DM814x OpenMax Components

User's Guide

OMX Components Version 05.02.00.xx

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards ought to be provided by the customer so as to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations and notices. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is neither responsible nor liable for any such use.

Resale of TI's products or services with *statements different from or beyond the parameters* stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products. www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Preface

Read This First

About This Document

This User's Guide serves as a software programmer's handbook for working with the DM814x OpenMax HDVPSS and HDVICP2 components. It provides the necessary information regarding how to effectively use the DM814x OpenMax HDVPSS Sub-system in customer systems and applications. It also provides details regarding the functionality and API interface of the components of the DM814x OpenMax HDVPSS Sub-system.

TI's implementation of OpenMax is based on the Khronos OpenMax Standard v1.1.2. It may be subsequently upgraded to newer versions of the standard as they are defined.

This document is intended for use by multimedia system integrators who wish to build advanced end-products using TI's DM814x SoC. It assumes that the reader is fluent in the C language, has a good working knowledge of multimedia frame-works and TI's multimedia SoC architectures (DM family). A prior knowledge of the Khronos OpenMax Standard will be very helpful.

How to Use This Document

This document includes the following chapters:

- Chapter 1 – Introduction – provides an overview of DM814x and the advantages of using the DM814x OpenMax Sub-system on this SoC. This section also includes information on the internal architecture of the software and its partitioning across the DM814x SoC.
- Chapter 2 - OpenMax Components – describes the components implemented by TI and their features. Also describes the data structures and interfaces to the OpenMax components of the DM814x OpenMax HDVPSS Sub-system.
- Chapter 3– API Reference – describes the data structures and interfaces to the OpenMax components of the DM814x OpenMax Sub-system

Note that the DM814x OpenMax Sub-system is being developed through a phased implementation approach. Certain features described in this document may not have been completely implemented yet. Important points on the status of the implementation is mentioned in the text of this document and marked in blue.

Abbreviations

Table of Abbreviations

Abbreviation	Description
CCSv5	Code Composer Studio Version 5
CIF	Common Intermediate Format
DVO1/2	Digital Video Output 1 & 2
EVM	Evaluation Module
HD	High Definition
HDMI	High Definition Multimedia Interface
HDVPSS	High Definition Video Processing Sub System
OMX	OpenMax
M2M	Memory to Memory
SD	Standard Definition
SOC	System on Chip
VFCC	Video Frame Capture Component
VFDC	Video Frame Display Component
VFPC	Video Frame Processing Component
VPDMA	Video Port Direct Memory Access
ADEC	Audio Decode Component
AENC	Audio Encode Component

Information about Cautions and Warnings

This book may contain cautions and warnings.

This is an example of a caution statement.

A caution statement describes a situation that could potentially damage your software or equipment.

CAUTION

This is an example of a warning statement.

A warning statement describes a situation that could potentially cause harm to you.

WARNING

The information in a caution or a warning is provided for your protection. Please read each caution and warning carefully.

Related Documentation

- See References section for related documentation.

Revision History

Version	Date	Revision History
0.1	August 09，2011	First Version

Trademarks

Code Composer Studio™ is a Trademark of Texas Instruments Incorporated

DSP/BIOS™ is a Trademark of Texas Instruments Incorporated

eXpressDSP™ is a Trademark of Texas Instruments Incorporated

TMS470™ is a Trademark of Texas Instruments Incorporated

All other trademarks are the property of the respective owner.

Contents

Chapter 1

Introduction

This chapter introduces the **DM814x OpenMax Sub-system**, including its usage, its internal architecture and partitioning across the DM814x SoC.

DM814x

DM814x is a highly integrated video SoC targeted at networked, high-definition (HD) video products – including surveillance video recorders, video conferencing and set-top boxes.

The DM814x HW includes a Cortex A8 Host processor, a graphics processing unit (GPU), a c674x DSP and a host of peripherals in addition to a dedicated video sub-system. Video subsystem (media controller) manage one specialized compression co-processor (HDVICP 2.0) and a dedicated HD video processing sub-system (HDVPSS).

The host processor runs a high level OS such as Linux. HDVICP2.0 represents the 2nd generation of TI's HD Video compression architecture. It is a compression engine that can handle several video formats including H.264, MPEG4, MPEG2, VC1, and RV and is scalable to different resolutions and frame-rates. HDVPSS handles video capture, display and HW accelerated video processing functions such as noise filtering, de-interlacing, scaling, and color processing. The c674x DSP typically runs audio functions and customer-specific algorithms such as Video Analytics, Face detection, pre and post processing algorithms and other video codecs that cannot be accelerated on the HDVICP 2.0.

Why OpenMax

A complex HW architecture such as DM814X demands that the SW be carefully designed to ensure high levels of performance and efficiency while simultaneously being easy to use and understand. With an efficient SW architecture, TI's customers can swiftly optimize TI-provided SW subsystems for their specific applications. DM814x is a highly optimized multi-core media processor with critical real-time software partitioned across multiple processors: Cortex A8, DSP, media controller in addition to the acceleration engines (HDVPSS and HD-VICP2). The OpenMax implementation simplifies this multi-processor architecture such that customers can develop their applications entirely on the Host A8 processor using standard OpenMax APIs without digging into the complexities of inter-processor communication, synchronization, and system partitioning. [Note: The implementation provided with this document is limited in features and intended for customers to understand the OpenMax call flow. The example IL client and application provided with this release working on the Cortex A8 and makes API calls to OpenMax components implemented on the media controller.]

OpenMax is based on the idea of 'active components'. Active components are data processing modules that can be connected to each other through standard, configurable data pipes allowing a continuous stream of media data to be processed from source to destination with very little intervention from the application. TI has adopted OpenMax APIs for media components. OpenMax has been popularized by the industry group Khronos.

TI's implementation of OpenMax is componentized. Each component is implemented to support the OpenMax Standard Non-Tunnel (SNT) design. The SNT design gives the application layer access to pre and post processed buffers going to and from the components. In addition, the APIs have been provided and designed to offer customers several choice of design parameters that trade-off end-product care-abouts such as latency, channel density and video quality. A further advantage of OpenMax is the ability to integrate the TI software seamlessly with open source media frameworks such as those based on GStreamer, Maemo, and Android.

OpenMax Overview

The Khronos OpenMax Working Group has defined a set of standard, open Application Programming Interfaces (APIs) for multimedia applications. OpenMax is a common specification for middleware applications such as codecs and media processing. The goal is to ensure that new products can be brought to market sooner and are easily portable across HW platforms.

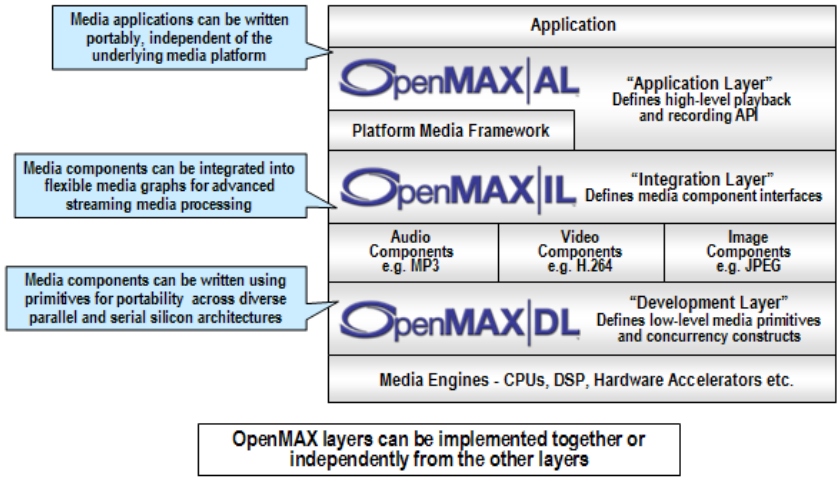


Figure: Three layers of OpenMax architecture (from http://www.khronos.org)

The OpenMax APIs are standardized at 3 layers:

1 OpenMax AL (Application Layer): A standardized interface between an application and multimedia middleware, where multimedia middleware provides the services needed to perform expected API functionality. To allow customers flexibility in implementing their applications, OpenMax AL is not mandated by TI. Customers are free to use their own application layers.

