OpenMAX™ Integration Layer Application Programming Interface Specification

Version 1.0

December 16, 2005
Document version 1.10

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 codecs used in embedded and/or mobile devices. The principal goal of the IL is to give codecs 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 codecs 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 codec 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 codecs portability across an array of platforms. The interface abstracts the hardware and software architecture in the system. Each codec 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 codecs themselves may be any combination of hardware or software and are completely transparent to the user. Without a standardized interface of this nature, codec vendors must write to proprietary or closed interfaces to integrate into mobile

devices. In this case, the portability of the codec 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 codecs
- 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 codecs and between codecs themselves

1.2.2 Design Philosophy

As previously stated, the key focus of the OpenMAX IL API is portability of media codecs. 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 most operating systems, this means an existing media framework.

Thus, much of the OpenMAX IL API is defined by requirements generated by the needs of media frameworks. Similarly, the IL is designed to allow the media framework layer to be as lightweight as possible. The result is an interface that is easily pluggable into most software stacks across operating system and framework boundaries. Likewise, several features of media frameworks were perceived to be handled at higher levels and not included in the API. Among these is the issue of file handling, which, if desired, may be easily added to the IL structure outside of the standard.

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 codecs to communicate directly with one another via tunneling affords implementation architectures even greater flexibility and efficiency.

1.2.3 Software Landscape

In most systems, a user-level media framework already exists. The OpenMAX IL API is designed to easily fit below these frameworks with little to no overhead between the interfaces. In most cases, the media framework provided by the operating system can be replaced with a thin layer that simply translates the API. Figure 1-1 illustrates the software landscape for the OpenMAX IL API.

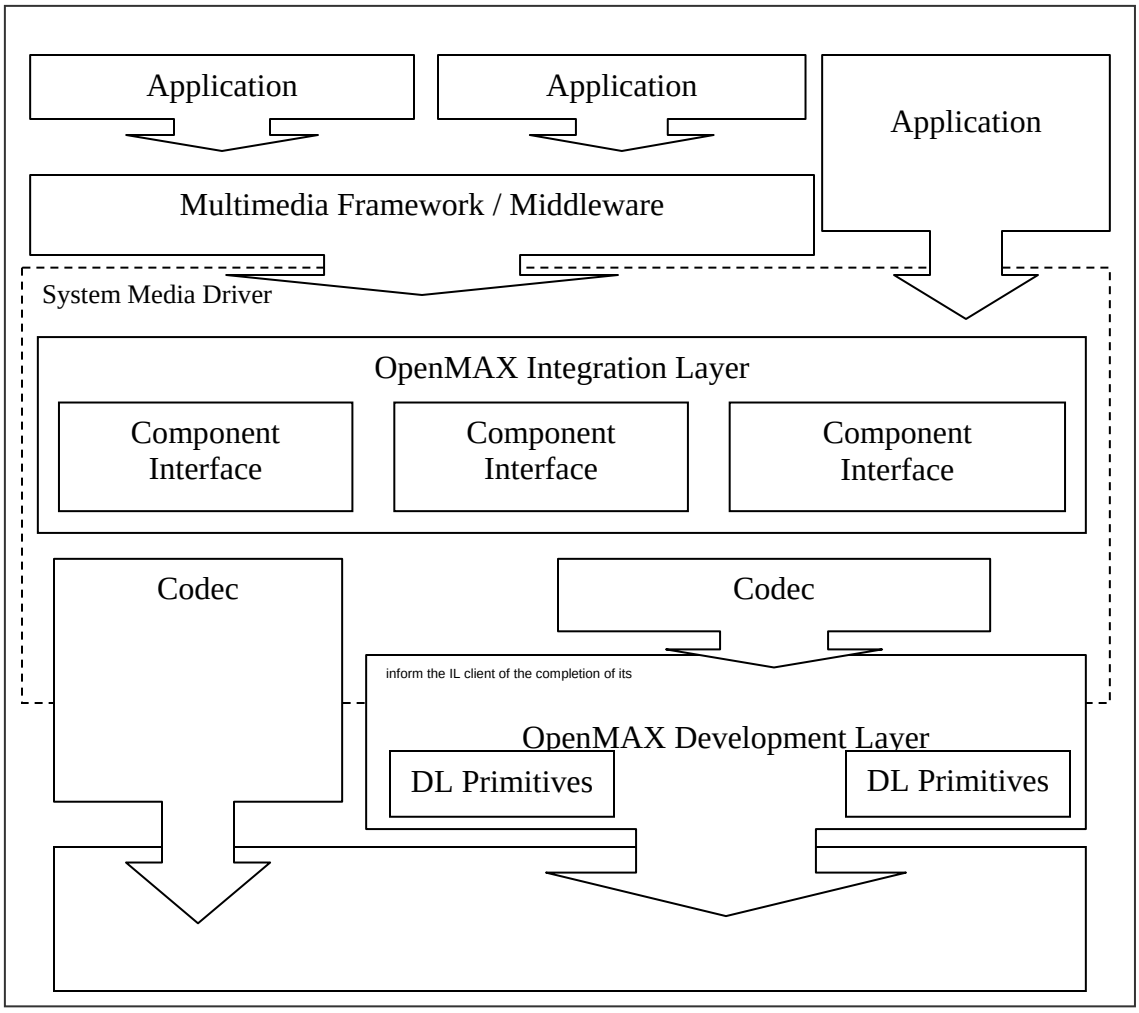


Figure 1-1. OpenMAX IL API Software Landscape

To remove possible reader confusion, the OpenMAX standard also defines a set of Development Layer (DL) primitives on which codecs can be built. The DL primitives and their full relationship to the IL are specified in other OpenMAX 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. The OSV is responsible for conformance testing of the standard reference OpenMAX components.

1.2.4.4 Original Equipment Manufacturers

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

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."

Table 1-1. Definitions of Commonly Used Words

1.4 Authors

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

- Gordon Grigor (ATI)
- Andrew Rostaing (Beatnik)
- Chris Grigg (Beatnik)
- Russell Tillitt (Beatnik)
- Roger Nixon (Broadcom)
- Brian Murray (Freescale)
- Norbert Schwagmann (Infineon)
- Mark Kokes (Nokia)
- Samu Kaajas (Nokia)
- Yeshwant Muthusamy (Nokia)
- Jim Van Welzen (NVIDIA)
- David Siorpaes (STMicroelectronics)
- Diego Melpignano (STMicroelectronics)
- Giulio Urlini (STMicroelectronics)
- Kevin Butchart (Symbian)
- Viviana Dudau (Symbian)
- David Newman (Texas Instruments)
- Leo Estevez (Texas Instruments)
- Richard Baker (Texas Instruments)

2 OpenMAX IL Introduction and Architecture

This section of the document describes the OpenMAX IL features and architecture.

2.1 OpenMAX IL Description

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.

2.1.1 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 or an application. The IL client interacts with a centralized IL entity called the core. The IL client uses the OpenMAX core for loading and unloading components, setting up direct communication between two OpenMAX components, and accessing the component's method functions.

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 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 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 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 component shall have at least one port to claim OpenMAX conformance. Although a vendor may provide an OpenMAX-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 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 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 implementation that provides these functions.

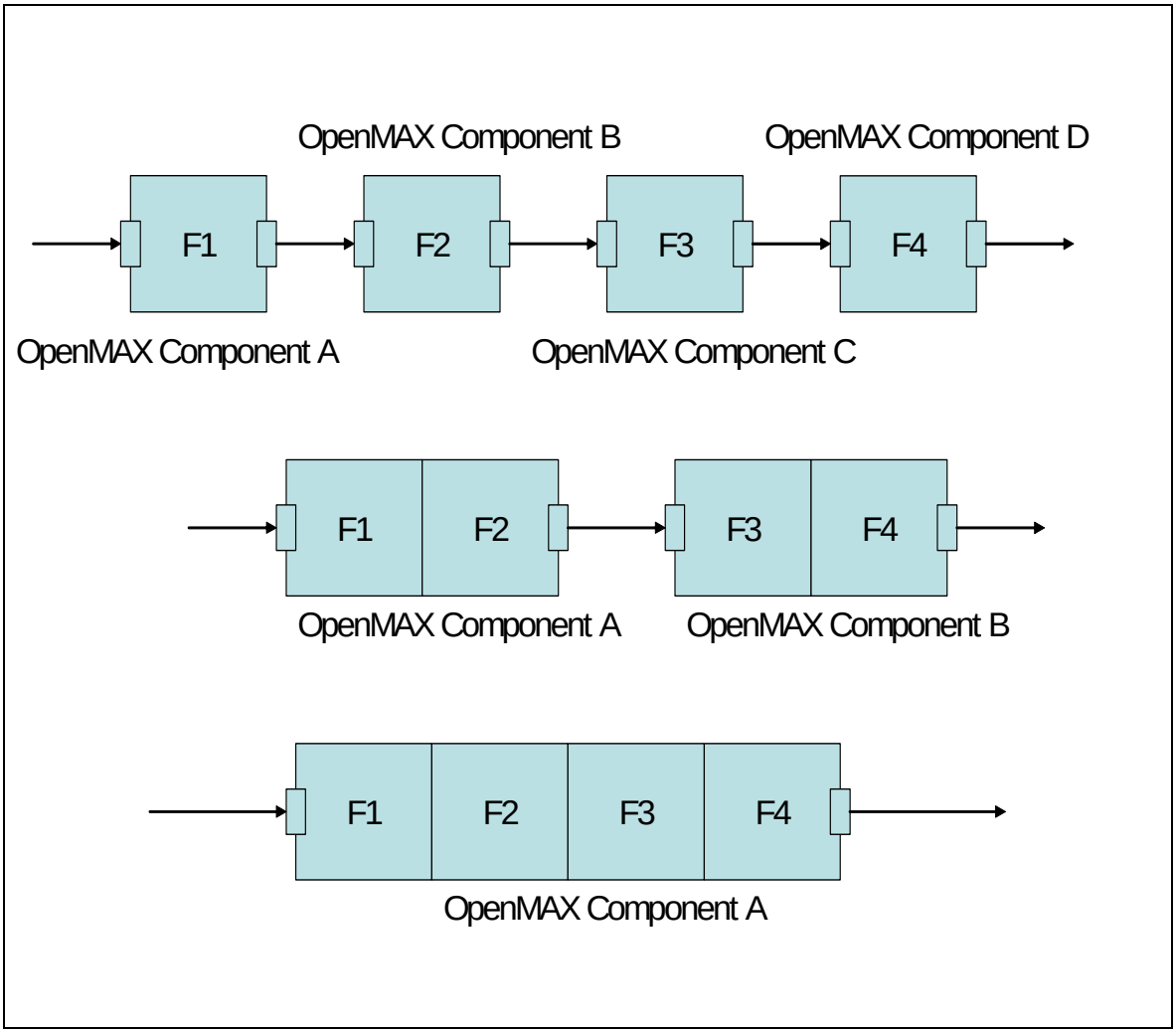


Figure 2-1. Possible Partitions for an OpenMAX Implementation

2.1.2 Key Vocabulary

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

2.1.2.1 Acronyms

Table 2-1 lists acronyms commonly used in describing the OpenMAX IL API.

Acronym	Meaning
IPC	Abbreviation of inter-processor communication.
OMX	Used as a prefix in the names of OpenMAX functions and structures. For example, a component may be place in the OMX_StateExecuting state.

Table 2-1. Acronyms

2.1.2.2 Key Definitions

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

Key word	Meaning
Accelerated component	OpenMAX components that wrap a function with a portion running on an accelerator. Accelerated components have special characteristics such as being able to support some types of tunneling.
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.
AMR	Abbreviation of adaptive multimedia retrieval, which is an adaptive multi-rate codec from the 3GGP consortium.
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 methods.
Main memory	Typically external memory that the host processor and the accelerator share.
OpenMAX component	A component that is intended to wrap functionality that is required in the target system. The OpenMAX wrapper provides a standard interface for the function being wrapped.
OpenMAX core	Platform-specific code that has the functionality necessary to locate and then load an OpenMAX 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.

Key word	Meaning
	In general, after the OpenMAX 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.
RTP	Abbreviation of real-time protocol, which is the Internet-standard protocol for the transport of real-time data, including audio and video.
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 components.

Table 2-2. Key Definitions

2.1.3 System Components

Figure 2-2 depicts the various types of communication enabled with OpenMAX. 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 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.

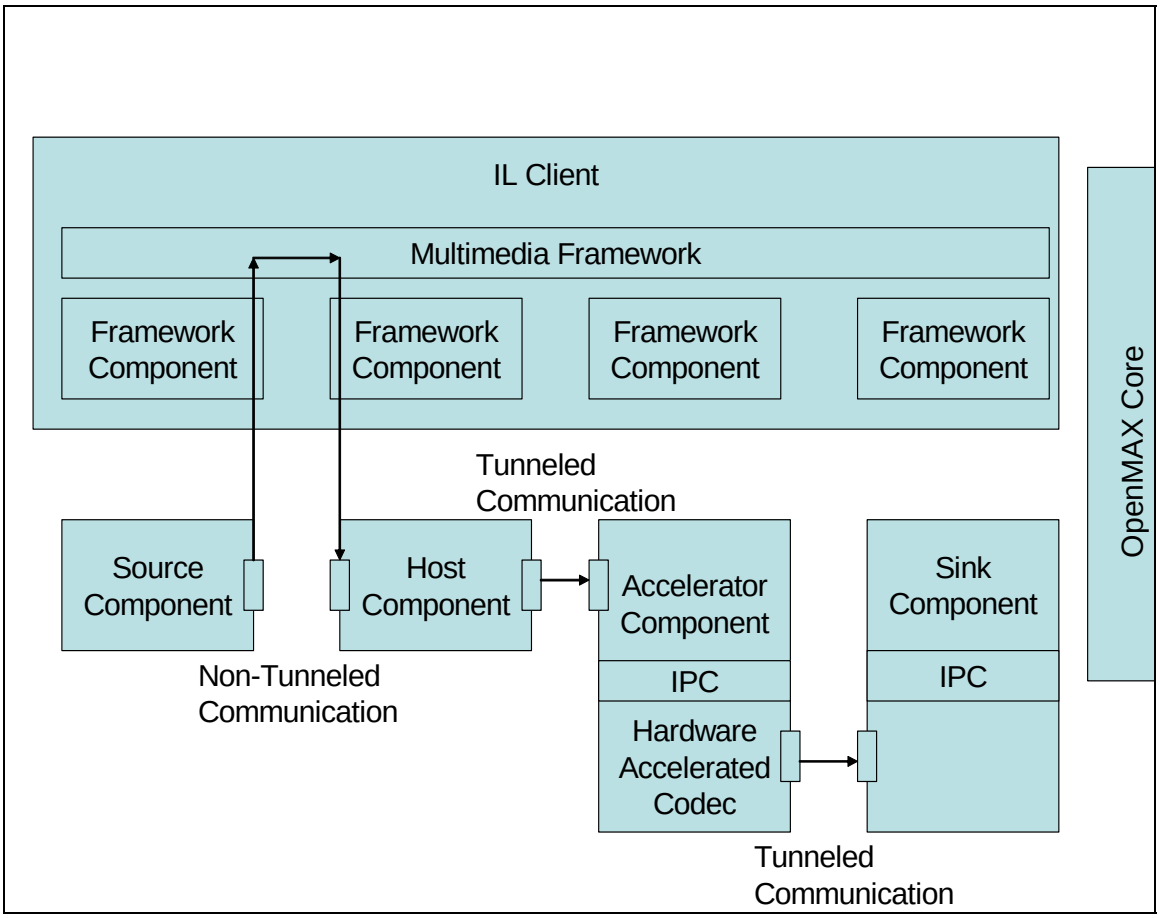


Figure 2-2. OpenMAX IL API System Components

2.1.3.1 Component Profiles

OpenMAX 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 implementers by simplifying the implementation. A base profile component does not need to implement tunneled communication.

2.1.4 Component States

Each OpenMAX 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 core. All other state transitions may then be achieved by communicating directly with the component.

A component can enter an invalid state when a state transition is made with invalid data. For example, if the callback pointers are not set to valid locations, the component may time out and alert the IL client of the error. The IL client shall stop, de-initialize, unload, and reload the component when the IL client detects an invalid state. Figure 2-3 depicts the invalid state as enterable from any state, although the only way to exit the invalid state is to unload and reload the component.

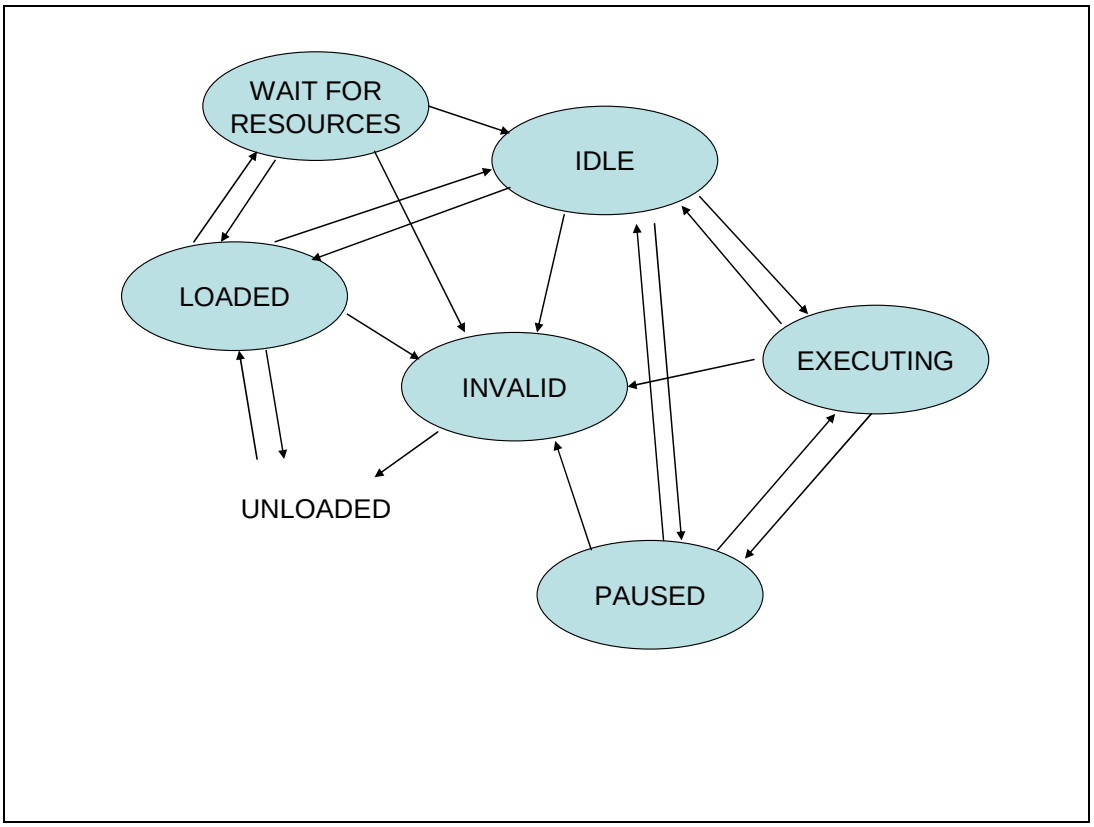


Figure 2-3. Component States

Transitioning into the IDLE state may fail since this state requires allocation of all operational resources. When the transition from LOADED to IDLE fails, the IL client may try again or may choose to put the component into the WAIT FOR RESOURCES state. Upon entering the WAIT FOR RESOURCE state, the component registers with a vendor-specific resource manager to alert it when resources have become available. The resource manager subsequently puts the component into the IDLE state. A command that the IL client sends controls all other state transitions except to INVALID.

The IDLE state indicates that the component has all of its needed resources but is not processing data. The EXECUTING state indicates that the component is pending reception of buffers to process data and will make required callbacks as specified in section 3. The PAUSED state maintains a context of buffer execution with the component without processing data or exchanging buffers. Transitioning from PAUSED to EXECUTING enables buffer processing to resume where the component left off.

Transitioning from EXECUTING or PAUSED to IDLE will cause the context in which buffers were processed to be lost, which requires the start of a stream to be reintroduced. Transitioning from IDLE to LOADED will cause operational resources such as communication buffers to be lost.

2.1.5 Component Architecture

只有一个入口可以控制component，那就是它的handle

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.

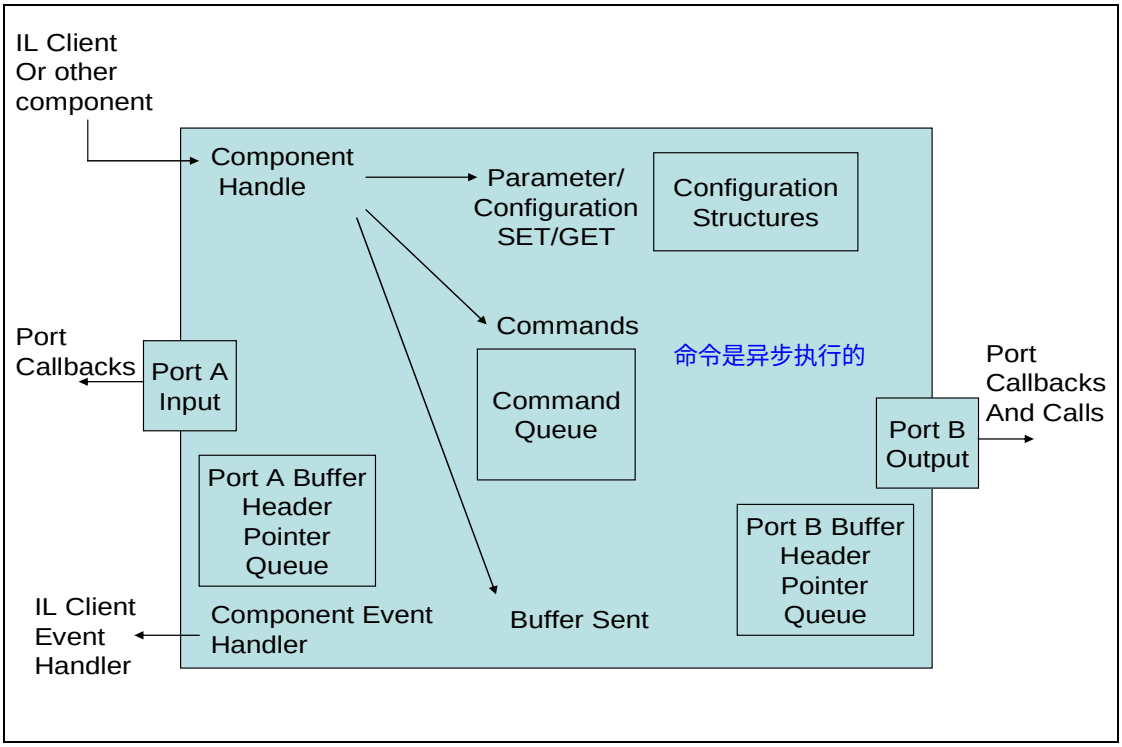


Figure 2-4. OpenMAX IL API Component Architecture

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

2.1.6 Communication Behavior

Configuration of a component may be accomplished once the handle to the component has been received from the OpenMAX 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 are always called from the IL client with `OMX_EmptyThisBuffer` (for more information, see section 3.2.2.17). Output ports are always called from the IL client with `OMX_FillThisBuffer` (for more information, see section 3.2.2.18). In an in-context implementation, callbacks to `OMX_EmptyBufferDone` or `OMX_FillBufferDone` will be made before the return. Figure 2-5 depicts the anticipated behavior for an in-context versus an out-of-context implementation. Note that the IL client should not make assumptions about return/callback sequences to enable heterogeneous integration of in-context and out-of-context OpenMAX components.

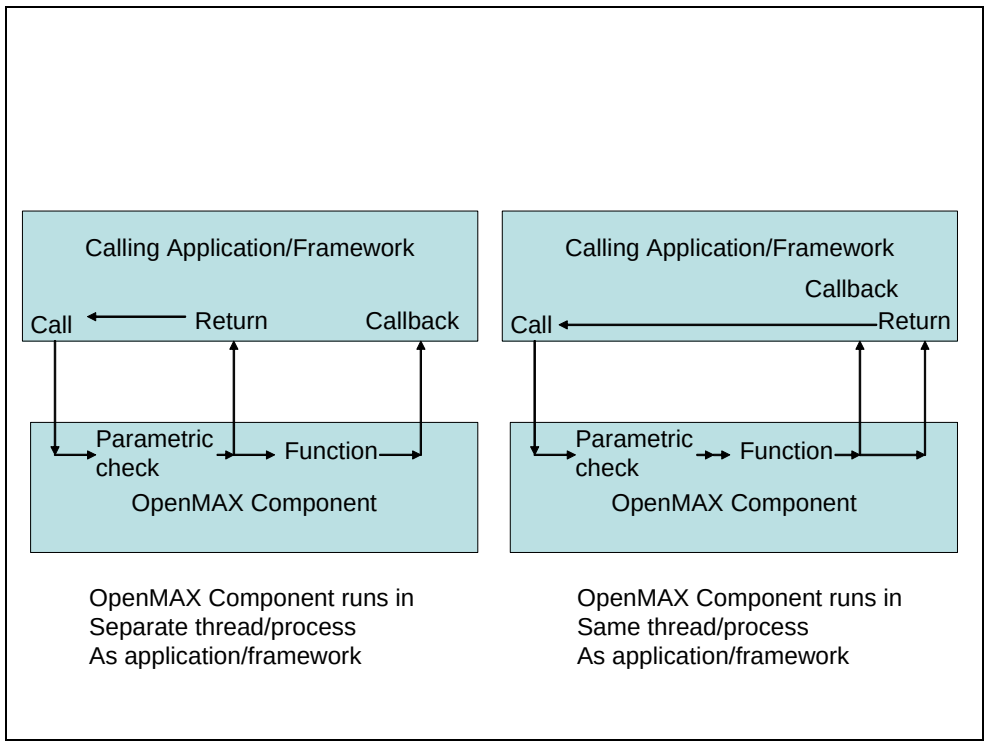


Figure 2-5. Out-of-Context versus In-Context Operation

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.

2.1.7 Tunneled Buffer Allocation and Sharing

This section describes buffer allocation for tunneling components and buffer sharing. For a given tunnel, exactly one port supplies the buffers and passes those buffers to the non-supplier port. In the simplest case, the supplier also allocates the buffers. Under the right circumstances, however, a tunneling component may choose to re-use buffers from one port on another to avoid memory copies and optimize memory usage. This practice is known as buffer sharing.

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.

Buffer sharing is implemented 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.

Strictly speaking, a component is only obligated to obey the external semantics required of it and may implement buffer sharing behind those semantics. More specifically, external semantics require that a component do the following:

- Provide buffers on all of its supplier ports.
- Accurately communicate buffer requirements on its ports.
- Pass a buffer from an output port to an input port with an `OMX_EmptyThisBuffer` call.
- Return a buffer from an input port to an output port with an `OMX_FillThisBuffer` call.

If a component chooses to share buffers, its implementation may fulfill those 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 `OMX_EmptyBufferDone` call.

OpenMAX 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.

2.1.7.1 Relevant Terms

This section describes terms used in discussions of tunneled buffer allocation and sharing. Figure 2-6 illustrates the concepts.

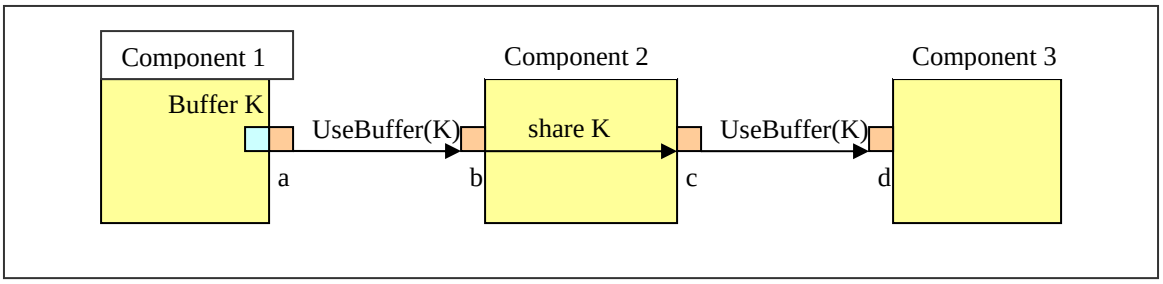


Figure 2-6. Example of Buffer Allocation and Sharing Relationships

Among a pair of ports that are tunneling, the port that calls UseBuffer on its neighbor is known as a *supplier port*. A buffer supplier port does not necessarily allocate its buffers; it may re-use buffer from another port on the same component. Ports a and c in Figure 2-6 illustrate supplier ports.

The port that receives the UseBuffer calls from its neighbor is known as a *non-supplier port*. Ports b and d Figure 2-6 illustrate non-supplier ports.

A port's *tunneling port* is the port neighboring it with which it shares a tunnel. For example, port b in Figure 2-6 is the tunneling port to port a. Likewise, port a is the tunneling port to port b.

An *allocator port* is a supplier port that also allocates its own buffers. Port a in Figure 2-6 is the only allocator port.

A *sharing port* is a port that re-uses buffers from another port on the same component. For example, port c in Figure 2-6 is a sharing port.

A *tunneling component* is a component that uses at least one tunnel.

The set of *buffer requirements* for a port includes the number of buffers required and the required size of each buffer. The maximum of multiple sets of buffer requirements is defined as the largest number of buffers mandated in any set combined with the largest size mandated in any set. One port retrieves buffer requirements from its tunneled port in a OMX_PORTDEFINITIONTYPE structure via an `OMX_GetParameter` call on the tunneled port's component. Note that one port may determine 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.

2.1.7.2 IL Client Component Setup

To set up tunneling components, the IL client shall 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 the loaded state to the idle state.

If the IL client does not operate in this manner, a tunneling component might never transition to idle because of the possible dependencies between components.

2.1.7.3 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 one 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 2-7.

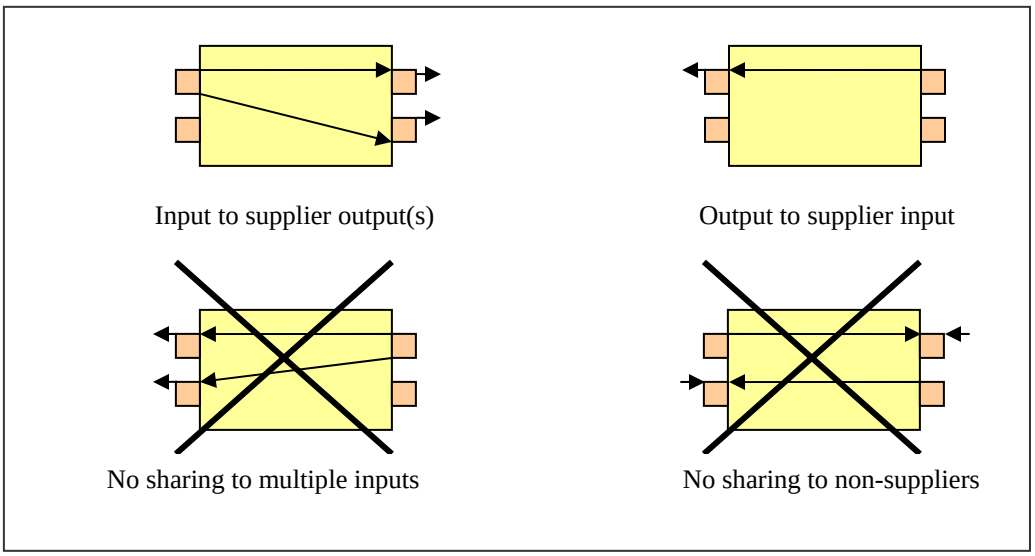


Figure 2-7. Possible Sharing Relationships

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). In Figure 2-8, 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.

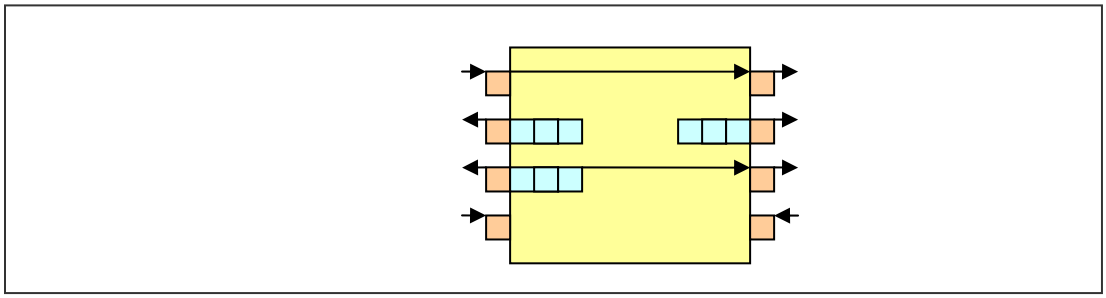


Figure 2-8. Determining Allocators

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 non-supplier port that it is tunneling with 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. 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.

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 `nBufferCount` 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 2-9. Note that this example is contrived to exercise every step and obligation outlined above, and is therefore more complex then most real use cases.

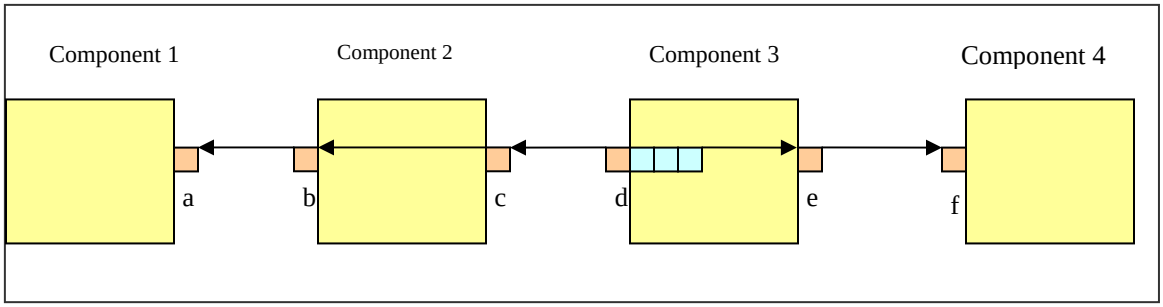


Figure 2-9. Example of Buffer Allocation

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.

2.1.7.4 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 that it is sharing with the output port by obeying the following rules:

- The output port calls `OMX_EmptyThisBuffer` on its tunneling port before the input port sends the corresponding `OMX_EmptyBufferDone` call to its tunneling port.
- The input port does not call `OMX_EmptyBufferDone` until all output ports on which the buffer is shared (i.e., via `OMX_EmptyThisBuffer` calls) return `OMX_EmptyBufferDone`.

2.1.7.5 Component Transition from Loaded to Idle State without Sharing

If a component does not share buffers, the component implementation reduces to a simpler set of steps and obligations than the case for sharing buffers.

When commanded to transition from loaded to idle, a non-sharing component performs the following operations in this order:

1. The component determines what buffering sharing it will implement, if any. In this case, there is no sharing.
2. The component determines which of its supplier ports, if any, are also allocator ports. All supplier ports are allocator ports.
3. The component allocates it buffers for each allocator port as follows:
    a. Since there is no sharing, the component does not ask the sharing port for requirements.
    b. The allocator determines the buffer requirements of its tunneled port via an `OMX_GetParameter` call.
    c. The allocator allocates buffers according to the maximum of its own requirements and the requirements of the tunneled ports.
    d. Since there is no sharing, no buffers must be passed to sharing ports.
    e. For every allocated buffer, the allocator port calls `OMX_UseBuffer` on its tunneling port.

All component obligations described for sharing components do not apply to non-sharing components.

2.1.8 Port Reconnection

Port reconnection enables a tunneled component to be replaced with another tunneled component without having to tear down surrounding components. In Figure 2-10, 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 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 component C input port may be connected to the component B2 output port. Then all ports may be given the enable command.

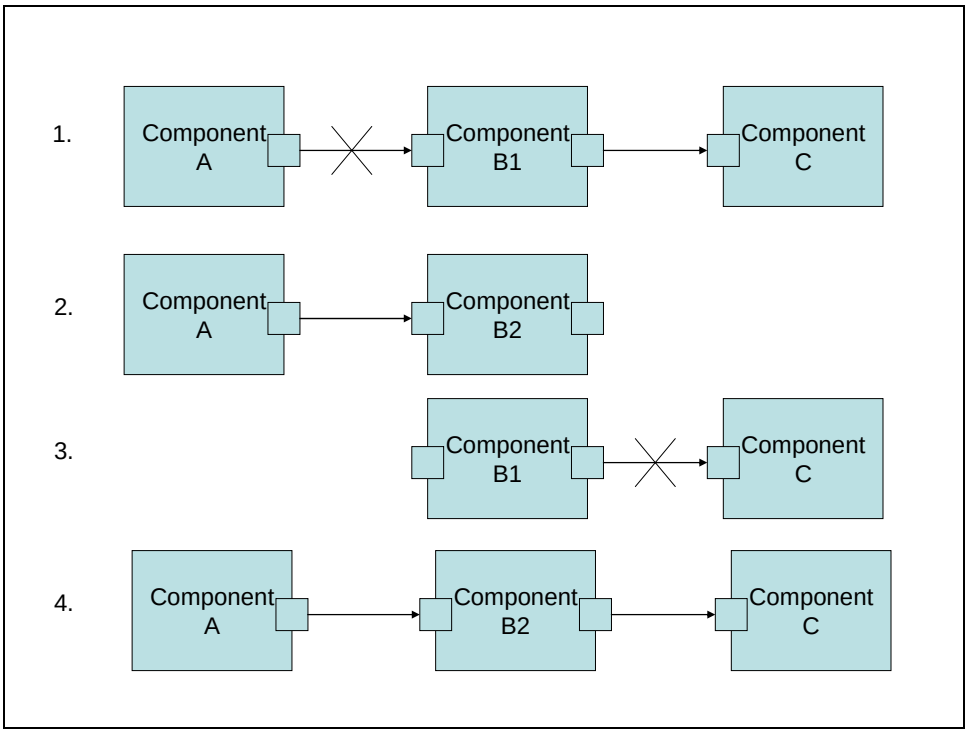


Figure 2-10. Port Reconnection

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 2-11 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, and component B1 resources may be de-allocated.

