0 Protocol Specification v.0.41", 2005.

DSI MIPI Display WG, "DSI Specification v.0.45", 2005.

11.4.6 Miscellaneous

BMP Microsoft Windows Bitmap (BMP) Format.

GIF87A GIF 87a, "Graphics Interchange Format, Version 87a," 1987.

GIF89A GIF 89a, "Graphics Interchange Format, Version 89a," 1989.

TIFF TIFF V.6.0, "Tagged Image File Format (TIFF) Specification, Version 6.0".

WebP WebP Image Format

11.4.7 SMIA

SMIA CCP2 SMIA CCP2, "Compact Camera Port 2 (CCP2) Specification 1.0."

SMIA CCP2/ER1 SMIA 1.0 CCP2/ER1, "Errata, Part 2 CCP2 Specification."

SMIA FUNC SMIA Functional, "Functional specification 1.0."

SMIA FUNC/ER1 SMIA Functional 1.0/ER1, "Errata for Part 1 Functional Specification."

SMIA CHAR SMIA Characterisation 1.0/V.A, "Characterisation Specification 1.0, Rev A."

SMIA SW/AP SMIA Software And Application 1.0, "Software And Application Specification 1.0."

11.4.8 W3C

PNG Portable Network Graphics (PNG) Specification (Second Edition), "Computer graphics and image processing, Portable Network Graphics (PNG): Functional specification," 2003.

11.5 VIDEO

11.5.1 3GPP

MBMS v1 3GPP TS 26.346 "MBMS Protocols and Codecs," v.1.5.0.

MBMS v2 3GPP TS 22.146 "Technical Specification Group Services and System Aspects; Multimedia Broadcast/Multicast Service." v.6.6.0.

11.5.2 AVS

AVS-M v1 AVS-M: Part 6 Video-Mobility, Stage 1: MMS service

AVS-M v2 AVS-M: Part 6 Video-Mobility, Stage 2: Streaming and conversational services

11.5.3 DLNA

HNv1.0 DLNA HNv1.0, "Home Networked Device Interoperability Guidelines v1.0," 2004.

11.5.4 ETSI

DVB-H v1 ETSI EN 302 304 V.1.1.1, DEN/JTC-DVB-155, "Digital Video Broadcasting (DVB), Transmission System for Handheld Terminals (DVB-H)," 2004.

DVB-H v2 ETSI ETS 300 468, RE/JTC-DVB-18, "Digital Video Broadcasting (DVB), Specification for Service Information (SI) in DVB systems," 1997.

DVB-H v3 ETSI EN 301 192 V.1.4.1, REN/JTC-DVB-157, "Digital Video Broadcasting (DVB), DVB specification for data broadcasting," 2004.

DVB-H v4 ETSI TS 101 154 V.1.7.1, RTS/JTC-DVB-170, "Digital Video Broadcasting (DVB), Implementation guidelines for the use of Video and Audio Coding in Broadcasting Applications based on the MPEG-2 Transport Stream," 2005.

DVB-H v5 ETSI TS 101 154 V.1.5.1, RTS/JTC-DVB-122, "Digital Video Broadcasting (DVB), Implementation guidelines for the use of Video and Audio Coding in Broadcasting Applications based on the MPEG-2 Transport Stream," 2004.

DVB-H v6 ETSI TS 102 005 V.1.1.1, DTS/JTC-DVB-124, "Digital Video Broadcasting

(DVB), Specification for the use of video and audio coding in DVB services delivered directly over IP," 2005.

DVB-H v7 ETSI TS 102 154 V.1.2.1, RTS/JTC-DVB-123, "Digital Video Broadcasting (DVB), Implementation guidelines for the use of Video and Audio Coding in Contribution and Primary Distribution Applications based on the MPEG-2 Transport Stream," 2004.

11.5.5 IETF

RFC1889 IETF RFC 1889, "RTP: A Transport Protocol for Real-Time Applications," 1996.

RFC2032 IETF RFC 2032, "RTP Payload Format for H.261 Video Streams," 1996.

RFC2038 IETF RFC 2038, "RTP Payload Format for MPEG1/MPEG2 Video," 1996.

RFC2190 IETF RFC 2190, "RTP Payload Format for H.263 Video Streams," 1997.