2 OpenMax IL (Integration Layer): A low-level interface for multimedia codecs and processing algorithms. It gives applications the ability to interface with compression modules (codecs), processing modules, sources, and sinks in a simplified manner. The modules themselves may be combination of software and HW accelerators (such as HDVPSS and HD-VICP) and are completely transparent to the user. IL provides system abstraction for components and implementation abstraction for applications. This is the ideal layer that allows customers flexibility with optimal system performance. OpenMax IL is implemented in the TI sub-system and the subject of this document.

3 OpenMax DL (Development Layer): An API which contains a comprehensive set of audio, video and imaging functions (such as FFTs and filters, color space conversion and video processing primitives) to enable the optimized implementation of codecs. To ensure optimal performance of codec implementations, TI's codecs are not based on OpenMax DL.

For additional details on OpenMax refer to [1].

OpenMax Integration Layer (IL)

The OpenMax IL API encapsulates each media processing module in a component interface. The standard was designed for codecs, sources (such as capture) and sinks (such as display). However it is extendible to general media processing functions also. 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 end-product media use case and mesh with existing graph-based media frameworks. The codec, source, sink or processing module may itself be any combination of hardware or software and are completely transparent to the user. The implementation of the component may also span processors (as in the case of DM814x). Much of the OpenMax IL API is defined by requirements of media frameworks. IL is designed to allow applications and media frameworks to be lightweight. The design is primarily designed to handle media. For example, handling file systems, networks, encryption etc are not part of an OpenMax IL subsystem although such extensions are possible outside of the standard.

The communication between components and between the application and components are designed to be asynchronous allowing multi-threaded/ multi-processor / HW accelerated implementations. Also the components are allowed to communicate directly with one another via any efficient 'proprietary tunneling' method leading to greater flexibility and efficiency.

The OpenMax IL API consists of two main segments: the core API and the component API. The OpenMax IL core is used for dynamically loading and unloading components and for facilitating component communication. Once loaded, the user communicates directly with the component using Component APIs. The core allows a user to establish communication tunnels between components (once set up, data flows between components without the involvement of the core). The components may each be a source, a sink, a codec or processing module, or a splitter or mixer. A component has parameters that can be set or queried. The parameters could control behavior (ex: codec bit-rate) or the actual execution state

of the component. Components can callback the application to return status, errors etc. Components have interfaces called ports to pass data to other components (through tunnels) or to the application.

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, component states and also to send commands to the component, receive event notifications, allocate buffers, establish communications with a single component port, and create tunnels between two component ports. OpenMax allows audio, video, and image data port as well as other ports (ex: meta-data).

Components and IL clients may communicate using either (i) Non-tunneled communications (between the IL client and a component) (ii) standard tunneling (standard mechanism for components to exchange data buffers directly with each other in a standard way) or (iii) Proprietary communication (non-standard direct data communications between two components). [Note: The implementation provided with this document allows only standard non tunneled communication between components].

This product is designed to support OpenMax base profile. Note that OpenMax compliance test suite has not been executed for the components.

For further details on OpenMax IL, refer to [2].

The OpenMax IL client is the key interface between the application layer on one hand and OpenMax components and the OpenMax core on the other. It controls the behavior and states of the OpenMax components. To understand the dynamic behavior of the OpenMax IL sub-system, it is important to understand the state machine of the OpenMax components.

OpenMax Component State Machine

Each OpenMax component can undergo a series of state transitions, as depicted in Figure 1-2. The job of moving the component from one state to another is the IL Client's. Every component is first considered to be UNLOADED. The component shall move to the LOADED state through a call to the OpenMax core. All other state transitions may then be achieved by communicating directly with the component.

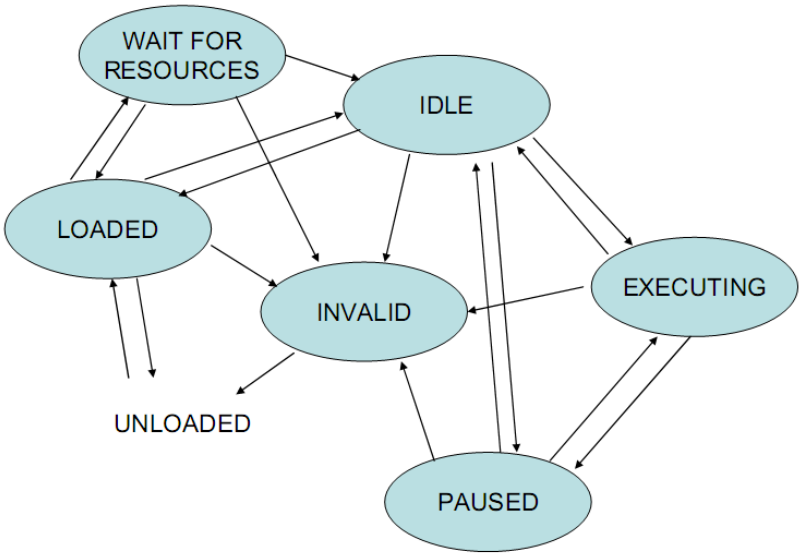


Figure: OpenMax component state diagram

Next the component will attempt to acquire all the resources it requires and transition from LOADED to IDLE State. If this state transition fails due to non-availability of resources, the client may try again or may choose to put the component into the WAIT FOR RESOURCES state. In the WAIT FOR RESOURCE state, for example, a semaphore set when the resource becomes available, may then allow the component to transition to IDLE state. From here, the component moves into the EXECUTING state indicating that the component is pending reception of buffers to process data and will make required callbacks (specified later) [Note: In the current implementation, all the resources are assumed to be available and the component moves directly from LOADED to IDLE state, i.e. It does not implement WAIT FOR RESOURCES]. The component may then be moved into a PAUSED state to maintain a context of buffer execution with the component without processing data or exchanging buffers [Note: PAUSED state is not implemented in TI's current implementation]. Buffer processing will resume when the component moves back from PAUSED to EXECUTING state. To stop the component, it must be transitioning from EXECUTING or PAUSED to IDLE – this will lead the buffers that were being processed to be lost.

Note that a component can enter an invalid state when a state transition is made with invalid data. The only way to exit the invalid state is to unload and reload the component. Note: INVALID state is not implemented in TI's current implementation. Behavior of the system for incorrect / corrupt data is not guaranteed in this version].

OpenMax Component Architecture

Figure 1-3 depicts the component architecture. The OpenMax component has a single handle into its data structure and array of functions. It makes multiple possible outgoing calls depending on the number of ports it has. It may call an IL Client's event handler. It may also make callbacks to a specified external function. Each port is also associated with a handle to a queue of pointers to buffer headers. The buffer headers point to the actual buffers. In Non tunneled architecture, buffers are allocated by component upon request by IL client through OMX_AllocateBuffer() APIs. Typically output port of component allocates the buffers and input port of connected component uses that buffer. All parameter or configuration calls are performed on a particular index and include a structure associated with that parameter or configuration.

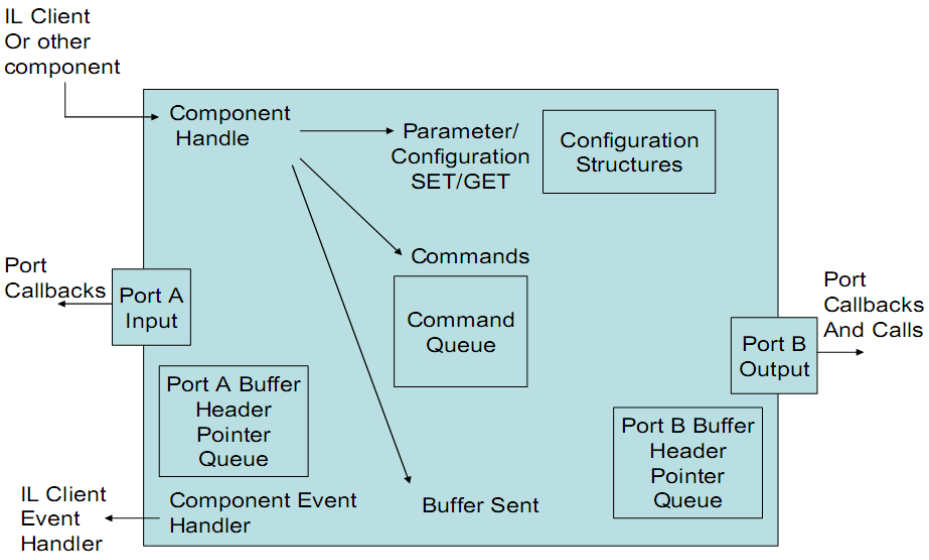


Figure: OpenMax Component Architecture

OpenMax component classes

The DM814x OpenMax Sub-system includes the following classes of OpenMax components. The user must note a key distinction between 'class of components' and 'individual OpenMax components'. Each individual component is a separate OpenMax component with a unique name from the viewpoint of the IL client. A class of components is a collection of components implemented in an identical manner in the TI implementation.