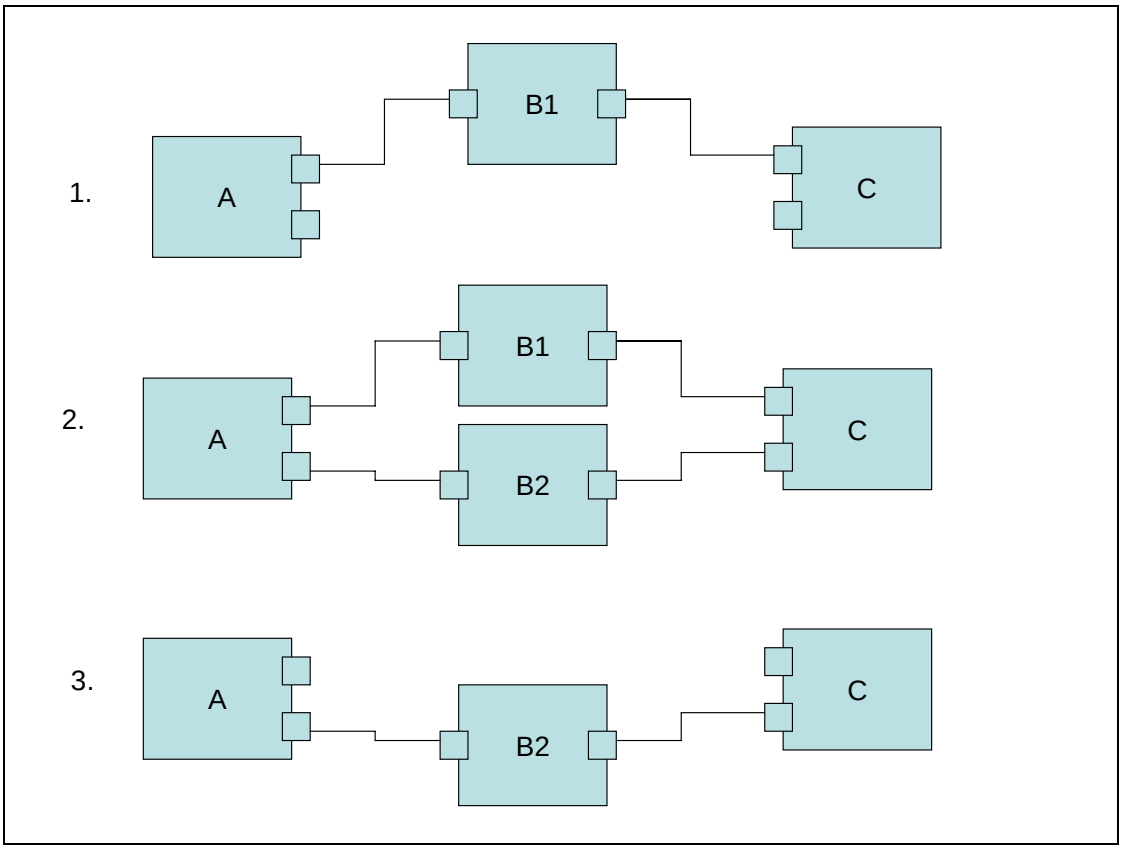


Figure 2-11. Reconnecting Components

2.1.9 Queues and Flush

A separate command queue enables 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 tunneled port when using tunneled communication. In Figure 2-12, assume that the 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 in the original order, and finally 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.

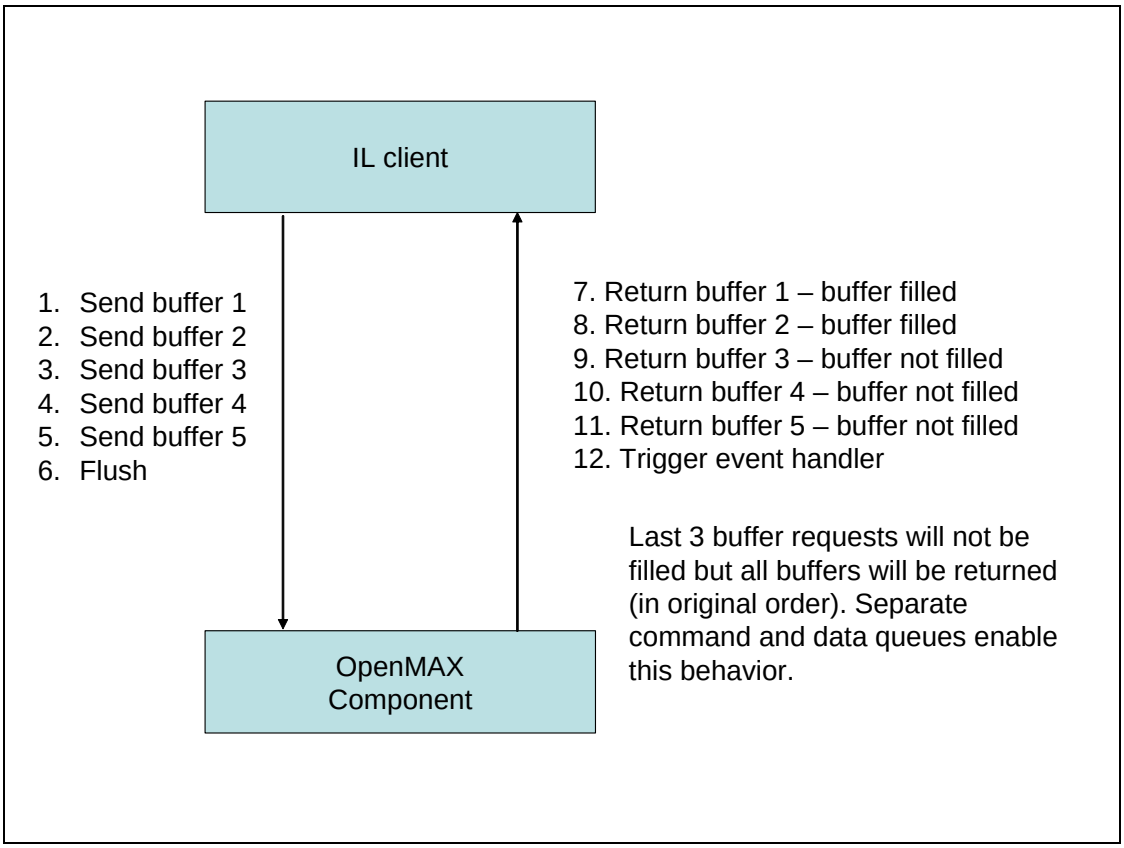


Figure 2-12. Flushing Buffers

2.1.10 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 components. The mark enables a component to send an event to the IL client when the marked buffer is encountered. Figure 2-13 depicts how this works.

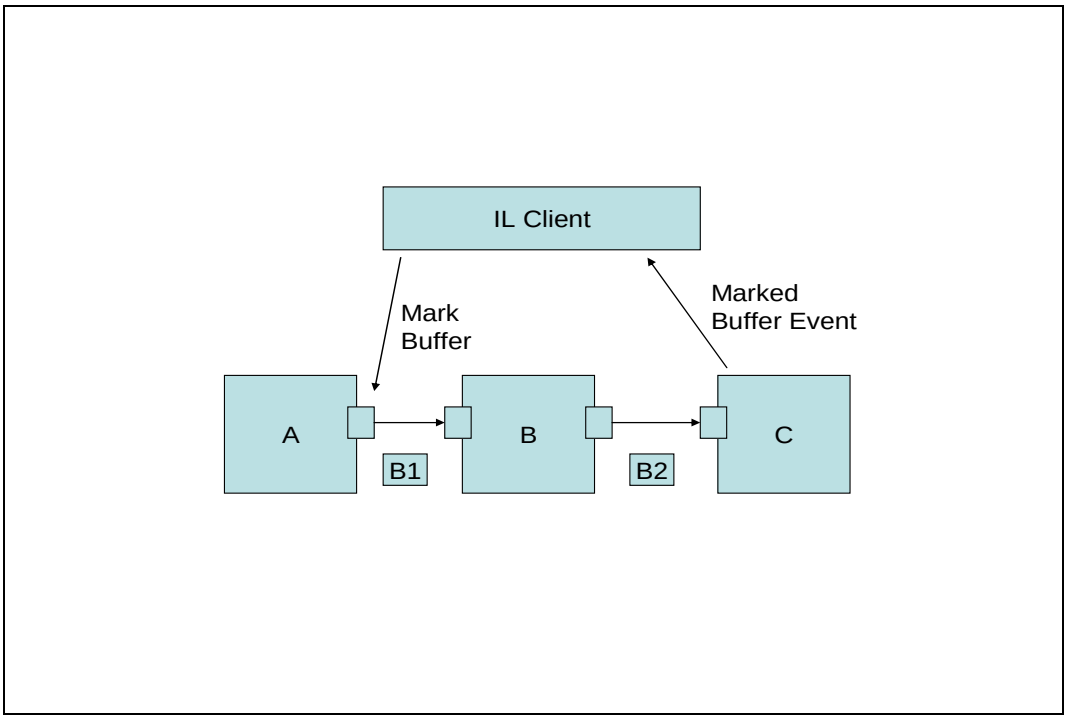


Figure 2-13. Marking Buffers

The IL client sends a command to mark a buffer. The next buffer sent from the output port of the component is marked B1. Component B processes the B1 buffer and provides the results in buffer B2 along with the mark. When component C receives the marked buffer B2 through its input port, the component does not trigger its event handler until it has processed the buffer.

2.1.11 Events and Callbacks

Six 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 execution of a command.
- *Marked buffer events* are triggered upon detection of a marked buffer by a component.
- A *port settings changed notification event* is generated when the component changes its port settings.
- A *buffer flag event* is triggered when an end of stream is encountered.
- A *resources acquired event* is generated when a component gets resources that it has been waiting for.

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

2.1.12 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.

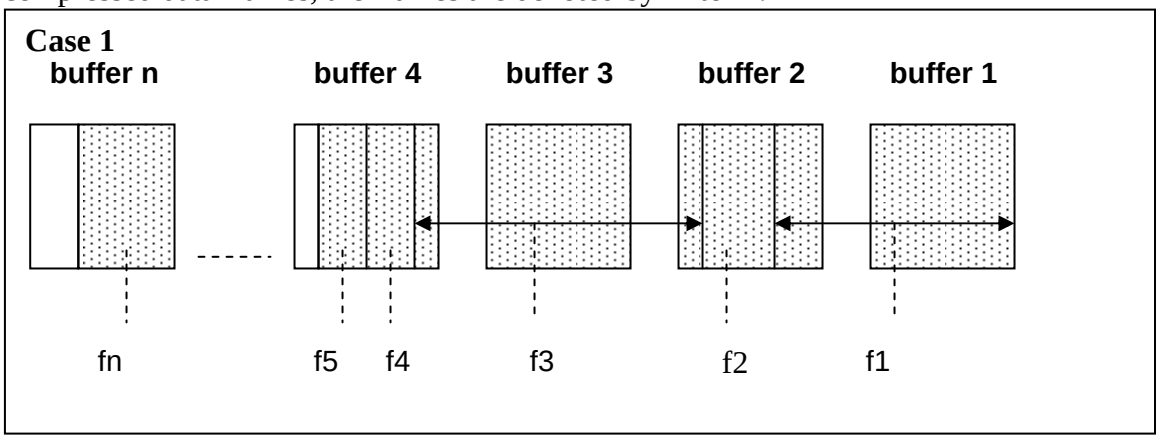
There are generally three cases that describe how a buffer can be filled with data. Each case presents its own benefits.

In all cases, the range and location of valid data in a buffer is defined by the `pBuffer`, `nOffset`, and `nFilledLength` parameters of the buffer header. The `pBuffer` parameter points to the start of valid data in the buffer. The `nOffset` parameter indicates the number of bytes between the start of the buffer and the start of valid data. The `nFilledLength` parameter specifies the number of contiguous bytes of valid data in the buffer. The valid data in the buffer is therefore located in the range `pBuffer + nOffset` to `pBuffer + nOffset + nFilledLength`.

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.

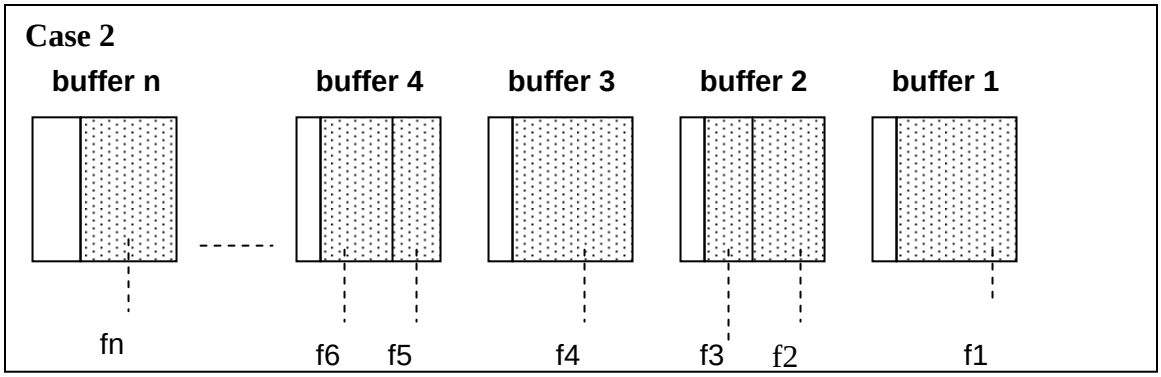
The shaded portion of the buffer represents data and the white portion denotes no data.

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.



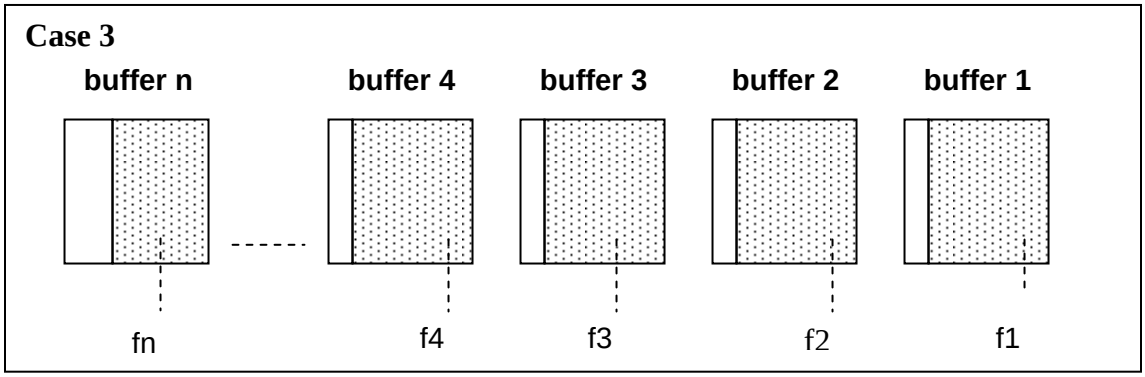
Case 1 provides a benefit when decoding for playback. The buffer can accommodate multiple frames and reduce the number of transactions required to buffer an amount of data for decoding. However, this case may require the decoder to parse the data when decoding the frames. It also may require the decoder 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.



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.



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.

2.1.13 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 in the 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.

2.1.14 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 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 2-14 illustrates the flow of time and data buffers in an example configuration of components.

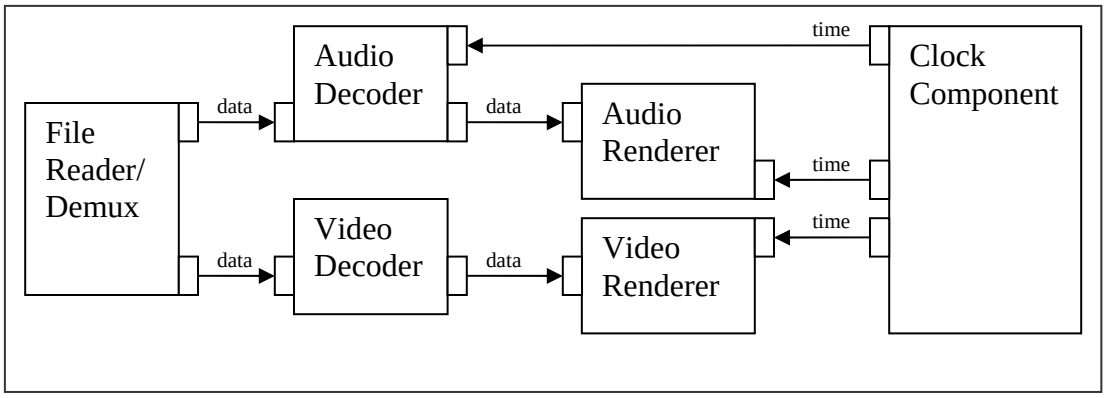


Figure 2-14. Flow of Time and Data Buffers

2.1.15 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.

2.1.16 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. 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.

2.1.17 Resource Management

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

2.1.17.1 Need for Resource Management

When a component is not allowed to go to idle state due to lack of resources, the IL client has cannot know what the limited resource is or which components are using that resource. Therefore, the IL client cannot, for example, free up resources for a mandatory audio stream to play without turning off all of the IL components or having specific

knowledge of IL component implementations, neither of which is a viable option. These situations necessitate IL resource management.

One of the goals of OpenMAX 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. For example, the IL client might have no information on whether a component is hardware accelerated or not.
- 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 1.0, "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:

- *Resource management* is responsible for managing the access of components to a limited resource. A resource manager will be aware of how much of a specific resource is available, which components are currently using the resource, and how much of the resource the components are using. A resource manager will recommend to policy which components should be pre-empted or resumed based on resource conflicts and availability.
- *Policy* is responsible for managing component chains or streams. The policy manager determines if a stream is allowed to run or resume based on information it receives from resource management, system configuration, requests from applications, or other factors.

2.1.17.2 Architectural Assumptions

The following discussion makes two architectural assumptions about the OpenMAX IL:

- Assumption 1: A framework exists that contains a policy manager between the applications and the OpenMAX IL.
- Assumption 2: A system can have one or more hardware platforms that are used by different OpenMAX components and that are managed by hardware vendor-specific resource manager(s).

These assumptions are illustrated in the high-level architecture shown in Figure 2-15. For systems that do not have a framework (that is, where user applications interface directly with the IL), version 1.0 of the OpenMAX IL API specification does not specify how resource management will be handled. Assumption 2 covers systems that have a single, centralized resource manager as well.

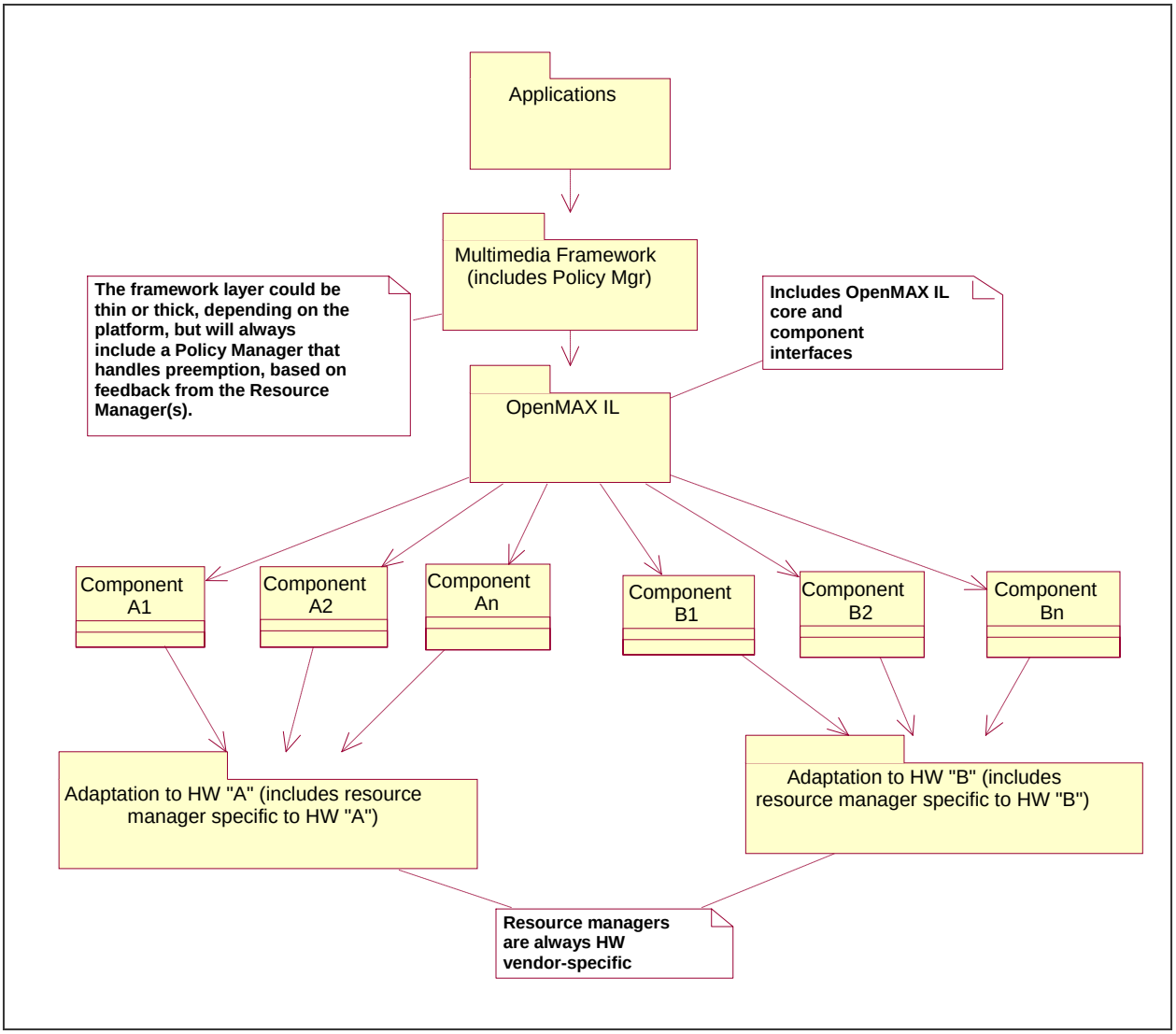


Figure 2-15. Architectural Assumptions

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

2.1.17.3 Component Priorities

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

The actual range of priorities can be left up to the platform, but the priority order is important and needs to be the same across IL implementations. 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.*

2.1.17.4 Behavioral Rules

The following behavior is defined on the IL layer:

- The `OMX_ErrorInsufficientResources` error is called only on a component that attempts to go to the idle state 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 the idle state, and the resources it requires need to be freed by preempting lower priority components.
- When a component that already has resources needs to be preempted, it will send the `OMX_ErrorResourcesPreempted` and `OMX_ErrorResourcesLost` errors to the IL client as it moves from the Executing or Paused state to the Idle state and from the Idle state to the Loaded state, respectively.
- 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 the idle state. When the client receives the notification that the component is in the idle state, it can try to move the rest of the components in that chain to the idle state as well. This automatic movement to the idle state 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 the loaded state, 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.

2.1.17.5 Hardware Vendor-Specific Resource Manager

To implement the behavioral rules, a hardware vendor-specific resource manager will need to exist below the IL layer 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.

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
- OMX_Core.h: OpenMAX IL core API
- OMX_Component.h: OpenMAX component API
- OMX_Audio.h: OpenMAX audio domain data structures
- OMX_IVCommon.h: OpenMAX structures common to image and video domains
- OMX_Video.h: OpenMAX video domain data structures
- OMX_Image.h: OpenMAX image domain data structures
- OMX_Other.h: OpenMAX other domain data structures (includes A/V synchronization)
- OMX_Index.h: Index of all OpenMAX-defined data structures

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

First, the OpenMAX data types are introduced. Next, the methods of the OpenMAX core are described. The methods that components implement are discussed in section 3.3. 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 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 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 Types

3.1.1 Enumerations

Five 32-bit integer 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 component (see section 3.1.1.1).
- `OMX_EVENTTYPE` includes events that can be generated inside an OpenMAX component and that are passed to the IL client through a callback function (see section 3.1.1.4).
- `OMX_BUFFERSUPPLIERTYPE` includes all the possibilities for the buffer supplier in the case of tunneled ports. A description of the use of this enumerative type can be found in section 3.1.1.5.
- `OMX_STATETYPE,` which is described in section 3.1.1.2.

Figure 3-1 shows the enumerations defined in OMX_Core.h.

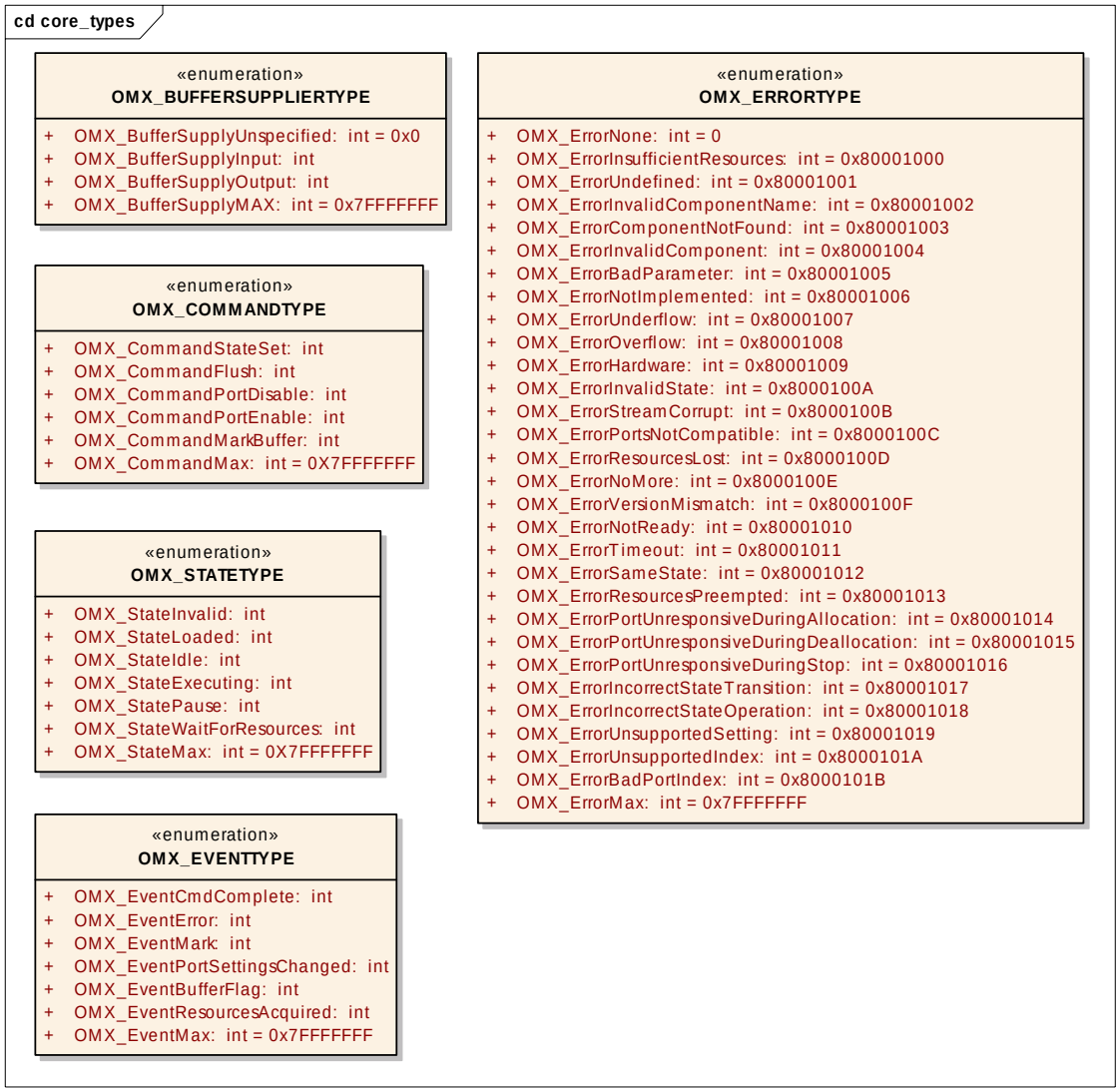


Figure 3-1. Enumerations Defined in OMX_Core.h

3.1.1.1 OMX_COMMANDTYPE

Table 3-1 represents the possible commands that an IL client can send to an OpenMAX component. Since commands are non-blocking, the OpenMAX component generates a command completion event via a callback function when the command has completed. Callbacks are defined in a dedicated structure; see section 3.1.2.7.

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 3-1. OpenMAX IL Commands

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

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

Table 3-2. Command Syntax

3.1.1.2 OMX_STATETYPE

Figure 3-2 illustrates the transitions among states that occur as a consequence of the IL client calling **OMX_SendCommand**(**OMX_StateSet**, <*state*>), where the new state for the component is passed as a parameter. A transition name surrounded by curly braces indicates that the transition is not triggered by a command sent by the IL client but is a consequence of internal component events.

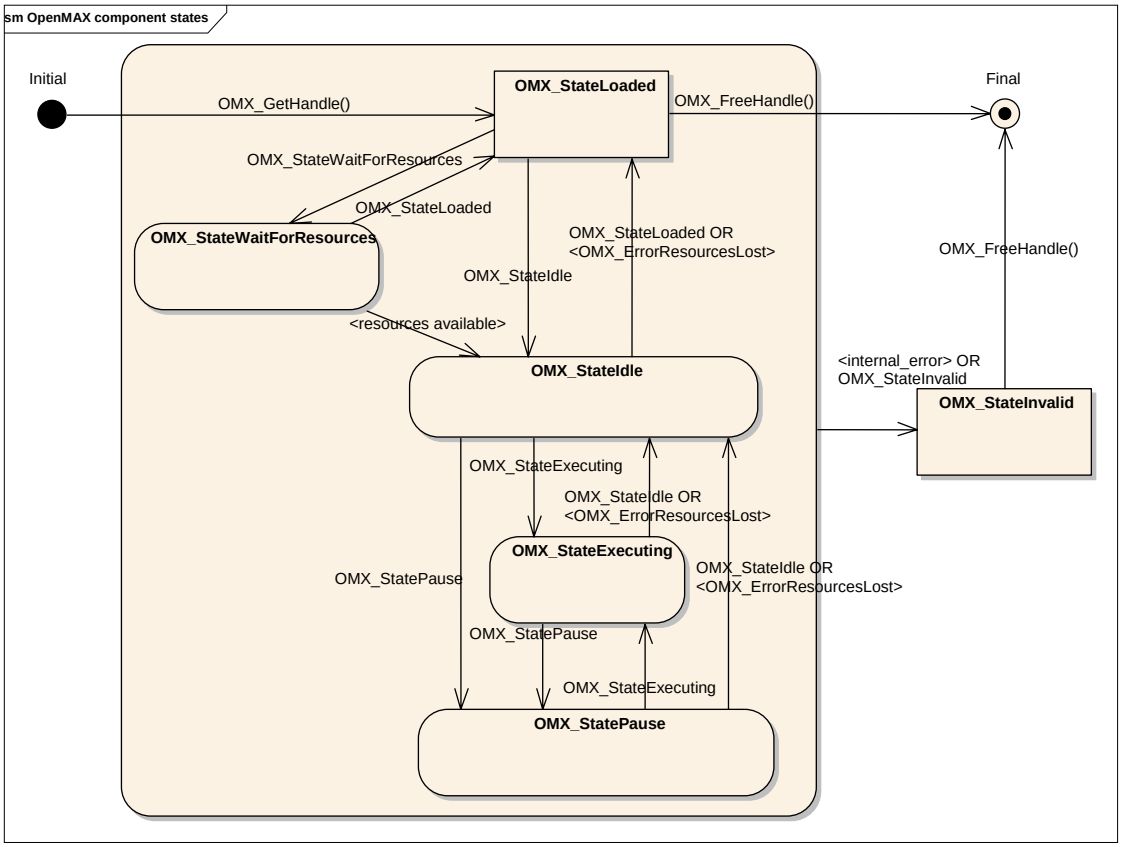


Figure 3-2. OpenMAX Component State Transitions

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 component.

Field Name	Description	Resources Allocated	Location of buffer
OMX_StateInvalid	Component is corrupt or has encountered an error from which it cannot recover.	Unknown	Unknown
OMX_StateLoaded	Component has been loaded but has no resources allocated.	No	Not available
OMX_StateIdle	Component has all 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_StateWaitFor Resources	Component is waiting for a resource to become available.	No	Not available

Table 3-3. OpenMAX Component States

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`, 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 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 must acquire all of its resources, including buffers, before completing the transition. Furthermore, before the transition can complete, the buffer supplier, which is always the IL client when not tunneling, must ensure that the non-supplier possesses all of its buffers. For a port connected to the IL client, the IL client may allocate the buffers itself and then pass them to the port via an `OMX_UseBuffer` call on the port, or it may

direct the port to perform the allocation via an OMX_AllocateBuffer call on the port. When a port is tunneling, the supplier port either allocates buffers itself or, if the port implements buffer sharing, re-uses buffers from a port on the same component. A tunneling supplier port then passes the buffers to the non-supplier port via an OMX_UseBuffer call on the non-supplier.

The number of buffers 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.

3.1.1.2.2 OMX_StateIdle

In the OMX_StateIdle state, the component is ready to be used, meaning that all necessary 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_StateInvalid and OMX_StateWaitForResources states.

3.1.1.2.2.1 OMX_StateIdle to OMX_StateLoaded

On a transition from OMX_StateIdle to OMX_StateLoaded, each buffer supplier must call OMX_FreeBuffer on the non-supplier port for each buffer residing at the non-supplier port. If the supplier allocated the buffer, it must free the buffer before calling OMX_FreeBuffer. If the non-supplier port allocated the buffer, it must free the buffer upon receipt of an OMX_FreeBuffer call. Furthermore, a non-supplier port must 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; the component communicates that the initiating OMX_SendCommand call has completed via a callback event.

3.1.1.2.2.2 OMX_StateIdle to OMX_StateExecuting

If the IL client requests a state transition from OMX_StateIdle to OMX_StateExecuting, the component shall begin transferring and processing data. 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 component is transferring and processing data buffers. 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. An IL client may transition a component in the OMX_StateExecuting state to either the OMX_StateIdle state or the OMX_StatePaused 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 `OMX_EmptyBufferDone` and `OMX_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.

3.1.1.2.4 OMX_StatePause

In this state, an OpenMAX 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. The component may still accept data buffers at its input, but such buffers will be queued only and not processed further. 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 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 resources, including buffers.

3.1.1.2.6 OMX_StateInvalid

In this state, the component has suffered internal corruption or an error from which it cannot recover. When it detects such a condition, the component transitions itself to OMX_StateInvalid and informs the IL client by generating an OMX_ErrorEvent event with the value OMX_ErrorInvalidState. When the IL client receives OMX_EventError indicating a transition to OMX_StateInvalid, it shall free all resources associated with that component and eventually call OMX_FreeHandle to release the handle associated with the component.

A component in the OMX_StateInvalid state shall fail every call made upon it and return an OMX_ErrorStateInvalid error message except for OMX_GetState, OMX_FreeBuffer, or OMX_ComponentDeinit. The IL client may also command a transition to the OMX_StateInvalid state explicitly via OMX_SendCommand. A component may transition between any state and the OMX_StateInvalid state.

3.1.1.3 OMX_ERRORTYPE

The OMX_ERRORTYPE enumeration shown in Table 3-4 defines the standard OpenMAX errors that all functions defined in the OpenMAX IL API return. 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.

Field Name	Value	Description
OMX_ErrorNone	0	The function returned successfully.
OMX_ErrorInsufficientResources	0x80001000	There were insufficient resources to perform the requested operation.
OMX_ErrorUndefined	0x80001001	There was an error but the cause of the error could not be determined.
OMX_ErrorInvalidComponentName	0x80001002	The component name string was invalid.
OMX_ErrorComponentNotFound	0x80001003	No component with the specified name string was found.
OMX_ErrorInvalidComponent	0x80001004	The component specified did not have a OMX_ComponentInit entry point, or the component did not correctly complete the OMX_ComponentInit call.

OMX_ErrorBadParameter	0x80001005	One or more parameters were invalid.
OMX_ErrorNotImplemented	0x80001006	The requested function is not implemented.
OMX_ErrorUnderflow	0x80001007	The buffer was emptied before the next buffer was ready.
OMX_ErrorOverflow	0x80001008	The buffer was not available when it was needed.
OMX_ErrorHardware	0x80001009	The hardware failed to respond as expected.
OMX_ErrorInvalidState	0x8000100A	The component is in the OMX_StateInvalid state.
OMX_ErrorStreamCorrupt	0x8000100B	The stream is found to be corrupt.
OMX_ErrorPortsNotCompatible	0x8000100C	Ports being set up for tunneled communication are incompatible.
OMX_ErrorResourcesLost	0x8000100D	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	0x8000100E	No more indices can be enumerated.
OMX_ErrorVersionMismatch	0x8000100F	The component detected a version mismatch.
OMX_ErrorNotReady	0x80001010	The component is not ready to return data at this time.
OMX_ErrorTimeout	0x80001011	A timeout occurred.
OMX_ErrorSameState	0x80001012	The component tried to transition into the state that it is currently in.
OMX_ErrorResourcesPreempted	0x80001013	Resources allocated to a component in the OMX_StateExecuting or OMX_Pause states have been pre-empted, causing the component to return to the OMX_StateIdle state.
OMX_ErrorPortUnresponsiveDuringAllocation	0x80001014	The non-supplier port deemed that it had waited an unusually long time for the supplier port to send it an allocated buffer via an OMX_UseBuffer call. A non-supplier port sends this error to the IL client via the EventHandler callback during the allocation of buffers on a transition from the LOADED to the IDLE state or on a port enable.