RFC2250 IETF RFC 2250, "RTP Payload Format for MPEG1/MPEG2 Video," 1998.

RFC2429 IETF RFC 2429, "RTP Payload Format for the 1998 Version of ITU-T Rec. H.263 Video (H.263+)," 1998.

RFC2431 IETF RFC 2431, "RTP Payload Format for BT.656 Video Encoding," 1998.

RFC2435 IETF/RFC 2435, "RTP Payload Format for JPEG-compressed Video," 1998.

RFC3189 IETF RFC 3189, "RTP Payload Format for DV (IEC 61834) Video," 2002.

RFC3497 IETF RFC 3497, "RTP Payload Format for Society of Motion Picture and Television Engineers (SMPTE) 292M Video", 2003.

RFC3551 IETF RFC 3551, "RTP Profile for Audio and Video Conferences with Minimal Control," 2003.

RFC3984 IETF RFC 3984, "RTP Payload Format for H.264 Video", 2005.

RFC4629 IETF RFC 4629, "RTP Payload Format for H.263 Video", 2007.

11.5.6 ISO

MPEG-1 Visual ISO/IEC JTC1/SC29/WG11 MPEG, International Standard IS 11172-2, "Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s, Part 2: Video," Ed. 1, 1993.

MPEG-2 Visual ISO/IEC JTC1/SC29/WG11 MPEG, International Standard IS 13818-2, "Generic coding of moving pictures and associated audio information, Part 2: Video," Ed. 2, 2000.

MPEG-4 Visual v1 ISO/IEC JTC1/SC29/WG11 MPEG, International Standard IS 14496-1/Amd 7, “Use of AVC (Advanced Video Coding) in MPEG-4 systems," Ed. 1, 2004.

MPEG-4 Visual v2 ISO/IEC JTC1/SC29/WG11 MPEG, International Standard IS 14496-2, "Coding of audio-visual objects, Part 2: Visual," Ed. 3, 2004.

MPEG-4 Visual v3 ISO/IEC JTC1/SC29/WG11 MPEG, International Standard IS 14496-10, "Coding of audio-visual objects, Part 10: Advanced Video Coding," Ed. 2, 2004.

MPEG-4 Visual v4	ISO/IEC JTC1/SC29/WG11 MPEG, International Standard IS 14496-15, "Coding of audio-visual objects, Part 15: Advanced Video Coding (AVC) file format," Ed. 1, 2004.
MPEG-21 Visual	ISO/IEC JTC1/SC29/WG11 MPEG, International Standard IS TR 21000-1, "Vision, Technologies and Strategy, Part 1," 2001.
MJPEG-2000 v1	ISO/IEC JTC1/SC29/WG1 MJPEG, International Standard IS 15444-3, "JPEG 2000 image coding system, Part 3: Motion JPEG 2000," Ed. 1, 2002.
MJPEG-2000 v2	ISO/IEC JTC1/SC29/WG1 MJPEG, International Standard IS 15444-3/Amd 2, "Motion JPEG 2000 derived from ISO base media file format," Ed. 1, 2003.

11.5.7 ITU

H.261	ITU-T H.261, "Video codec for audiovisual services at p x 64 kbit/s," 1993.
H.262	ITU-T H.262, "Generic coding of moving pictures and associated audio information: Video," 2000.
H.263	ITU-T H.263, "Video coding for low bit rate communication," 2005.
H.264	ITU-T H.264, "Advanced video coding for generic audiovisual services," 2005.

11.5.8 MISC

RV	Real Video 10 Codec
WMV	Windows Media Video
VC1	Society of Motion Picture and Television Engineers, "VC-1 Compressed Video Bitstream Format and Decoding Process", SMPTE 421M.
RFC4425	RTP Payload Format for Video Codec 1 (VC-1).
VP8	VP8 Video Codec

11.6 JAVA

11.6.1 Multimedia

JSR-135	JCP/JSR-135: Mobile Media API 1.1, 2003
JSR-234	JCP/JSR-234: Advanced Multimedia Supplements, 2005

11.6.2 Broadcast

JSR-272	JCP/JSR-272: Mobile Broadcast Service API for Handheld Terminals, 2005