1. Video Frame Capture Component (VFCC) Class – is the class of components that manages Video Capture from an external source such as a camera. It is implemented on the media controller. The actual component name invoked will be different depending on the nature of the capture – Examples: Multiplexed video (a feature of DM814x), and Non-multiplexed video will be separate components.

2. Video Frame Display Component (VFDC) – Video display sink. Implemented on the media controller. The DM814x allows up to three displays. Each display is an independent instance of the VFDC class of components. . [In this release, VFDC component has been integrated and tested with 2 HD displays only].

3. Video Frame Processing Component (VFPC) – Video frame processing class of components. Implemented on the media controller. Features such as scalar, deinterlacers and Noise filter are individual components within the VFPC class of components.

4. CTRL component (CTRL) - The control components are the ones that has no omx input/output ports. This is not a data processing component. The job of the CTRL is to configure & start the control functionalities such as external video decoder (TVP, SIL etc) and display controller configurations on the DM814x.

5. Video Decode Component (VDEC) – This component is responsible for video bit stream decoding. This component works on one frame per buffer basis. Currently It supports H264 and MPEG2 decode.

6. Video Encode Component (VENC) - This component is responsible for video bit stream encoding. This component works on one frame per buffer basis. Currently It supports only H264 encode.

7. Audio Decode Component (ADEC) – This component is responsible for audio bit stream decoding. This component works on one frame per buffer basis. This component supports only single buffer per input/output port. Currently it supports MP3 and AAC-LC decode.

8. Audio Encode Component (AENC) – This component is responsible for audio encoding. This component works on one frame per buffer basis. This component supports only single buffer per input/output port. Currently it supports AAC-LC encode.

The DM814x provides multiple graphics display ports, each of which may be flexibly alpha-blended with video. The graphics ports are each accessible through the HLOS APIs – i.e. Linux Frame Buffer device (FBDev) in TI's software implementation. The OpenMax APIs for a specific display's OpenMax VFDC component allow the user to specify the actual graphics plane that a specific display may be blended with.

Chapter 2

OpenMax Components

This section provided a description of the OpenMax components available on the DM814x OpenMax Multimedia Sub-system.

General Information about OMX components

Each OpenMax component implemented by TI has a unique name that also specifies its location in the hierarchy of OpenMax components.

- Video Frame Display Component
- Video Frame Capture Component
- Video Frame Processing Componnet
- Video Control Componnet
- Video Encode Component
- Video Decode Component
- Audio Decode Component
- Audio Encode Component

Following OMX APIs are used to create and configure the OpenMax components.

- **Creating the component**

API : OMX_GetHandle – create the component instance. Component name is unique for each component, which is used to create the component. Component handle returned in this call will be used for all other OpenMax APIs.

Example : `OMX_GetHandle(&pAppData->pDisHandle, "OMX.TI.VPSSM3.VFDC",`

`pAppData->disILComp, &pAppData->pCb);`

- **Configuring the parameters**

API : OMX_SetParameter

Example : `OMX_SetParameter(pAppData->pDisHandle, (OMX_INDEXTYPE) OMX_TI_IndexParamVFDCDriverInstId, &driverId);`

For Configuring different parameter, different index types can be used and associated structures can be passed in above API. Component handle is used to distinguish between different components.

2.1 Video Frame Display Component (VFDC)

OMX Component Name : OMX.TI.VPSSM3.VFDC

Video frame display component takes the input buffer from the memory and displays that buffer on the external device like TV . The VFDC component is a sink component that has input ports but no output ports. The job of the VFDC is to display the frames provided to it.

OMX component implementation on media controller and APIs has been available on A8/Linux. There is one instance of the display component for each display attached to the system. DM814x allows up to 3 displays and hence allows the IL client to instantiate up to 3 VFDC components (each identified by a unique instance ID).

After displaying a frame, the buffers corresponding to that frame are freed up. The VFDC runs on the media controller using one active thread. This thread is normally sleeping and activated on the posting of an event in the callback function invoked by the display driver. When activated, the component de-queue any buffers returned by the driver and queues a new set of buffers from the input port into the display driver. In case there is no new frame available to queue (ex: when frames are generated slower than the display rate), it queues the last displayed buffer (still owned by the OMX component) again for display. It frees frames that are already displayed back into the corresponding input data pipe.

DM814x supports 3 display components simultaneously:

- Two HD display components (OMX_VIDEO_DISPLAY_ID_HD0 and OMX_VIDEO_DISPLAY_ID_HD1). The input ports can only be of data type YUV422P. The maximum resolution supported in these ports is 1080p60. These components are not available in this release.
- One SD display (OMX_VIDEO_DISPLAY_ID_SD0) (not validated in the current release)
- VFDC support different standard resolution like 1080p60, 1080i60, 720p60 etc for HD VENC and SD resolution for SD VENC

Different paths within VFDC could be configured using display controller. Once the display operation is started, the VFDC always retains the last buffer and displays the same buffer continuously till the application gives a new buffer to display.

Features Supported

- YUV422 interleaved format
- Interlaced and progressive displays
- Resolution up to 1080P@60FPS display on HD VENC D_DVO1/DVO2 through Bypass paths
- Field merged interlaced buffer mode

Limitations

- VFDC supports only one handle per instance. This means that a specific driver could be opened only once.
- Support queuing mechanism. Application may queue multiple buffers and VFDC displays the buffers one after another sequentially in order the buffers are queued.

- Once the display operation is started, the display driver always retains the last buffer and displays the same buffer continuously till the application gives a new buffer to display.
- Application should stop display operation before it could dequeue the last buffer from the component.
- Before the display operation is started, the application has to queue a minimum set of buffers. This operation is called priming.
- The minimum of number buffers required could defer from driver to driver. Generally this is equal to 1 buffer and the recommended value is equal to 3 buffers

Configuration

Control interface provides the necessary means of controlling mechanism for an individual component. This interface provides a set of functions to manage the Display component instances, like creation, deletion etc, function to connect the Display component to specific data interface, functions to control & configure and functions to query and receive different types of statistics, diagnostics and state related information.

All these control mechanism to the Display component is through OpenMax core sending control messages to Display component. On processing the message display component may respond with the acknowledgements with the result of processing.

These configuration parameters are categorized into 3 different classifications:

1) OMX Core Parameters – These are OMX mandatory base class parameters that must be initialized prior to requesting the component instance handle.

   For setting basic parameters in OMX, following API / Index is used. It uses standard OMX structure for defining the parameters.

```
OMX_SetParameter(pAppData->pDisHandle,
OMX_IndexParamPortDefinition,                    &paramPort);
```

Following are the common parameters frequently changed:

Field Name	Description	Values
format.video.nFrameWidth	Number of columns for display	Integer value of range 16 to 1920
format.video.nFrameHeight	Number of lines for display	Integer value of range 16 to 1080
format.video.eColorFormat	Format of input buffer	OMX_COLOR_FormatYC

		bYCr
nBufferCountActual	Number of input buffers	Integer value Min = 4, Max 32
format.video.nStride	Pitch of video data	It should be twice of Width, as only supported format is YUV 422.
inbufsize	Size of the input buffer	Stride * Height

2) VFDC Core Parameters – These are VFDC mosaic parameters, which upon instantiation are initialized to default parameters. Typically these parameter values are overridden according to the required application use case.

OMX Index: **OMX_TI_IndexParamVFDCCreateMosaicLayout**

Config Structure : OMX_PARAM_VFDC_CREATEMOSAICLAYOUT:

. Following are the parameters for this index, and values prescribed.