OMX_ErrorPortUnresponsiveDuringDeallocation	0x80001015	The non-supplier port deemed that it had waited an unusually long time for the supplier port to request the de-allocation of a buffer header via a OMX_FreeBuffer call. A non-supplier port sends this error to the IL client via the EventHandler callback during the de-allocation of buffers on a transition from the IDLE to LOADED state or on a port disablement.
OMX_ErrorPortUnresponsiveDuringStop	0x80001016	The supplier port deemed that it had waited an unusually long time for the non-supplier port to return a buffer via an EmptyThisBuffer or FillThisBuffer call. A supplier port sent this error to the IL client via the EventHandler callback during the disabling of a port, either on a transition from the IDLE to LOADED state or on a port disablement.
OMX_ErrorIncorrectStateTransition	0x80001017	A state transition was attempted that is not allowed.
OMX_ErrorIncorrectStateOperation	0x80001018	A command or method was attempted that is not allowed during the present state.
OMX_ErrorUnsupportedSetting	0x80001019	One or more values encapsulated in the parameter or configuration structure are unsupported.
OMX_ErrorUnsupportedIndex	0x8000101A	The parameter or configuration indicated by the given index is unsupported.
OMX_ErrorBadPortIndex	0x8000101B	The port index that was supplied is incorrect.
OMX_ErrorPortUnpopulated	0x8000101C	The port has lost one or more of its buffers and is thus unpopulated.

Table 3-4. OpenMAX Error Codes

3.1.1.4 OMX_EVENTTYPE

The OMX_EVENTTYPE enumeration shown in Table 3-5 includes the event types that an OpenMAX component can generate. Section 3.1.2.7 describes events that the OpenMAX 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.

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 the OMX_StateWaitForResources state to the OMX_StateIdle state.

Table 3-5. OpenMAX Event Types

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. In case of a component state change, the new state that the component has entered is returned as an event parameter. A component that transitions to the OMX_StateInvalid state does not generate this event.

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_ErrorInvalidState error if the component transitions to the OMX_StateInvalid state.
- 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 generates 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 as soon as component port settings change. For example, a video decoder may not know *a priori* the output frame size and frame rate, as these parameters are coded in the input bit stream. As soon as such parameters are parsed, the component changes the values of the configuration structures of its output port and sends the `OMX_EventPortSettingsChanged` event to the IL client.

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 receives this event, it is ready to change state into the OMX_StateIdle, and it waits for all the buffers to be allocated and assigned to its ports.

3.1.1.5 OMX_BUFFERSUPPLIERTYPE

The OMX_BUFFERSUPPLIERTYPE enumerative type 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.

Field Name	Value	Description
`OMX_BufferSupplyUnspecified`	0x0	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.

Table 3-6. OpenMAX Buffer Supplier Type Used in Tunnel Setup

3.1.2 Structures

This section discusses the data structures defined in the OpenMAX core. The first two fields of each OpenMAX data structure denote the size of the structure and the version of type OMX_VERSIONTYPE, which is defined in section 3.1.2.4. The entity that allocates an OpenMAX structure is responsible for filling in these two values.

3.1.2.1 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;
```

3.1.2.2 OMX_COMPONENTINITTYPE Type Definition

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);
```

3.1.2.2.1 pName

pName contains the string name of the component and has limit of 128 bytes (including '\0').

3.1.2.2.2 pInitialize

pInitialize contains the pointer to the initialization function of the component.

3.1.2.3 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.

3.1.2.4 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 specification version under which the structure is defined. For OpenMAX IL version 1.0, the specification version is 1.0.0.0. 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;
```

3.1.2.4.1 nVersionMajor

`nVersionMajor` identifies the major version number.

3.1.2.4.2 nVersionMinor

`nVersionMinor` identifies the minor version number.

3.1.2.4.3 nRevision

`nRevision` identifies the revision number.

3.1.2.4.4 nStep

`nStep` identifies the step number.

3.1.2.5 OMX_PRIORITYMGMTTYPE

The OMX_PRIORITYMGMTTYPE type 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, all of the other components in the same group shall transition 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;
```

3.1.2.5.1 nGroupPriority

The value of `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.

3.1.2.5.2 nGroupID

The value for `nGroupID` is a unique ID for all components in the same component group.

3.1.2.6 OMX_BUFFERHEADERTYPE

In the context of a single port, 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, each pair of tunneling ports share buffer headers; otherwise, the same buffer transferred over multiple ports will have distinct buffer headers associated with it for each port. 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_U32 nOutputPortPrivate;
    OMX_U32 nInputPortPrivate;
    OMX_HANDLETYPE hMarkTargetComponent;
    OMX_PTR pMarkData;
    OMX_U32 nTickCount;
    OMX_TICKS nTimeStamp;
    OMX_U32 nFlags;
    OMX_U32 nOutputPortIndex;
    OMX_U32 nInputPortIndex;
} OMX_BUFFERHEADERTYPE;
```

3.1.2.6.1 pBuffer

`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` in section 3.1.2.6.4.

3.1.2.6.2 nAllocLen

`nAllocLen` is the total size of the allocated buffer in bytes, including valid and unused byte.

3.1.2.6.3 nFilledLen

`nFilledLen` is the total size of valid bytes currently in the buffer starting from the location specified by `pBuffer` and `nOffset`.

3.1.2.6.4 nOffset

`nOffset` is the start offset of valid data in bytes from the start of the buffer. A pointer to the valid data may be obtained by adding `nOffset` to `pBuffer`.

3.1.2.6.5 pAppPrivate

`pAppPrivate` is a pointer to an IL client private structure.

3.1.2.6.6 pPlatformPrivate

`pPlatformPrivate` is a pointer to a platform private structure. The core that allocated this buffer header structure uses this pointer.

3.1.2.6.7 pOutputPortPrivate

`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.

3.1.2.6.8 pInputPortPrivate

`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.

3.1.2.6.9 hMarkTargetComponent

`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 should compare its own handle to this handle and emit the mark if the handles match. A component should propagate this field from an input buffer to its associated output buffer.

3.1.2.6.10 pMarkData

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 should propagate this field from an input buffer to its associated output buffer.

3.1.2.6.11 nTickCount

`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.

3.1.2.6.12 nTimeStamp

`nTimeStamp` is a 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.

3.1.2.6.13 nFlags

The `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
```

3.1.2.6.13.1 OMX_BUFFERFLAG_EOS

A component sets EOS when it has no more data to emit on a particular output port. Thus, an output port shall set EOS on the last buffer it emits. The determination by a component of when an output port should cease sending data is implementation specific.

3.1.2.6.13.2 OMX_BUFFERFLAG_STARTTIME

The source of a stream (e.g., a de-multiplexing component) sets the OMX_BUFFERFLAG_STARTTIME flag 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 STARTTIME flag shall perform an `OMX_SetConfig` call on its sync port using `OMX_ConfigTimeClientStartTime` and pass the timestamp for the buffer.

3.1.2.6.13.3 OMX_BUFFERFLAG_DECODEONLY

The source of a stream (e.g., a de-multiplexing component) sets the OMX_BUFFERFLAG_DECODEONLY flag 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.

3.1.2.6.13.4 OMX_BUFFERFLAG_DATACORRUPT

The OMX_BUFFERFLAG_DATACORRUPT flag is set when the IL client identifies the data in the associated buffer as corrupt.

3.1.2.6.13.5 OMX_BUFFERFLAG_ENDOFFRAME

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.

3.1.2.6.14 nOutputPortIndex

`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.

3.1.2.6.15 nInputPortIndex

`nInputPortIndex` contains the port index of the input 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 `nInputPortIndex` is undefined.

3.1.2.7 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;
```

3.1.2.7.1 nPorts

`nPorts` is the number of ports of a given port domain (audio, video, image, or other) for the component.

3.1.2.7.2 nStartPortNumber

`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 `nStartNumber`.

3.1.2.8 OMX_CALLBACKTYPE

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

- An asynchronous command triggered by the IL client has completed successfully or failed and generated an error. Commands include those sent by `OMX_SendCommand` and those implied by IL client calls to `EmptyThisBuffer` or `FillThisBuffer`.
- An error unassociated with a command triggered by the IL client has occurred. For example, the component has suffered an unrecoverable error and is transitioning to the OMX_StateInvalid state.

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 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;
```

3.1.2.8.1 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. `nData1` carries the value of `OMX_COMMANDTYPE` that has been completed or `OMX_ERRORTYPE`. `nData2` carries further event parameters, e.g., `OMX_STATETYPE`. `pEventData` contains event specific data. The `pEventData` pointer may contain additional data associated with the event (e.g., mark-specific data). A call to `EventHandler` is a blocking call, so the IL client should respond within five msec 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	The handle of the component that calls this function.
eEvent	The event that the component is communicating to the IL client.

nData1 The first integer event-specific parameter. See Table 3-7 for the meaning in the context of each event.

nData2 The second integer event-specific parameter. See Table 3-7 for the meaning in the context of each event. The default value is 0 if not used.

pEventData A pointer to additional event-specific data. See Table 3-7 for the meaning in the context of each event.

Table 3-7 lists the parameters used in each event.

eEvent	**nData1**	**nData2**	**pEventData**
`OMX_EventCmdComplete`	`OMX_CommandStateSet`	State reached	Null
	`OMX_CommandFlush`	Port index	Null
	`OMX_CommandPortDisable`	Port index	Null
	`OMX_CommandPortEnable`	Port index	Null
	`OMX_CommandMarkBuffer`	Port index	Null
`OMX_EventError`	Error code	0	Null
`OMX_EventMark`	0	0	Data linked to the mark, if any
`OMX_EventPortSettingsChanged`	port index	0	Null
`OMX_EventBufferFlag`	port index	`nFlags` unaltered	Null
`OMX_EventResourcesAcquired`	0	`0`	Null

Table 3-7. Event Parameter Usage

3.1.2.8.2 EmptyBufferDone

A component uses the `EmptyBufferDone` callback to pass a buffer from an input port back to the IL client. A component sets the `nOffset` and `nFilledLength` values of the buffer header to reflect the portion of the buffer it consumed; for example, `nFilledLength` is set equal to 0x0 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 or to OMX_StateInvalid.
- The IL client flushes or disables a port.

The `EmptyBufferDone` call is a blocking call that should return from within five msec. 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_OUT OMX_HANDLETYPE hComponent,
 OMX_OUT OMX_PTR pAppData,
 OMX_OUT OMX_BUFFERHEADERTYPE* pBuffer)
```

The parameters are as follows.

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

3.1.2.8.3 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 n`FilledLength` of the buffer header to reflect the portion of the buffer it filled; for example, `nFilledLength` is equal to 0x0 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 or to OMX_StateInvalid.
- The IL client flushes or disables a port.

The `FillBufferDone` call is a blocking call that should return from within five msec. 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_OUT OMX_HANDLETYPE hComponent,
OMX_OUT OMX_PTR pAppData,
OMX_OUT OMX_BUFFERHEADERTYPE* pBuffer)
```

The parameters are as follows.

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

3.1.2.9 OMX_PARAM_BUFFERSUPPLIERTYPE

The OMX_PARAM_BUFFERSUPPLIERTYPE structure is used to communicate buffer supplier settings or buffer supplier preferences.

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;
```

3.1.2.9.1 nPortIndex

`nPortIndex` represents the port that this structure applies to.

3.1.2.9.2 eBufferSupplier

`eBufferSupplier` is a field that contains the index of the buffer supplier, if input or output.

3.1.2.10 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;
```

3.1.2.10.1 nTunnelFlags

The `nTunnelFlags` integer parameter 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.

3.1.2.10.2 eSupplier

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.

3.1.2.11 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;
    union {
      OMX_AUDIO_PORTDEFINITIONTYPE audio;
          OMX_VIDEO_PORTDEFINITIONTYPE video;
          OMX_IMAGE_PORTDEFINITIONTYPE image;
          OMX_OTHER_PORTDEFINITIONTYPE other;
    } format;
} OMX_PARAM_PORTDEFINITIONTYPE;
```

3.1.2.11.1 nPortIndex

`nPortIndex` is a read-only field the identifies the port. 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.

3.1.2.11.2 eDir

`eDir` is a read-only field that indicates the direction (`OMX_DirInput` or `OMX_DirOutput`) for the port.

3.1.2.11.3 nBufferCountActual

`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 no less than `nBufferCountMin` for this field.

3.1.2.11.4 nBufferCountMin

`nBufferCountMin` is a read-only field that specifies the minimum number of buffers that the port requires. The component shall define this non-zero default value.

3.1.2.11.5 nBufferSize

`nBufferSize` is a read-only field that specifies the minimum size in bytes for buffers that are allocated for this port. .

3.1.2.11.6 bEnabled

`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.

3.1.2.11.7 bPopulated

`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.

3.1.2.11.8 eDomain

`eDomain` is a read-only field that indicates the domain of the port. This field determines the contents of the `format` union explained in section 3.1.2.11.9.

3.1.2.11.9 format

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.

3.1.3 OMX_PORTDOMAINTYPE

Table 3-8 enumerates the fields used in the OMX_PARAM_PORTDEFINITIONTYPE structure to define the domain of the port.

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.

Table 3-8. Port Domain Names

3.1.4 OMX_HANDLETYPE

The OMX_HANDLETYPE structure defines the component handle as seen by the IL client. The component handle is used 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 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, although some may return an error because the state is inappropriate for the access.

3.2 OpenMAX Core Methods/Macros

The OpenMAX core implements the main interface for an IL client that wants to use OpenMAX components. For efficiency, OpenMAX IL defines a set of OpenMAX core macros that map on one-to-one basis to most OpenMAX 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/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 component port.
- Send an empty buffer to an OpenMAX component port.
- Send commands to a component.
- Get the actual state of the component.

- Get references to OpenMAX component-proprietary parameters.

The OpenMAX Core also implements methods for the following:

- Initializing/de-initializing the whole OpenMAX IL Core
- Getting an OpenMAX component handle
- Releasing an OpenMAX component handle
- Detecting all OpenMAX components available on the platform at run time
- Setting up data tunnels among OpenMAX components

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 3-9 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 should transition to the OMX_StateInvalid state when a critical error occurs. 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.

	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	OMX_ComponentDeInit	OMX_Init	OMX_Deinit	OMX_ComponentNameEnum	OMX_GetHandle	OMX_FreeHandle	OMX_SetupTunnel	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		
OMX_ErrorInsufficient Resources		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			
OMX_ErrorInvalid ComponentName																		X				
OMX_ErrorComponent NotFound																		X				
OMX_ErrorInvalidComponent	X	X	X	X	X	X	X	X	X	X	X	X	X	X				X	X	X		
OMX_ErrorBadParameter	X	X	X	X	X	X	X	X	X	X	X	X	X	X			X	X	X	X		
OMX_ErrorNotImplemented																				X		
OMX_ErrorUnderflow																					X	
OMX_ErrorOverflow																					X	
OMX_ErrorHardware																					X	X
OMX_ErrorInvalidState	X	X	X	X	X	X	X		X	X	X	X	X	X						X	X	X
OMX_ErrorStreamCorrupt																					X	
OMX_ErrorPorts NotCompatible																				X		
OMX_ErrorResourcesLost																					X	
OMX_ErrorNoMore			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																	
OMX_ErrorTimeout		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X		
OMX_ErrorSameState																					X	
OMX_ErrorResources Preempted																					X	
OMX_ErrorPortUnResponsive DuringAllocation																					X	
OMX_ErrorPortUnresponsive DuringDeallocation																					X	
OMX_ErrorPortUnresponsive DuringStop																					X	
OMX_ErrorIncorrectState Transition		X																				
OMX_ErrorIncorrectState Operation				X					X	X		X	X	X						X		
OMX_ErrorUnsupported Setting				X		X																
OMX_ErrorUnsupportedIndex			X	X	X	X	X															
OMX_ErrorBadPortIndex		X	X	X	X	X	X		X	X	X	X	X							X		
OMX_ErrorPortUnpopulated																					X	

Table 3-9. Error Codes

3.2.2 Macros

This section describes the OpenMAX core macros.

Table 3-10 defines which macros may be called on a component in each component state.

	OMX_GetComponent	OMX_SendCommand	OMX_GetParameter	OMX_SetParameter	OMX_GetConfig	OMX_SetConfig	OMX_GetExtension	OMX_GetState	OMX_UseBuffer	OMX_AllocateBuff	OMX_FreeBuffer	OMX_EmptyThisBuf	OMX_FillThisBuff	OMX_ComponentDeI	OMX_SetupTunnel
OMX_StateLoaded	X	X	X	X	X	X	X	X	X	X	X			X	X
OMX_StateIdle	X	X	X		X	X	X	X			X	X	X	X	
OMX_StateExecuting	X	X	X		X	X	X	X			X	X	X	X	
OMX_StatePaused	X	X	X		X	X	X	X			X	X	X	X	
OMX_StateWaitForResources	X	X	X	X	X	X	X	X	X	X	X			X	
OMX_StateInvalid								X			X			X	
Disabled Port	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Table 3-10. Valid Component Calls

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 msec.

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 of less than 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 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 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 version structure that the component will populate. `SpecVersion` is the version of the specification that the component was built against. Note that this value may or may not match the version of the structure. For example, if the component was built against the version 2.0 specification but the IL client, which creates the structure, was built against the version 1.0 specification, the versions would be different.
pComponentUUID [out]	A pointer to the universal unique identifier (UUID) of the component, which the component will fill in. The UUID is a unique identifier that is set at run time for the component and is unique to each instance of the component.

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,
      &CompUUID);
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 msec. The component normally executes the command outside the context of the call, though a solution without threading may elect to execute it in context. In either case, 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. If the component fails to execute the command, the component generates an `OMX_EventError` and passes the appropriate error as a parameter.

The component may elect to queue commands for later execution. The only restriction is that the completion shall be done in the same order as the requests arrived.

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 3.3.6.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.

If the component successfully transitions to the new state, it notifies the IL client of the new state via the OMX_EventCmdComplete event, indicating OMX_CommandStateSet for `nData1` and the new state for `nData2`. If a state transition fails, the component shall notify the IL client of the error that prevented it via OMX_EventError event. 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

This 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 -1, the component shall flush all ports.

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

For each port that the component successfully flushes, the component shall send an `OMX_EventCmdComplete` event, indicating `OMX_CommandFlush` for `nData1` and the individual port index for `nData2`, even if the flush resulted from using a value of -1 for `nParam`. If a flush fails, the component shall notify the IL client of the error via an OMX_EventError event.

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 -1, the component shall disable all ports. A disabled port has no buffers and is not connected to either the IL client or another port via a tunnel. A disabled port does not allocate buffers 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 or set up a tunnel on it regardless of the component state. Thus the `OMX_CommandPortDisable` command, in co-operation with `OMX_CommandPortEnable`, is useful for the dynamic reconfiguration or re-tunneling of a port.

The port must 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 IL client shall return any buffers it is holding to the supplier port via `OMX_EmptyThisBuffer`/`OMX_FillThisBuffer` if tunneling or `EmptyBufferDone`/`FillBufferDone` if not tunneling. Then, the IL client shall wait for the supplier port 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 IL client shall wait for the non-supplier port to return all buffers via `OMX_EmptyThisBuffer` or `OMX_FillThisBuffer`. Then, the IL client shall free the buffers via `OMX_FreeBuffer` before completing the disable command.

For each port that the component successfully disables, the component shall send an OMX_EventCmdComplete event indicating OMX_CommandPortDisable for `nData1` and the individual port index for `nData2`, even if using a value of -1 for `nParam` caused the port to be disabled. If the disable operation fails, the component shall notify the IL client of the error via the OMX_EventError event.

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 -1, the component shall enable all ports. 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_WaitForResources state to the OMX_IdleState.
- 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 must 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_WaitForResources, 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_Executing state, then that port shall begin transferring buffers.

For each port that the component successfully enables, the component shall send an OMX_EventCmdComplete event, indicating OMX_CommandPortEnable for `nData1` and the individual port index for `nData2`, even if using the value of -1 for `nParam` caused the enable operation. If a port enablement operation fails, the component shall notify the IL client of the error via OMX_EventError event.

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 port that the component successfully marks a buffer, the component shall send an OMX_EventCmdComplete event indicating OMX_CommandPortMarkBuffer for `nData1` and the individual port index for `nData2`. If a mark operation fails, the component shall notify the IL client of the error via OMX_EventError event.

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.

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.

```
/* 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 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 msec.

The `OMX_GetParameter` macro is defined as follows.

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

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.
ComponentParameterStructure [in,out]	A pointer to the IL client-allocated structure that the component fills

Section 3.3.7 describes the corresponding function that each component implements.

3.2.2.8.1 Prerequisites for This Method

The macro can be invoked when the component is in any state except the OMX_StateInvalid state.

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);
}
```

3.2.2.9 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 msec.

The `OMX_SetParameter` macro is defined as follows.

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

The parameters are as follows.

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

Section 3.3.8 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.

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);
```

3.2.2.10 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 `nParamIndex` 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 msec.

The `OMX_GetConfig` macro is defined as follows.

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

The parameters are as follows.

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

Section 3.3.9 describes the corresponding function that each component implements.

3.2.2.10.1 Prerequisites for This Method

The macro can be invoked when the component is in any state except the OMX_StateInvalid state.

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);
```

3.2.2.11 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 msec.

The `OMX_SetConfig` macro is defined as follows.

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

The parameters are as follows.

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

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

3.2.2.11.1 Prerequisites for This Method

The macro can be invoked when the component is in any state except the OMX_StateInvalid state.

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);
```

3.2.2.12 OMX_GetExtensionIndex

The `OMX_GetExtensionIndex` macro will invoke a component to translate from a standardized OpenMAX or vendor-specific extension string for a configuration or a parameter into an OpenMAX structure 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 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 msec.

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 3.3.11 describes the corresponding function that each component implements.

3.2.2.12.1 Prerequisites for This Method

The macro can be invoked when the component is in any state except the OMX_StateInvalid state.

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 msec.

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 3.3.12 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 already allocated by the IL client or a buffer already 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.

The `OMX_UseBuffer` 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 msec.

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.
pBuffer [in]	A pointer to the memory buffer area to be used.

Section 3.3.14 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[j]);
}
```

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. This call is a blocking call that shall be performed 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 it 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 the OMX_StateExecuting, the OMX_StatePause, or the OMX_StateIdle states.

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 msec.

The `OMX_AllocateBuffer` macro is defined as follows.

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

The parameter are as follows.

Paramter	Description
hComponent [in]	The handle of the 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]	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 3.3.15 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. 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, but may result in the port sending an OMX_ErrorPortUnpopulated error if the call is not performed as described. The call is made from buffer supplier ports when tunneling to release buffer headers from the port that the supplier port is tunneling with.

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

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 3.3.16 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 `nFilledLength` field of the buffer header will not be zero. If the buffer contains no data, the value of

nFilledLength is 0x0. The OMX_EmptyThisBuffer macro is invoked to pass buffers containing data when the component is in or making a transition to the OMX_StateExecuting or in the OMX_StatePaused state.

When a port is non-tunneled, buffers sent to OMX_EmptyThisBuffer are returned to the IL client with the EmptyBufferDone callback once they have been emptied.

When a port is tunneled, buffers sent to OMX_EmptyThisBuffer are sent to the tunneled port once they are emptied so long as the component is in the OMX_StateExecuting state. Buffers are returned to the input port that supplied them using OMX_EmptyThisBuffer whenever the tunneled port is flushed or disabled. Buffers are also returned to the input port that supplied them when the component calling OMX_FillThisBuffer is transitioning from the OMX_StateExecuting state or the OMX_StatePaused state to the OMX_StateIdle state.

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 msec.

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 3.3.17 describes the corresponding function that each component implements.

3.2.2.17.1 Prerequisites for This Method

The component must be in the appropriate state as shown in Table 3-10.

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 is invoked to pass buffers containing no data when the component is in or making a transition to the OMX_StateExecuting state or is in the OMX_StatePaused state.

When a port is non-tunneled, buffers sent to `OMX_FillThisBuffer` return to the IL client with the `FillBufferDone` callback once they have been filled.

When a port is tunneled, buffers sent to `OMX_FillThisBuffer` are sent to the tunneled port once they are filled so long as the component is in the OMX_StateExecuting state. Buffers are returned to the output port that supplied them using `OMX_FillThisBuffer` whenever the tunneled port is flushed or disabled. Buffers are also returned to the output port that supplied them when the component that calls `OMX_FillThisBuffer` is transitioning from the OMX_StateExecuting state or OMX_StatePaused state to the OMX_StateIdle state.

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 msec.

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 input port that receives the buffer.

Section 3.3.18 describes the corresponding function that each component implements.

3.2.2.18.1 Prerequisites for This Method

The component must be in the appropriate state as shown in Table 3-10.

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]);
   }
}
```

3.2.3 Functions

This section describes the functions in the OpenMAX IL API.

3.2.3.1 OMX_Init

The `OMX_Init` method initializes the OpenMAX core. `OMX_Init` shall be the first call made into OpenMAX and should be executed only one time without an intervening `OMX_Deinit` call. If `OMX_Init` is called twice, `OMX_ErrorNone` is returned but the init request is ignored. The core should return from this call within 20 msec.

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 error will be returned. The OpenMAX 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 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 `OMX_Deinit` method de-initializes the OpenMAX core. `OMX_Deinit` should be the last call made into the OpenMAX core after all OpenMAX-related resources have been released. The core should return from this call within 20 msec. While it may be preferable to have the core command each of the components back to the loaded state and then de-initialize them, doing so may require more than the recommended 20 msec call time. It further requires the OpenMAX core to track all component handles, which may add unnecessary complexity for some platforms.

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 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 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 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 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 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 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 msec.

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.

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

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

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 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 msec. The IL client should call `OMX_FreeHandle` only when the component is in the OMX_StateLoaded or the OMX_StateInvalid state; calling `OMX_FreeHandle` from any other state may result in the component taking longer than the recommended 20 msec execution time, and is provided only as a failure recovery mechanism.

`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 freed.

3.2.3.5.1 Prerequisites for This Method

The component should be in the OMX_StateLoaded or the OMX_StateInvalid 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 component's `ComponentTunnelRequest()` method to set up the tunnel.

When setting up non-tunneled communication for an input port, the value of the `hOutput` parameter shall be 0x0. When setting up non-tunneled communication for an output port, the value of `hInput` shall be 0x0.

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.

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 component should return from this call within 20 msec.

This method is unsupported by base profile components, which shall return `OMX_ErrorNotImplemented`.

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. If the value of this parameter is 0x0, the `hPortInput` port on the `hInput` component will be set up for non-tunneled communication.
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. If the value of this parameter is 0x0, the `hPortOutput` port on the `hOutput` component will be set up for non-tunneled communication.
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.

3.2.3.6.2 Results/Outputs for This Method

If the method returns successfully when both an output and input component are supplied, tunneled or proprietary communication has been set up between the specified output and input ports. When only an output or an input component is supplied or if an error occurs during processing, the ports are set up for non-tunneled communication.

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);
```

3.3 OpenMAX Component Methods and Structures

OpenMAX 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.

3.3.1 nSize

`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 an output from a function.

3.3.2 nVersion

`nVersion` is the version of the OpenMAX specification that the structure is built against. The creator of this structure is responsible for initializing this value. Every user of this structure should verify that it knows how to use the exact version of this structure.

3.3.3 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.

3.3.4 pApplicationPrivate

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

3.3.5 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 for a description of its semantics.

GetComponentVersion is defined as follows.

```
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);
```

3.3.6 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 for a description of its semantics.

SendCommand is defined as follows.

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

3.3.7 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 for a description of its semantics.

GetParameter is defined as follows.

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

3.3.8 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 3.2.2.9 for a description of its semantics.

SetParameter is defined as follows.

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

3.3.9 GetConfig

The IL client calls the GetConfig component method via the `OMX_GetConfig` core macro. See the definition of OMX_GetConfig in section 3.2.2.10 for a description of its semantics.

GetConfig is defined as follows.

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

3.3.10 SetConfig

The IL client calls the SetConfig component method via the `OMX_SetConfig` core macro. See the definition of `OMX_SetConfig` in section 3.2.2.11 for a description of its semantics.

SetConfig is defined as follows.

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

3.3.11 GetExtensionIndex

The IL client calls the GetExtenstionIndex 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.

GetExtensionIndex is defined as follows.

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

3.3.12 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 for a description of its semantics.

GetState is defined as follows.

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

3.3.13 ComponentTunnelRequest

The `OMX_ComponentTunnelRequest` method will interact with another OpenMAX 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 interop profile-conformant component shall support tunneling to a component with compatible parameters. 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`.
2. Set the OMX_PORTTUNNELFLAG_READONLY flag to indicate that buffers from this output port are read-only and that the buffers cannot be shared through components or modified.

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 parameter for the `pTunnelComp` parameter, the port should be set up for non-tunneled communication with the IL client.

The component should return from this call within five msec.

ComponentTunnelRequest is defined as follows.

```
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);
```

The parameters are 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.

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 callbacks field is filled by the output port component. The read-only flag can be applied by both components.

3.3.13.1 Prerequisites for This Method

The component shall be in the OMX_StateLoaded state.

3.3.13.2 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);
```

3.3.14 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 for a description of its semantics.

UseBuffer is defined as follows.

```
    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);
```

3.3.15 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 for a description of its semantics.

AllocateBuffer is defined as follows.

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

3.3.16 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 for a description of its semantics.

FreeBuffer is defined as follows.

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

3.3.17 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 for a description of its semantics.

EmptyThisBuffer is defined as follows.

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

3.3.18 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 for a description of its semantics.

FillThisBuffer is defined as follows.

```
OMX_ERRORTYPE (*FillThisBuffer)(
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_BUFFERHEADERTYPE* pBuffer);
```

3.3.19 SetCallbacks

The `SetCallbacks` method will allow the core to transfer the callback structure from the IL client to the component. This is a blocking call. The component should return from this call within five msec.

SetCallbacks is defined as follows.

```
    OMX_ERRORTYPE (*SetCallbacks)(
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_CALLBACKTYPE* pCallbacks,
            OMX_IN  OMX_PTR 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.

3.3.19.1 Prerequisites for This Method

The component shall be in the OMX_StateLoaded state.

3.3.19.2 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_0;
pComp->nSize = sizeof(OMX_COMPONENTTYPE);
OMX_ComponentRegistered[i].pInitialize(hHandle);
pComp->SetCallbacks(hHandle, pCallBacks, pAppData);
```

3.3.20 ComponentDeinit

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

ComponentDeinit is defined as follows.

```
    OMX_ERRORTYPE (*ComponentDeInit)(
            OMX_IN  OMX_HANDLETYPE hComponent);
```

The single parameter is as follows.

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

3.3.20.1 Prerequisites for This Method

There are no prerequisites for this method. The IL client may execute this function regardless of component state so that de-initialization is guaranteed even on components that are unresponsive to state changes. However, executing ComponentDeinit when the component is in the OMX_StateLoaded state is recommended for proper shutdown.