Field Name	Description	Values
nLayoutId	Mosaic layout ID	0
nNumWindows	Number of windows in mosaic	1
nDisChannelNum	Display channel	0
sMosaicWinFmt .winStartX	Horizontal start	0 - 1920
sMosaicWinFmt .winStartY	Vertical start	0 - 1080
sMosaicWinFmt .winWidth	Width in pixels	0 - 1920
sMosaicWinFmt .winHeight	Number of lines in a window	0 - 1080
sMosaicWinFmt .pitch.VFDC_YUV_INT_ADDR_	Pitch in bytes for each of the sub-window buffers	Width * 2
sMosaicWinFmt .dataFormat	Data format for window	VFDC_DF_YUV422I_YUYV
sMosaicWinFmt .bpp	Bits per pixels for each window	VFDC_BPP_BITS16
sMosaicWinFmt	Window priority in case of	0

.priority	overlapping windows	

3) VFDC Index Parameters – These are VFDC specific parameters, which are configured according to the required application use case.

OMX Index : **OMX_TI_IndexParamVFDCDriverInstId**

Decription: set/select the disply path driver ID as well as the VENC display mode.

Field Name	Description	Values
nDrvInstID	The physical output port for a VFDC instance. This decides display is on HDMI or DVO2 (LCD) or SD.	0 = VPS_DISP_INST_BP0 1 = VPS_DISP_INST_BP1 2 = VPS_DISP_INST_SEC1
eDispVencMode	The video output format	OMX_DC_MODE_1080P_60 OMX_DC_MODE_720P_60 OMX_DC_MODE_1080I_60 OMX_DC_MODE_1080P_30

Default properties :

Field Name	Description	Values
nDrvInst	Driver Instance ID	VPS_DISP_INST_BP0, VPS_DISP_INST_BP1, VPS_DISP_INST_SEC1
eBufAllocPref	Buffer allocator preferences on this port	OMX_BASE_BUFFER_ALLOCATION_PREFERENCE_DEFAULT
bReadOnlyBuffers	buffers on this port will be read only	OMX_TRUE
nWaterMark	Watermark level on each port	1
eBufMemoryType	Buffer memory type	OMX_BUFFER_MEMORY_DEFAULT
hBufHeapPerPort		
eDataAccessMode	Whether frame mode or stream mode data	OMX_BASE_DATA_ACCESS_MODE_FRAME
bDMAAccessedBuffer	Whether to access data buffers via DMA.	OMX_FALSE

pMetaDataFieldTypesArr		NULL

2.2 Video Frame Capture Component (VFCC)

Video frame capture component takes the input from the external sources such as cameras, DVD players, TVP etc and puts the capture images in the memory. VFCC is an optimized media controller capture component for DM814X, which can address the broad market/specific customer requirements. The VFCC component is a source component that has omx output ports but no input ports. The job of the VFCC is to capture the frames provided to it through the video input ports (VIP) on the DM814x.

The DM814x contains 2 VIP ports – VIP1 and VIP2 that supports max. 24-bit interfaces. Each of the ports can be re-configured as two 8 bit VIP ports – in that case, these ports are named VIP1A, VIP1B, VIP2A and VIP2B. The SW model is to normally create instance of the VFCC component for each VIP PORT. Since DM814x allows up to 4 VIP ports, the IL client can instantiate up to 4 VFCC components with configuration to use specific VIP port.

The capture component is implemented as follows. A periodic timer wakes up a timer interrupt service routine that checks all video ports for available captured buffers and issues a call back function that posts an event. The posted event triggers the active processing thread of the VFCC. When activated, the component thread de-queue the captured buffers and subsequently queues a new set of buffers from the output port into the capture driver. The frequency of the periodic timer is configurable. Also, the calling back option of the driver is configurable and can be either unconditional or on data availability.

DM814x also supports non-muxed video. In this case, each video port (In the case of 8b configuration: VIP1A, VIP1B, VIP2A, VIP2B or in the case of 16/24b configuration: VIP1 and VIP2 capture the video from a single camera input. Note that it is possible for VIP1 to be configured as 8b – i.e. split into VIP1A and VIP1B while VIP2 remains a single 16/24b port. Likewise it is possible for VIP2 to be configured as 8b – i.e. split into VIP2A and VIP2B while VIP1 remains a single 16/24b port.

OMX component implementation on media controller and APIs has been available on A8/Linux

Features supported

- **Input Video Source Formats**
  - § YUV422 8-bit embedded sync mode
  - § YUV422 16-bit embedded sync mode
  - § YUV422 8-bit 2x/4x pixel multiplxed mode
  - § YUV422 8-bit 4x line multiplexed mode

- **Output Video formats**
  - § YUV422 YUYV interleaved format
  - § YUV420 Semi-planer format

- **In-line video processing features**
  - § Chroma-down sampling
- **Other features**
  - § multi-port capture on VIP0 & VIP1 (Port A, Port B), with ability to configure.
  - § Interlaced as well as progressive capture
  - § Non-multiplexed capture upto 1080P60 (1920x1080) resolution Multi-channel - upto 16CH D1 (NTSC/PAL) using 4 VIP ports (VIP0/A, VIP0/B, VIP1/A, VIP1/B)
  - § Per frame info to user like - field ID, captured frame width x height, timestamp, logical channel ID
  - § For YUV input, optional chroma downsampling is supported

§ Per channel frame-dropping. Example, for a 60fps video source, 30fps, 15fps, 7fps capture

Input to Output Combinations support

Input Format	Output format
YUV422 8/16-bit embedded sync mode	YUV422 YUYV interleaved format (optionally scaled)
	YUV420 Semi-planer format (optionally scaled)
YUV422 8/16-bit embedded sync mode - MULTI-CH modes - pixel mux, line mux	YUV422 YUYV interleaved format (SCALING, CHR_DS, CSC NOT SUPPORTED in MULTI-CH modes)

Configuration

Control interface provides the necessary means of controlling mechanism for an individual component. This interface provides a set of functions to manage the Capture component instances, like creation, deletion etc, function to connect the Capture component to specific data interface, functions to control & configure and functions to query and receive different types of statistics, diagnostics and state related information.

All these control mechanism to the Capture component is through OpenMAX core sending control messages to capture module. On processing the message capture module may respond with the acknowledgements with the result of processing. VFCC configuration parameters are categorized into 2 different classifications:

1. OMX Core Parameters – These are OMX mandatory base class parameters that must be initialized prior to requesting the component instance handle.

Index – `OMX_IndexParamPortDefinition`

Following fields can be modified for above index.

Field Name	Description	Values
format.video.nFrameWidth	Number of columns for display	Integer value of range 16 to 1920
format.video.nFrameHeight	Number of lines for display	Integer value of range 16 to 1080
format.video.eColorFormat	Format of output buffer	OMX_COLOR_FormatYUV420SemiPlanar OMX_COLOR_FormatYCbYCr
nBufferCountActual	Number of input buffers	Integer value

		Min = 5, Max 32
format.video.nStride	Pitch of video data	For 420 capture – Width For 422 capture - 2* Width
nBufferSize	Size of the input buffer	For 420 -width * Height * 3/2 For 422 – Stride * Height

2. VFCC Index Parameters - These are VFCC class specific parameters, which are configured according to the required application use case. These parameters are accessed using a index and field_name combination.

   OMX Index : OMX_TI_IndexParamVFCCHwPortID

Field Name	Description	Values
eHwPortId	The Video Input Interface from where capture image/video is obtained.	OMX_VIDEO_CaptureHWPortVIP1_PORTA OMX_VIDEO_CaptureHWPortVIP2_PORTA

OMX Index : **OMX_TI_IndexParamVFCCHwPortProperties**

Field Name	Description	Values
eCaptMode	Multiplex Mode	OMX_VIDEO_CaptureModeSC_NON_MUX OMX_VIDEO_CaptureModeMC_LINE_MUX
eVifMode	Multiplex Mode	OMX_VIDEO_CaptureVifMode_08BIT OMX_VIDEO_CaptureVifMode_16BIT OMX_VIDEO_CaptureVifMode_24BIT
eInColorFormat		OMX_COLOR_FormatYCbYCr
eScanType	Video scan type	OMX_VIDEO_CaptureScanTypeProgressive
nMaxWidth	MAX Capture width	1920
nMaxHeight	Max capture height	1080
nMaxChnlsP	Max num	1

erHwPort	channels/handle	

Default properties :

Field Name	Description	Values
nDrvInst	Driver Instance ID	VPS_CAPT_INST_VIP_ALL
eBufAllocPref	Buffer allocator preferences on this port	OMX_BASE_BUFFER_ALLOCATION_PREFERENCE_DEFAULT
bReadOnlyBuffers	buffers on this port will be read only	OMX_TRUE
nWaterMark	Watermark level on each port	1
eBufMemoryType	Buffer memory type	OMX_BUFFER_MEMORY_DEFAULT
hBufHeapPerPort		
eDataAccessMode	Whether frame mode or stream mode data	OMX_BASE_DATA_ACCESS_MODE_FRAME
bDMAAccessedBuffer	Whether to access data buffers via DMA.	OMX_FALSE
pMetaDataFieldTypesArr		NULL
nNumMetaDataFields		0

2.3 Video Frame Processing Component (VFPC)

Video frame processing component takes the input from the memory processes the input like scale the image, chroma up samples the image and puts it back to the memory. The VFPC components are generic components has both input and output OMX ports. Various memory to memory operations such as noise filtering, scaling, deinterlacing etc operations are performed by this component. VFPC has the capability to generate single or multiple outputs. VFPC is an active component and the data processing thread wakes up due to either data availability at the input side or due to the periodic wake ups. On getting callbacks from the driver, the component de-queue all the buffers from the driver and frees the input buffers.

- § OMX component implementation on media controller and APIs has been available on A8/Linux
- § VFPC could be opened multiple times – supports multiple handles (N) for the same component - Each handle can have different configuration
- § VFPC supports queuing of input request from multiple sources.
- § Also, either on data notification or periodically the component provides the next set of buffers to the HW for processing

The following VFPC components are released in the package

2.3.1 VFPC-Scalar (SC5):

Features supported for VFPC-SC5

- § Scaling up to maximum 1920 pixels in horizontal direction & 1080 pixels in the vertical direction
- § Downscaling up to 1/8x.
- § Supports only a fixed standard set of coefficients for the scalar.
- § Chroma up sampling from YUV420 semiplanar to YUYV422 interleaved format.
- § Supports horizontal and vertical cropping of the image before scaling.
- § Supports different types of scalar like poly phase and running average.
- § Multi-channel support – up to 16 channels per handle
- § Support dynamic resolution change on both input and output side

Limitation

- § Interlaced image at input or output not supported.
- § Application loading coefficients is not supported
- § No configuration support for SC algorithms such as poly-phase or running average (Supported only Fixed configuration)

§ Pitch of input and output buffer should be multiple of 16.

Default properties:

Field Name	Description	Values
nDrvInst	Driver Instance ID	VPS_M2M_INST_SEC0_SC5_WB2
pDriverProperties->nNumCurInputBufPerProcess	Number of input buffers required by the driver in each process call.	OMX_VFPC_INDTXSCWB_NUM_CURRENT_INPUT_FRAMES_PER_PROCESS (1)
pDriverProperties->nNumHistoryBufPerProcess	Number of input buffers required by the driver in each process call. This number of history	OMX_VFPC_INDTXSCWB_NUM_HISTORY_FRAMES_PER_PROCESS (0)
pDriverProperties->nNumOutputPerProcess	Number of output buffers required by the driver in each process call	OMX_VFPC_INDTXSCWB_NUM_OUTPUT_FRAMES_PER_PROCESS (1)
pDriverProperties->auOutputSubSampleFactor[0]	Indicates default subsampling info that is to applied . Subsampling may be 1/2, 1/4 etc	OMX_VFPC_INDTXSCWB_SUBSAMPLEFACTOR_OUTPUT0 (1)
pDriverProperties->sInBufProp[0].eColorFormat	The Color format of the I/O buf	OMX_COLOR_FormatYUV420SemiPlanar
pDriverProperties->sInBufProp[0].nMaxFrameHeight	Max frame height of the I/O buf	OMX_VFPC_DEFAULT_INPUT_FRAME_HEIGHT (1080)
pDriverProperties->sInBufProp[0].nMaxFrameWidth	Max frame width of the I/O buf	OMX_VFPC_DEFAULT_INPUT_FRAME_WIDTH (1920)
pDriverProperties->sInBufProp[0].eBufferMemType	The MemType of the I/O buf - Tled Vs Non-Tiled	OMX_BUFFER_MEMORY_TILED_8BIT
pDriverProperties->sOutBufProp[0].eColorFormat	The Color format of the I/O buf	OMX_COLOR_FormatYCbYCr
pDriverProperties->sOutBufProp[0].nMaxFrameHeight	Max frame height of the I/O buf	OMX_VFPC_DEFAULT_OUTPUT_FRAME_HEIGHT (1080)
pDriverProperties->sOutBufProp[0].nMaxFrameWidth	Max frame width of the I/O buf	OMX_VFPC_DEFAULT_OUTPUT_FRAME_WIDTH (1920)
pDriverProperties->sOutBufProp[0].eBufferMemType	The MemType of the I/O buf - Tled Vs Non-Tiled	OMX_BUFFER_MEMORY_DEFAULT

2.3.2 VFPC-Noise Filter (NF)

Features Supported

§ Input formats: YUV422, non-tiled memory, YUYV interleaved data - this is the only input format supported by the NSF hardware

§ Output formats: YUV420T, tiled memory, YUV420 semi-planer data - this is the only output format supported by the NSF hardware

- Tiler support for YUV420 output, YUV420 previous filtered input
- Multi-channel support – up to 16 channels per handle
- Configurable input size (width, height, startX, startY, pitch) per channel
- Output size is always same as input size
- Configurable noise filter processing parameters like filter strength, filter threshold per channel
- The NF hardware supports spatial as well as temporal noise filtering.
- Configurable noise filter operation mode per channel like temporal NF bypass, spatial NF bypass, all NF bypass, i.e. only chroma downsample
- When temporal noise filtering is enabled, the hardware needs the previous noise filtered output as one of the inputs.
- When temporal noise filter is disabled (OMX_NSF_BYPASS_MODE_SNF_TNF), this previous noise filtered frame is not required. It is possible to bypass both spatial as well as temporal noise filter
- (OMX_NSF_BYPASS_MODE_SNF_TNF), i.e. NF can be used for only YUV422 to YUV420 chroma down-sampling. In this case too, previous noise filtered frame is not required

Limitation

- Slice based NF is not supported.
- Width/pitch/height should be multiple of 32 pixels.

Default values :

Field Name	Description	Values
nDrvInst	Driver Instance ID	VPS_M2M_INST_NF0
pDriverProperties->nNumCurInputBufPerProcess	Number of input buffers required by the driver in each process call.	OMX_VFPC_NF_NUM_CUR_INPUTBUFS (1)
pDriverProperties->nNumHistoryBufPerProcess	Number of input buffers required by the driver in each process call. This number of history	OMX_VFPC_NF_HIS_INPUTBUFS (1)
pDriverProperties->sHistoryProp[0].nFrameListIndex	The index into the frameList that will be used as history buffer	0
pDriverProperties->sHistoryProp[0].eFrameListType	The enum identifying if nFrameListIndex refers to the inFrameList or the outFrameList	OMX_VFPC_FRAMELIST_TYPE_OUTPUT
pDriverProperties->nNumOutputPerProcess	Number of output buffers required by the driver in each process call	OMX_VFPC_NF_NUM_OUTBUFS (1)
pDriverProperties->auOutputSubSampleFactor[0]	Indicates default subsampling info that is to applied .	OMX_VFPC_NF_SUBSAMPLEFACTOR_OUTPUT0 (1)

	Subsampling may be 1/2, 1/4 etc	
pDriverProperties->sInBufProp[0].eColorFormat	The Color format of the I/O buf	OMX_COLOR_FormatYCbYCr
pDriverProperties->sInBufProp[0].nMaxFrameHeight	Max frame height of the I/O buf	OMX_VFPC_DEFAULT_INPUT_FRAME_HEIGHT (1080)
pDriverProperties->sInBufProp[0].nMaxFrameWidth	Max frame width of the I/O buf	OMX_VFPC_DEFAULT_INPUT_FRAME_WIDTH (1920)
pDriverProperties->sInBufProp[0].eBufferMemType	The MemType of the I/O buf - Tled Vs Non-Tiled	OMX_BUFFER_MEMORY_DEFAULT
pDriverProperties->sOutBufProp[0].nMaxFrameHeight	Max frame height of the I/O buf	OMX_VFPC_DEFAULT_INPUT_FRAME_HEIGHT (1080)
pDriverProperties->sOutBufProp[0].nMaxFrameWidth	Max frame width of the I/O buf	OMX_VFPC_DEFAULT_INPUT_FRAME_WIDTH (1920)
pDriverProperties->sOutBufProp[0].eColorFormat	The Color format of the I/O buf	OMX_COLOR_FormatYUV420SemiPlanar
pDriverProperties->sOutBufProp[0].eBufferMemType	The MemType of the I/O buf - Tled Vs Non-Tiled	OMX_BUFFER_MEMORY_TILED_8BIT

§

2.3.3 VFPC-DEI MQ Dual Out

VFPC DEI Dual Output Paths: As shown in below figures, the VFPC DEI takes in YUYV422/YUV420 interlaced/progressive input via the DEI path and provides two scaled version of the deinterlaced/bypassed outputs - one via writeback path 0 and another via VIP 0.