3.3.20.2 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);
OMX_OSAL_Free(pComp);
```

3.4 Calling Sequences

This section describes how the IL client, the OpenMAX 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 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 3-3 shows how an IL client should initialize an OpenMAX component.

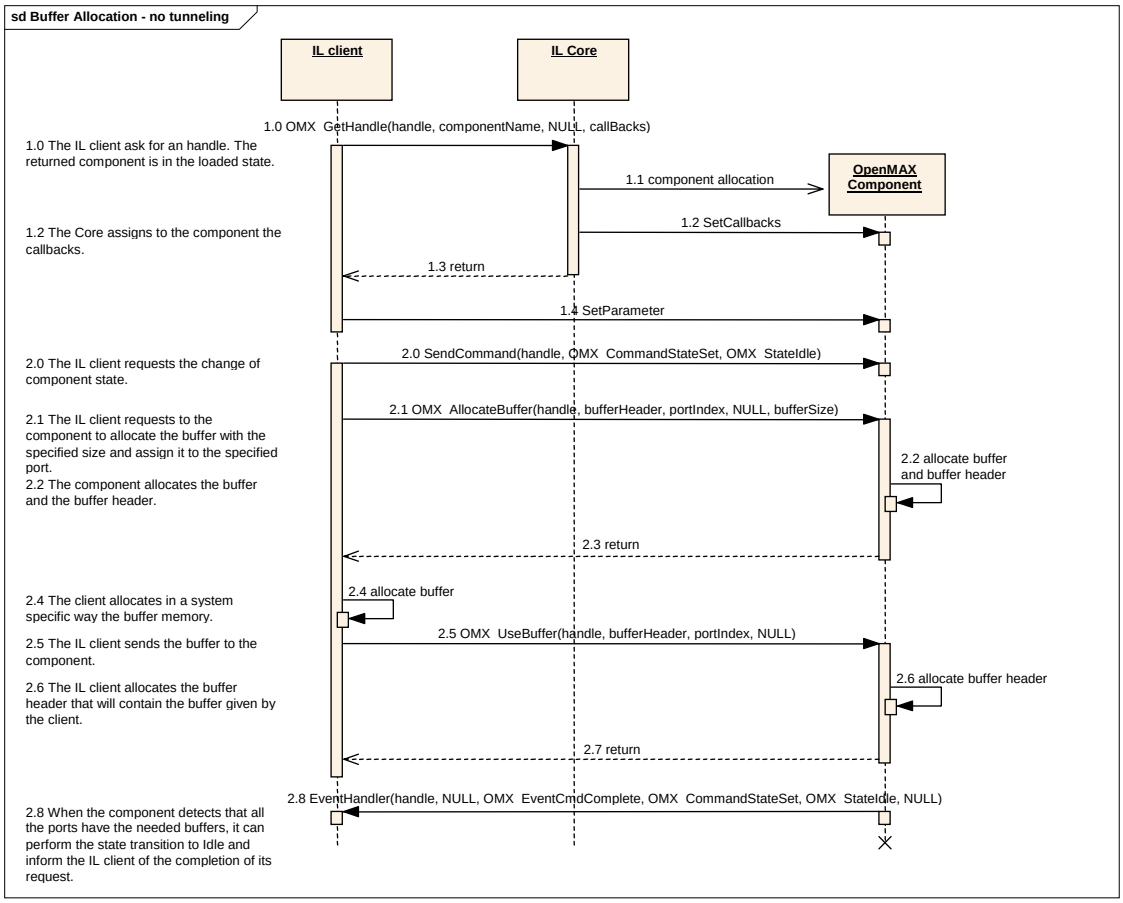


Figure 3-3. Component Initialization

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 setup 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 shall have allocated a buffer and passed it to the component. 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 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 3-4, 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.

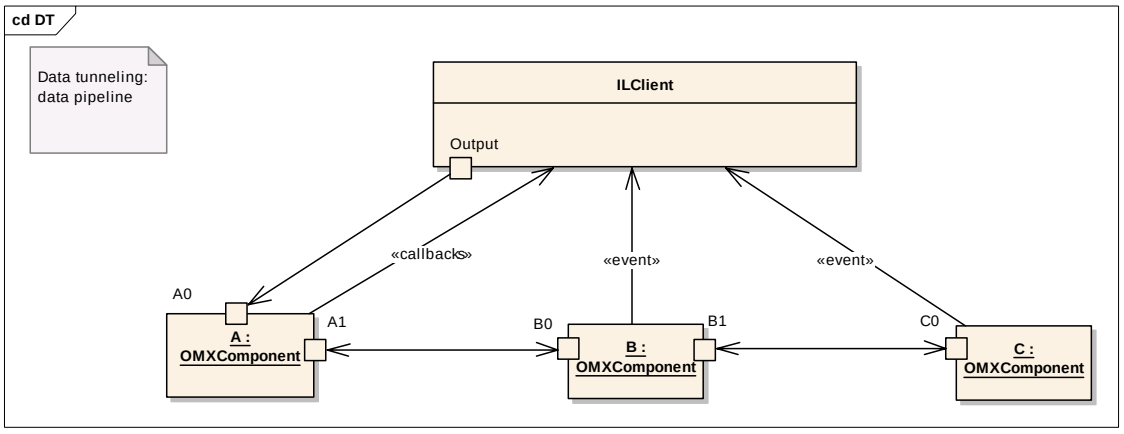


Figure 3-4. Example of Data Tunneling Among OpenMAX Components

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 input 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 buffers 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. Figure 3-5 illustrates the setup process:

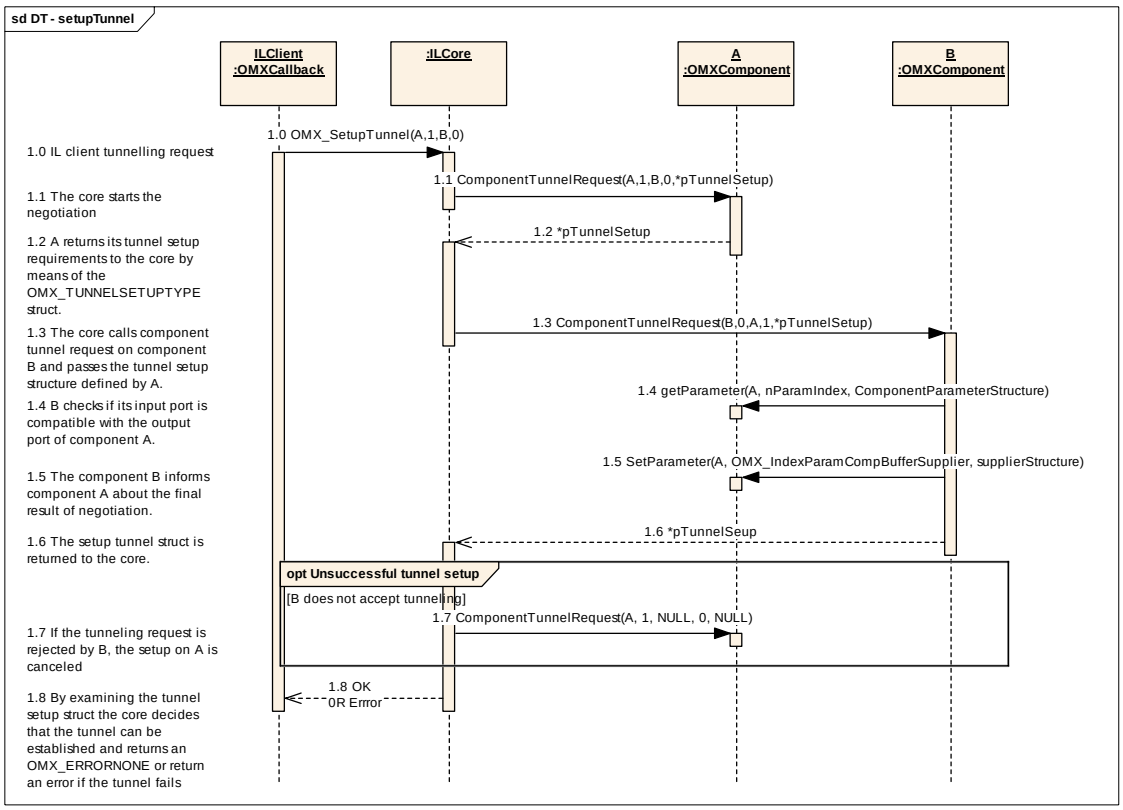


Figure 3-5. Tunnel Setup

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 3.1.2.9 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 3-6:

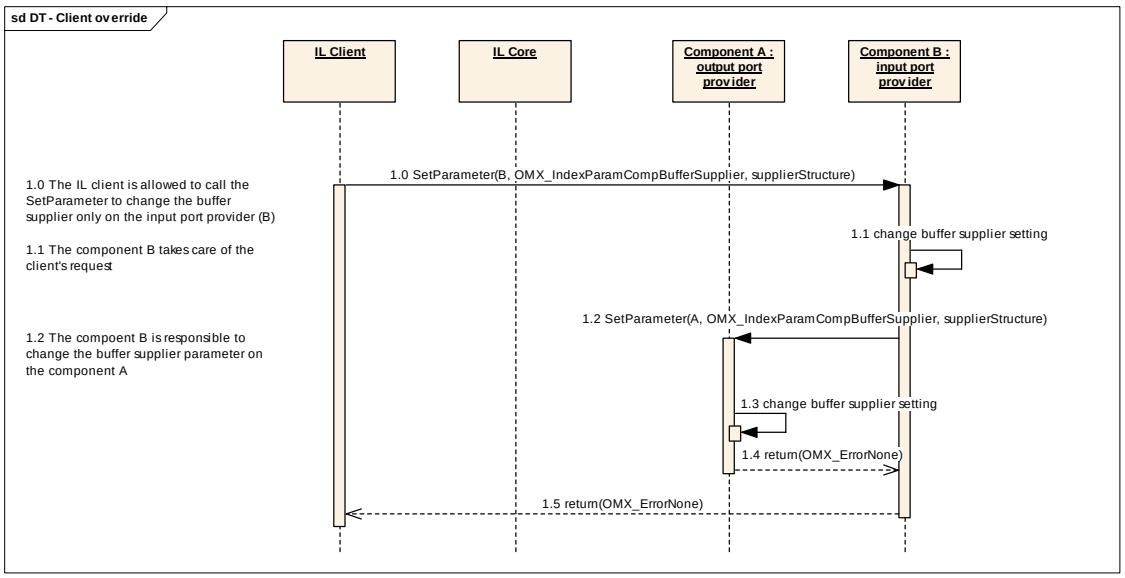


Figure 3-6. IL Client Buffer Supplier Override

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`.

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 3-7 illustrates the state transition behavior in which the tunnels are already created and configured.

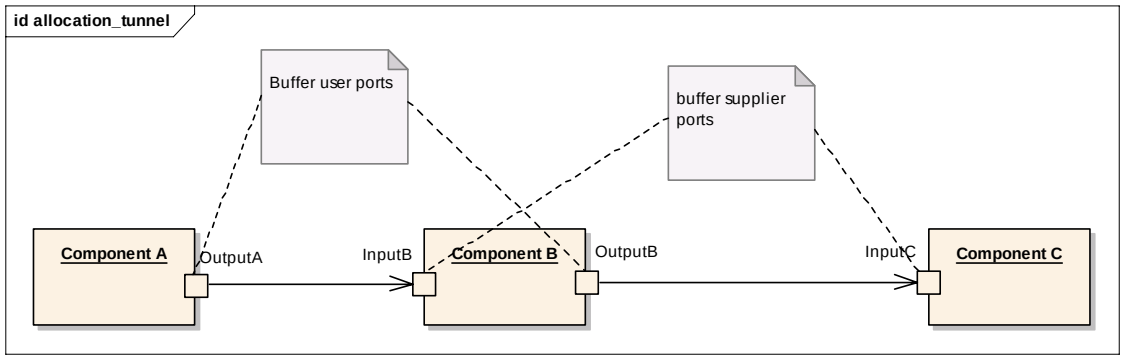


Figure 3-7. Tunneling Example

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 3-8 illustrates the behavior of each tunneled component during the state transition.

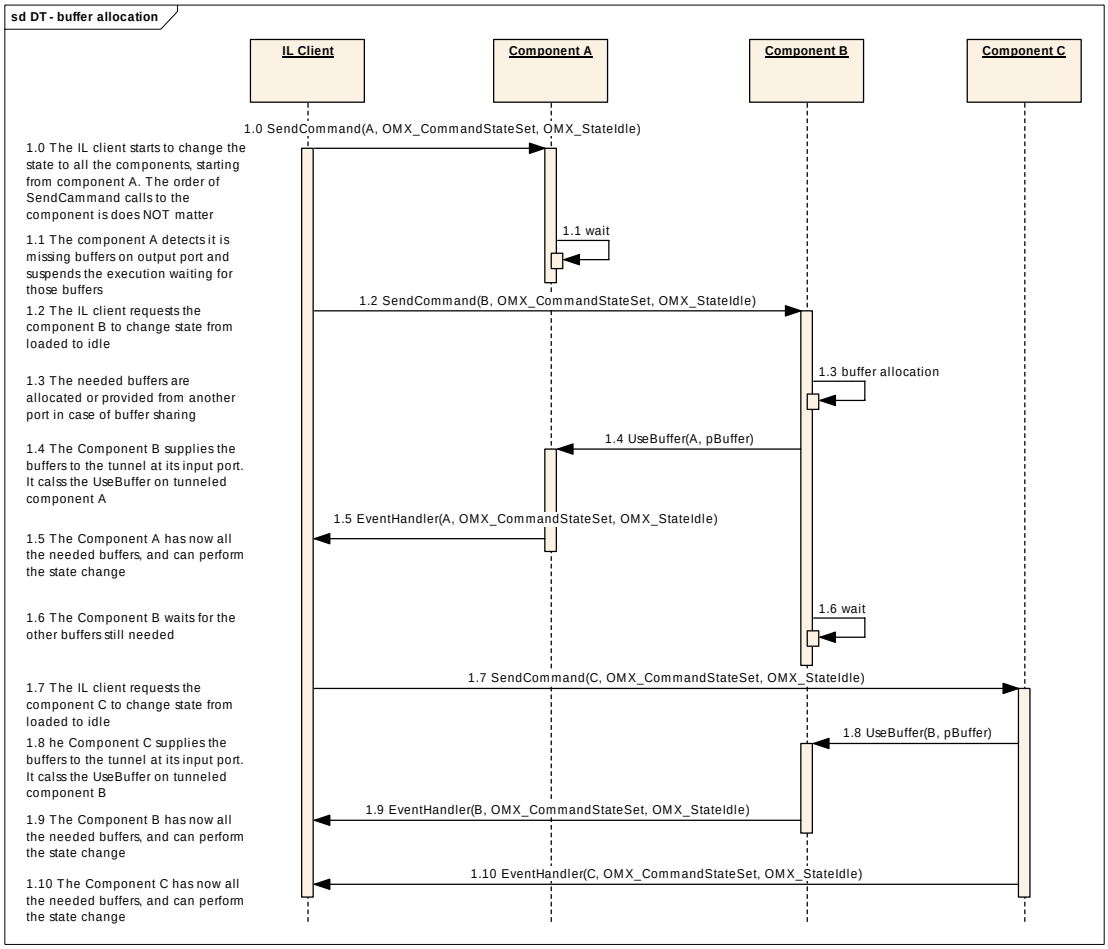


Figure 3-8. State Transition to Idle in the Case of Tunneled Component s

Each supplier port on a component shall pass its buffers 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 3-8, 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 buffers.

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 `UseBuffer` on the output port of component A. Then, component B waits for all of the needed buffers on its output port.

Now component A has all of the needed buffers, 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 gives the buffers needed to 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 obtained all of the needed buffers.

3.4.2 Data Flow

OpenMAX 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 an `OMX_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 3-9 illustrates the dynamic behavior related to data flow.

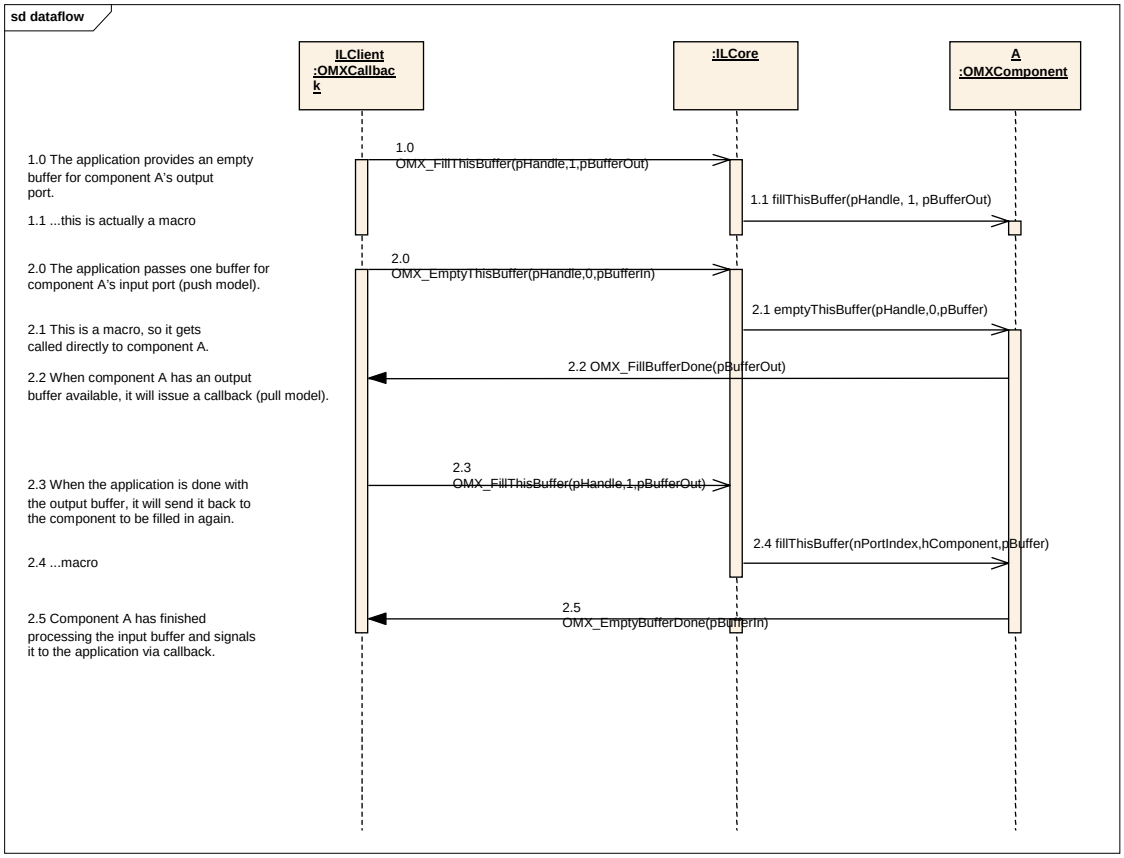


Figure 3-9. Data Flow Between Non-tunneled Components

3.4.2.2 Tunneled Data Flow

In data tunneling, OpenMAX 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 3-10 illustrates this process.

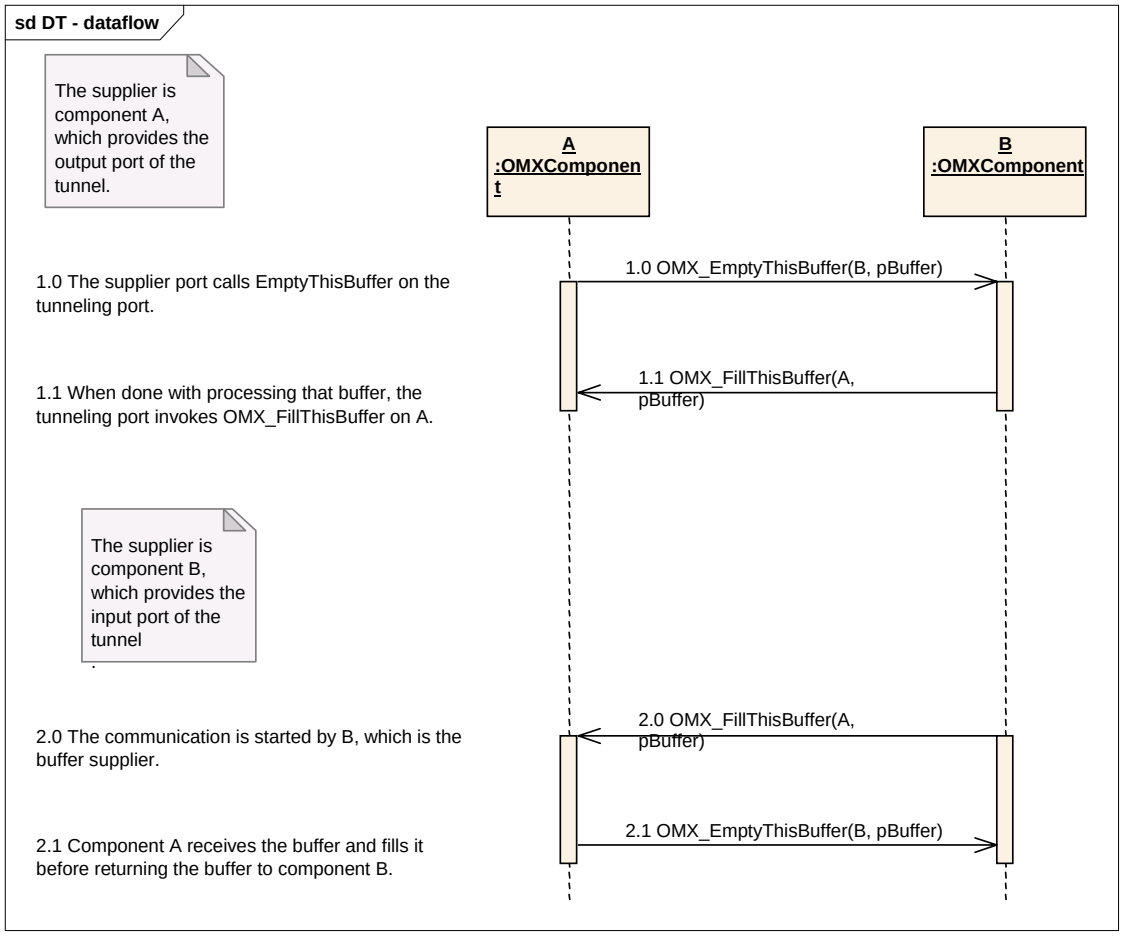


Figure 3-10. Data Flow Between Tunneled Components

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 3-11.

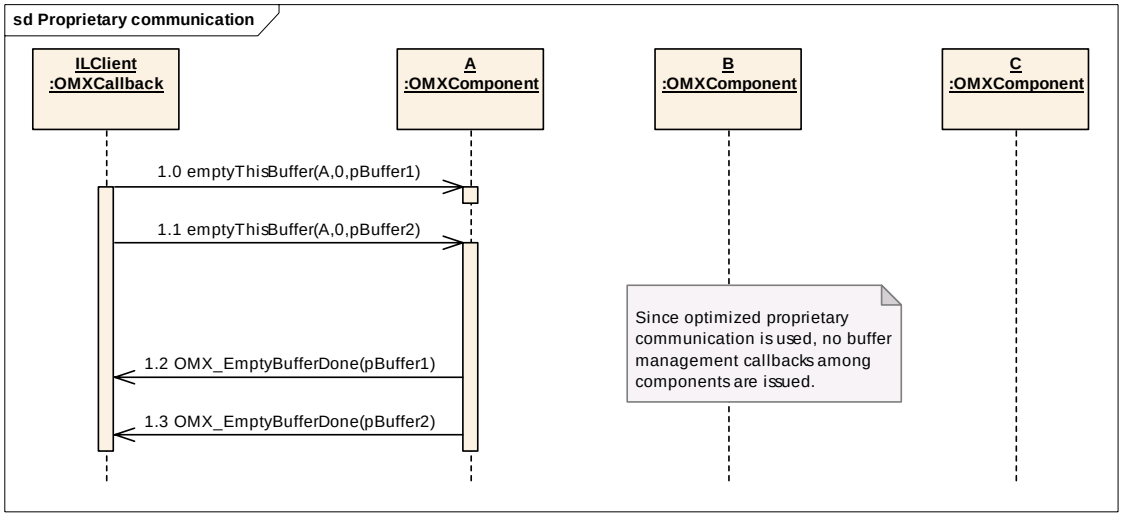


Figure 3-11. Data Flow with Proprietary Communication Between Components

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 `OMX_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 `OMX_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`.

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 3-12 illustrates non-tunneled de-initialization.

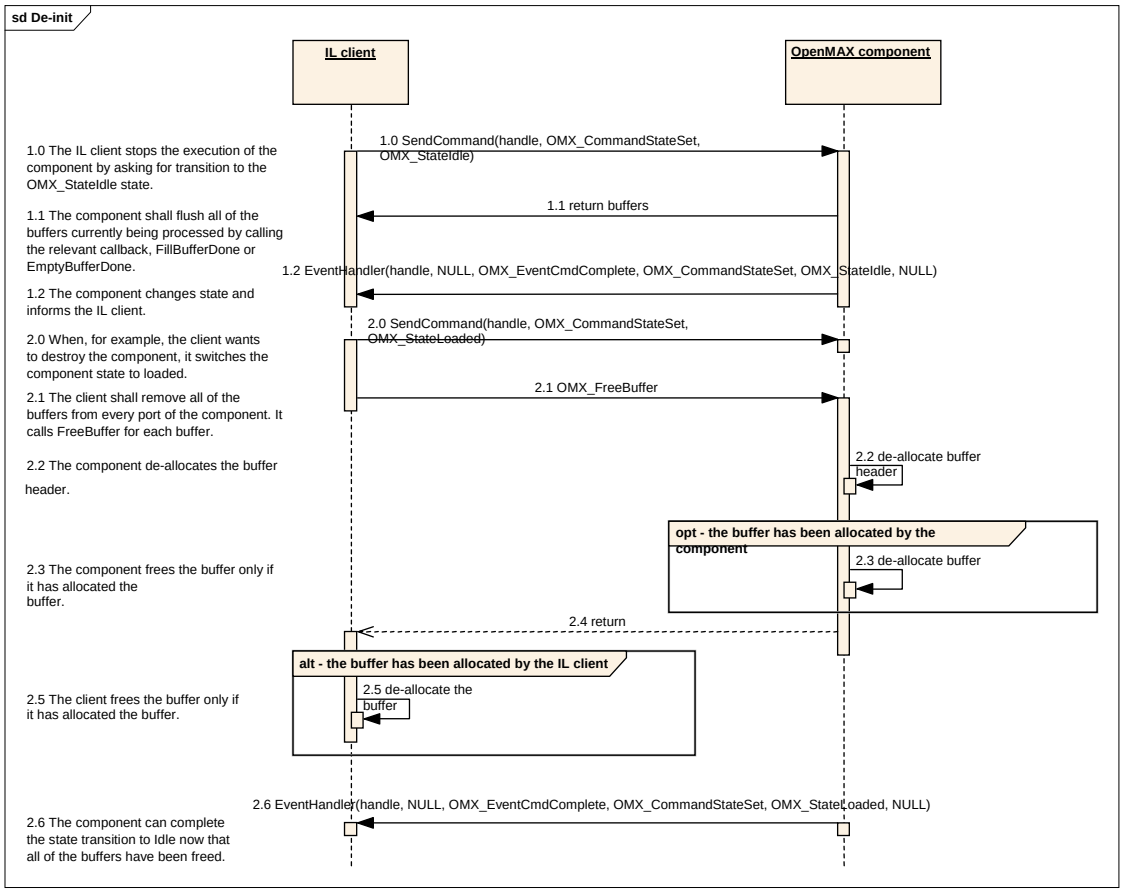


Figure 3-12. De-initialization of Non-tunneled Components

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

Figure 3-13 illustrates the component de-initialization for a port that is tunneled.

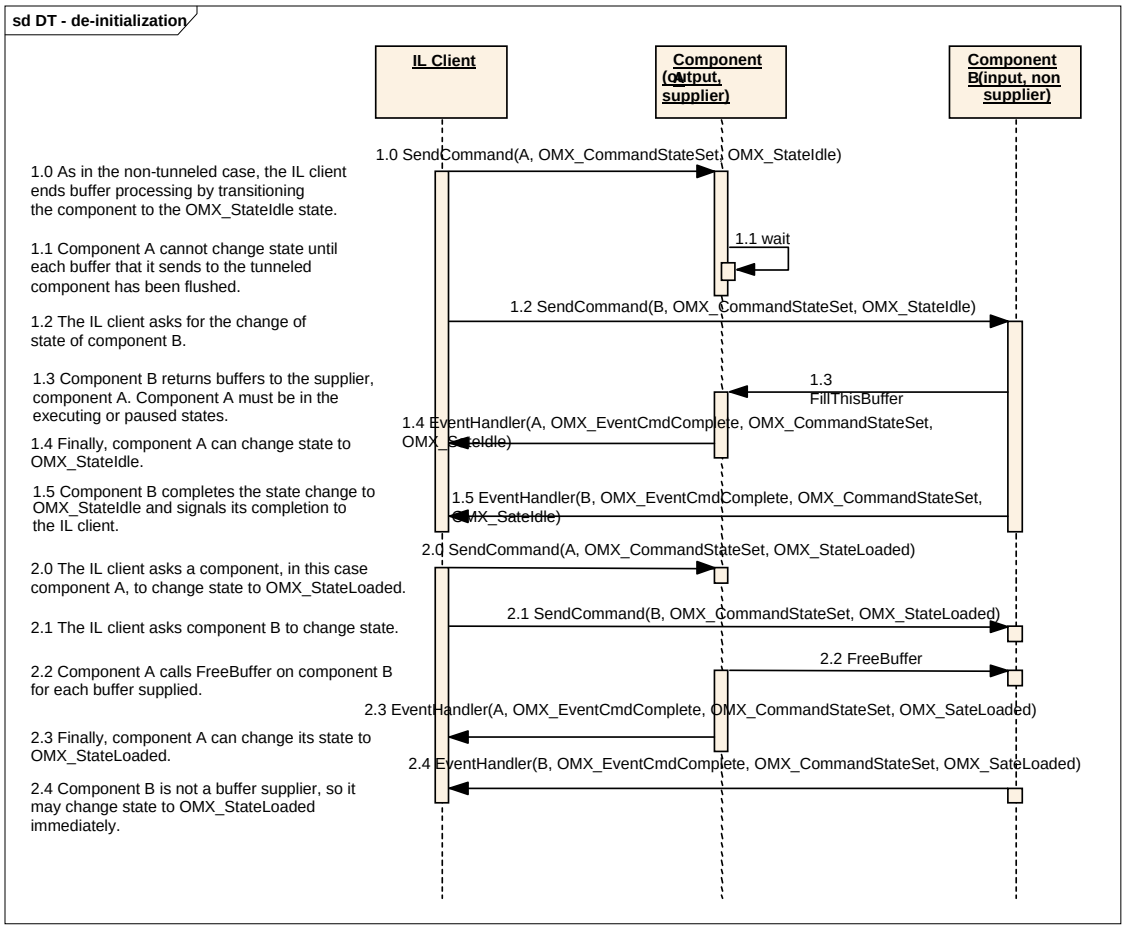


Figure 3-13. De-initialization of Tunneled Components

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 3-14 illustrates the behavior of enabling and disabling tunneled ports.

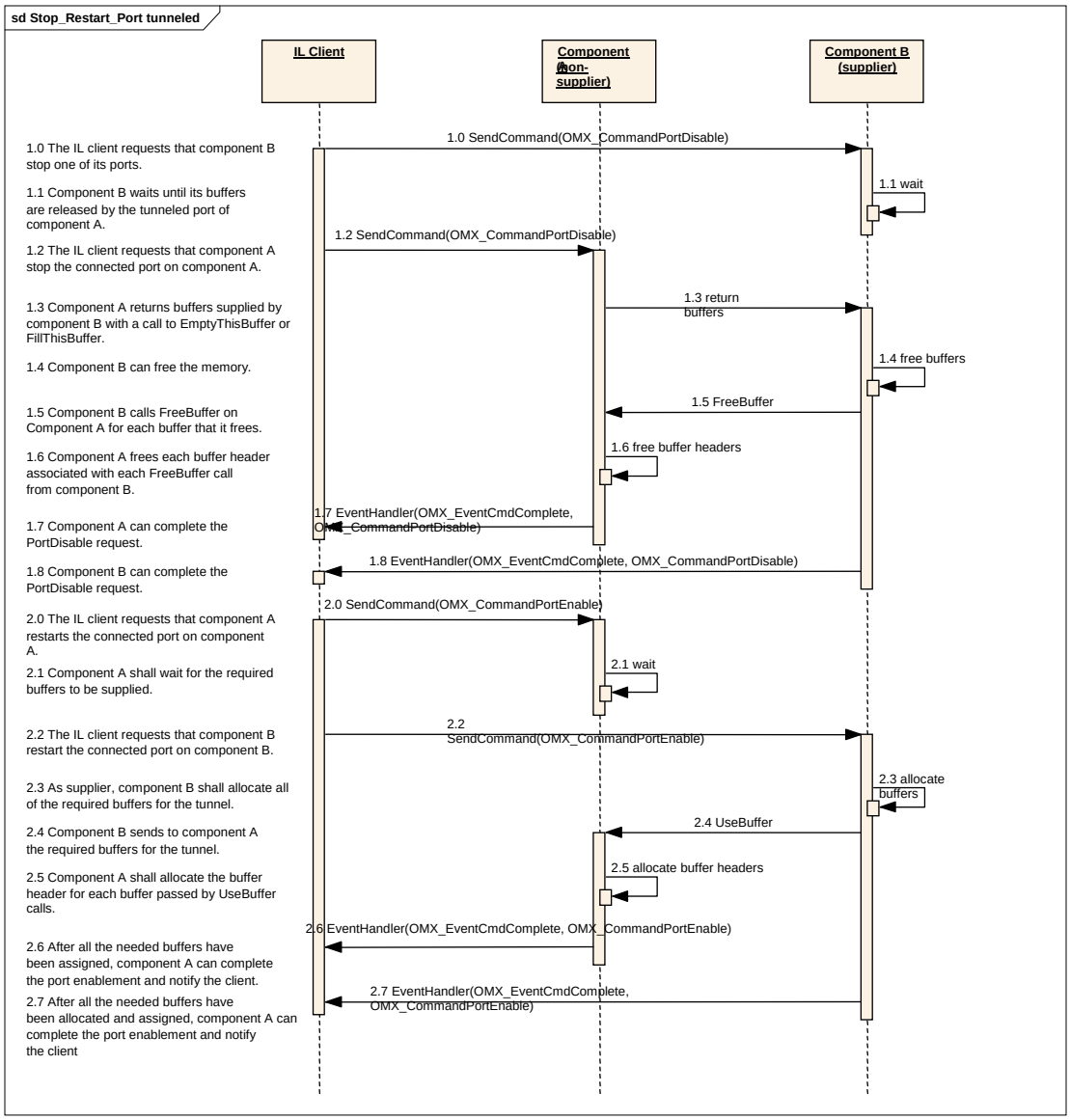


Figure 3-14. Disablement and Enablement of Tunneled Ports

3.4.4.2 Non-tunneled Port Disablement and Enablement

Figure 3-15 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 3.3.15, 3.3.14, and 3.3.16, respectively.

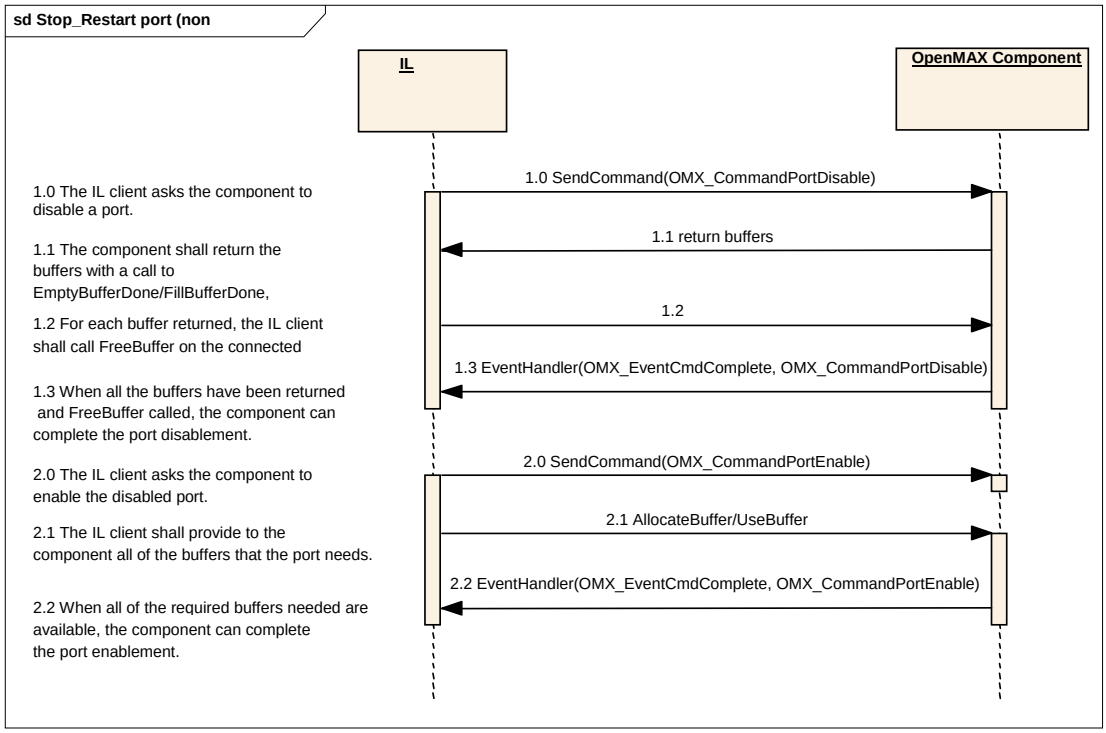


Figure 3-15. Disablement and Enablement of Non-tunneled Ports

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 3-16 shows how a video decoder and a video renderer, both of which exchange data through the IL client, should dynamically change their port settings.

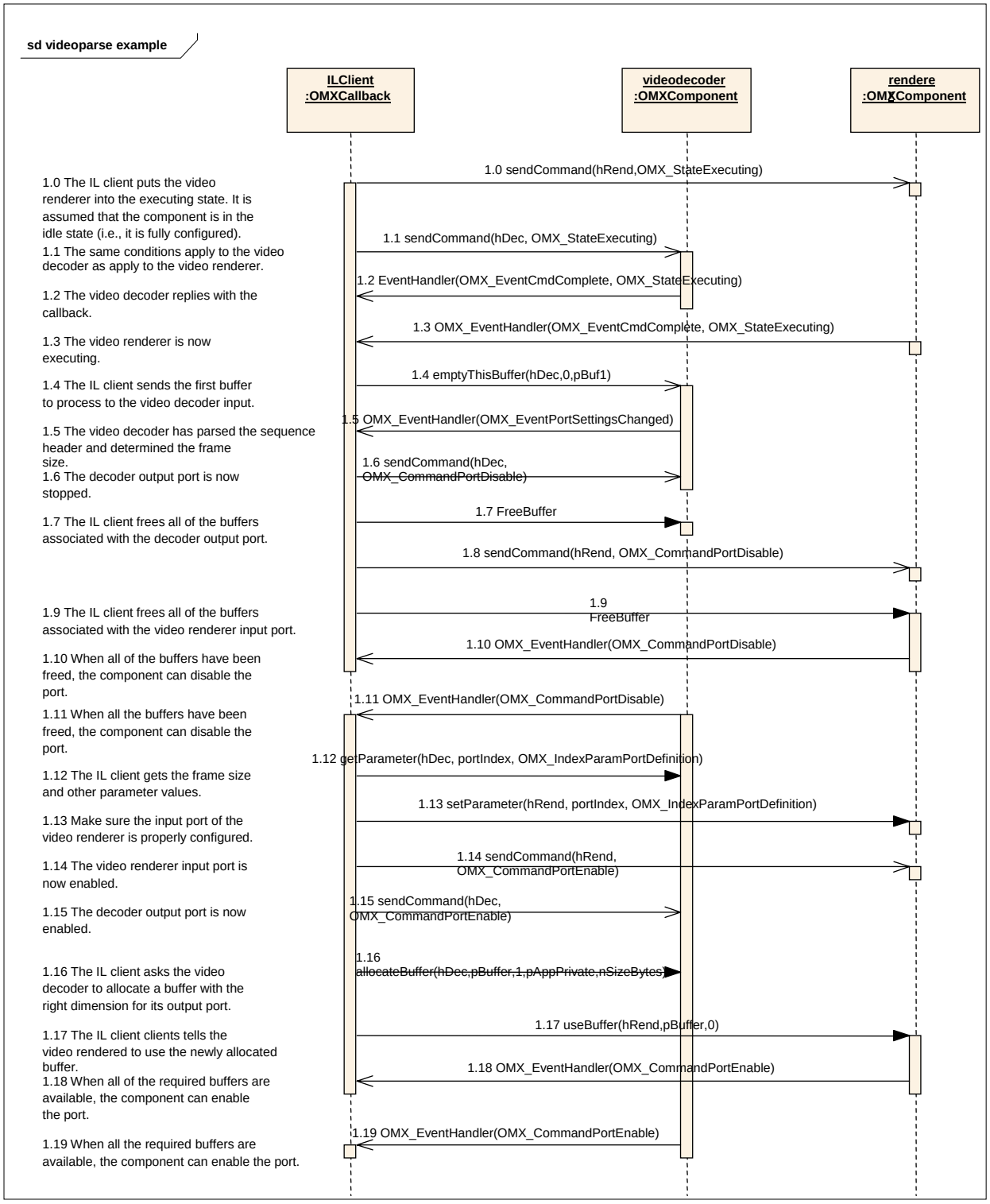


Figure 3-16. Dynamic Port Reconfiguration

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 decoder component will parse it and change its output port settings accordingly.

The OpenMAX decoder component shall then notify the IL client by generating the `OMX_PortSettingsChanged` 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 shall then read the new port settings with `OMX_GetConfig` and allocate one or more buffers with the right dimensions for the output port. Once the buffers are allocated, they will be also communicated to the video renderer using `OMX_UseBuffer` (1.17). The input port of the video renderer shall also be set up with `OMX_SetConfig` (1.18).

Finally, ports can be enabled and normal processing resumes.

3.4.6 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 standard definition. An IL client may use the resource manager entry points.

Figure 3-17 proposes the behavior of an IL client that ignores the resource manager. The resource manager handles the component internally only, and the IL client has to take no special action.

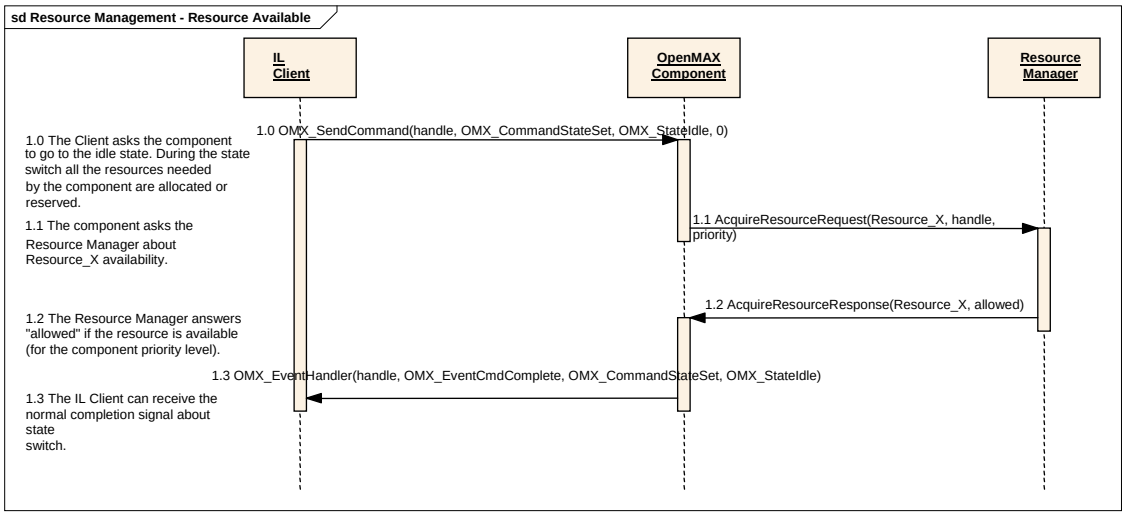


Figure 3-17. Transition from Loaded to Idle with Resource Management

In Figure 3-17, the IL client is unaware of the existence of a resource manager. In the implementation of the OpenMAX component, an asynchronous call to the resource manager is implemented.

The OpenMAX component provides a callback to the resource manager, which receives the signal for the completion of the request.

Figure 3-17 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 3-18 presents a more complex use case.

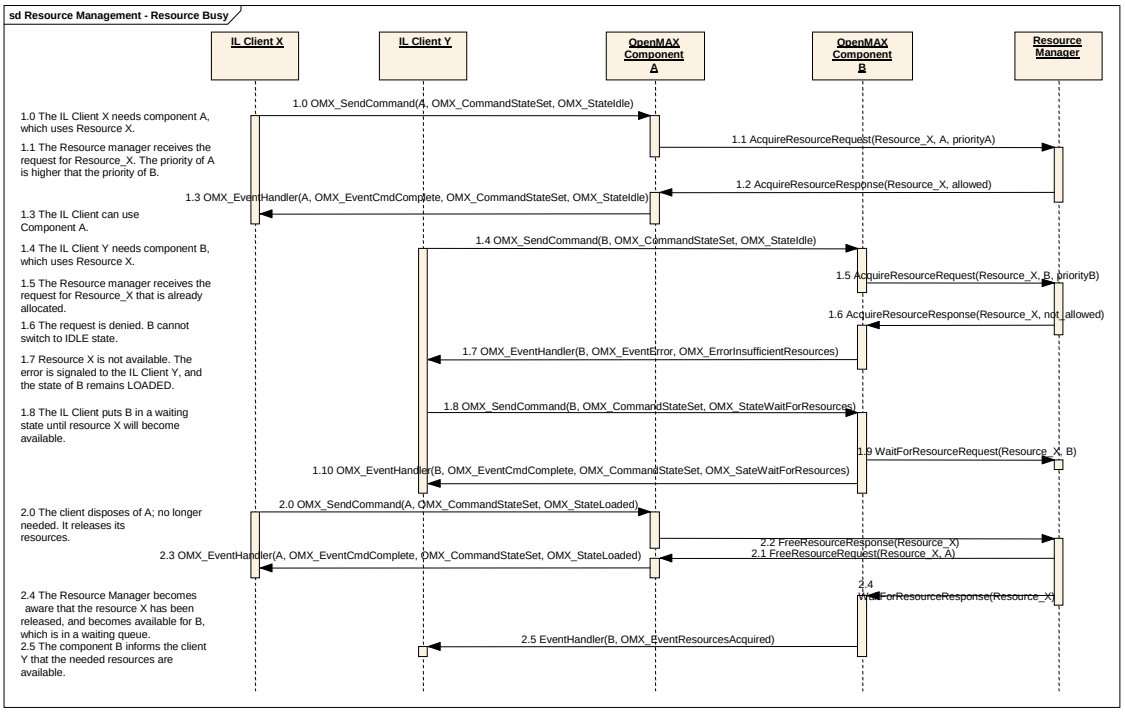


Figure 3-18. Busy Resource Management

In Figure 3-18, two different OpenMAX 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 Loaded to Idle 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 the idle state 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 the Loaded state.

Figure 3-18 also shows the behavior of components when resource X becomes available. Component A changes state to Loaded 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 3-18, 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 standard in general.

Figure 3-19 presents an example of a client that actively uses the resource management API.

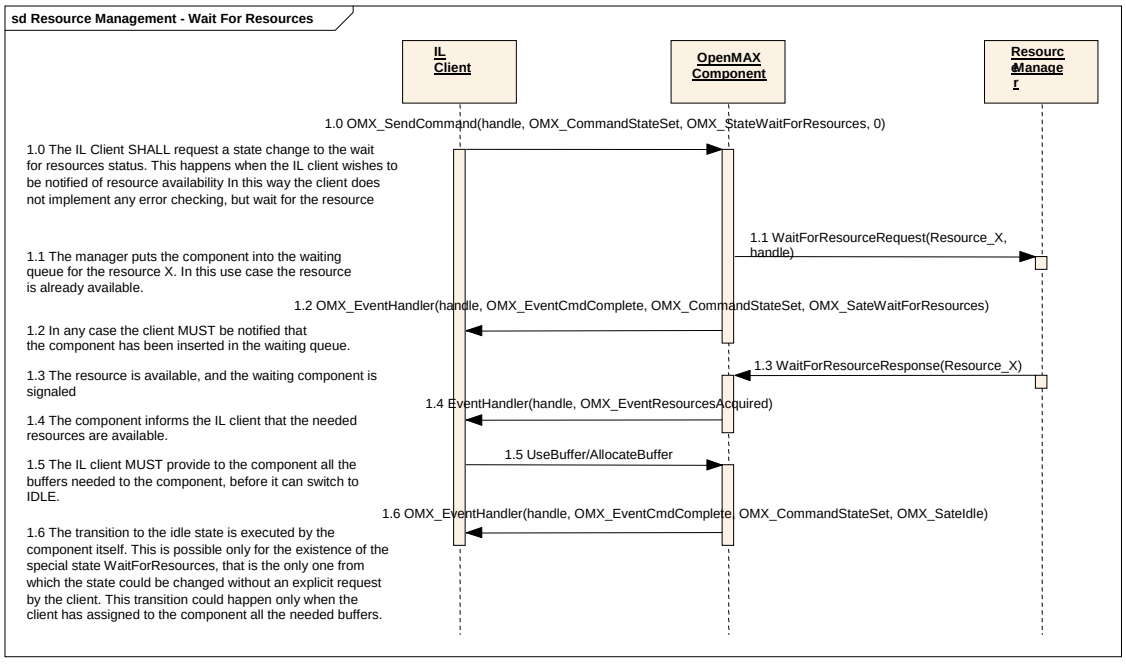


Figure 3-19. State Change from Loaded to WaitForResources

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 OMX_EventHandler callbacks that correspond to two different state changes.

The two functions `WaitForResourceRequest` and `WaitForResourceResponse` in Figure 3-19 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 Loaded, as shown in Figure 3-20.

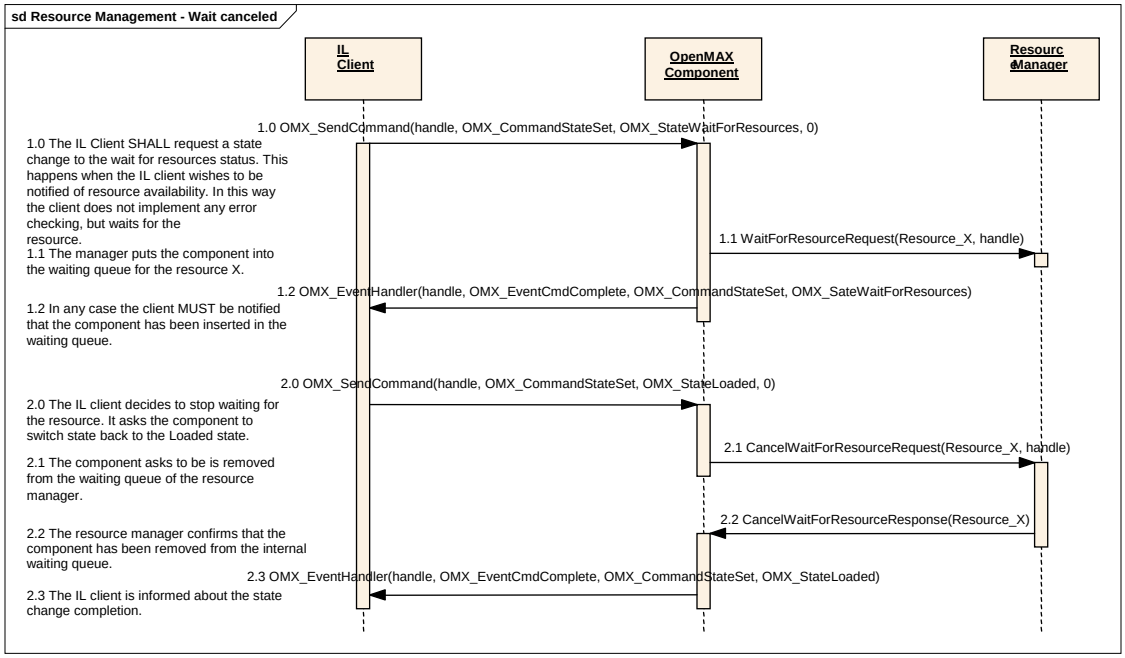


Figure 3-20. Remove Component from Waiting Status

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 standard components are shown in this example.

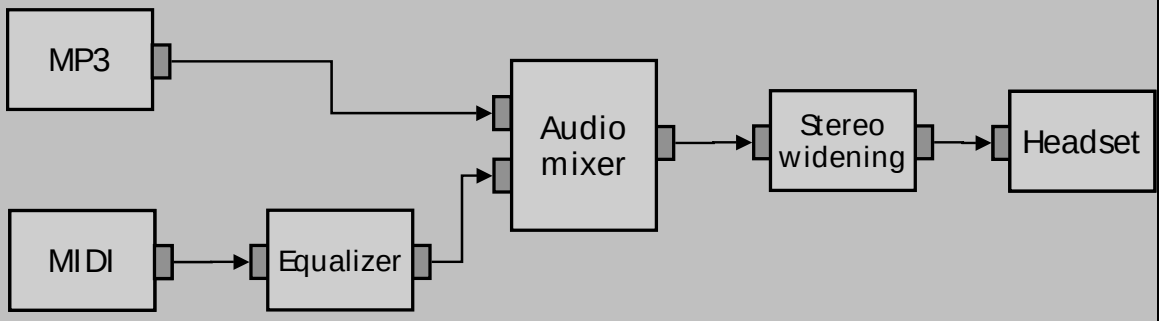


Figure 4-1. Audio Payback Processing Chain

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 codecs.

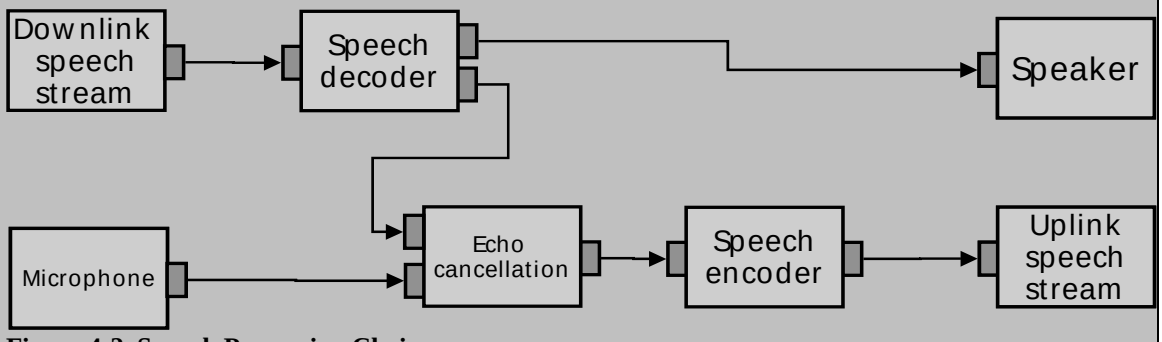


Figure 4-2. Speech Processing Chain

4.1.2 Special Issues

Some audio formats have special or unique requirements that are different from other audio formats, or even from other domains. These issues are described in the following sections.

4.1.2.1 Minimum Buffer Payload Size for Uncompressed Data

OpenMAX 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 msec. This means that an output port of a PCM component shall produce at least five msec of audio data for each buffer. The minimum payload size is applied only for PCM (i.e., `OMX_AUDIO_CodingADPCM`) and not for any other formats.

4.1.2.2 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 the MIDI OpenMAX component using the `nFileSize` field of the OMX_AUDIO_PARAM_MIDITYPE structure. For more information on the OMX_AUDIO_PARAM_MIDITYPE structure, see section 4.1.30.

4.1.3 General Enumerations

`OMX_AUDIO_CODINGTYPE` is the enumeration used to define the possible audio coding. 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`.

Field Name	Value	Description	References to Standard(s)
`OMX_AUDIO_CodingUnused`	0	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

Field Name	Value	Description	References to Standard(s)
OMX_AUDIO_CodingAMR		Any variant of AMR encoded data	AMR-NB , AMR-WB
OMX_AUDIO_CodingGSMFR		Any variant of GSM Full-Rate (i.e., GSM610)	GSM-FR
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_CodingTDMAEF		Any variant	TDMA-EFR

Field Name	Value	Description	References to Standard(s)
R		of TDMA Enhanced Full-Rate encoded data (TIA/EIA-136-410)	
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
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	MPEG-2 AAC , MPEG-4 AAC HE-AAC v1 ,

Field Name	Value	Description	References to Standard(s)
		data	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
OMX_AUDIO_CodingMax	0x7FFFFFFF		

Table 4-1. Audio Coding Types

4.1.4 OMX_AUDIO_PORTDEFINITIONTYPE

The OMX_AUDIO_PORTDEFINITION 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.

OMX_AUDIO_PORTDEFINITIONTYPE is defined as follows.