Features Supported

- **Input Formats**
  - § YUV422 Interleaved
  - § YUV420 Semi-Planar
- **Output Formats**
  - § YUV422 Interleaved on WB0/1
  - § YUV422 Interleaved on VIP0/1
  - § YUV420 Semi-Planar on VIP0/1
  - § YUV420 Semi-Planar Tiled on VIP0/1
- **Other Features**
  - § DEI in deinterlacing mode
  - § DEI in progressive bypass mode
  - § Line averaging and field averaging mode of DEI operation
  - § Optional scaling using SC1 and SC2.
  - § Scaling up to 1920 maximum pixels in horizontal direction
  - § Different types of scalar like poly phase and running average
  - § Horizontal and vertical cropping of the image before scaling

- § Enable/disable of DRN
- § Frame drop feature on WB0/1 and VIP0/1 outputs to enable load balancing
- § Multi-channel (up to 8 channels per instance)

- **Runtime Cofiguration**
  - § Output resolution change on WB0
  - § Output resolution change on VIP0

Configuration

Control interface provides the necessary means of controlling mechanism for an individual component. This interface provides a set of functions to manage the Scalar component instances, like creation, deletion etc, function to connect the Scalar component to specific data interface, functions to control & configure and functions to query and receive different types of statistics, diagnostics and state related information.

All these control mechanism to the VFPC class component is through OpenMAX core sending control messages to scalar module. On processing the message frame processing module may respond with the acknowledgements with the result of processing. VFPC component(s) implement the OpenMAX APIS for these control messages such as,

Configuration parameters are categorized into 2 different classifications:

1. OMX Core Parameters – These are OMX mandatory base class parameters that must be initialized prior to requesting the component instance handle.

   OMX Index : **OMX_IndexParamPortDefinition**

Field Name	**Description**	**Values**
format.video.nFrameWidth	Number of columns display	Integer value of range 16 to 1920
format.video.nFrameHeight	Number of lines for displa	Integer value of range 16 to 1080
format.video.eColorFormat	Format of output buffer	OMX_COLOR_FormatYUV SemiPlanar OMX_COLOR_FormatYCb
nBufferCountActual	Number of input buffers	Integer value Min = 5
format.video.nStride	Pitch of video data	For 420– Width For 422 - 2* Width
nBufferSize	Size of the input buffer	For 420 -width * Height * 3/2 For 422 – Stride * Height

2. VFPC Index Parameters - These are VFPC class specific parameters, which are configured according to the required application use case. These parameters are accessed using a index and field_name combination.

OMX Index : **OMX_TI_IndexParamVFPCNumChPerHandle**

OMX_PARAM_VFPC_NUMCHANNELPERHANDLE: Enumerates the number of channels the VFPC-SC component processes

Field Name	**Description**	**Values**
nNumChannelsPerHandle	Number of channels to processed	1 (All VFPC class compon support only 1 channel instance)

Dynamic configuration

API: OMX_SetConfig
OMX Index **: OMX_TI_IndexConfigVidChResolution**
Description: API to Configure the resolution related parameters of a specific channel (port).
Assumption - Any driver channel of VFPC module will have a max of two input ports and a max of two output ports

Field Name	**Description**	**Values**
eDir	Indicates which end of channel the configura values apply to.	OMX_DirInput OMX_DirOutput
nChId	Indicates the channel ir for which this configura values apply.	0 – All VFPC compon support 1 channel per insta
Frm0Width	Width of the frame at the port corresponding to channel	Integer value of range 16 to 1920
Frm0Height	Height of the frame at first port corresponding to channel.	Integer value of range 16 to 1920
Frm0Pitch	Height of the frame at first port corresponding to channel.	For 420– Width For 422 - 2* Width
Frm1Width	Width of the frame at second port correspondir	Integer value of range

	the channel	16 to 1920 (Curre supported for output port VFPC-DEIM components)
Frm1Height	Height of the frame at second port correspondir the channel.	Integer value of range 16 to 1920 (Curre supported for output port VFPC-DEIM components)
Frm1Pitch	Height of the frame at second port correspondir the channel.	For 420– Width For 422 - 2* Width
FrmStartX	Horizontal StartOffset	Integer value of range 0 to width (Applies only at input end of the channel)
FrmStartY	Vertical StartOffset	Integer value of range 0 to height(Applies only at input end of the channel)
FrmCropWidth	Crop Width	Integer value of range 0 to width - FrmStartX (App only at the input end of channel)
FrmCropHeight	Crop Height	Integer value of range 0 to height - FrmStartY(App only at the input end of channel)

OMX Index **: OMX_TI_IndexConfigAlgEnable**

Description: Flag to enable/disable the Algorithm/IP

Field Name	**Description**	**Values**
nChId	Indicates the channel index for which configuration values apply.	0 – All VFPC components suppo channel per instance
bAlgBy	If set to true, the HW IP is set in bypass.	0 – Indicates that the Bypas disabled, HW IP is enabled 1 – Indicates that the Bypas enabled, HW IP is used.

OMX Index **: OMX_TI_IndexConfigSubSamplingFactor**

Description: To set the Sub sampling factor of a specific port. This is used to drop the output frame rate of the component in the order of 1/2, 1/3, 1/4 etc. Please not that validated only for 1/2(reduce to half) configuration. The sub sampling factor shall be an integer. For example: 1 – no drop, 2 – alternative frame drops etc.

Default Values of omxVfpcDriverProperties_t for OMX_VFPC_DEIDUALOUT_MEDIUM_COMP_NAME component.

Field Name	**Description**	**Values**
nPortIndex	Index of the conce output port.	0 or 1
nSubSamplingFactor	Reduxtion of output fr rate in the order of nSubSamplingFactor)	1 – no subsampling 2 – subsampling by half (tested configuration)
Field Name	**Description**	**Values**
nDrvInst	Driver Instance ID	**TI_814x**:VPS_M2M_INST _MAIN_DEI_SC1_SC3_ WB0_VIP0 **TI_816x**: VPS_M2M_INST_AUX_D EI_SC2_SC4_WB1_VIP1
pDriverProperties->nNumCurInputBufPerProcess	Number of input buffers required by the driver in each process call.	OMX_VFPC_DEIDUALOUT_MEDIUM_NUM_CURRENT_INPUT_FRAMES_PER_PROCESS (1)
pDriverProperties->nNumHistoryBufPerProcess	Number of input buffers required by the driver in each process call. This number of history buffers required per process call	OMX_VFPC_DEIDUALOUT_MEDIUM_NUM_HISTORY_FRAMES_PER_PROCESS (0)
pDriverProperties->nNumOutputPerProcess	Number of output buffers required by the driver in each process call	OMX_VFPC_DEIDUALOUT_MEDIUM_NUM_OUTPUT_FRAMES_PER_PROCESS (2)
pDriverProperties->auOutputSubSampleFactor[0]	Indicates default subsampling info that is to applied . Subsampling may be 1/2, 1/4 etc.	OMX_VFPC_DEIDUALOUT_SUBSAMPLEFACTOR_OUTPUT0 (2)
pDriverProperties->auOutputSubSampleFactor[1]	Indicates default subsampling info that is to applied . Subsampling may be 1/2, 1/4 etc	OMX_VFPC_DEIDUALOUT_SUBSAMPLEFACTOR_OUTPUT1 (2)
pDriverProperties->sHistoryProp[0].nFrameLi	The index into the frameList that will be	0

stIndex	used as history buffer	
pDriverProperties->sHistoryProp[0].eFrameListType	The enum identifying if nFrameListIndex refers to the inFrameList or the outFrameList	OMX_VFPC_FRAMELIST_TYPE_INPUT
pDriverProperties->sInBufProp[0].eColorFormat	The Color format of the I/O buf	OMX_COLOR_FormatYUV420SemiPlanar
pDriverProperties->sInBufProp[0].nMaxFrameWidth	Max frame width of the I/O buf	OMX_VFPC_DEFAULT_INPUT_FRAME_WIDTH (1920)
pDriverProperties->sInBufProp[0].nMaxFrameHeight	Max frame height of the I/O buf	OMX_VFPC_DEFAULT_INPUT_FRAME_HEIGHT (1080)
pDriverProperties->sInBufProp[0].eBufferMemType	The MemType of the I/O buf - Tled Vs Non-Tiled	OMX_BUFFER_MEMORY_TILED_8BIT
pDriverProperties->sOutBufProp[0].eColorFormat	The Color format of the I/O buf	OMX_COLOR_FormatYCbYCr
pDriverProperties->sOutBufProp[0].nMaxFrameWidth	Max frame width of the I/O buf	OMX_VFPC_DEFAULT_OUTPUT_FRAME_WIDTH (1920)
pDriverProperties->sOutBufProp[0].nMaxFrameHeight	Max frame height of the I/O buf	OMX_VFPC_DEFAULT_OUTPUT_FRAME_HEIGHT (1080)
pDriverProperties->sOutBufProp[0].eBufferMemType	The MemType of the I/O buf - Tled Vs Non-Tiled	OMX_BUFFER_MEMORY_DEFAULT
pDriverProperties->sOutBufProp[1].eColorFormat	The Color format of the I/O buf	OMX_COLOR_FormatYUV420SemiPlanar
pDriverProperties->sOutBufProp[1].nMaxFrameWidth	Max frame width of the I/O buf	OMX_VFPC_DEFAULT_OUTPUT_FRAME_WIDTH (1920)
pDriverProperties->sOutBufProp[1].nMaxFrameHeight	Max frame height of the I/O buf	OMX_VFPC_DEFAULT_OUTPUT_FRAME_HEIGHT (1080)
pDriverProperties->sOutBufProp[1].eBufferMemType	The MemType of the I/O buf - Tled Vs Non-Tiled	OMX_BUFFER_MEMORY_TILED_8BIT

2.4 Control Components (CTRL)

The CTRL component is a control component that has no omx input/output ports. This is not a data processing component. The job of the CTRL is to configure & start the control functionalities such as external video decoder (TVP, SIL etc) and display controller configurations on the DM814x.

The following CTRL components are released in the package

1. **CTRL- TVP (External Video Decoder):**

   This component supports the configuration & control of external video decoders such as TVP 5158 (muxed capture), TVP 7002 (non muxed 1080i60 capture, component input), SIL 9135 (non muxed 1080P60 capture, HDMI input).

   NOTE: It's preferred to move OMX.TI.VPSSM3.CTRL.TVP componet to loaded to idle state before OMX.TI.VPSSM3.VFCC move to idle. Also move OMX.TI.VPSSM3.CTRL.TVP componet to execute to idle state after OMX.TI.VPSSM3.VFCC move to idle

2. **CTRL- DC (Display Controller):**

   This component supports the Display controller configuration for various display paths in side the DM814x display controller HW

   Supports configuring the following display controller paths

   1. BP0 to on-chip HDMI display
   2. BP1 to on-chip DVO2 and route to LCD display on the catalog Daughter card

   Primary display is always on-chip HDMI

   Secondary one is DVO2.

Configuration

OMX Index : OMX_TI_IndexParamCTRLVidDecInfo

Description: Configure the video decoder parameters

OMX_CTRL_VIDEO_DECODER_STD: Defines the supported video decoder standards.

Field Name	Description	Values
nPortIndex	Index of the concerned output port	0 or 1
videoStandard	Video Standard	OMX_VIDEO_DECODER_STD_1080P_60 OMX_VIDEO_DECODER_STD_1080I_60 OMX_VIDEO_DECODER_STD_AUTO_DET
videoDecoderId	Decoder interface on board	OMX_VID_DEC_TVP7002_DRV OMX_VID_DEC_SII9135_DRV OMX_VID_DEC_TVP5158_DRV

Features Supported

- Connecting multiplexers, VCOMP, CIG and COMP modules statically and dynamically (but not at run time, i.e. after display is started)
- Supports setting modes and synchronizing multiple VENCs
- All HD VENCs support upto 720p60, 1080p30, 1080i60 and 1080p60 modes. Other modes are not supported.

Limitation

- Does not support configuring different modes on the tied VENCs like 1080P@60 FPS on DVO1 and 720P@60 FPS on DV02 could not be tied (synchronized)
- Run time configuration of VCOMP, CIG and blenders is not supported
- CPROC features are not supported. CPROC is currently put in simple bypass mode - does only color space conversion. Note that CPROC module is available only on TI814X.
- Run time switching of input path at the multiplexer and graphics enable/disable at the COMP is not supported.
- Runtime configuration of VCOMP, CIG and Blenders is not supported

2.5 Video Decoder Component (VDEC)

The VDEC component is a video decode component that has one input port and one output port. The job of the VDEC is to decode the encoded frames provided to it. In other words, it encapsulates the Video Decoder.

The following should be noted about the VDEC component –

- Supports H264, MPEG4, H263 and MPEG2 decode. It does not support any other compression format
- Supports only frame level decoding. Encoded data should be given as a whole frame, i.e, stream parsing and marking frame boundaries needs to happen outside the component. Slice based decoding is not supported
- Supports decoding of only progressive content. Interlaced content decoding is not supported
- Supports codec supported color format. Format supported in this release is OMX_COLOR_FormatYUV420PackedSemiPlanar. It is based on decoder's capability.
- Does not support run time configuration of its dynamic parameters using the OMX_GetConfig and OMX_SetConfig apis
- Without processing at the most 8 buffers can be queued up.
- Video decoder o/p buffers have padding and alignment requirement. Padding is required both in Horizontal (PadX) and Vertical (PadY) direction. Given below is a computation of the buffer size for 420 o/p buffers of VDEC. This is calculated as:

{ALIGN((Width + 2 * PadX), 128)} * {ALIGN(height, 16) + 4 * PadY} * (3 / 2)

Where ALIGN(value, alignment) is a macro that ensures the given *'value'* is adjusted to the next multiple of '*alignment*'. The user should set the appropriate buffer size in the VDEC output port parameters based on the above formula. The table below documents the Horizontal (PadX) and Vertical (PadY) padding required for various compression formats supported by VDEC. The application should allocate output buffers taking this into consideration.

Compression Format	Horizontal Padding (PadX)	Vertical Padding (PadY)
H264	32	24
MPEG4	16	16
H263	16	16
MPEG2	8	8
VC1	32	40
MJPEG	0	0

The stride (or pitch) of the output frame generated by VDEC also varies based on the compression format. The table below summarizes the stride for various compression formats. This is expressed as a function of Width (resolution width) and PadX (padding in horizontal direction).

Compression Format	Horizontal Padding (PadX)	Stride (Pitch)
H264	32	(Width + (2 * PadX) + 127) & 0xFFFFFF80;
MPEG4	16	(Width + (2 * PadX) + 127) & 0xFFFFFF80
H263	16	(Width + (2 * PadX) + 127) & 0xFFFFFF80
MPEG2	8	(Width + 15) & 0xFFFFFFF0
VC1	32	(Width + (2 * PadX) + 127) & 0xFFFFFF80
MJPEG	0	(Width + 15) & 0xFFFFFFF0

Users are recommended to go through individual codec user guides for more details.

For setting all the parameters supported by codec, two new indices have been added, which allows complete structure of static and dynamic

parameters to be configured using codec standard static and dynamic parameter structure. In the decode_display example static parameter change code is provided.