```
typedef struct OMX_AUDIO_PORTDEFINITIONTYPE {
    OMX_STRING cMIMEType;
    OMX_NATIVE_DEVICETYPE pNativeRender;
    OMX_BOOL bFlagErrorConcealment;
    OMX_AUDIO_CODINGTYPE eEncoding;
} OMX_AUDIO_PORTDEFINITIONTYPE;
```

The parameters for OMX_AUDIO_PORTDEFINITIONTYPE are defined as follows.

- `cMIMEType` is the MIME type of data for the port.
- `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 component.
- `eEncoding` is the type of data expected for this port (e.g., PCM, AMR, MP3, and so forth).

4.1.5 OMX_AUDIO_PARAM_PORTFORMATTYPE

OMX_AUDIO_PARAM_PORTFORMATTYPE is the structure for the port format parameter. This structure enumerates the various data input/output formats that the port supports.

This parameter call can be use 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` 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 prefer`ence formats are provided with lower values of nIndex.`

For `OMX_SetParameter`, the field is `nIndex` 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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `nIndex` indicates the enumeration index for the format from 0x0 to N-1.
- `eEncoding` is the type of data expected for this port (e.g., PCM, AMR, MP3, and so forth).

4.1.6 OMX_AUDIO_PARAM_PCMMODETYPE

The OMX_AUDIO_PARAM_PCMMODETYPE structure is used to set or query the current or default 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 msec.

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_CHANNELMAPPINGTYPE
eChannelMapping[OMX_AUDIO_MAXCHANNELS];
} OMX_AUDIO_PARAM_PCMMODETYPE;
```

4.1.6.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_PCMMODETYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.

- `nChannels` is the number of channels of audio (mono, stereo, multi-channel).
- `eNumData` indicates whether the PCM data is signed or unsigned.
- `eEndian` indicates whether PCM data is in little- or big-endian order.
- `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 4-2 identifies the PCM mode.

Field Name	Value	Description
`OMX_AUDIO_PCMModeLinear`	0	Linear PCM encoded data
`OMX_AUDIO_PCMModeALaw`		A law PCM encoded data (G.711)
`OMX_AUDIO_PCMModeMULaw`		μ law PCM encoded data (G.711)
`OMX_AUDIO_PCMModeMax`	0x7FFFFFFF	

Table 4-2. PCM Mode

- `eChannelMapping` is the audio channel mapping enumeration. A component will indicate the order of the audio channels as shown in Table 4-3. 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.

Field Name	Value	Description
`OMX_AUDIO_ChannelNone`	0	Unused or empty
`OMX_AUDIO_ChannelLF`	0x1	Left front
`OMX_AUDIO_ChannelRF`	0x2	Right front
`OMX_AUDIO_ChannelCF`	0x3	Center front
`OMX_AUDIO_ChannelLS`	0x4	Left surround
`OMX_AUDIO_ChannelRS`	0x5	Right surround
`OMX_AUDIO_ChannelLFE`	0x6	Low frequency effects
`OMX_AUDIO_ChannelCS`	0x7	Back surround
`OMX_AUDIO_ChannelLR`	0x8	Left rear
`OMX_AUDIO_ChannelRR`	0x9	Right rear
`OMX_AUDIO_ChannelMax`	0x7FFFFFFF	

Table 4-3. Audio Channel Mapping

4.1.6.2 Dependencies

The structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.6.3 Functionality

The OMX_AUDIO_PARAM_PCMMODETYPE structure sets the parameters of PCM audio.

4.1.6.4 Error Conditions

On processing the OMX_AUDIO_PARAM_PCMMODETYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.6.5 Post-processing Conditions

The characteristics of the PCM codec component at the port indicated by `nPortIndex` are fully specified.

4.1.7 OMX_AUDIO_PARAM_MP3TYPE

The OMX_AUDIO_PARAM_MP3TYPE structure is used to set or query the current or default 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_PARAM_MP3TYPE;
```

4.1.7.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_MP3TYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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 and MP3 use this value, although the names are more appropriate for MP3. Table 4-4 shows the values.

Mode	Value	Description
`OMX_AUDIO_ChannelModeStereo`	0	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.
OMX_AUDIO_ChannelModeMax	0x7FFFFFFF	

Table 4-4. Audio Channel Mode

4.1.7.2 Dependencies

The structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.7.3 Functionality

The OMX_AUDIO_PARAM_MP3TYPE structure sets the parameters of the MP3 codec.

4.1.7.4 Error Conditions

On processing the OMX_AUDIO_PARAM_MP3TYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.7.5 Post-processing Conditions

The characteristics of the MP3 codec component at the port indicated by `nPortIndex` are fully specified.

4.1.8 OMX_AUDIO_PARAM_AACPROFILETYPE

The OMX_AUDIO_PARAM_AACPROFILETYPE structure is used to set or query the current or default 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;
```

4.1.8.1 Parameter Definitions

The parameters for the OMX_AUDIO_PARAM_AACPROFILETYPE structure are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is a read-only value containing the index of the port.
- `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 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 4-5 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;"`

Define Name	Value	Description
OMX_AUDIO_AACToolNone	0x00000000	No AAC tools allowed (encoder configuration) or active (optional decoder information output).
OMX_AUDIO_AACToolMS	0x00000001	Mid/Side (MS) joint coding tool.
OMX_AUDIO_AACToolIS	0x00000002	Intensity Stereo (IS) tool.
OMX_AUDIO_AACToolTNS	0x00000004	Temporal Noise Shaping (TNS) tool.
OMX_AUDIO_AACToolPNS	0x00000008	MPEG-4 Perceptual Noise Substitution (PNS) tool.
OMX_AUDIO_AACToolLTP	0x00000010	MPEG-4 Long Term Prediction (LTP) tool.
OMX_AUDIO_AACToolAll	0x7FFFFFFF	All AAC tools allowed or active.

Table 4-5. AAC Tool Usage

- `nAACERtools` is the AAC Error Resilience tool usage. Table 4-6 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;"`

Define Name	Value	Description
OMX_AUDIO_AACERNone	0x00000000	No AAC ER tools allowed/used
OMX_AUDIO_AACERVCB11	0x00000001	Virtual Code Books for AAC section data (VCB11)
OMX_AUDIO_AACERRVLC	0x00000002	Reversible Variable Length Coding (RVLC)
OMX_AUDIO_AACERHCR	0x00000004	Huffman Codeword Reordering (HCR)
OMX_AUDIO_AACERAll	0x7FFFFFFF	All AAC ER tools allowed/used

Table 4-6. AAC Error Resilience Tool Usage

- `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 4-7 shows the values and descriptions.

Field Name	Value	Description
OMX_AUDIO_AACObjectNull	0	Null - not used
OMX_AUDIO_AACObjectMain	1	AAC Main object/profile
OMX_AUDIO_AACObjectLC	2	AAC Low Complexity object/profile (MPEG-4: AAC profile)
OMX_AUDIO_AACObjectSSR	3	AAC Scalable Sample Rate object/profile
OMX_AUDIO_AACObjectLTP	4	AAC Long Term Prediction object
OMX_AUDIO_AACObjectHE	5	High Efficiency AAC (object type SBR, MPEG-4: HE-AAC profile)
OMX_AUDIO_AACObjectScalable	6	AAC Scalable object
OMX_AUDIO_AACObjectERLC	17	ER AAC Low Complexity object (Error Resilient AAC-LC)
OMX_AUDIO_AACObjectLD	23	AAC Low Delay object (Error Resilient)
OMX_AUDIO_AACObjectHE_PS	29	AAC High Efficiency with Parametric Stereo coding (HE-AAC v2, object type PS)
OMX_AUDIO_AACObjectMax	0x7FFFFFFF	

Table 4-7. AAC Profile Type

- `eAACStreamFormat` is the enumeration of OMX_AUDIO_AACSTREAMFORMATTYPE for the AAC stream format. Table 4-8 shows the field names and values.

Field Name	Value	Description
OMX_AUDIO_AACStreamFormatMP2ADTS	0	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

Field Name	Value	Description
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)
OMX_AUDIO_AACStreamFormatMax	0x7FFFFFFF	

Table 4-8. AAC Stream Format Type

- `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 4.1.7.

4.1.8.2 Dependencies

The OMX_AUDIO_PARAM_AACPROFILETYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.8.3 Functionality

The OMX_AUDIO_PARAM_AACPROFILETYPE structure sets the parameters of the AAC codec.

4.1.8.4 Error Conditions

On processing the OMX_AUDIO_PARAM_AACPROFILETYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.

- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.8.5 Post-processing Conditions

The characteristics of the AAC codec component at the port indicated by `nPortIndex` are fully specified.

4.1.9 OMX_AUDIO_PARAM_VORBISTYPE

The OMX_AUDIO_PARAM_VORBISTYPE structure is used to set or query the current or default 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_CHANNELMODETYPE eChannelMode;
} OMX_AUDIO_PARAM_VORBISTYPE;
```

4.1.9.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_VORBISTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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.

4.1.9.2 Dependencies

The OMX_AUDIO_PARAM_VORBISTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.9.3 Functionality

The OMX_AUDIO_PARAM_VORBISTYPE structure sets the parameters of the Vorbis codec.

4.1.9.4 Error Conditions

On processing the OMX_AUDIO_PARAM_VORBISTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.9.5 Post-processing Conditions

The characteristics of the Vorbis codec component at the port indicated by `nPortIndex` are fully specified.

4.1.10 OMX_AUDIO_PARAM_WMATYPE

The OMX_AUDIO_PARAM_WMATYPE structure is used to set or query the current or default 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_U32 nChannels;
    OMX_U32 nBitRate;
    OMX_AUDIO_WMAFORMATTYPE eFormat;
} OMX_AUDIO_PARAM_WMATYPE;
```

4.1.10.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_WMATYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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 4-9 shows the field names and values.

Field Name	Value	Description
`OMX_AUDIO_WMAFormatUnused`	0	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_WMAFormatMax	0x7FFFFFFF	For future versions of Windows Media audio codecs.

Table 4-9. Windows Media Audio Codec Version

4.1.10.2 Dependencies

The OMX_AUDIO_PARAM_WMATYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using OMX_SetParameter only when the component is in the OMX_StateLoaded state.

4.1.10.3 Error Conditions

On processing the OMX_AUDIO_PARAM_WMATYPE structure, the following error conditions can occur:

- OMX_ErrorBadParameter if one or more fields of the structure are incorrect.
- OMX_ErrorUnSupportedIndex when the feature is unsupported.
- OMX_ErrorInvalidState when the OMX_SetParameter function is called and the component is in the OMX_StateInvalid state.
- 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_ErrorVersionMismatch when the nVersion field of the structure does not match the expected version for the component.
- OMX_ErrorNoMore when the OMX_GetParameter function is called and the value of nPortIndex exceeds the number of audio ports for the component.
- OMX_ErrorUnsupportedSetting when a field in the structure is unsupported by the component during an OMX_SetParameter call.

4.1.10.4 Post-processing Conditions

The characteristics of the WMA codec component at the port indicated by nPortIndex are fully specified.

4.1.11 OMX_AUDIO_RATYPE

The OMX_AUDIO_RATYPE structure is used to set or query the current or default 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_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_PARAM_RATYPE;
```

4.1.11.1 Parameter Definitions

The parameters for OMX_AUDIO_RATYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex`: is the read-only value containing the index of the port.
- `nChannels` is the number of audio channels.
- `nSamplingRate` is the sampling rate of the source data. The values allowed for the sampling rate are 8000 Hz, 11025 Hz, 22050Hz, and 44100Hz.
- `nBitsPerFrame` is the value for bits per frame. The range is 46-12288 bits per frame.
- `nSamplePerFrame` is the value for samples per frame. The values allowed for the samples per frame are 256, 512, or 1024.
- `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.

4.1.11.2 Dependencies

The OMX_AUDIO_RATYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.11.3 Functionality

The OMX_AUDIO_RATYPE structure sets the parameters of the RealAudio codec.

4.1.11.4 Error Conditions

On processing the OMX_AUDIO_RATYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.11.5 Post-processing Conditions

The characteristics of the RealAudio codec component at the port indicated by `nPortIndex` are fully specified.

4.1.12 OMX_AUDIO_PARAM_SBCTYPE

The Subband codec (SBC) is a mandatory audio codec for applications that supports 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 or default 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;
```

4.1.12.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_SBCTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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`.
- `bChannelMode` is the audio channel mode.
- `eSBCAllocType` is the enumeration of the adaptive bit allocation algorithm. Table 4-10 shows the field names and values.

Field Name	Value	Description
`OMX_AUDIO_SBCAllocMethodLoudness`	0	Loudness allocation method
`OMX_AUDIO_SBCAllocMethodSNR`	1	Signal-to-noise ratio (SNR) allocation method
`OMX_AUDIO_SBCAllocMethodMax`	0x7FFFFFFF	

Table 4-10. Adaptive Bit Allocation Algorithm Values

4.1.12.2 Dependencies

The OMX_AUDIO_PARAM_SBCTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.12.3 Functionality

This OMX_AUDIO_PARAM_SBCTYPE structure configures the parameters of the SBC codec.

4.1.12.4 Error Conditions

On processing the OMX_AUDIO_PARAM_SBCTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.12.5 Post-processing Conditions

The characteristics of the SBC codec component at the port indicated by `nPortIndex` are fully specified.

4.1.13 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 or default 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_AUDIO_PARAM_ADPCMTYPE;
```

4.1.13.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_ADPCMTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `nChannels` is the number of channels of audio (mono, stereo).
- `nBitrate` is the bit rate of the encoded ADPCM audio. If the bit rate is variable or unknown, this parameter has the value 0.
- `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.

4.1.13.2 Dependencies

The OMX_AUDIO_PARAM_ADPCMTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.13.3 Functionality

The OMX_AUDIO_PARAM_ADPCMTYPE structure sets the parameters of a generic ADPCM codec.

4.1.13.4 Error Conditions

On processing the OMX_AUDIO_PARAM_ADPCMTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.13.5 Post-processing Conditions

The characteristics of the ADPCM codec component at the port indicated by `nPortIndex` are fully specified.

4.1.14 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 or default 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_AUDIO_PARAM_G723TYPE;
```

4.1.14.1 Parameter Definitions

The parameters of OMX_AUDIO_PARAM_G723TYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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 4-11 identifies bit rate values.

Field Name	Value	Description
`OMX_AUDIO_G723ModeUnused`	0	Rate unused or unknown
`OMX_AUDIO_G723ModeLow`		5.3 kbps
`OMX_AUDIO_G723ModeHigh`		6.3 kbps

Table 4-11. G.723 Bit Rate Values

- `bHiPassFilter` enables high-pass filter preprocessing in the encoder.

4.1.14.2 Dependencies

The OMX_AUDIO_PARAM_G723TYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.14.3 Functionality

The OMX_AUDIO_PARAM_G723TYPE structure sets the parameters of the ITU-G.723.1 codec.

4.1.14.4 Error Conditions

On processing the OMX_AUDIO_PARAM_G723TYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

- OMX_ErrorNoMore when the OMX_GetParameter function is called and the value of nPortIndex exceeds the number of audio ports for the component.
- OMX_ErrorUnsupportedSetting when a field in the structure is unsupported by the component during an OMX_SetParameter call.

4.1.14.5 Post-processing Conditions

The characteristics of the codec component at the port indicated by nPortIndex are fully specified.

4.1.15 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 or default 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;
```

4.1.15.1 Parameter Definitions

The parameters of OMX_AUDIO_PARAM_G726TYPE are defined as follows.

- 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.
- nVersion is the version of the structure.
- nPortIndex is the read-only value containing the index of the port.
- nChannels is the number of channels of audio (mono, stereo).
- eG726Mode is the bit rate of the encoded speech. Table 4-12 identifies the bit rate values.

•

Field Name	Value	Description
OMX_AUDIO_G726ModeUnused	0	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

Table 4-12. G.726 Bit Rate Values

4.1.15.2 Dependencies

The OMX_AUDIO_PARAM_G726TYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.15.3 Functionality

The OMX_AUDIO_PARAM_G726TYPE structure sets the parameters of the ITU-G.726 codec.

4.1.15.4 Error Conditions

On processing the OMX_AUDIO_PARAM_G726TYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.15.5 Post-processing Conditions

The characteristics of the codec component at the port indicated by `nPortIndex` are fully specified.

4.1.16 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 or default 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 eAnnex;
    OMX_BOOL bHiPassFilter;
} OMX_AUDIO_PARAM_G729TYPE;
```

4.1.16.1 Parameter Definitions

The parameters of OMX_AUDIO_PARAM_G729TYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `nChannels` is the number of channels of audio (mono, stereo).
- `bDTX` enables Discontinuous Transmission when Annex B of the standard is used.
- `eAnnex` identifies the standard annexes used. Table 4-13 identifies the standard annexes.

Field Name	Value	Description
`OMX_AUDIO_G729`	0	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

Table 4-13. Standard Annexes

- `bHiPassFilter` enables High-Pass filter preprocessing in the encoder.

4.1.16.2 Dependencies

The OMX_AUDIO_PARAM_G729TYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.16.3 Functionality

The OMX_AUDIO_PARAM_G729TYPE structure sets the parameters of the ITU-G.729 codec.

4.1.16.4 Error Conditions

On processing the OMX_AUDIO_PARAM_G729TYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.16.5 Post-processing Conditions

The characteristics of the codec component at the port indicated by `nPortIndex` are fully specified.

4.1.17 OMX_AUDIO_PARAM_AMRTYPE

The Adaptive Multi-Rate coder is defined in 3GPP standards as having two 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.

The OMX_AUDIO_PARAM_AMRTYPE structure is used to set or query the current or default 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_AUDIO_AMRBANDMODETYPE eAMRBandMode;
    OMX_AUDIO_AMRDTXMODETYPE  eAMRDTXMode;
    OMX_AUDIO_AMRFRAMEFORMATTYPE eAMRFrameFormat;
} OMX_AUDIO_PARAM_AMRTYPE;
```

4.1.17.1 Parameter Definitions

The parameters of OMX_AUDIO_PARAM_AMRTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `nChannels` is the number of channels of audio (mono, stereo).
- `eAMRBandMode` is the bit rate of the encoded speech. Table 4-14 shows the bit rate values.

Field Name	Value	Description
`OMX_AUDIO_AMRBandModeUnused`	0	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

Field Name	Value	Description
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_AMRBandModeMax	0x7FFFFFFF	5.15 kbps

Table 4-14. Adaptive Multi-Rate Bit Rate Values

- eAMRDTXMode identifies the AMR Discontinuous Transmission mode and voice activity detection (VAD) type. Table 4-15 describes the modes and types.

Field Name	Value	Description
OMX_AUDIO_AMRDTXModeUsed		DTX used or unused
OMX_AUDIO_AMRDTXModeOnVAD1		Use Type 1 VAD
OMX_AUDIO_AMRDTXModeOnVAD2		Use Type 2 VAD
OMX_AUDIO_AMRDTXModeOnAuto		VAD type automatic
OMX_AUDIO_AMRDTXModeAsEFR		DTX frames as EFR (3GPP 26.101, frame type equals 8,9,10)
OMX_AUDIO_AMRDTXModeMax	0x7FFFFFFF	

Table 4-15. Adaptive Multi-Rate Discontinuous Transmission Mode and VAD Type

- eAMRFrameFormat identifies the encoded frame format. Table 4-16 shows the frame formats.

Field Name	Value	Description
OMX_AUDIO_AMRFrameFormatConformance	0	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)

Field Name	Value	Description
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 3267, sec. 5)
OMX_AUDIO_AMRFrameFormatRTPPayload		RTP payload format (RFC 3267, sec. 4)
OMX_AUDIO_AMRFrameFormatITU		ITU frame format
OMX_AUDIO_AMRDTXModeMax	0x7FFFFFFF	

Table 4-16. Encoded Frame Format

4.1.17.2 Dependencies

The OMX_AUDIO_PARAM_AMRTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.17.3 Functionality

The OMX_AUDIO_PARAM_AMRTYPE structure sets the parameters of the AMR codec.

4.1.17.4 Error Conditions

On processing the OMX_AUDIO_PARAM_AMRTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.17.5 Post-processing Conditions

The characteristics of the codec component at the port indicated by `nPortIndex` are fully specified.

4.1.18 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 msec. The coding algorithm is RPE-LTP.

The OMX_AUDIO_PARAM_GSMFRTYPE structure is used to set or query the current or default 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_AUDIO_PARAM_GSMFRTYPE;
```

4.1.18.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_GSMFRTYPE as defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `bDTX` enables Discontinuous Transmission (3GPP 46.031, 46.032).

4.1.18.2 Dependencies

The OMX_AUDIO_PARAM_GSMFRTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.18.3 Functionality

The OMX_AUDIO_PARAM_GSMFRTYPE structure sets the parameters of the GSM Full-Rate codec.

4.1.18.4 Error Conditions

On processing the OMX_AUDIO_PARAM_GSMFRTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.18.5 Post-processing Conditions

The characteristics of the codec component at the port indicated by `nPortIndex` are fully specified.

4.1.19 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 msec. 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 or default 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;
```

4.1.19.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_GSMEFRTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `bDTX` enables Discontinuous Transmission (3GPP 46.041, 46.042).
- `bHiPassFilter` enables High-Pass filter preprocessing in the encoder.

4.1.19.2 Dependencies

The OMX_AUDIO_PARAM_GSMEFRTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.19.3 Functionality

The OMX_AUDIO_PARAM_GSMEFRTYPE structure sets the parameters of the GSM Enhanced Full-Rate codec.

4.1.19.4 Error Conditions

On processing the OMX_AUDIO_PARAM_GSMEFRTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

- OMX_ErrorNoMore when the OMX_GetParameter function is called and the value of nPortIndex exceeds the number of audio ports for the component.
- OMX_ErrorUnsupportedSetting when a field in the structure is unsupported by the component during an OMX_SetParameter call.

4.1.19.5 Post-processing Conditions

The characteristics of the codec component at the port indicated by nPortIndex are fully specified.

4.1.20 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 msec. The coding algorithm is VSELP.

The OMX_AUDIO_PARAM_GSMHRTYPE structure is used to set or query the current or default 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;
```

4.1.20.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_GSMHRTYPE are defined as follows.

- 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.
- nVersion is the version of the structure.
- nPortIndex is the read-only value containing the index of the port.
- bDTX enables Discontinuous Transmission (3GPP 46.041, 46.042).
- bHiPassFilter enables High-Pass filter preprocessing in the encoder.

4.1.20.2 Dependencies

The OMX_AUDIO_PARAM_GSMHRTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.20.3 Functionality

The OMX_AUDIO_PARAM_GSMHRTYPE structure sets the parameters of the GSM Half-Rate codec.

4.1.20.4 Error Conditions

On processing the OMX_AUDIO_PARAM_GSMHRTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.20.5 Post-processing Conditions

The characteristics of the codec component at the port indicated by `nPortIndex` are fully specified.

4.1.21 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 msec. The coding algorithm is VSELP.

The OMX_AUDIO_PARAM_TDMAFRTYPE structure is used to set or query the current or default 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_BOOL bDTX;
    OMX_BOOL bHiPassFilter;
} OMX_AUDIO_PARAM_TDMAFRTYPE;
```

4.1.21.1 Parameter Definitions

The parameters of OMX_AUDIO_PARAM_TDMAFRTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `bDTX` enables Discontinuous Transmission.
- `bHiPassFilter` enables High-Pass filter preprocessing in the encoder.

4.1.21.2 Dependencies

The OMX_AUDIO_PARAM_TDMAFRTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.21.3 Functionality

The OMX_AUDIO_PARAM_TDMAFRTYPE structure sets the parameters of the TDMA Full-Rate codec.

4.1.21.4 Error Conditions

On processing the OMX_AUDIO_PARAM_TDMAFRTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.21.5 Post-processing Conditions

The characteristics of the codec component at the port indicated by `nPortIndex` are fully specified.

4.1.22 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 msec. The coding algorithm is ACELP.

The OMX_AUDIO_PARAM_TDMAEFRTYPE structure is used to set or query the current or default 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_BOOL bDTX;
    OMX_BOOL bHiPassFilter;
} OMX_AUDIO_PARAM_TDMAEFRTYPE;
```

4.1.22.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_TDMAEFRTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `bDTX` enables Discontinuous Transmission.
- `bHiPassFilter` enables High-Pass filter preprocessing in the encoder.

4.1.22.2 Dependencies

The OMX_AUDIO_PARAM_TDMAEFRTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.22.3 Functionality

The OMX_AUDIO_PARAM_TDMAEFRTYPE structure sets the parameters of the TDMA Enhanced Full-Rate codec.

4.1.22.4 Error Conditions

On processing the OMX_AUDIO_PARAM_TDMAEFRTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.22.5 Post-processing Conditions

The characteristics of the codec component at the port indicated by `nPortIndex` are fully specified.

4.1.23 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 msec. The coding algorithm is VSELP.

The OMX_AUDIO_PARAM_PDCFRTYPE structure is used to set or query the current or default 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_BOOL bDTX;
    OMX_BOOL bHiPassFilter;
} OMX_AUDIO_PARAM_PDCFRTYPE;
```

4.1.23.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_PDCFRTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `bDTX` enables Discontinuous Transmission.
- `bHiPassFilter` enables High-Pass filter preprocessing in the encoder.

4.1.23.2 Dependencies

The OMX_AUDIO_PARAM_PDCFRTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.23.3 Functionality

The OMX_AUDIO_PARAM_PDCFRTYPE structure sets the parameters of the PDC Full-Rate codec.

4.1.23.4 Error Conditions

On processing the OMX_AUDIO_PARAM_PDCFRTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.23.5 Post-processing Conditions

The characteristics of the codec component at the port indicated by `nPortIndex` are fully specified.

4.1.24 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 msec. The coding algorithm is ACELP.

The OMX_AUDIO_PARAM_PDCEFRTYPE structure is used to set or query the current or default 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_BOOL bDTX;
    OMX_BOOL bHiPassFilter;
} OMX_AUDIO_PARAM_PDCEFRTYPE;
```

4.1.24.1 Parameter Definitions

The parameters of OMX_AUDIO_PARAM_PDCEFRTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `bDTX` enables Discontinuous Transmission.
- `bHiPassFilter` enables High-Pass filter preprocessing in the encoder.

4.1.24.2 Dependencies

The OMX_AUDIO_PARAM_PDCEFRTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.24.3 Functionality

The OMX_AUDIO_PARAM_PDCEFRTYPE structure sets the parameters of the PDC Enhanced Full-Rate codec.

4.1.24.4 Error Conditions

On processing the OMX_AUDIO_PARAM_PDCEFRTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.24.5 Post-processing Conditions

The characteristics of the codec component at the port indicated by `nPortIndex` are fully specified.

4.1.25 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 msec. The coding algorithm is PSI-CELP.

The OMX_AUDIO_PARAM_PDCHRTYPE structure is used to set or query the current or default 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_PDCEFRTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_BOOL bDTX;
    OMX_BOOL bHiPassFilter;
} OMX_AUDIO_PARAM_PDCEFRTYPE;
```

4.1.25.1 Parameter Definitions

The parameters of OMX_AUDIO_PARAM_PDCHRTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `bDTX` enables Discontinuous Transmission.
- `bHiPassFilter` enables High-Pass filter preprocessing in the encoder.

4.1.25.2 Dependencies

The OMX_AUDIO_PARAM_PDCHRTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.25.3 Functionality

The OMX_AUDIO_PARAM_PDCHRTYPE structure sets the parameters of the PDC Full-Rate codec.

4.1.25.4 Error Conditions

On processing the OMX_AUDIO_PARAM_PDCHRTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.25.5 Post-processing Conditions

The characteristics of the codec component at the port indicated by `nPortIndex` are fully specified.

4.1.26 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 msec. 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 or default 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_PARAMQCELP8TYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_AUDIO_CDMARATETYPE eCDMARate;
    OMX_U32 nMinBitRate;
    OMX_U32 nMaxBitRate;
} OMX_AUDIO_PARAMQCELP8TYPE;
```

4.1.26.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_QCELP8TYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `eCDMARate` is the frame rate or type. Table 4-17 shows the frame rate values.

Field Name	Value	Description
OMX_AUDIO_CDMARateBlank	0	Blank frame
OMX_AUDIO_CDMARateFull	4	Rate 1
OMX_AUDIO_CDMARateHalf	3	Rate 1/2
OMX_AUDIO_CDMARateQuarter	2	Rate 1/4
OMX_AUDIO_CDMARateEighth	1	Rate 1/8
OMX_AUDIO_CDMARateErasure	14	Erasure frame (due to channel errors)
OMX_AUDIO_CDMARateMax	0x7FFFFFFF	

Table 4-17. QCELP8 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. This value shall be greater than or equal to the minimal rate. The default value is 4.

4.1.26.2 Dependencies

The OMX_AUDIO_PARAM_QCELP8TYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.26.3 Functionality

The OMX_AUDIO_PARAM_QCELP8TYPE structure sets the parameters of the QCELP8 codec.

4.1.26.4 Error Conditions

On processing the OMX_AUDIO_PARAM_QCELP8TYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.26.5 Post-processing Conditions

The characteristics of the codec component at the port indicated by `nPortIndex` are fully specified.

4.1.27 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 msec. 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 or default 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_PARAMQCELP13TYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_AUDIO_CDMARATETYPE eCDMARate;
    OMX_U32 nMinBitRate;
    OMX_U32 nMaxBitRate;
} OMX_AUDIO_PARAMQCELP13TYPE;
```

4.1.27.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_QCELP13TYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `eCDMARate` is the frame rate or type. Table 4-18 shows the frame rate values.

Field Name	Value	Description
OMX_AUDIO_CDMARateBlank	0	Blank frame
OMX_AUDIO_CDMARateFull	4	Rate 1
OMX_AUDIO_CDMARateHalf	3	Rate 1/2
OMX_AUDIO_CDMARateQuarter	2	Rate 1/4
OMX_AUDIO_CDMARateEighth	1	Rate 1/8
OMX_AUDIO_CDMARateErasure	14	Erasure frame (due to channel errors)
OMX_AUDIO_CDMARateMax	0x7FFFFFFF	

Table 4-18. QCELP13 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.

4.1.27.2 Dependencies

The OMX_AUDIO_PARAM_QCELP13TYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.27.3 Functionality

The OMX_AUDIO_PARAM_QCELP13TYPE structure sets the parameters of the QCELP13 codec.

4.1.27.4 Error Conditions

On processing the OMX_AUDIO_PARAM_QCELP13TYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.27.5 Post-processing Conditions

The characteristics of the codec component at the port indicated by `nPortIndex` are fully specified.

4.1.28 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 msec. 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 or default 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_PARAMEVRCTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_AUDIO_CDMARATETYPE eCDMARate;
    OMX_U32 nMinBitRate;
    OMX_U32 nMaxBitRate;
    OMX_U32 bNoiseSuppressor;
} OMX_AUDIO_PARAMEVRCTYPE;
```

4.1.28.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_EVRCTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `eCDMARate` is the frame rate or type. Table 4-19 shows the frame rate values.

Field Name	Value	Description
OMX_AUDIO_CDMARateBlank	0	Blank frame
OMX_AUDIO_CDMARateFull	4	Rate 1
OMX_AUDIO_CDMARateHalf	3	Rate 1/2
OMX_AUDIO_CDMARateEighth	1	Rate 1/8
OMX_AUDIO_CDMARateErasure	14	Erasure frame (due to channel errors)
OMX_AUDIO_CDMARateMax	0x7FFFFFFF	

Table 4-19. Enhanced Variable Speech Frame Rate Values

- `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.
- `bNoiseSuppressor` enables the encoder's noise suppressor preprocessing as a part of the encoder.

4.1.28.2 Dependencies

The OMX_AUDIO_PARAM_EVRCTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure only be set using `OMX_SetParameter` may when the component is in the OMX_StateLoaded state.

4.1.28.3 Functionality

The OMX_AUDIO_PARAM_EVRCTYPE structure sets the parameters of the Enhanced Variable Speech Coder (EVRC) speech codec.

4.1.28.4 Error Conditions

On processing the OMX_AUDIO_PARAM_EVRCTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.28.5 Post-processing Conditions

The characteristics of the codec component at the port indicated by `nPortIndex` are fully specified.

4.1.29 OMX_AUDIO_PARAMSMVTYPE

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 msec. 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_PARAMSMVTYPE structure is used to set or query the current or default 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_PARAMSMVTYPE is defined as follows.

```
typedef struct OMX_AUDIO_PARAMSMVTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_AUDIO_CDMARATETYPE eCDMARate;
    OMX_U32 nMinBitRate;
    OMX_U32 nMaxBitRate;
    OMX_U32 bNoiseSuppressor;
} OMX_AUDIO_PARAMSMVTYPE;
```

4.1.29.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAMSMVTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `eCDMARate` is the frame rate or type. Table 4-20 identifies the frame rate values.

Field Name	Value	Description
`OMX_AUDIO_CDMARateBlank`	0	Blank frame
`OMX_AUDIO_CDMARateFull`	4	Rate 1
`OMX_AUDIO_CDMARateHalf`	3	Rate 1/2
`OMX_AUDIO_CDMARateEighth`	1	Rate 1/8
`OMX_AUDIO_CDMARateErasure`	14	Erasure frame (due to channel errors)
`OMX_AUDIO_CDMARateMax`	0x7FFFFFFF	

Table 4-20. Selectable Mode Vocoder Frame Rate Values

- `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.
- `bNoiseSuppressor` enables the encoder's noise suppressor preprocessing as a part of the encoder.

4.1.29.2 Dependencies

The OMX_AUDIO_PARAMSMVTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.29.3 Functionality

The OMX_AUDIO_PARAMSMVTYPE structure sets the parameters of the Selectable Mode Vocoder codec.

4.1.29.4 Error Conditions

On processing the OMX_AUDIO_PARAMSMVTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.29.5 Post-processing Conditions

The characteristics of the codec component at the port indicated by `nPortIndex` are fully specified.

4.1.30 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;
```

4.1.30.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_MIDITYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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; see OMX_Types.h), 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 4-21 shows the MIDI file format.

Field Name	Value	Description
`OMX_AUDIO_MIDIFormatUnknown`	0	MIDI format is unknown or not used.
`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)
OMX_AUDIO_MIDIFormatMax	0x7FFFFFFF	Allowance for expansion in the number of MIDI file formats

Table 4-21. MIDI File Format

4.1.30.2 Dependencies

The OMX_AUDIO_PARAM_MIDITYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using `OMX_SetParameter` only when the component is in the OMX_StateLoaded state.

4.1.30.3 Error Conditions

On processing the OMX_AUDIO_PARAM_MIDITYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is not supported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.30.4 Post-processing Conditions

The characteristics of the MIDI IL component at the port indicated by `nPortIndex` are fully specified.

4.1.31 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: It includes a pointer to data, namely the DLS, which is outside the structure. This is because DLS soundbanks can get 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;
```

4.1.31.1 Parameter Definitions

The parameters for OMX_AUDIO_PARAM_MIDILOADUSERSOUNDTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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 4-22 identifies the MIDI soundbanks.

Field Name	Value	Description
`OMX_AUDIO_MIDISoundBankUnused`	0	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_MIDISoundBankMobileDLSPlusOptions		Mobile DLS, using the specification-defined optional feature set
OMX_AUDIO_MIDISoundBankMax	0x7FFFFFFF	Allowance for expansion in the number of soundbank types

Table 4-22. MIDI Soundbanks

- 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 4-23 shows the MIDI soundbank layouts.

Field Name	Value	Description
OMX_AUDIO_MIDISoundBankLayoutUnused	0	Unknown/unused soundbank layout type.
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.
OMX_AUDIO_MIDISoundBankLayoutMax	0x7FFFFFFF	Allowance for expansion in the number of soundbank layout types.

Table 4-23. MIDI Soundbank Layouts

4.1.31.2 Dependencies

The OMX_AUDIO_PARAM_MIDILOADUSERSOUNDTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set using OMX_SetParameter only when the component is in the OMX_StateLoaded state.

4.1.31.3 Error Conditions

On processing the OMX_AUDIO_PARAM_MIDILOADUSERSOUNDTYPE structure, the following error conditions can occur:

- OMX_ErrorBadParameter if one or more fields of the structure are incorrect.
- OMX_ErrorUnSupportedIndex when the feature is unsupported.
- OMX_ErrorInvalidState when the OMX_SetParameter function is called and the component is in the OMX_StateInvalid state.

- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.

4.1.31.4 Post-processing Conditions

The characteristics of the MIDI IL component at the port indicated by `nPortIndex` are fully specified.

4.1.32 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;
```

4.1.32.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_MIDIIMMEDIATEEVENTTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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.`

4.1.32.2 Dependencies

The OMX_AUDIO_CONFIG_MIDIIMMEDIATEEVENTTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set at any time using `OMX_SetConfig` as long as the component is not in the OMX_StateInvalid state.

4.1.32.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_MIDIIMMEDIATEEVENTTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is not supported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetConfig` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

4.1.32.4 Post-processing Conditions

The live MIDI event array is rendered by the MIDI engine, or the MIP message contained in the buffer is processed.

4.1.33 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;
```

4.1.33.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_MIDISOUNDBANKPROGRAMTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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.

4.1.33.2 Dependencies

The OMX_AUDIO_CONFIG_MIDISOUNDBANKPROGRAMTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set at any time using `OMX_SetConfig` as long as the component is not in the OMX_StateInvalid state.

4.1.33.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_MIDISOUNDBANKPROGRAMTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.

- `OMX_ErrorInvalidState` when the `OMX_SetConfig` function is called and the component is in the OMX_StateInvalid state.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetConfig` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

4.1.33.4 Post-processing Conditions

The specified MIDI channel has a soundbank and program associated with it.

4.1.34 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_U64 nStopTime;
    OMX_U16 nChannelMuteMask;
    OMX_U16 nChannelSoloMask;
    OMX_U32 nTrack0031MuteMask;
    OMX_U32 nTrack3263MuteMask;
    OMX_U32 nTrack0031SoloMask;
    OMX_U32 nTrack3263SoloMask;
} OMX_AUDIO_CONFIG_MIDICONTROLTYPE;
```

4.1.34.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_MIDICONTROLTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `sPitchTransposition` is the pitch transposition in semitones, stored as Q22.10 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 a Q14.17 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 a Q22.10 fixed-point number based on the 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.
- 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.

4.1.34.2 Dependencies

The OMX_AUDIO_CONFIG_MIDICONTROLTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure may be set at any time using OMX_SetConfig as long as the component is not in the OMX_StateInvalid state.

4.1.34.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_MIDICONTROLTYPE structure, the following error conditions can occur:

- OMX_ErrorBadParameter if one or more fields of the structure are incorrect.
- OMX_ErrorUnSupportedIndex when the feature is unsupported.
- OMX_ErrorInvalidState when the OMX_SetConfig function is called and the component is in the OMX_StateInvalid state.
- OMX_ErrorVersionMismatch when the nVersion field of the structure does not match the expected version for the component.

- `OMX_ErrorNoMore` when the `OMX_GetConfig` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

4.1.34.4 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.

4.1.35 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;
```

4.1.35.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_MIDISTATUSTYPE are defined as follows.

- `nSize` is the size of the structure in bytes. This value shall be specified when this structure is used as an output from a function.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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 4-24 describes the payback states.

Field Name	Value	Description
`OMX_AUDIO_MIDIPlayBackStateUnknown`	0	Unknown/unused MIDI playback state, or state does not map to one of the defined states.
`OMX_AUDIO_MIDIPlayBackStateClosedEngaged`		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_MIDIPlayBackStateOpenEngaged		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_MIDIPlayBackStatePlayingPartially		Best-effort playback due to SP-MIDI/DLS resource constraints
OMX_AUDIO_MIDIPlayBackStatePlayingSilently		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.
OMX_AUDIO_MIDIPlayBackStateMax	0x7FFFFFFF	Allowance for expansion in the number of playback states.

Table 4-24. MIDI Playback States

4.1.35.2 Dependencies

The OMX_AUDIO_CONFIG_MIDISTATUSTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure contains read-only parameters and therefore cannot be set.

4.1.35.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_MIDISTATUSTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is not supported.
- `OMX_ErrorInvalidState` when the `OMX_GetConfig` function is called and the component is in the OMX_StateInvalid state.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetConfig` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is not supported by the component during an `OMX_SetConfig` call.

4.1.35.4 Post-processing Conditions

The IL client has the status of the MIDI engine.

4.1.36 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 4.1.37, 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;
```

4.1.36.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_MIDIMETAEVENTTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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.

4.1.36.2 Dependencies

The OMX_AUDIO_CONFIG_MIDIMETAEVENTTYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure cannot be set.

4.1.36.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_MIDIMETAEVENTTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetConfig` function is called and the component is in the OMX_StateInvalid state.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

- `OMX_ErrorNoMore` when the `OMX_GetConfig` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

4.1.36.4 Post-processing Conditions

The IL client knows the type, track number, and size of the meta events in the MIDI file.

4.1.37 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 4.1.36. 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;
```

4.1.37.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_MIDIMETAEVENTDATATYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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

4.1.37.2 Dependencies

The OMX_AUDIO_CONFIG_MIDIMETAEVENTDATATYPE structure may be queried at any time that the component is not in the OMX_StateInvalid state. The structure cannot be set.

4.1.37.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_MIDIMETAEVENTDATATYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetConfig` function is called and the component is in the OMX_StateInvalid state.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetConfig` function is called and the value of `nPortIndex` exceeds the number of audio ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

4.1.37.4 Post-processing Conditions

The IL client has access to the required meta event data in the MIDI file.

4.1.38 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;
```

4.1.38.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_VOLUMETYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.

- `nPortIndex`: is the read-only value containing the index of the port.
- `bLinear` is the volume to be set on a linear (0-100) or a logarithmic scale (millidecibel, 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.

4.1.38.2 Dependencies

The OMX_AUDIO_CONFIG_VOLUMETYPE structure may be queried using `OMX_GetConfig` and set using `OMX_SetConfig` at any time after the component has been loaded.

4.1.38.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_VOLUMETYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetConfig` function is called and the component is not in a valid state.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetConfig` function is called and the value of `nPortIndex` exceeds the number of ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

4.1.38.4 Post-processing Conditions

In case of an `OMX_SetConfig` call using this structure, the volume of the port is set. In case of an `OMX_GetConfig` call using this structure, the IL client can determine the current volume setting for the port.

4.1.39 OMX_AUDIO_CONFIG_CHANNELVOLUMETYPE

The OMX_AUDIO_CONFIG_CHANNELVOLUMETYPE structure is used to adjust the audio volume for a channel.

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;
```

4.1.39.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_CHANNELVOLUMETYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `nChannel` is the channel to select in the range 0 to N-1 using OMX_ALL to apply volume settings 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 millidecibel (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.

4.1.39.2 Dependencies

The OMX_AUDIO_CONFIG_CHANNELVOLUMETYPE structure may be queried using `OMX_GetConfig` and set using `OMX_SetConfig` at any time after the component has been loaded.

4.1.39.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_CHANNELVOLUMETYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetConfig` function is called and the component is not in a valid state.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetConfig` function is called and the value of `nPortIndex` exceeds the number of ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

4.1.39.4 Post-processing Conditions

In case of an `OMX_SetConfig` call using this structure, the volume of the audio channel is set. In case of an `OMX_GetConfig` call using this structure, the IL client can determine the current volume setting for the channel.

4.1.40 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;
```

4.1.40.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_BALANCETYPE are as follows.

- `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.

- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port. 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.

4.1.40.2 Dependencies

The OMX_AUDIO_CONFIG_BALANCETYPE structure may be queried using `OMX_GetConfig` and set using `OMX_SetConfig` at any time after the component has been loaded.

4.1.40.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_BALANCETYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetConfig` function is called and the component is not in a valid state.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetConfig` function is called and the value of `nPortIndex` exceeds the number of ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

4.1.40.4 Post-processing Conditions

In case of an `OMX_SetConfig` call using this structure, the balance setting is set to the specified value. In case of an `OMX_GetConfig` call using this structure, the IL client can determine the current balance setting parameter.

4.1.41 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;
```

4.1.41.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_MUTETYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port. 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).

4.1.41.2 Dependencies

The OMX_AUDIO_CONFIG_MUTETYPE structure may be queried using `OMX_GetConfig` and set using `OMX_SetConfig` at any time after the component has been loaded.

4.1.41.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_MUTETYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetConfig` function is called and the component is not in a valid state.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetConfig` function is called and the value of `nPortIndex` exceeds the number of ports for the component.

- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

4.1.41.4 Post-processing Conditions

In case of an `OMX_SetConfig` call using this structure, the specified audio port is muted or unmuted according to the structure data. In case of an `OMX_GetConfig` call using this structure, the IL client can determine if the port is muted or not.

4.1.42 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;
```

4.1.42.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_CHANNELMUTETYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port. 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 volume settings 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.

4.1.42.2 Dependencies

The OMX_AUDIO_CONFIG_CHANNELMUTETYPE structure may be queried using `OMX_GetConfig` and set using `OMX_SetConfig` at any time after the component has been loaded.

4.1.42.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_CHANNELMUTETYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetConfig` function is called and the component is not in a valid state.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetConfig` function is called and the value of `nPortIndex` exceeds the number of ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

4.1.42.4 Post-processing Conditions

In case of an `OMX_SetConfig` call using this structure, the specified audio channel is muted or unmuted according to the structure data. In case of an `OMX_GetConfig` call using this structure, the IL client can determine if the channel is muted or not.

4.1.43 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;
```

4.1.43.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_LOUDNESSTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `bLoudness` enable the loudness if set to OMX_TRUE or disables the loudness effect if set to OMX_FALSE.

4.1.43.2 Dependencies

The OMX_AUDIO_CONFIG_LOUDNESSTYPE structure may be queried using `OMX_GetConfig` and set using `OMX_SetConfig` at any time after the component has been loaded.

4.1.43.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_LOUDNESSTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetConfig` function is called and the component is not in a valid state.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetConfig` function is called and the value of `nPortIndex` exceeds the number of ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

4.1.43.4 Post-processing Conditions

In case of an `OMX_SetConfig` call using this structure, the loudness effect is enabled or disabled according to the struct data. In case of an `OMX_GetConfig` call using this structure, the IL client can determine if the loudness effect is enabled.

4.1.44 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;
```

4.1.44.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_BASSTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port..
- `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.

4.1.44.2 Dependencies

The OMX_AUDIO_CONFIG_BASSTYPE structure may be queried using `OMX_GetConfig` and set using `OMX_SetConfig` at any time after the component has been loaded.

4.1.44.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_BASSTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetConfig` function is called and the component is not in a valid state.

- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetConfig` function is called and the value of `nPortIndex` exceeds the number of ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

4.1.44.4 Post-processing Conditions

In case of a SetConfig call using this structure, the bass level is set. In case of a GetConfig call using this structure, the IL client can determine the level.

4.1.45 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;
```

4.1.45.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_TREBLETYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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.

4.1.45.2 Dependencies

The OMX_AUDIO_CONFIG_TREBLETYPE structure may be queried using `OMX_GetConfig` and set using `OMX_SetConfig` at any time after the component has been loaded.

4.1.45.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_TREBLETYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetConfig` function is called and the component is not in a valid state.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetConfig` function is called and the value of `nPortIndex` exceeds the number of ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

4.1.45.4 Post-processing Conditions

In case of an `OMX_SetConfig` call using this structure, the treble level is set. In case of an `OMX_GetConfig` call using this structure, the IL client can determine the level.

4.1.46 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;
```

4.1.46.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_EQUALIZERTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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.

4.1.46.2 Dependencies

The OMX_AUDIO_CONFIG_EQUALIZERTYPE structure may be queried using `OMX_GetConfig` and set using `OMX_SetConfig` at any time after the component has been loaded.

4.1.46.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_EQUALIZERTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetConfig` function is called and the component is not in a valid state.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

- OMX_ErrorNoMore when the OMX_GetConfig function is called and the value of nPortIndex exceeds the number of ports for the component.
- OMX_ErrorUnsupportedSetting when a field in the structure is unsupported by the component during an OMX_SetConfig call.

4.1.46.4 Post-processing Conditions

In case of an OMX_SetConfig call using this structure, the graphic equalizer algorithm parameters are set. In case of an OMX_GetConfig call using this structure, the IL client can determine algorithm parameters.

4.1.47 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;
```

4.1.47.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_STEREOWIDENINGTYPE are defined as follows.

- 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.
- nVersion is the version of the structure.
- nPortIndex is the read-only value containing the index of the port.
- 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 4-25.

Field Name	Value	Description
OMX_AUDIO_StereoWideningHeadphones		Stereo widening for headphones.
OMX_AUDIO_StereoWidenindLoudspeakers		Stereo widening for two closely spaced loudspeakers.
OMX_AUDIO_StereoWideningMax	0x7FFFFFFF	Allowance for expansion in the number of stereo widening types.

Table 4-25. Stereo Widening Processing Type

- 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.

4.1.47.2 Dependencies

The OMX_AUDIO_CONFIG_STEREOWIDENINGTYPE structure may be queried using OMX_GetConfig and set using OMX_SetConfig at any time after the component has been loaded.

4.1.47.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_STEREOWIDENINGTYPE structure, the following error conditions can occur:

- OMX_ErrorBadParameter if one or more fields of the structure are incorrect.
- OMX_ErrorUnSupportedIndex when the feature is unsupported.
- OMX_ErrorInvalidState when the OMX_SetConfig function is called and the component is not in a valid state.
- OMX_ErrorVersionMismatch when the nVersion field of the structure does not match the expected version for the component.
- OMX_ErrorNoMore when the OMX_GetConfig function is called and the value of nPortIndex exceeds the number of ports for the component.
- OMX_ErrorUnsupportedSetting when a field in the structure is unsupported by the component during an OMX_SetConfig call.

4.1.47.4 Post-processing Conditions

In case of an OMX_SetConfig call using the OMX_AUDIO_CONFIG_STEREOWIDENINGTYPE structure, the stereo widening

algorithm parameters are set. In case of an `OMX_GetConfig` call using this structure, the IL client can determine algorithm parameters.

4.1.48 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;
```

4.1.48.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_CHORUSTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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. The range of values is 0-100.
- `nFeedback` is the feedback from the chorus output to the input in percentage.

4.1.48.2 Dependencies

The OMX_AUDIO_CONFIG_CHORUSTYPE structure may be queried using `OMX_GetConfig` and set using `OMX_SetConfig` at any time after the component has been loaded.

4.1.48.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_CHORUSTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetConfig` function is called and the component is not in a valid state.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetConfig` function is called and the value of `nPortIndex` exceeds the number of ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

4.1.48.4 Post-processing Conditions

In case of an `OMX_SetConfig` call using this structure, the chorus algorithm parameters are set. In case of an `OMX_GetConfig` call using this structure, the IL client can determine the algorithm parameters.

4.1.49 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;
```

4.1.49.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_REVERBERATIONTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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.

4.1.49.2 Dependencies

The OMX_AUDIO_CONFIG_REVERBERATIONTYPE structure may be queried using `OMX_GetConfig` and set using `OMX_SetConfig` at any time after the component has been loaded.

4.1.49.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_REVERBERATIONTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetConfig` function is called and the component is not in a valid state.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when the `OMX_GetConfig` function is called and the value of `nPortIndex` exceeds the number of ports for the component.

- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

4.1.49.4 Post-processing Conditions

In case of an `OMX_SetConfig` call using this structure, the reverberation algorithm parameters are set. In case of an `OMX_GetConfig` call using this structure, the IL client can determine the algorithm parameters.

4.1.50 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;
```

4.1.50.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_ECHOCANCELATIONTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `eEchoCancelation` is the enumeration for enabling/disabling echo cancellation and selecting the mode, as shown in Table 4-26.

Field Name	Value	Description
`OMX_AUDIO_EchoCanOff`	0	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).

Field Name	Value	Description
OMX_AUDIO_EchoCanMax	0x7FFFFFFF	Allowance for expansion with additional types.

Table 4-26. Echo Cancellation Values

4.1.50.2 Dependencies

The OMX_AUDIO_CONFIG_ECHOCANCELATIONTYPE structure may be queried using OMX_GetConfig and set using OMX_SetConfig at any time after the component has been loaded.

4.1.50.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_ECHOCANCELATIONTYPE structure, the following error conditions can occur:

- OMX_ErrorBadParameter if one or more fields of the structure are incorrect.
- OMX_ErrorUnSupportedIndex when the feature is unsupported.
- OMX_ErrorInvalidState when the OMX_SetConfig function is called and the component is not in a valid state.
- OMX_ErrorVersionMismatch when the nVersion field of the structure does not match the expected version for the component.
- OMX_ErrorNoMore when the OMX_GetConfig function is called and the value of nPortIndex exceeds the number of ports for the component.
- OMX_ErrorUnsupportedSetting when a field in the structure is unsupported by the component during an OMX_SetConfig call.

4.1.50.4 Post-processing Conditions

In case of an OMX_SetConfig call using the OMX_AUDIO_CONFIG_ECHOCANCELATIONTYPE structure, echo cancellation is enabled or disabled according to the struct data. In case of an OMX_GetConfig call using this structure, the IL client can determine if the processing is enabled.

4.1.51 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;
```

4.1.51.1 Parameter Definitions

The parameters for OMX_AUDIO_CONFIG_NOISEREDUCTIONTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `bNoiseReduction` enables noise reduction processing if set to OMX_TRUE or disables noise reduction processing if set to OMX_FALSE.

4.1.51.2 Dependencies

The OMX_AUDIO_CONFIG_NOISEREDUCTIONTYPE structure may be queried using `OMX_GetConfig` and set using `OMX_SetConfig` at any time after the component has been loaded.

4.1.51.3 Error Conditions

On processing the OMX_AUDIO_CONFIG_NOISEREDUCTIONTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnSupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when `OMX_SetConfig` function is called and the component is not in a valid state.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

- `OMX_ErrorNoMore` when the `OMX_GetConfig` function is called and the value of `nPortIndex` exceeds the number of ports for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

4.1.51.4 Post-processing Conditions

In case of an `OMX_SetConfig` call using the OMX_AUDIO_CONFIG_NOISEREDUCTIONTYPE structure, noise reduction processing is enabled or disabled according to the struct data. In case of an `OMX_GetConfig` call using this structure, the IL client can determine if the processing is enabled.

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 configurations 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 4-27 lists the uncompressed 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_Format12bitRGB444`	12 bits per pixel RGB format with colors stored as Red 11:8, Green 7:4, and Blue 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_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_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_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_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_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_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_Format32bitBGRA8888	32 bits per pixel ARGB format with colors stored as Alpha 31:24 Red 23:16, Green 15:8, and Blue 7:0.
OMX_COLOR_Format32bitARGB8888	24 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. 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, 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. U and V pixels are sub-sampled by a factor of two both horizontally and vertically.
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, 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 interleaved U and V pixels. U and V pixels are sub-sampled by a factor of two both horizontally and vertically.
OMX_COLOR_FormatYUV422Planar	YUV planar format, organized with three separate planes for each color component, namely Y, U, and V.
OMX_COLOR_FormatYUV422PackedPlanar	YUV planar format, organized with three separate planes for each color component, namely Y, U, and V. This format differs from OMX_COLOR_FormatYUV422Planar in that each slice of data shall contain a plane of Y, U, and V data, 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 interleaved U and V pixels.
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_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.

Table 4-27. Uncompressed Data Formats

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 a 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_FormatYUV411PackedPlanar`, and `OMX_COLOR_FormatYUV422PackedPlanar`. 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. 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 a `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 4-28 describes the index values that relate to video.

Index	Description
OMX_IndexParamCommonDeblocking	Used with OMX_GetParameter and OMX_SetParameter to access OMX_CONFIG_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_CONFIG_SENSORMODETYPE. The mode of the sensor controls the resolution 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.
OMX_IndexConfigCommonColorFormatConversion	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.

Index	Description
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_IndexConfigCommonFrameStabilisation	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.
OMX_IndexConfigCommonMirror	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_MIRRORTYPE.

Index	Description
	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_IndexConfigCommonCrop	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_SALEFACTORTYPE. 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.
OMX_IndexConfigCommonExposure	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_EXPOSURECON

Index	Description
	TROLTYPE. 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. Backlight 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

Index	Description
	of pixels in video or image data.
OMX_IndexConfigCommonExclusionRect	Used with OMX_GetConfig and OMX_SetConfig to access OMX_CONFIG_EXCLUSIONRECTTYPE. 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.

Table 4-28. Index Values for Video

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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `bDeblocking` is a Boolean value that enables or disables de-blocking.

4.2.5.2 Dependencies

The parameter may be queried using `OMX_GetParameter` or set using `OMX_SetParameter` at any time that the component is initialized.

4.2.5.3 Error Conditions

On processing the OMX_ PARAM_DEBLOCKINGTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorTimeout` if the component did not respond in time.

4.2.5.4 Post-processing Conditions

The de-blocking filter used when processing image or video data for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetParameter`.

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, a 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_CONFIG_INTERLEAVETYPE;
```

4.2.6.1 Parameters

The parameters for OMX_PARAM_INTERLEAVETYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- nPortIndex is the read-only value containing the index of the port.
- 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.

4.2.6.2 Dependencies

The parameter may be queried using `OMX_GetParameter` or set using `OMX_SetParameter` at any time that the component is initialized.

4.2.6.3 Error Conditions

On processing the OMX_PARAM_INTERLEAVETYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorTimeout` if the component did not respond in time.

4.2.6.4 Post-processing Conditions

Interleaving or de-interleaving for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetParameter`.

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_FRAMESIZETYPE sFrameSize;
} OMX_PARAM_SENSORMODETYPE;
```

4.2.7.1 Parameters

The parameters for OMX_ PARAM_SENSORMODETYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.

- `nPortIndex` is the read-only value containing the index of the port.
- `nFrameRate` is the frame rate of the image sensor in frames per second.
- `sFrameSize` is the resolution of the image sensor mode.

4.2.7.2 Dependencies

The parameter may be queried using `OMX_GetParameter` or set using `OMX_SetParameter` at any time that the component is initialized.

4.2.7.3 Error Conditions

On processing the OMX_ PARAM_SENSORMODETYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetParameter` call.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorTimeout` if the component did not respond in time.

4.2.7.4 Post-processing Conditions

The sensor mode used when processing image or video data for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetParameter`.

4.2.8 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;
```

4.2.8.1 Parameters

The parameters for OMX_CONFIG_COLORCONVERSIONTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only field indicating the index of the port.
- `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

4.2.8.2 Dependencies

The parameter may be queried using `OMX_GetConfig` or set using `OMX_SetConfig` at any time that the component is initialized.

4.2.8.3 Error Conditions

On processing the OMX_CONFIG_COLORCONVERSIONTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.
- `OMX_ErrorTimeout` if the component did not respond in time.
- `OMX_NotReady` if an `OMX_SetConfig` operation that has not completed processing. The caller should retry the `OMX_GetConfig` or `OMX_SetConfig` call.

4.2.8.4 Post-processing Conditions

The color conversion used when processing image or video data for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetConfig`.

4.2.9 OMX_SCALEFACTORTYPE

Scaling is used to stretch or shrink video or image data on the specified input or output port.

OMX_SCALEFACTORTYPE is defined as follows.

```
typedef struct OMX_SCALEFACTORTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_S32 xWidth;
    OMX_S32 xHeight;
}OMX_SCALEFACTORTYPE;
```

4.2.9.1 Parameters

The parameters for OMX_SCALEFACTORTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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%.

4.2.9.2 Dependencies

The parameter may be queried using `OMX_GetConfig` or set using `OMX_SetConfig` at any time that the component is initialized.

4.2.9.3 Error Conditions

On processing the OMX_SCALEFACTORTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect. This error may also occur when the component does not support the scaling factor requested.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.
- `OMX_ErrorTimeout` if the component did not respond in time.
- `OMX_NotReady` if an `OMX_SetConfig` operation has not completed processing. The caller should retry the `OMX_GetConfig` or `OMX_SetConfig` call.

4.2.9.4 Post-processing Conditions

The scaling used when processing image or video data for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetConfig`.

4.2.10 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;
```

4.2.10.1 Parameters

The parameters for OMX_CONFIG_IMAGEFILTERTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `eImageFilter` is the enumerated valued indicating the image filter used. Table 4-29 details the values that can be selected for the image filter.

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.

Table 4-29. Image Filter Values

4.2.10.2 Dependencies

The parameter may be queried using OMX_GetConfig or set using OMX_SetConfig at any time that the component is initialized.

4.2.10.3 Error Conditions

On processing the OMX_CONFIG_IMAGEFILTERTYPE structure, the following error conditions can occur:

- OMX_ErrorBadParameter if one or more fields of the structure are incorrect.
- OMX_ErrorUnsupportedIndex when the feature is unsupported.
- OMX_ErrorVersionMismatch when the nVersion field of the structure does not match the expected version for the component.
- OMX_ErrorUnsupportedSetting when a field in the structure is unsupported by the component during an OMX_SetConfig call.
- OMX_ErrorTimeout if the component did not respond in time.
- OMX_NotReady if an OMX_SetConfig operation has not completed processing. The caller should retry the OMX_GetConfig or OMX_SetConfig call.

4.2.10.4 Post-processing Conditions

The image filtering used when processing image or video data for the component on the port specified by nPortIndex is configured explicitly when set using OMX_SetConfig.

4.2.11 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;
```

4.2.11.1 Parameters

The parameters for OMX_CONFIG_COLORENHANCEMENTTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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.

4.2.11.2 Dependencies

The parameter may be queried using `OMX_GetConfig` or set using `OMX_SetConfig` at any time that the component is initialized.

4.2.11.3 Error Conditions

On processing the OMX_CONFIG_COLORENHANCEMENTTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.
- `OMX_ErrorTimeout` if the component did not respond in time.
- `OMX_NotReady` if an `OMX_SetConfig` operation has not completed processing. The caller should retry the `OMX_GetConfig` or OMX_`SetConfig` call.

4.2.11.4 Post-processing Conditions

The color enhancement used when processing image or video data for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetConfig`.

4.2.12 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;
```

4.2.12.1 Parameters

The parameters for OMX_CONFIG_COLORKEYTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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.

4.2.12.2 Dependencies

The parameter may be queried using `OMX_GetConfig` or set using `OMX_SetConfig` at any time that the component is initialized.

4.2.12.3 Error Conditions

On processing the OMX_CONFIG_COLORKEYTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

- `OMX_ErrorTimeout` if the component did not respond in time.
- `OMX_NotReady` if an `OMX_SetConfig` operation has not completed processing. The caller should retry the `OMX_GetConfig` or `OMX_SetConfig` call.

4.2.12.4 Post-processing Conditions

The color key used when processing image or video data for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetConfig`.

4.2.13 OMX_CONFIG_COLORBLENDTYPE

Color blending is used to perform arithmetic operations between two sources when mixing image or video data.

```
typedef struct OMX_CONFIG_COLORBLENDTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nRGBAlphaConstant;
    OMX_COLORBLENDTYPE  eColorBlend;
} OMX_CONFIG_COLORBLENDTYPE;
```

OMX_CONFIG_COLORBLENDTYPE is defined as follows.

4.2.13.1 Parameters

The parameters for OMX_CONFIG_COLORBLENDTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `nRGBAlphaConstant` is the 32-bit per color channel constant alpha value for blending when the OMX_COLORBLENDTYPE is OMX_ColorBlendAlphaConstant.
- `eColorBlend` is the enumerated valued indicating the color blend operation used. Table 4-30 details the values that can be selected for color blending.

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_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).

Table 4-30. Color Blending Values

4.2.13.2 Dependencies

The parameter may be queried using OMX_GetConfig or set using OMX_SetConfig at any time that the component is initialized.

4.2.13.3 Error Conditions

On processing the OMX_CONFIG_COLORBLENDTYPE structure, the following error conditions can occur:

- OMX_ErrorBadParameter if one or more fields of the structure are incorrect.
- OMX_ErrorUnsupportedIndex when the feature is unsupported.
- OMX_ErrorVersionMismatch when the nVersion field of the structure does not match the expected version for the component.
- OMX_ErrorUnsupportedSetting when a field in the structure is unsupported by the component during an OMX_SetConfig call.
- OMX_ErrorTimeout if the component did not respond in time.
- OMX_NotReady if an OMX_SetConfig operation has not completed processing. The caller should retry the OMX_GetConfig or OMX_SetConfig call.

4.2.13.4 Post-processing Conditions

The color blend used when processing image or video data for the component on the port specified by nPortIndex is configured explicitly when set using OMX_SetConfig.

4.2.14 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_SENSORMODE 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;
```

4.2.14.1 Parameters

The parameters for OMX_FRAMESIZETYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `nWidth` is the width of the rectangle in pixels.
- `nHeight` is the height of the rectangle in pixels.

4.2.14.2 Dependencies

The parameter may be queried using `OMX_GetConfig` or set using `OMX_SetConfig` at any time that the component is initialized.

4.2.14.3 Error Conditions

On processing the OMX_FRAMESIZETYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.
- `OMX_ErrorTimeout` if the component did not respond in time.
- `OMX_NotReady` if an `OMX_SetConfig` operation has not completed processing. The caller should retry the `OMX_GetConfig` or `OMX_SetConfig` call.

4.2.14.4 Post-processing Conditions

The frame size used when processing image or video data for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetConfig`.

4.2.15 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;
```

4.2.15.1 Parameters

The parameters for OMX_CONFIG_ROTATIONTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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°. Rotation is clockwise.

4.2.15.2 Dependencies

The parameter may be queried using `OMX_GetConfig` or set using `OMX_SetConfig` at any time that the component is initialized.

4.2.15.3 Error Conditions

On processing the OMX_CONFIG_ROTATIONTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect. This error may occur when the value specified in `nRotation` is not supported.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.
- `OMX_ErrorTimeout` if the component did not respond in time.
- `OMX_NotReady` if an `OMX_SetConfig` operation has not completed processing. The caller should retry the `OMX_GetConfig` or `OMX_SetConfig` call.

4.2.15.4 Post-processing Conditions

The angle of rotation used when processing image or video data for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetConfig`.

4.2.16 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;
```

4.2.16.1 Parameters

The parameters for OMX_CONFIG_MIRRORTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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 4-31 identifies the mirroring 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_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.

Table 4-31. Mirror Type Values

4.2.16.2 Dependencies

The parameter may be queried using OMX_GetConfig or set using OMX_SetConfig at any time that the component is initialized.

4.2.16.3 Error Conditions

On processing the OMX_CONFIG_MIRRORTYPE structure, the following error conditions can occur:

- OMX_ErrorBadParameter if one or more fields of the structure are incorrect.
- OMX_ErrorUnsupportedIndex when the feature is unsupported.
- OMX_ErrorVersionMismatch when the nVersion field of the structure does not match the expected version for the component.
- OMX_ErrorUnsupportedSetting when a field in the structure is unsupported by the component during an OMX_SetConfig call.
- OMX_ErrorTimeout if the component did not respond in time.
- OMX_NotReady if an OMX_SetConfig operation has not completed processing. The caller should retry the OMX_GetConfig or OMX_SetConfig call.

4.2.16.4 Post-processing Conditions

The mirroring used when processing image or video data for the component on the port specified by nPortIndex is configured explicitly when set using OMX_SetConfig.

4.2.17 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;
```

4.2.17.1 Parameters

The parameters for OMX_CONFIG_POINTTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `nX` is the X-coordinate location in pixels in the horizontal direction.
- `nY` is the Y-coordinate location in pixels in the vertical direction.

4.2.17.2 Dependencies

The parameter may be queried using `OMX_GetConfig` or set using `OMX_SetConfig` at any time that the component is initialized.

4.2.17.3 Error Conditions

On processing the OMX_CONFIG_POINTTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.
- `OMX_ErrorTimeout` if the component did not respond in time.
- `OMX_NotReady` if an `OMX_SetConfig` operation has not completed processing. The caller should retry the `OMX_GetConfig` or `OMX_SetConfig` call.

4.2.17.4 Post-processing Conditions

The point location specified by the X,Y coordinates used when processing image or video data for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetConfig`.

4.2.18 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;
```

4.2.18.1 Parameters

The parameters for OMX_CONFIG_RECTTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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.

4.2.18.2 Dependencies

The parameter may be queried using `OMX_GetConfig` or set using `OMX_SetConfig` at any time that the component is initialized.

4.2.18.3 Error Conditions

On processing the OMX_CONFIG_RECTTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

- `OMX_ErrorTimeout` if the component did not respond in time.
- `OMX_NotReady` if an `OMX_SetConfig` operation has not completed processing. The caller should retry the `OMX_GetConfig` or `OMX_SetConfig` call.

4.2.18.4 Post-processing Conditions

The rectangle used when processing image or video data for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetConfig`.

4.2.19 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;
```

4.2.19.1 Parameters

The parameters for OMX_CONFIG_FRAMESTABTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `bStab` is the Boolean value that enables or disables frame stabilization.

4.2.19.2 Dependencies

The parameter may be queried using `OMX_GetConfig` or set using `OMX_SetConfig` at any time that the component is initialized.

4.2.19.3 Error Conditions

On processing the OMX_CONFIG_FRAMESTABTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.

- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during a OMX_SetConfig call.
- `OMX_ErrorTimeout` if the component did not respond in time.
- `OMX_NotReady` if an `OMX_SetConfig` operation has not completed processing. The caller should retry the `OMX_GetConfig` or `OMX_SetConfig` call.

4.2.19.4 Post-processing Conditions

The frame stabilization used when processing image or video data for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetConfig`.

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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `eWhiteBalControl` is the enumerated valued indicating the type of white balance control used. Table 4-32 details the values that can be selected for 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_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.

Table 4-32. White Balance Control

4.2.20.2 Dependencies

The parameter may be queried using `OMX_GetConfig` or set using `OMX_SetConfig` at any time that the component is initialized.

4.2.20.3 Error Conditions

On processing the OMX_CONFIG_WHITEBALCONTROLTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect. This error may also occur when a specific white balance control is not supported.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

- `OMX_ErrorTimeout` if the component did not respond in time.
- `OMX_NotReady` if an `OMX_SetConfig` operation has not completed processing. The caller should retry the `OMX_GetConfig` or `OMX_SetConfig` call.

4.2.20.4 Post-processing Conditions

The frame stabilization used when processing image or video data for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetConfig`.

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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `eExposureControl` is an enumerated value that selects the type of exposure used. Table 4-33 details the values that can be selected for exposure.

OMX_EXPOSURECONTROLTYPE Enumerated Value	**Description**
`OMX_ExposureControlOff`	Disables exposure control
`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_ExposureControlSmallAperture	Exposure when using a small aperture on the camera

Table 4-33. Exposure Control

4.2.21.2 Dependencies

The parameter may be queried using OMX_GetConfig or set using OMX_SetConfig at any time that the component is initialized.

4.2.21.3 Error Conditions

On processing the OMX_CONFIG_EXPOSURECONTROLTYPE structure, the following error conditions can occur:

- OMX_ErrorBadParameter if one or more fields of the structure are incorrect.
- OMX_ErrorUnsupportedIndex when the feature is unsupported.
- OMX_ErrorVersionMismatch when the nVersion field of the structure does not match the expected version for the component.
- OMX_ErrorUnsupportedSetting when a field in the structure is unsupported by the component during an OMX_SetConfig call.
- OMX_ErrorTimeout if the component did not respond in time.
- OMX_NotReady if an OMX_SetConfig operation has not completed processing. The caller should retry the OMX_GetConfig or OMX_SetConfig call.

4.2.21.4 Post-processing Conditions

The exposure used when processing image or video data for the component on the port specified by nPortIndex is configured explicitly when set using OMX_SetConfig.

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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `nContrast` is the value for contrast. The range of values is -100 to 100. The value 0x0 indicates no contrast change to pixel data.

4.2.22.2 Dependencies

The parameter may be queried using `OMX_GetConfig` or set using `OMX_SetConfig` at any time that the component is initialized.

4.2.22.3 Error Conditions

On processing the OMX_CONFIG_CONTRASTTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.
- `OMX_ErrorTimeout` if the component did not respond in time.
- `OMX_NotReady` if an `OMX_SetConfig` operation has not completed processing. The caller should retry the `OMX_GetConfig` or `OMX_SetConfig` call.

4.2.22.4 Post-processing Conditions

The contrast used when processing image or video data for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetConfig`.

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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `nBrightness` is the value for brightness in the range 0% to 100%, where 0% produces all black pixels and 100% produces entirely white.

4.2.23.2 Dependencies

The parameter may be queried using `OMX_GetConfig` or set using `OMX_SetConfig` at any time that the component is initialized.

4.2.23.3 Error Conditions

On processing the OMX_CONFIG_BRIGHTNESSTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.
- `OMX_ErrorTimeout` if the component did not respond in time.
- `OMX_NotReady` if an OMX_SetConfig operation has not completed processing. The caller should retry the `OMX_GetConfig` or `OMX_SetConfig` call.

4.2.23.4 Post-processing Conditions

The brightness applied by the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetConfig`.

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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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.

4.2.24.2 Dependencies

The parameter may be queried using `OMX_GetConfig` or set using `OMX_SetConfig` at any time that the component is initialized.

4.2.24.3 Error Conditions

On processing the OMX_CONFIG_BACKLIGHTTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.

- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during a OMX_SetConfig call.
- `OMX_ErrorTimeout` if the component did not respond in time.
- `OMX_NotReady` if an `OMX_SetConfig` operation has not completed processing. The caller should retry the `OMX_GetConfig` or `OMX_SetConfig` call.

4.2.24.4 Post-processing Conditions

The backlight of the display for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetConfig`.

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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `nGamma` is the value for gamma in the range of -100 to 100. The value 0x0 indicates no gamma change to pixel data. A value of -100 produces all black pixels, and a value of 100 produces all white pixels.

4.2.25.2 Dependencies

The parameter may be queried using `OMX_GetConfig` or set using `OMX_SetConfig` at any time that the component is initialized.

4.2.25.3 Error Conditions

On processing the OMX_CONFIG_GAMMATYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.
- `OMX_ErrorTimeout` if the component did not respond in time.
- `OMX_NotReady` if an OMX_SetConfig operation has not completed processing. The caller should retry the `OMX_GetConfig` or `OMX_SetConfig` call.

4.2.25.4 Post-processing Conditions

The gamma used when performing color format conversion for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetConfig`.

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_SATURATIONTTYPE;
```

4.2.26.1 Parameters

The parameters for OMX_CONFIG_SATURATIONTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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.

4.2.26.2 Dependencies

The parameter may be queried using `OMX_GetConfig` or set using `OMX_SetConfig` at any time that the component is initialized.

4.2.26.3 Error Conditions

On processing the OMX_CONFIG_SATURATIONTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.
- `OMX_ErrorTimeout` if the component did not respond in time.
- `OMX_NotReady` if an `OMX_SetConfig` operation has not completed processing. The caller should retry the `OMX_GetConfig` or `OMX_SetConfig` call.

4.2.26.4 Post-processing Conditions

The saturation used when performing color format conversion for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetConfig`.

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.

- `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.
- `nVersion` is the version of the structure.

- `nPortIndex` is the read-only value containing the index of the port.
- `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.

4.2.27.2 Dependencies

The parameter may be queried using `OMX_GetConfig` or set using `OMX_SetConfig` at any time that the component is initialized.

4.2.27.3 Error Conditions

On processing the OMX_CONFIG_LIGHTNESSTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorTimeout` if the component did not respond in time.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.
- `OMX_NotReady` if is an `OMX_SetConfig` operation has not completed processing. The caller should retry the `OMX_GetConfig` or `OMX_SetConfig` call.

4.2.27.4 Post-processing Conditions

The type of lightness used when performing color format conversion for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetConfig`.

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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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`.
- `nAlpha` indicates the alpha value used when blending planes, if the blending operation uses global alpha. For information on blending operations, see section 4.2.13.

4.2.28.2 Dependencies

The parameter may be queried using `OMX_GetConfig` or set using `OMX_SetConfig` at any time that the component is initialized.

4.2.28.3 Error Conditions

On processing the OMX_CONFIG_PLANEBLENDTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.
- `OMX_ErrorTimeout` if the component did not respond in time.
- `OMX_NotReady` if an `OMX_SetConfig` operation has not completed processing. The caller should retry the `OMX_GetConfig` or `OMX_SetConfig` call.

4.2.28.4 Post-processing Conditions

The type of plane blending used when enabled for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetConfig`.

4.2.29 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;
```

4.2.29.1 Parameters

The parameters for OMX_CONFIG_DITHERTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `eDither` is the type of dithering used when performing color format conversion. Table 4-34 details the values that can be selected for dithering.

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

Table 4-34. Dithering Values

4.2.29.2 Dependencies

The parameter may be queried using `OMX_GetConfig` or set using `OMX_SetConfig` at any time that the component is initialized.

4.2.29.3 Error Conditions

On processing the OMX_CONFIG_DITHERTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorUnsupportedIndex` when the feature is unsupported.
- `OMX_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorUnsupportedSetting` when a field in the structure is unsupported by the component during an `OMX_SetConfig` call.

- `OMX_ErrorTimeout` if the component did not respond in time.
- `OMX_NotReady` if an `OMX_SetConfig` operation has not completed processing. The caller should retry the `OMX_GetConfig` or `OMX_SetConfig` call.

4.2.29.4 Post-processing Conditions

The type of dithering used when performing color format conversion for the component on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetConfig`.

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 General Enumerations

The `OMX_VIDEO_CODINGTYPE` enumeration defines the video coding types supported in OpenMAX IL version.1.0. If `OMX_VIDEO_CodingUnused` is selected, then the coding selection shall be done in a vendor-specific way. Table 4-35 shows the OpenMAX-supported video compression formats.

Field Name	Value	Coding Type Descriptions	References to Standards
`OMX_VIDEO_CodingUnused`	0x0	No coding applied. Use `eColorFormat`	Not available
`OMX_VIDEO_CodingAutoDetect`	0x1	Auto-detection by the OpenMAX component	Not available
`OMX_VIDEO_CodingMPEG2`	0x2	MPEG-2, also known as H.262 video format	MPEG2
`OMX_VIDEO_CodingH263`	0x3	ITU H.263 video format	H263
`OMX_VIDEO_CodingMPEG4`	0x4	MPEG-4 video format	MPEG4
`OMX_VIDEO_CodingWMV`	0x5	All versions of the Windows Media video format	WMV
`OMX_VIDEO_CodingRV`	0x6	All versions of the RealVideo® format	RV
`OMX_VIDEO_CodingAVC`	0x7	ITU H.264/AVC video format	H264

OMX_VIDEO_CodingMJPEG	0x8	Motion JPEG video format	MJPEG
OMX_VIDEO_CodingMax	0x7FFFFFFF	Maximum value	N/A

Table 4-35. Supported Video Compression Formats

The OMX_VIDEO_PICTURETYPE enumeration defines the video picture types supported in OpenMAX IL version.1.0. Table 4-36 describes the supported video picture types.

Field Name	Value	Picture Type Descriptions
OMX_VIDEO_PictureTypeI	0x01	General I-frame type
OMX_VIDEO_PictureTypeP	0x02	General P-frame type
OMX_VIDEO_PictureTypeB	0x04	General B-frame type
OMX_VIDEO_PictureTypeSI	0x08	H.263 SI-frame type
OMX_VIDEO_PictureTypeSP	0x10	H.263 SP-frame type
OMX_VIDEO_PictureTypeEI	0x20	H.264 EI-frame type
OMX_VIDEO_PictureTypeEP	0x40	H.264 EP-frame type
OMX_VIDEO_PictureTypeS	0x80	MPEG-4 S-frame type
OMX_VIDEO_PictureTypeMax	0x7FFFFFFF	Maximum value

Table 4-36. Supported Video Picture Types

4.3.2 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 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 4-37 identifies the video indices.

OpenMAX IL Indices (OMX_Index.h)	Corresponding OpenMAX IL Video Structures (OMX_Video.h)
OMX_IndexParamVideoPortFormat	OMX_VIDEO_PARAM_PORTFORMATTYPE
OMX_IndexParamQuantization	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_IndexParamVideoCorrection	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
OMX_IndexParamVideoAvc	OMX_VIDEO_PARAM_AVCTYPE
OMX_IndexParamVideoH263	OMX_VIDEO_PARAM_H263TYPE

Table 4-37. Video Indices

4.3.3 Video Use Cases Examples

Figure 4-3 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.

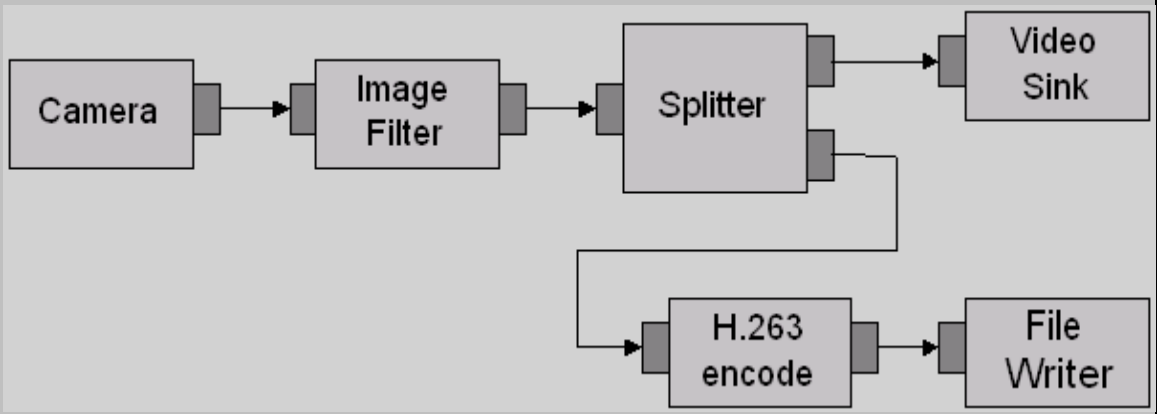


Figure 4-3. H.263 Video Encode Use Case

Figure 4-3 shows six components, namely the camera, the image filter, the splitter, the H.263 video encoder, the file writer, and the video sink.

Figure 4-4 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.

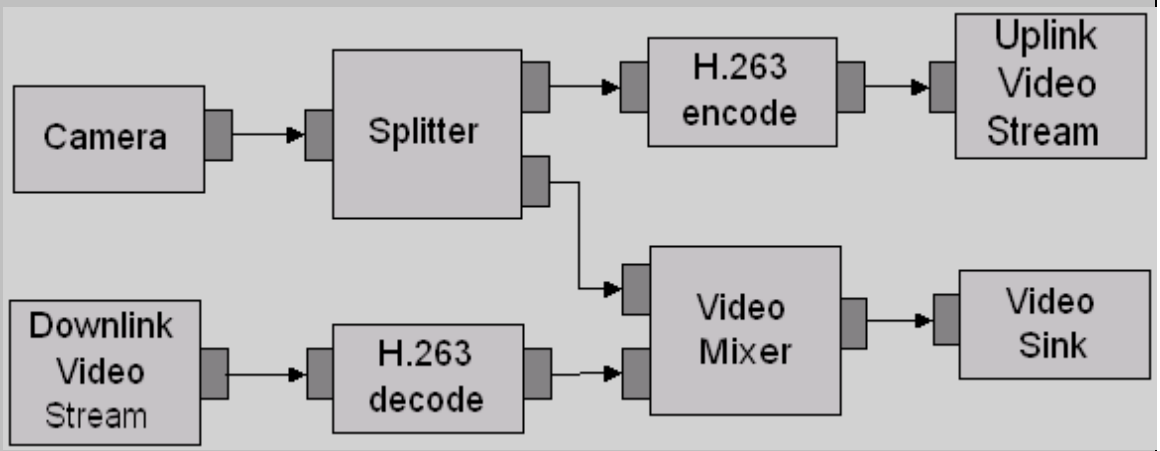


Figure 4-4. Video Conferencing Use Case

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.

4.3.4 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 or default definition of a video port for a component using the `OMX_GetParameter` function. It is also used to set the definition of a video port for a 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_Index_ParamVideoPort.

OMX_VIDEO_PORTDEFINITIONTYPE is defined as follows.

```
typedef struct OMX_VIDEO_PORTDEFINITIONTYPE {
    OMX_STRING cMIMEType;
    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_PORTDEFINITIONTYPE;
```

4.3.4.1 Parameters

The parameters for OMX_VIDEO_PORTDEFINITIONTYPE are defined as follows.

- `cMIMEType` is the MIME type of data for the port.
- `pNativeRender` is the read-only field containing a reference to the platform-specific native renderer. This is NULL if a platform specific native renderer is not available.
- `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 the read-only 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. The value for nStride cannot be 0x0.

- 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 the minimum buffer payload for uncompressed data, see section 4.2.2.
- 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 frame rate specified is that used on the port if the data is not compressed. 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.

4.3.4.2 Dependencies

The parameter may be queried at any time that the component is initialized. The parameter may only be set using OMX_SetParameter when the component is in the OMX_StateLoaded state.

4.3.4.3 Error Conditions

On processing the OMX_VIDEO_PORTDEFINITIONTYPE structure, the following error conditions can occur:

- OMX_ErrorBadParameter if one or more fields of the structure are incorrect.
- OMX_ErrorNotImplemented when the feature is unsupported.
- OMX_ErrorInvalidState when the OMX_SetParameter function is called and the component is in the OMX_StateInvalid state.
- 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_ErrorVersionMismatch when the nVersion field of the structure does not match the expected version for the component.
- OMX_ErrorBadParameter if one or more fields of the structure are incorrect.
- OMX_ErrorNoMore when the OMX_GetParameter function is called and the value of nPortIndex exceeds the number of video ports for the component.

4.3.4.4 Post-processing Conditions

The generic characteristics for the port indicated by nPortIndex are fully specified when set using OMX_SetParameter.

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_VIDEO_PARAM_PORTFORMATTYPE;
```

4.3.5.1 Parameters

The parameters for OMX_VIDEO_PARAM_PORTFORMATTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `eCompressionFormat` is the compression format used on the port. This essentially refers to the file extension. 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 4-35.

- `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 4-33.

4.3.5.2 Dependencies

The OMX_VIDEO_PARAM_PORTFORMATTYPE structure may be queried at any time that the component is not in the NULL state. The structure may be set using `OMX_SetParameter` when the component is in the OMX_StateLoaded state.

4.3.5.3 Error Conditions

On processing the OMX_VIDEO_PARAM_PORTFORMATTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorNotImplemented` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorNoMore` when the `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of video ports for the component.

4.3.5.4 Post-processing Conditions

The OMX_VIDEO_PARAM_PORTFORMATTYPE structure has no post-processing conditions.

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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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

The parameter may be queried at any time that the component is not in the NULL state. The parameter may only be set using `OMX_SetParameter` when the component is in the OMX_StateLoaded state after `OMX_VIDEO_PORTDEFINITIONTYPE` has been set for the port. This parameter is only applicable to certain video encoders, which include MPEG-2 and MPEG-4.

4.3.6.3 Error Conditions

On processing the OMX_VIDEO_PARAM_QUANTIZATIONTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorNotImplemented` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

4.3.6.4 Post-processing Conditions

The quantization characteristics for a video encoder on the port specified by `nPortIndex` are configured explicitly when set using `OMX_SetParameter`.

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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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

The parameter may be queried at any time that the component is not in the NULL state. The parameter may only be set using `OMX_SetParameter` when the component is in the OMX_StateLoaded state after `OMX_VIDEO_PORTDEFINITIONTYPE` has been set for the port. This parameter is only applicable to certain video encoders, such as H.261 and H.263.

4.3.7.3 Error Conditions

On processing the OMX_VIDEO_PARAM_VIDEOFASTUPDATETYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorNotImplemented` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

4.3.7.4 Post-processing Conditions

The fast update characteristics for a video encoder on the port specified by `nPortIndex` are configured explicitly when set using `OMX_SetParameter`.

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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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 4-38 enumerates the possible video bit rate control types for `OMX_VIDEO_CONTROLRATETYPE`.

Field Name	Value	Bit Rate Control Descriptions
OMX_Video_ControlRateDisable	0x0	Disable
OMX_Video_ControlRateVariable	0x1	Variable bit rate
OMX_Video_ControlRateConstant	0x2	Constant bit rate
OMX_Video_ControlRateVariableSkipFrames	0x3	Variable bit rate with frame skipping
OMX_Video_ControlRateConstantSkipFrames	0x4	Constant bit rate with frame skipping
OMX_Video_ControlRateMax	0x7FFFFFFF	Maximum value

Table 4-38. Supported Video Bit Rate Control Types

- `nTargetBitrate` is the target bit rate for video encoding in units of bits per second. For certain video encoding standards, this field is not applicable.

4.3.8.2 Dependencies

The parameter may be queried at any time that the component is not in the NULL state. The parameter may only be set using `OMX_SetParameter` when the component is in the OMX_StateLoaded state after `OMX_VIDEO_PORTDEFINITIONTYPE` has been set for the port; 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).

4.3.8.3 Error Conditions

On processing the OMX_VIDEO_PARAM_BITRATETYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorNotImplemented` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

4.3.8.4 Post-processing Conditions

The bit rate control characteristics for the video encoder on the port specified by `nPortIndex` are configured explicitly when set using `OMX_SetParameter`.

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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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 4-39 enumerates the possible video motion vector types for `OMX_VIDEO_MOTIONVECTORTYPE`.

Field Name	Value	Motion Vector Descriptions
`OMX_Video_MotionVectorPixel`	0x0	Full pixel motion vectors
`OMX_Video_MotionVectorHalfPel`	0x1	Half pixel motion vectors
`OMX_Video_MotionVectorQuarterPel`	0x2	Quarter pixel motion vectors
`OMX_Video_MotionVectorEighthPel`	0x3	Eighth pixel motion vectors
`OMX_Video_MotionVectorMax`	0x7FFFFFFF	Maximum value

Table 4-39. Supported Video Motion Vector Types

- `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

The parameter may be queried at any time that the component is not in the NULL state. The parameter may only be set using `OMX_SetParameter` when the component is in the OMX_StateLoaded state after `OMX_VIDEO_PORTDEFINITIONTYPE` has been set for the port. This parameter is only applicable to certain video encoders, which include MPEG2 and MPEG4.

4.3.9.3 Error Conditions

On processing the OMX_VIDEO_PARAM_MOTIONVECTORTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorNotImplemented` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

4.3.9.4 Post-processing Conditions

The motion vector search range for a video encoder on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetParameter`.

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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `eRefreshMode` is the enumeration for the type of intra-refresh mode. Table 4-40 shows the possible values for `OMX_VIDEO_INTRAREFRESHTYPE.`

Field Name	Value	Intra-Refresh Descriptions
OMX_VIDEO_IntraRefreshCyclic	0	Cyclic intra-refresh
OMX_VIDEO_IntraRefreshAdaptive	1	Adaptive intra-refresh
OMX_VIDEO_IntraRefreshBoth	2	Cyclic and Adaptive intra-refresh
OMX_VIDEO_IntraRefreshMax	0x7FFFFFFF	Maximum value

Table 4-40. Supported Video Intra-Refresh Types

- `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

The parameter may be queried at any time that the component is not in the NULL state. The parameter may only be set using `OMX_SetParameter` when the component is in the OMX_StateLoaded state after `OMX_VIDEO_PORTDEFINITIONTYPE` has been set for the port. This parameter is only applicable to certain video encoders, which includes MPEG4.

4.3.10.3 Error Conditions

On processing the OMX_VIDEO_PARAM_INTRAREFRESHTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorNotImplemented` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

4.3.10.4 Post-processing Conditions

The intra refresh for video encoding on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetParameter`.

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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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 portioning.
- `bEnableRVLC` is a Boolean value that enables or disables reversible variable-length coding.

4.3.11.2 Dependencies

The parameter may be queried at any time that the component is not in the NULL state. The parameter may only be set using OMX_`SetParameter` when the component is in the OMX_StateLoaded state after `OMX_VIDEO_PORTDEFINITIONTYPE` has been set for the port. This parameter is only applicable to certain video encoders, which include MPEG4.

4.3.11.3 Error Conditions

On processing the OMX_VIDEO_PARAM_ERRORCORRECTIONTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.•
- `OMX_ErrorNotImplemented` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

4.3.11.4 Post-processing Conditions

The error correction used for video encoding on the port specified by `nPortIndex` is configured explicitly when set using `OMX_SetParameter`.

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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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

The parameter may be queried at any time that the component is not in the NULL state. The parameter may only be set using `OMX_SetParameter` when the component is in the OMX_StateLoaded state after `OMX_VIDEO_PORTDEFINITIONTYPE` has been set for the port. This parameter is only applicable to certain video encoders, which include MPEG4 and other derivations of MPEG4.

4.3.12.3 Error Conditions

On processing the OMX_VIDEO_PARAM_VBSMCTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorNotImplemented` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

4.3.12.4 Post-processing Conditions

The variable size blocks used for motion compensation during video encoding on the port specified in `nPortIndex` are configured explicitly when set using `OMX_SetParameter`.

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.

- `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.
- `nVersion` is the version of the structure.

- `nPortIndex` is the read-only value containing the index of the port.
- `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 4-41 shows the possible H.263 video profile types for `OMX_VIDEO_H263PROFILETYPE`.

Field Name	Value	H.263 Profile Descriptions
`OMX_VIDEO_H263ProfileBaseline`	0x01	H.263 Baseline Profile: H.263 (V1), no optional modes
`OMX_VIDEO_H263ProfileH320Coding`	0x02	H.263 Coding Efficiency (H.320) Backward Compatibility Profile: H.263+ (V2), includes annexes I, J, L.4, and T
`OMX_VIDEO_H263ProfileBackwardCompatible`	0x04	H.263 BackwardCompatible: Backward Compatibility Profile: H.263 (V1), includes annex F
`OMX_VIDEO_H263ProfileISWV2`	0x08	H.263 Interactive Streaming Wireless Profile: H.263+ (V2), includes annexes I, J, K, and T
`OMX_VIDEO_H263ProfileISWV3`	0x10	H.263 Interactive Streaming Wireless Profile: H.263++ (V3), includes profile 3 and annexes V and W.6.3.8
`OMX_VIDEO_H263ProfileHighCompression`	0x20	H.263 Conversational High Compression Profile: H.263++ (V3), includes profiles 1 and 2 and annexes D and U
`OMX_VIDEO_H263ProfileInternet`	0x40	H.263 Conversational Internet Profile: H.263++ (V3), includes profile 5 and annex K
`OMX_VIDEO_H263ProfileInterlace`	0x80	H.263 Conversational Interlace Profile: H.263++ (V3), includes profile 5 and annex W.6.3.11
`OMX_VIDEO_H263ProfileHighLatency`	0x100	H.263 High Latency Profile: H.263++ (V3), includes profile 6 and annexes O.1 and P.5
`OMX_VIDEO_H263ProfileMax`	0x7FFFFFFF	Maximum value

Table 4-41. Supported H.263 Profile Types

- `eLevel` is the maximum processing level that an encoder or decoder supports for a particular profile. Table 4-42 shows the possible H.263 video level types.

Field Name	Value	H.263 Level Descriptions
OMX_VIDEO_H263Level10	0x01	H.263 level 10
OMX_VIDEO_H263Level20	0x02	H.263 level 20
OMX_VIDEO_H263Level30	0x04	H.263 level 30
OMX_VIDEO_H263Level40	0x08	H.263 level 40
OMX_VIDEO_H263Level50	0x10	H.263 level 50
OMX_VIDEO_H263Level60	0x20	H.263 level 60
OMX_VIDEO_H263Level70	0x40	H.263 level 70
OMX_VIDEO_H263LevelMax	0x7FFFFFFF	Maximum value

Table 4-42. Supported H.263 Level Types

- `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 4-36.
- `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.

4.3.13.2 Dependencies

The parameter may be queried at any time that the component is not in the NULL state. The parameter may only be set using `OMX_SetParameter` when the component is in the OMX_StateLoaded state after `OMX_VIDEO_PORTDEFINITIONTYPE` has been set for the port. This parameter is only applicable when the port is configured for H.263.

4.3.13.3 Error Conditions

On processing the OMX_VIDEO_PARAM_H263TYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorNotImplemented` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

4.3.13.4 Post-processing Conditions

The video encode or decode parameters for H.263 on the port specified by n`PortIndex` are configured explicitly when set using `OMX_SetParameter`.

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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `nPFrames` is the number of P frames between I frames.
- `nBFrames` is the number of B frames between I frames.
- `eProfileLevel` is the maximum processing level that an encoder or decoder supports for a particular profile. Table 4-43 shows the possible MPEG-2 video profile types in `OMX_VIDEO_MPEG2PROFILETYPE`.

Field Name	Value	MPEG-2 Profile Descriptions
OMX_VIDEO_MPEG2ProfileSimple	0x01	Simple profile
OMX_VIDEO_MPEG2ProfileMain	0x02	Main profile
OMX_VIDEO_MPEG2Profile422	0x04	4:2:2 profile
OMX_VIDEO_MPEG2ProfileSNR	0x08	SNR profile
OMX_VIDEO_MPEG2ProfileSpatial	0x10	Spatial profile
OMX_VIDEO_MPEG2ProfileHigh	0x20	High profile
OMX_VIDEO_MPEG2ProfileMax	0x7FFFFFFF	Maximum value

Table 4-43. Supported MPEG-2 Profile Types

- eLevel is the maximum processing level that an MPEG-2 encoder or decoder supports for a particular profile. Table 4-44 shows the possible MPEG-2 video level types in OMX_VIDEO_MPEG2LEVELTYPE.

Field Name	Value	MPEG-2 Level Descriptions
OMX_VIDEO_MPEG2LevelLL	0x01	Low level
OMX_VIDEO_MPEG2LevelML	0x02	Main level
OMX_VIDEO_MPEG2LevelH14	0x04	High 1440 level
OMX_VIDEO_MPEG2LevelHL	0x08	High level
OMX_VIDEO_MPEG2LevelMax	0x7FFFFFFF	Maximum level

Table 4-44. Supported MPEG-2 Level Types

4.3.14.2 Dependencies

The parameter may be queried at any time that the component is not in the NULL state. The parameter may only be set using OMX_SetParameter when the component is in the OMX_StateLoaded state after OMX_VIDEO_PORTDEFINITIONTYPE has been set for the port. This parameter is only applicable when the port is configured for MPEG-2.

4.3.14.3 Error Conditions

On processing the OMX_VIDEO_PARAM_MPEG2TYPE structure, the following error conditions can occur:

- OMX_ErrorBadParameter if one or more fields of the structure are incorrect.
- OMX_ErrorNotImplemented when the feature is unsupported.
- OMX_ErrorInvalidState when the OMX_SetParameter function is called and the component is in the OMX_StateInvalid state.
- 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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

4.3.14.4 Post-processing Conditions

The video encoding or decoding parameters for MPEG-2 on the port specified by `nPortIndex` are configured explicitly when set using `OMX_SetParameter`.

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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `nSliceHeaderSpacing` is the number of macroblocks in a slice (H263+ Annex K). Make this value 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 4-45 shows the possible MPEG-4 video profile types in `OMX_VIDEO_MPEG4PROFILETYPE`.

Field Name	Value	MPEG-4 Profile Descriptions
`OMX_VIDEO_MPEG4ProfileSimple`	0x01	MPEG-4 Simple Profile, Levels 1-3
`OMX_VIDEO_MPEG4ProfileSimpleScalable`	0x02	MPEG-4 Simple Scalable Profile, Levels 1-2
`OMX_VIDEO_MPEG4ProfileCore`	0x04	MPEG-4 Core Profile, Levels 1-2
`OMX_VIDEO_MPEG4ProfileMain`	0x08	MPEG-4 Main Profile, Levels 2-4
`OMX_VIDEO_MPEG4ProfileNbit`	0x10	MPEG-4 N-bit Profile, Level 2
`OMX_VIDEO_MPEG4ProfileScalableTexture`	0x20	MPEG-4 Scalable Texture Profile, Level 1
`OMX_VIDEO_MPEG4ProfileSimpleFace`	0x40	MPEG-4 Simple Face Animation Profile, Levels 1-2
`OMX_VIDEO_MPEG4ProfileSimpleFBA`	0x80	MPEG-4 Simple Face and Body Animation (FBA) Profile, , Levels 1-2
`OMX_VIDEO_MPEG4ProfileBasicAnimated`	0x100	MPEG-4 Basic Animated Texture Profile, Levels 1-2

OMX_VIDEO_MPEG4ProfileHybrid	0x200	MPEG-4 Hybrid Profile, Levels 1-2
OMX_VIDEO_MPEG4ProfileAdvancedRealTime	0x400	MPEG-4 Advanced Real Time Simple Profiles, Levels 1-4
OMX_VIDEO_MPEG4ProfileCoreScalable	0x800	MPEG-4 Core Scalable Profile, Levels 1-3
OMX_VIDEO_MPEG4ProfileAdvancedCoding	0x1000	MPEG-4 Advanced Coding Efficiency Profile, Levels 1-4
OMX_VIDEO_MPEG4ProfileAdvancedCore	0x2000	MPEG-4 Advanced Core Profile, Levels 1-2
OMX_VIDEO_MPEG4ProfileAdvancedScalable	0x4000	MPEG-4 Advanced Scalable Texture, Levels 2-3
OMX_VIDEO_MPEG4ProfileMax	0x7FFFFFFF	Maximum value

Table 4-45. Supported MPEG-4 Profile Types

- eLevel is the maximum processing level that an encoder or decoder supports for a particular MPEG-4 profile. Table 4-46 shows the possible MPEG-4 video level types in OMX_VIDEO_MPEG4LEVELTYPE.

Field Name	Value	MPEG-4 Level Descriptions
OMX_VIDEO_MPEG4Levell	0x01	Level 1
OMX_VIDEO_MPEG4Level2	0x02	Level 2
OMX_VIDEO_MPEG4Level3	0x04	Level 3
OMX_VIDEO_MPEG4Level4	0x08	Level 4
OMX_VIDEO_MPEG4LevelMax	0x7FFFFFFF	Max level

Table 4-46. Supported MPEG-4 Level Types

- nAllowedPictureTypes identifies the picture types allowed in the bit stream. For more information on picture types, see Table 4-36.
- nHeaderExtension specifies the number of consecutive video packet headers within a VOP.
- bReversibleVLC is a Boolean value that enables or disables the use of reversible variable-length coding

4.3.15.2 Dependencies

The parameter may be queried at any time that the component is not in the NULL state. The parameter may only be set using `OMX_SetParameter` when the component is in the OMX_StateLoaded state after `OMX_VIDEO_PORTDEFINITIONTYPE` has been set for the port. This parameter is only applicable when the port is configured for MPEG-4.

4.3.15.3 Error Conditions

On processing the OMX_VIDEO_PARAM_MPEG4TYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorNotImplemented` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

4.3.15.4 Post-processing Conditions

The video encoding or decoding parameters for MPEG-4 on the port specified by `nPortIndex` are configured explicitly when set using `OMX_SetParameter`.

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.

- `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.
- `nVersion` is the version of the structure.

- nPortIndex is the read-only value containing the index of the port.
- eFormat is the enumerated format of the data stream. Table 4-47 shows the possible Windows Media video format types for OMX_VIDEO_WMVFORMATTYPE.

Field Name	Value	Windows Media Video Format Descriptions
OMX_VIDEO_WMVFormatUnused	0x01	Format unused or unknown
OMX_VIDEO_WMVFormat7	0x02	Windows Media video format 7
OMX_VIDEO_WMVFormat8	0x04	Windows Media video format 8
OMX_VIDEO_WMVFormat9	0x08	Windows Media video format 9
OMX_VIDEO_WMVFormatMax	0x7FFFFFFF	Maximum level

Table 4-47. Supported Windows Media Video Format Types

4.3.16.2 Dependencies

The parameter may be queried at any time that the component is not in the NULL state. The parameter may only be set using OMX_SetParameter when the component is in the OMX_StateLoaded state after OMX_VIDEO_PORTDEFINITIONTYPE has been set for the port. This parameter is only applicable when the port is configured for Windows Media video.

4.3.16.3 Error Conditions

On processing the OMX_VIDEO_PARAM_WMVTYPE structure, the following error conditions can occur:

- OMX_ErrorBadParameter if one or more fields of the structure are incorrect.
- OMX_ErrorNotImplemented when the feature is unsupported.
- OMX_ErrorInvalidState when the OMX_SetParameter function is called and the component is in the OMX_StateInvalid state.
- 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_ErrorVersionMismatch when the nVersion field of the structure does not match the expected version for the component.

4.3.16.4 Post-processing Conditions

The video encoding or decoding parameters for Windows Media video on the port specified by nPortIndex are configured explicitly when set using OMX_SetParameter.

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_BOOL bEnablePostFilter;
    OMX_BOOL bEnableLatencyMode;
} OMX_VIDEO_PARAM_RVTYPE;
```

4.3.17.1 Parameters

The parameters for OMX_VIDEO_PARAM_RVTYPE are defined as follows.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `eFormat` is the video format. Table 4-48 shows the possible RealVideo video format types in `OMX_VIDEO_RVFORMATTYPE`.

Field Name	Value	RV Format Descriptions
`OMX_VIDEO_RVFormatUnused`	0x01	Format unused or unknown
`OMX_VIDEO_RVFormat8`	0x02	RealVideo 8 format
`OMX_VIDEO_RVFormat9`	0x04	RealVideo 9 format
`OMX_VIDEO_RVFormatMax`	0x7FFFFFFF	Maximum level

Table 4-48. Supported RealVideo Format Types

- `bEnablePostFilter` is a Boolean value that enables or disables the post filter.
- `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

The parameter may be queried at any time that the component is not in the NULL state. The parameter may only be set using `OMX_SetParameter` when the component is in the OMX_StateLoaded state after `OMX_VIDEO_PORTDEFINITIONTYPE` has been set for the port. This parameter is only applicable when the port is configured for RealVideo.

4.3.17.3 Error Conditions

On processing the OMX_VIDEO_PARAM_RVTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorNotImplemented` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

4.3.17.4 Post-processing Conditions

The video encoding or decoding parameters for RealVideo on the port specified by `nPortIndex` are configured explicitly when set using `OMX_SetParameter`.

4.3.18 OMX_VIDEO_PARAM_AVCTYPE

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.

- `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.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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.

- `nRefIdxTrailing` 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.
- `nRefIdxForward` 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.
- `eProfileLevel` is the profile used for the types of AVC encoding or decoding that are supported. Table 4-49 shows the possible AVC video profile types in `OMX_VIDEO_AVCPROFILETYPE`.

Field Name	Value	AVC Profile Descriptions
`OMX_VIDEO_AVCProfileBaseline`	0x01	Baseline profile
`OMX_VIDEO_AVCProfileMain`	0x02	Main profile
`OMX_VIDEO_AVCProfileExtended`	0x04	Extended profile
`OMX_VIDEO_AVCProfileHigh`	0x08	High profile
`OMX_VIDEO_AVCProfileHigh10`	0x10	High 10 profile
`OMX_VIDEO_AVCProfileHigh422`	0x20	High 4:2:2 profile
`OMX_VIDEO_AVCProfileHigh444`	0x40	High 4:4:4 profile
`OMX_VIDEO_AVCProfileMax`	0x7FFFFFFF	Maximum value

Table 4-49. Supported AVC Profile Types

- `eLevel` is the maximum processing level that an AVC encoder or decoder supports for a particular profile. Table 4-50 shows the possible AVC video level types in `OMX_VIDEO_AVCLEVELTYPE`.

Field Name	Value	AVC Level Descriptions
`OMX_VIDEO_AVCLevel1`	0x01	AVC level 1
`OMX_VIDEO_AVCLevel1b`	0x02	AVC level 1b
`OMX_VIDEO_AVCLevel11`	0x04	AVC level 1.1
`OMX_VIDEO_AVCLevel12`	0x08	AVC level 1.2
`OMX_VIDEO_AVCLevel13`	0x10	AVC level 1.3
`OMX_VIDEO_AVCLevel2`	0x20	AVC level 2
`OMX_VIDEO_AVCLevel21`	0x40	AVC level 2.1
`OMX_VIDEO_AVCLevel22`	0x80	AVC level 2.2

OMX_VIDEO_AVCLevel3	0x100	AVC level 3
OMX_VIDEO_AVCLevel31	0x200	AVC level 3.1
OMX_VIDEO_AVCLevel32	0x400	AVC level 3.2
OMX_VIDEO_AVCLevel4	0x800	AVC level 4
OMX_VIDEO_AVCLevel41	0x1000	AVC level 14.1
OMX_VIDEO_AVCLevel42	0x2000	AVC level 4.2
OMX_VIDEO_AVCLevel5	0x4000	AVC level 5
OMX_VIDEO_AVCLevel51	0x8000	AVC level 5.1
OMX_VIDEO_AVCLevelMax	0x7FFFFFFF	Maximum value

Table 4-50. Supported AVC Level Types

- 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 is a Boolean value that enables or disables specification of the method used in the derivation process for luma motion vectors for B_Skip, B_Direct_16x16, and B_Direct_8x8 as specified in sub-clause 8.4.1.2 of the AVC specification.
- 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 4-51 shows the possible AVC video coding loop filter types in OMX_VIDEO_AVCLOOPFILTERTYPE.

Field Name	Value	AVC Loop Filter Level Descriptions
OMX_VIDEO_AVCLoopFilterEnable	0x01	Enables AVC loop filter
OMX_VIDEO_AVCLoopFilterDisable	0x02	Disables AVC loop filter
OMX_VIDEO_AVCLoopFilterDisableSliceBoundary	0x04	Disables AVC loop filter on slice boundary
OMX_VIDEO_AVCLevelMax	0x7FFFFFFF	Maximum level

Table 4-51. Supported AVC Loop Filter Types

4.3.18.2 Dependencies

The parameter may be queried at any time that the component is not in the NULL state. The parameter may only be set using `OMX_SetParameter` when the component is in the OMX_StateLoaded state after `OMX_VIDEO_PORTDEFINITIONTYPE` has been set for the port. This parameter is only applicable when the port is configured for AVC.

4.3.18.3 Error Conditions

On processing the OMX_VIDEO_PARAM_AVCTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorNotImplemented` when the feature is unsupported.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.

4.3.18.4 Post-processing Conditions

The video encoding or decoding parameters for AVC on the port specified by `nPortIndex` are configured explicitly when set using `OMX_SetParameter`.

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 Parameter and Configuration Indices

The header OMX_Index.h contains the enumeration `OMX_INDEXTYPE`, which contains all standard index values used with the OpenMAX IL version 1.0 core functions `OMX_GetParameter`, `OMX_SetParameter`, `OMX_GetConfig`, and `OMX_SetConfig`. Table 4-52 shows the index values that relate to imaging.

OpenMAX IL Indices (OMX_Index.h)	Corresponding OpenMAX IL Image Structures (OMX_Image.h)
OMX_IndexParamImagePortFormat	OMX_IMAGE_PARAM_PORTFORMATYPE
OMX_IndexParamImagePort	OMX_IMAGE_PORTDEFINITIONTYPE
OMX_IndexParamImageInit	OMX_IMAGE_PARAM_TYPE
OMX_IndexParamFlashControl	OMX_IMAGE_PARAM_FLASHCONTROLTYPE
OMX_IndexConfigFocusControl	OMX_IMAGE_CONFIG_FOCUSCONTROLTYPE
OMX_IndexParamQFactor	OMX_IMAGE_PARAM_QFACTORTYPE
OMX_IndexParamQuantizationTable	OMX_IMAGE_PARAM_QUANTIZATIONTABLETYPE
OMX_IndexParamHuffmanTable	OMX_IMAGE_PARAM_HUFFMANTTABLETYPE

Table 4-52. Image Indices

For example, `OMX_IndexParamImagePortFormat` index is used with `OMX_GetParameter` and `OMX_SetParameter` to access `OMX_IMAGE_PARAM_PORTFORMATYPE`.

4.4.2 Image Use Case Example

Figure 4-5 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.

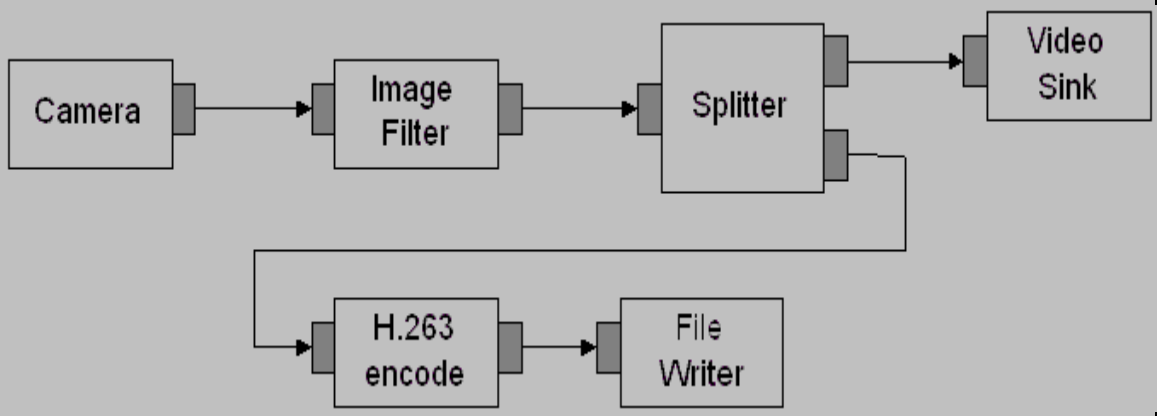


Figure 4-5. Image Filtering and JPEG Encoding Use Case

Figure 4-5 shows six components, namely the camera, the image filter, the splitter, the JPEG encoder, the file writer, and the image sink.

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 PortDefinition structure defines all of the parameters necessary for the compliant component to set up an input or an output image path. If additional vendor specific data is required, it should be transmitted to the component using the `CustomCommand` function. Compliant components will pre-populate this structure with optimal values during the `OMX_GetParameter()` command.

OMX_IMAGE_PORTDEFINITIONTYPE is defined as follows.

```
typedef struct OMX_IMAGE_PORTDEFINITIONTYPE {
    OMX_STRING cMIMEType;
    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_IMAGE_PORTDEFINITIONTYPE;
```

4.4.3.1 Parameters

The parameters for OMX_IMAGE_PORTDEFINITIONTYPE are defined as follows.

- `cMIMEType` is the multipurpose Internet mail extensions (MIME) type of data on the port.
- `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 read-only 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.
- `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 component supports error concealment. This flag is returned by a component upon invoking `OMX_GetParameters`; it is ignored on `OMX_SetParameter` calls.
- `bFlagErrorConcealment` enables error concealment if it is supported for the port.
- `eCompressionFormat` is the enumeration describing the compression format used on the port. When `OMX_IMAGE_CodingUnused` is specified, the `eColorFormat` field is valid. Table 4-53 shows the supported image compression formats.

Field Name	Value	Compression Format Description	Reference to Standard
`OMX_IMAGE_CodingUnused`	0x0	No coding applied, use `eColorFormat`	Not available
`OMX_IMAGE_CodingAutoDetect`	0x1	Auto detection by the OpenMAX component	Not available
`OMX_IMAGE_CodingJPEG`	0x2	JPEG/JFIF image format	JPEG
`OMX_IMAGE_CodingJPEG2K`	0x3	JPEG 2000 image format	JPEG2K
`OMX_IMAGE_CodingEXIF`	0x4	EXIF image format	EXIF
`OMX_IMAGE_CodingTIFF`	0x5	TIFF image format	TIFF
`OMX_IMAGE_CodingGIF`	0x6	Graphics image format	GIF
`OMX_IMAGE_CodingPNG`	0x7	PNG image format	PNG
`OMX_IMAGE_CodingLZW`	0x8	LZW image format	LZW
`OMX_IMAGE_CodingBMP`	0x9	Windows Bitmap format	BMP
`OMX_IMAGE_CodingMax`	0x7FFFFFFF	Maximum value	Not available

Table 4-53. Supported Image Compression Formats

- `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`.

4.4.3.2 Dependencies

The structure may be queried at any time that the component is not in the NULL state. The structure may only be set using `OMX_SetParameter` when the component is in the OMX_StateLoaded state.

4.4.3.3 Error Conditions

On processing the OMX_IMAGE_PORTDEFINITIONTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when `OMX_GetParameter` function is called and the value of `nPortIndex` exceeds the number of ports for the component.

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.

- `nSize` is the size of the structure in bytes.
- `nVersion` is the version of the structure.

- `nPortIndex` is the read-only value containing the index of the port.
- `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 4-35.
- `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.

4.4.4.2 Dependencies

The structure may be queried at any time that the component is not in the NULL state. The structure may be set using `OMX_SetParameter` when the component is in the OMX_StateLoaded state.

4.4.4.3 Error Conditions

On processing the OMX_IMAGE_PARAM_PORTFORMATTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when OMX_GetParameter function is called and the value of `nPortIndex` exceeds the number of ports for the component.

4.4.4.4 Post-processing Conditions

The OMX_IMAGE_PARAM_PORTFORMATTYPE structure has no post-processing conditions.

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.

- `nSize` is the size of the structure in bytes.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `eFlashControl` is an enumeration for the flash control modes. Table 4-54 shows the supported image flash controls.

Field Name	Value	Flash Control Description
`OMX_IMAGE_FlashControlOn`	0x0	Strobe at every shot
`OMX_IMAGE_FlashControlOff`	0x1	Strobe off
`OMX_IMAGE_FlashControlAuto`	0x2	Strobe according to environment light
`OMX_IMAGE_FlashControlRedEyeReduction`	0x3	Pre-shot strobes
`OMX_IMAGE_FlashControlFillin`	0x4	Flash for background/ foreground effect
`OMX_IMAGE_FlashControlTorch`	0x5	Flash is always on
`OMX_IMAGE_FlashControlMax`	0x7FFFFFFF	Maximum value

Table 4-54. Supported Image Flash Controls

4.4.5.2 Dependencies

The OMX_IMAGE_PARAM_FLASHCONTROLTYPE structure may be queried at any time that the component is not in the NULL state. The structure may only be set using `OMX_SetParameter` when the component is in the OMX_StateLoaded state.

4.4.5.3 Error Conditions

On processing the OMX_IMAGE_PARAM_FLASHCONTROLTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when OMX_GetParameter function is called and the value of `nPortIndex` exceeds the number of ports for the component.

4.4.5.4 Post-processing Conditions

The OMX_IMAGE_PARAM_FLASHCONTROLTYPE structure has no post-processing conditions.

4.4.6 OMX_IMAGE_PARAM_FOCUSCONTROLTYPE

OMX_IMAGE_PARAM_FOCUSCONTROLTYPE controls the focus mode and range.

OMX_IMAGE_PARAM_FOCUSCONTROLTYPE is defined as follows.

```
typedef struct OMX_IMAGE_PARAM_FOCUSCONTROLTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_IMAGE_FOCUSCONTROLTYPE eFocusControl;
    OMX_U32 nFocusSteps;
    OMX_U32 nFocusStepIndex;
} OMX_IMAGE_PARAM_FOCUSCONTROLTYPE;
```

4.4.6.1 Parameters

The parameters for OMX_IMAGE_PARAM_FOCUSCONTROLTYPE are defined as follows.

- `nSize` is the size of the structure in bytes.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `eFocusControl` is an enumeration that specifies the image focus controls. Table 4-55 shows the supported image focus controls.

•

Field Name	Value	Focus Control Description
OMX_IMAGE_FocusControlOn	0x0	Manual focus on
OMX_IMAGE_FocusControlOff	0x1	Manual focus off
OMX_IMAGE_FocusControlAuto	0x2	Auto focus on
OMX_IMAGE_FocusControlAutoLock	0x3	Auto focus lock
OMX_IMAGE_FocusControlCentroid	0x4	Focus on region
OMX_IMAGE_FocusControlMax	0x7FFFFFFF	Maximum value

Table 4-55. Supported Image Focus Controls

- `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.

4.4.6.2 Dependencies

The OMX_IMAGE_PARAM_FOCUSCONTROLTYPE structure may be queried at any time that the component is not in the NULL state. The structure may be set using `OMX_SetParameter` when the component is in the OMX_StateLoaded state.

4.4.6.3 Error conditions

On processing the OMX_IMAGE_PARAM_FOCUSCONTROLTYPE structure, the following error conditions can occur:

- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when OMX_GetParameter function is called and the value of `nPortIndex` exceeds the number of ports for the component.

4.4.6.4 Post-processing Conditions

The OMX_IMAGE_PARAM_FOCUSCONTROLTYPE structure has no post-processing conditions.

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.

- `nSize` is the size of the structure in bytes.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `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.

4.4.7.2 Dependencies

The OMX_IMAGE_PARAM_QFACTORTYPE structure may be queried at any time that the component is not in the NULL state. The structure may only be set using `OMX_SetParameter` when the component is in the OMX_StateLoaded state.

4.4.7.3 Error Conditions

On processing the OMX_IMAGE_PARAM_QFACTORTYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when OMX_GetParameter function is called and the value of `nPortIndex` exceeds the number of ports for the component.

4.4.7.4 Post-processing Conditions

The OMX_IMAGE_PARAM_QFACTORTYPE structure has no post-processing conditions.

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_QUANTIZATIONTABLETTYPE {
    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.

- `nSize` is the size of the structure in bytes.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `eQuantizationTable` is an enumeration for the quantization table type, which defines luma or chroma table types. Table 4-56 shows the supported image quantization table types.

Field Name	Value	Quantization Table Description
`OMX_IMAGE_QuantizationTableLuma`	0x0	Quantize luma coefficients
`OMX_IMAGE_QuantizationTableChroma`	0x1	Quantize chroma coefficients
`OMX_IMAGE_QuantizationTableMax`	0x7FFFFFFF	Max value

Table 4-56. Supported Image Quantization Table Types

- `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 Dependencies

The OMX_IMAGE_PARAM_QUANTIZATIONTABLETYPE structure may be queried at any time that the component is not in the NULL state. The structure may only be set using `OMX_SetParameter` when the component is in the OMX_StateLoaded state.

4.4.8.3 Error Conditions

On processing the OMX_IMAGE_PARAM_QUANTIZATIONTABLETYPE structure, the following error conditions can occur:

- `OMX_ErrorBadParameter` if one or more fields of the structure are incorrect.
- `OMX_ErrorInvalidState` when the `OMX_SetParameter` function is called and the component is in the OMX_StateInvalid state.
- `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_ErrorVersionMismatch` when the `nVersion` field of the structure does not match the expected version for the component.
- `OMX_ErrorNoMore` when OMX_GetParameter function is called and the value of `nPortIndex` exceeds the number of ports for the component.

4.4.8.4 Post-processing Conditions

The OMX_IMAGE_PARAM_QUANTIZATIONTABLETYPE structure has no post-processing conditions.

4.4.9 OMX_IMAGE_PARAM_HUFFMANTABLETYPE

The OMX_IMAGE_PARAM_HUFFMANTABLETYPE structure is used to set the Huffman variable code length type used for JPEG.

OMX_IMAGE_PARAM_HUFFMANTABLETYPE 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_HUFFMANTABLETYPE are defined as follows.

- `nSize` is the size of the structure in bytes.
- `nVersion` is the version of the structure.
- `nPortIndex` is the read-only value containing the index of the port.
- `eHuffmanTable` is an enumeration for the Huffman table types. The same Huffman table is applied for chroma and luma components. Table 4-57 shows the supported Huffman table types.

Field Name	Value	Huffman Table Description
OMX_IMAGE_HuffmanTableAC	0x0	Huffman encode AC coefficients
OMX_IMAGE_HuffmanTableDC	0x1	Huffman encode DC coefficients
OMX_IMAGE_HuffmanTableMax	0x7FFFFFFF	Maximum value

Table 4-57. Supported Huffman Table Types

- 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 Dependencies

The OMX_IMAGE_PARAM_HUFFMANTABLETYPE structure may be queried at any time that the component is not in the NULL state. The structure may only be set using OMX_SetParameter when the component is in the OMX_StateLoaded state.

4.4.9.3 Error Conditions

On processing the OMX_IMAGE_PARAM_HUFFMANTABLETYPE structure, the following error conditions can occur:

- OMX_ErrorBadParameter if one or more fields of the structure are incorrect.
- OMX_ErrorInvalidState when the OMX_SetParameter function is called and the component is in the OMX_StateInvalid state.
- 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_ErrorVersionMismatch when the nVersion field of the structure does not match the expected version for the component.
- OMX_ErrorNoMore when OMX_GetParameter function is called and the value of nPortIndex exceeds the number of ports for the component.

4.4.9.4 Post-processing Conditions

The OMX_IMAGE_PARAM_HUFFMANTABLETYPE structure has no post-processing conditions.

5 OpenMAX Component Extension APIs

5.1 Description of the Extension Process

An OpenMAX component may support any setting defined in the OpenMAX specification. Vendors can add to the list of parameters and configurations not included in the official header files. These additions are referred to as *extensions*.

Any extensions approved by OpenMAX are considered OpenMAX extensions. Any extensions not approved by OpenMAX are vendor-defined extensions.

Any vendor that develops OpenMAX components may add to the list of standard indexes a collection of one or more custom parameters or configuration indexes. Each index shall have a value greater than the value of `OMX_IndexIndexVendorStartUnused` and less than the value of `OMX_IndexMax` - 1.

Each custom parameter or configuration index may apply to one of the four existing domains, namely audio, video, image, and "other". It may also apply a parameter or configuration that does not belong to any known domain. For example, file access could be a domain, where parameter and configuration index values operate on a file port.

A vendor-specific 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.

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 vendor extension section of the component documentation.

5.1.1 GetExtensionIndex

The `OMX_GetExtensionIndex` method will translate a vendor-specific configuration or parameter string into an OpenMAX structure index. There is no requirement for the component to support this command for the indexes already found in the OMX_INDEXTYPE enumeration, thus reducing a component's memory footprint. The component may support all vendor-supplied extension indexes not found in the master OMX_INDEXTYPE enumeration that it supports. This is a blocking call. The component should return from this call within five msec.

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	The string that the component will translate into a 32-bit index.

[in]	`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

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 custom header file. The header file is to be used with the component that implements the vendor-specific feature.

If the data container is simply a pointer to a memory area, the IL client shall know how to manage the data. A use guide for vendor-specific parameters shall be published in the component description document.

Each custom 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.

5.2 Examples of Using Extension Querying API

This section shows sample code for extension APIs.

5.2.1 Sample Code Showing Calling Sequence

The following sample code shows an example of calling an extension API.

```
/* Set the vendor-specific filename parameter
   on a reader */
OMX_U32 eIndexParamFilename;
OMX_PTR oFileName;

OMX_GetExtensionIndex(
      hFileReaderComp,
      "OMX.CompanyXYZ.index.param.filename",
      &eIndexParamFilename);
OMX_SetParameter(hComp, eIndexParamFilename, &oFileName);
```

This following code sample shows how to use a vendor-specific parameter. The code passes a file name to a component. The file name string does not belong to any OpenMAX domain; it used only for this example.

```
/* Get the vendor-specific mp3 faster
   decoding feature settings */
OMX_U32 eIndexParamFasterDecomp;
OMX_CUSTOM_AUDIO_STRUCTURE oFasterDecompParams;

OMX_GetExtensionIndex(
      hMp3DecoderComp,
      "OMX.CompanyXYZ.index.param.fasterdecomp",
      &eIndexParamFasterDecomp);
OMX_GetParameter(hMp3DecoderComp, eIndexParamFasterDecomp,
      &oFasterDecompParams);
```

In this second 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 OpenMAX Generic Components

This section specifies a set of components, including features, names, and operations, that is standardized by the group for increased cross-platform application and codec portability.

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.

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.

Table 6-1. Seek Modes

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. 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.
2. Seek to the desired location through the use of `OMX_SetConfig` on `OMX_IndexConfigTimePosition` requesting the desired timestamp.
3. Flush all components.
4. 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.

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 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 one audio and one video 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.

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.

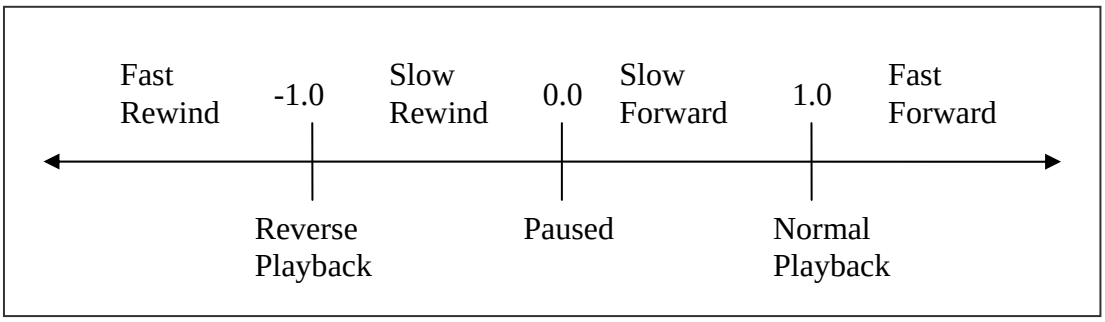


Figure 6-1. Mapping Time Scale Factors to Trick Modes

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_U32 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_WaitingOnStartTime, 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.

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:

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.

Table 6-2. Clock State Values

An `OMX_GetConfig` execution on OMX_TIME_CONFIG_CLOCKSTATETYPE queries the current clock state.

An `OMX_SetConfig` execution on 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.

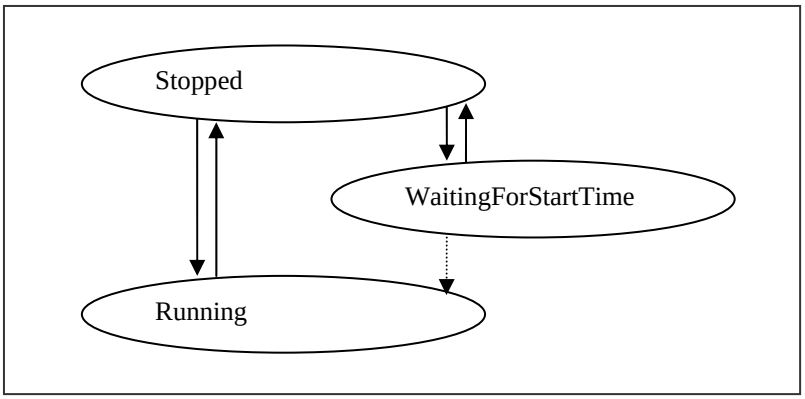


Figure 6-2. Clock State Transitions

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_ClockRunning: 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_WaitingOnStartTime: 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_ClockRunning. 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

The clock component can accept both a video and an audio reference clock, supplied respectively by a video component and an audio component. 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_IndexConfigTimeCurrentAudioReference and OMX_IndexConfigTimeCurrentVideoReference and passing the following structure:

```
typedef struct OMX_TIME_CONFIG_TIMESTAMPTYPE {
            OMX_U32 nSize;
            OMX_VERSIONTYPE nVersion;
            OMX_TICKS nReferenceTimestamp;
} OMX_TIME_CONFIG_TIMESTAMPTYPE;
```

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 component that is providing the reference clock.

The IL client controls which reference clock the clock component uses (if any) via the OMX_IndexConfigTimeActiveRefClock configuration and the following structure:

```
typedef struct OMX_TIME_CONFIG_ACTIVEREFCLOCKTYPE {
            OMX_U32 nSize;
            OMX_VERSIONTYPE nVersion;
            OMX_TIME_REFCLOCKTYPE eClock;
} OMX_TIME_CONFIG_ACTIVEREFCLOCKTYPE;
```

Possible `eClock` values include those shown in Table 6-3:

OMX_TIME_REFCLOCKTYPE Value	**Interpretation**
`OMX_TIME_RefClockNone`	Not using a reference clock
`OMX_TIME_RefClockAudio`	Using audio reference clock.
`OMX_TIME_RefClockVideo`	Using video reference clock

Table 6-3. Reference Clock Values

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.

6.2.4.1 Media Time Updates

A clock component sends a client a media time update, as either the fulfillment of a request or a scale change notification, over its clock port via the following structure:

```
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;
```

`eUpdateType` indicates the reason for the update and as one of the values shown in Table 6-4:

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.

Table 6-4. Update Type

The nClientPrivate field contains the client private pointer specified by the request that generated this media time update if this update is a request fulfillment, or NULL otherwise.

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.

6.2.4.2 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).

6.2.4.3 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.

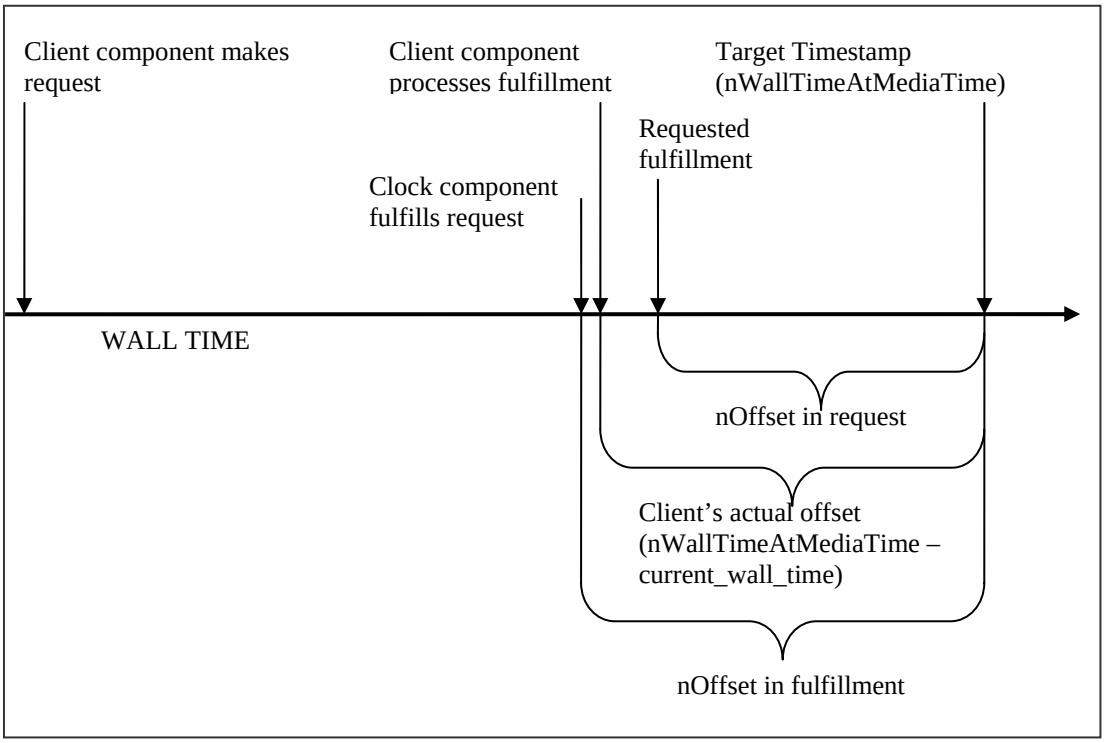


Figure 6-3. Timeline for Request and Fulfillment of Media Time Update

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.

6.2.4.4 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.

6.2.4.5 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 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.

6.2.5.1 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}. If an IL client specified an offset when it started the clock component (e.g., to implement a pre-roll), then the clock component adds this offset as W_{now} + Offset. The clock component stores the ultimate reference/wall time pair, representing the base of extrapolation, for later use as $<R_{base}, W_{base}>$ where:

$R_{\text{base}} = R_{\text{now}}$

$W_{\text{base}} = W_{\text{now}} + \text{Offset}$

The clock component calculates the instantaneous media time, M_{now}, by querying the wall clock, W_{now}, and extrapolating from the last reference, modulated by the current scale, *Scale*, as follows:

$M_{now} = R_{base} + Scale * (W_{now} - W_{base})$

6.2.5.2 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}:

$R_{\text{base}} = M_{\text{now}}$

$W_{\text{base}} = W_{\text{now}}$

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*

6.2.5.3 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 6-5 should be used according to 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

Table 6-5. Media Time Request Scale

6.2.6 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 6-4. 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 a pair of renderers, which deliver the data to hardware.

The renderers coordinate with the clock component to implement smooth synchronized audio-video delivery. The renderers and audio decoder are clients of the clock component (connected on their respective clock ports) so they may watch for scale changes. The video renderer also uses the clock component to time delivery of video frames via media time requests.

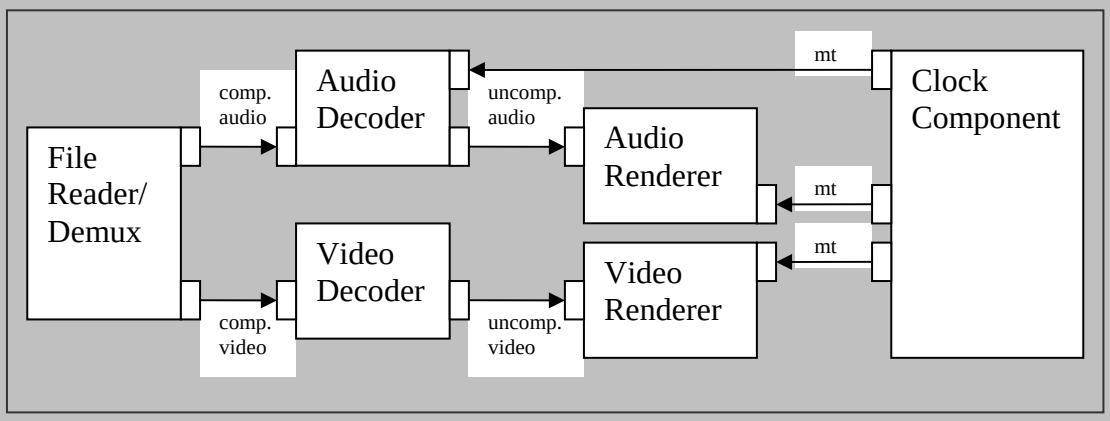


Figure 6-4. Example Use Case of Audio-Video File Playback

The audio and video renderers 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 renderer, however, shall coordinate with the clock component when delivering video frames. For each frame that the renderer shall deliver at a particular timestamp, the following occurs:

1. The renderer 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 renderer that references the original timestamp and includes the private pointer.

3. The renderer receives the media time update, de-references the private pointer to obtain the frame data, and delivers the frame. The renderer 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 audio decoder 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 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.

7.1 SPEECH

7.1.1 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.

7.1.2 3GPP2

SMV	3GPP2-SMV, “Selectable Mode Vocoder (SMV) Service Option for Wideband Spread Spectrum Communication Systems”, 3GPP2 C.S0030-0, 2004.

7.1.3 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.

7.1.4 ITU

G.711	ITU-G.723.1, “Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s”, 1996.
G.723.1	ITU-G.726, “40, 32, 24, 16 kbit/s adaptive differential pulse code modulation (ADPCM)”, 1990.

G.726	ITU-G.729, “Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear-prediction (CS-ACELP)”', 1996.
G.729	ITU-G711, “Pulse code modulation (PCM) of voice frequencies “, 1988.

7.1.5 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.

7.1.6 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.

7.2 AUDIO

7.2.1 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.

7.2.2 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

7.3 SYNTHETIC AUDIO

7.3.1 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.

7.4 IMAGE

7.4.1 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.

7.4.2 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.

7.4.3 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.

7.4.4 JEITA

EXIF JEITA, Japanese Electronics and Information Technology Industries Association, “EXIF (Exchangeable Image File Format) 2.2”, 2002.

7.4.5 MIPI

CSI MIPI Camera WG, "CSI 2.0 Protocol Specification v.0.41", 2005.

DSI MIPI Display WG, "DSI Specification v.0.45", 2005.

7.4.6 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".

7.4.7 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."

7.4.8 W3C

PNG — Portable Network Graphics (PNG) Specification (Second Edition), "Computer graphics and image processing, Portable Network Graphics (PNG): Functional specification," 2003.

7.5 VIDEO

7.5.1 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.

7.5.2 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

7.5.3 DLNA

HNv1.0 — DLNA HNv1.0, "Home Networked Device Interoperability Guidelines v1.0," 2004.

7.5.4 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.

7.5.5 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.

7.5.6 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.

7.5.7 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.

7.5.8 MISC

RV Real Video 10 Codec

WMV Windows Media Video

7.6 JAVA

7.6.1 Multimedia

JSR-135 JCP/JSR-135: Mobile Media API 1.0, 2003

JSR-234 JCP/JSR-234: Advanced Multimedia Supplements, 2005

7.6.2 Broadcast

JSR-272 JCP/JSR-272: Mobile Broadcast Service API for Handheld Terminals, 2005