For selecting the codec, compression format should be specified in parameter setting using OMX_IndexParamPortDefinition. Please refer to decode example.

For H264 - OMX_VIDEO_CodingAVC

For MPEG4 – OMX_VIDEO_CodingMPEG4

For MPEG2 - OMX_VIDEO_CodingMPEG2

For H263 - OMX_VIDEO_CodingH263

For VC1 - OMX_VIDEO_CodingWMV

2.6 Video Encoder Component (VDEC)

The VENC component is a video encode component that has one input port and one output port. The job of the VENC is to encode the raw frames provided to it. In other words, it encapsulates the Video Encoder

The following should be noted about the VENC component –

- Supports H264 and MPEG4 encode. It does not support any other compression format
- Supports only YUV420 semi planner input format (OMX_COLOR_FormatYUV420PackedSemiPlanar)
- Supports encoding of only progressive content. Interlaced content encoding is not supported
- Supports encoding of I & P frames. Does not support B frames
- Supports run time configuration of all dynamic parameters supported by the codec using OMX_GetConfig and OMX_SetConfig apis
- Without processing at the most 8 buffers can be queued up.
- Does not support slice mode encoding.

VENC – Setting & controlling parameters

As mentioned earlier, VENC supports run time configuration of its dynamic parameters using the OMX_GetConfig & OMX_SetConfig apis. However, not all standard OMX indices & associated CONFIG structures have been supported. Those that are supported are mentioned in the table below. Note that OMX_GetConfig returns the current settings whereas OMX_SetConfig is used to apply the new settings.

```
eError = OMX_SetConfig( hComponent,  OMX_IndexConfigX,
(OMX_PTR) OMX_VIDEO_CONFIG_Y);
```

It is recommended to do a OMX_GetConfig call before the corresponding OMX_SetConfig call in order to determine the current settings.

For selecting the codec type, compression format should be specified in parameter setiing using OMX_IndexParamPortDefinition index. This is specified in following wiki link. Encode example provided in SDK provides example for this.

For setting all the parameters supported by codec, two new indices have been added, which allows complete structure of static and dynamic parameters to be configured using codec standard static and dynamic parameter structure. Following link has example of using these indices. For structure definition and parameters settings, please refer the specific codec user guide.

http://processors.wiki.ti.com/index.php/VENC

Index OMX_IndexConfigX	**CONFIG structure OMX_VIDEO_CONFIG_Y**	**Description**	**Unit**
OMX_IndexConfigVideoBitrate	OMX_VIDEO_CONFIG_BITRATETYPE	nEncodeBitrate is the target bit rate for the Video Encoder.	Bits per second. E.g. nEncodeBitrate = 2000000 for bit rate of 2 mbps
OMX_IndexConfigVideoFramerate	OMX_CONFIG_FRAMERATETYPE	xEncodeFramerate is the target frame rate for the Video Encoder	In Q16 as per OMX specifications. E.g, xEncodeFramerate = 60*65536 for frame rate of 60 fps
OMX_IndexConfigVideoAVCIntraPeriod	OMX_VIDEO_CONFIG_AVCINTRAPERIOD	nPFrames is the number of P frames between 2 I frames (intraFrameInterval)	Any integer value. Eg, nPFrames=29 results in an I frame once every 30 frames
OMX_IndexConfigVideoIntraVOPRefresh	OMX_CONFIG_INTRAREFRESHVOPTYPE	IntraRefreshVOP = TRUE forces an IDR frame to be generated by the Video Encoder	
OMX_TI_IndexConfigVideoDynamicParams	OMX_VIDEO_CONFIG_DYNAMICPARAMS	This is a custom extension (not part of OMX standard). It encapsulates the Video Encoder's dynamic parameters structure. It enables setting the codec's dynamic parameters directly. See OMX_TI_Video.h	

		for the definition of this strcuture	

2.7 Audio Decoder Component (ADEC)

The ADEC component is a audio decode component that has one input port and one output port. The job of the ADEC is to decode the encoded frames provided to it. In other words, it encapsulates the Audio Decoder.

The following should be noted about the ADEC component –

- Supports MP3 and AAC-LC decode. It does not support any other compression format.
- Supports single input/output port with single input buffer and single output buffer.
- Supports bit-stream bucket based decoding. Encoded data should be given as a pool of 4KB input buffer, the component will consume x amount of bytes required to decode a frame. Application will refill the buffer to maintain again 4KB input buffer pool.
- Decoded output PCM is Stereo channels with 16-bit bit-precision per sample. If the stream is mono, internal codec performs mono to stereo conversion
- Does not support run time configuration of its dynamic parameters using the OMX_GetConfig and OMX_SetConfig apis
- In AAC-LC, raw data format is not supported.

2.8 Audio Encoder Component (AENC)

The AENC component is a audio encode component that has one input port and one output port. The job of the AENC is to enode the audio PCM frames provided to it. In other words, it encapsulates the Audio Encoder.

The following should be noted about the AENC component –

- Supports AAC-LC encode. It does not support any other compression format

Chapter 3

API Reference

This section provided a description of the OpenMax components available on the DM814x OpenMax Multimedia Sub-system.

General Information about OMX components

TI's implementation of the DM814x OpenMax Multimedia Sub-system is based on OpenMax IL v1.1.2. The OpenMax IL v1.1.2 specifications are included in this release in the directory $OMXINSTALL/docs. This section supplements the specifications with additional details on additional features/extensions/constraints of TI's implementation. The rest of this chapter will frequently cross reference sections of the OpenMax v1.1.2 specifications (appropriate section quoted in square braces).

General Information

The standard OpenMax APIs are located in the directory $OMXINSTALLDIR/interfaces/openMaxv11.

Enumerations

The Enumerations used in our code are identical to those specified by the specifications.

OpenMax IL APIs

The following is the list of OpenMax IL APIs and some notes on their implementation status where appropriate.

OMX_ComponentNameEnum (cComponentName, nNameLength, nIndex)

The OMX_ComponentNameEnum method will enumerate through all the names of recognized valid components in the system. This function is provided as a means to detect all the components in the system run-time. There is no strict ordering to the enumeration order of component names, although each name will only be enumerated once. If the OMX core supports run-time installation of new components, it is only required to detect newly installed components when the first call to enumerate component names is made (i.e. when nIndex is 0x0).

The core should return from this call in 20 msec.

Parameters:

[out] *cComponentName* pointer to a null terminated string with the component name. The names of the components are strings less than 127 bytes in length plus the trailing null for a maximum size of 128 bytes. An example of a valid component name is "OMX.TI.AUDIO.DSP.MIXER\0". Names are assigned by the vendor, but shall start with "OMX." and then have the Vendor designation next.
[in] *nNameLength* number of characters in the cComponentName string. With all component name strings restricted to less than 128 characters (including the trailing null) it is recomended that the caller provide a input string for the cComponentName of 128 characters.
[in] *nIndex* 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 is returned. The value of nIndex is 0 to (N-1), where N is the number of valid installed components in the system.

Returns:

OMX_ERRORTYPE If the command 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 OMX error will be returned.

OMX_Deinit (void)

The OMX_Deinit method is used to deinitialize the OMX core. It shall be the last call made into OMX. In the event that the core determines that thare are components loaded when this call is made, the core may return with an error rather than try to unload the components.

The core should return from this call within 20 msec.

Returns:

OMX_ERRORTYPE If the command successfully executes, the return code will be OMX_ErrorNone. Otherwise the appropriate OMX error will be returned.

OMX_FreeHandle (hComponent)

The OMX_FreeHandle method will free a handle allocated by the OMX_GetHandle method. If the component reference count goes to zero, the component will be unloaded from memory.

The core should return from this call within 20 msec when the component is in the OMX_StateLoaded state.

Parameters:

[in] *hComponent* Handle of the component to be accessed. This is the component handle returned by the call to the GetHandle function.

Returns:

OMX_ERRORTYPE If the command successfully executes, the return code will be OMX_ErrorNone. Otherwise the appropriate OMX error will be returned.

OMX_GetComponentsOfRole (role, pNumComps, compNames)

[This API is not implemented in this release].

The OMX_GetComponentsOfRole method will return the number of components that support the given role and (if the compNames field is non-NULL) the names of

those components. The call will fail if an insufficiently sized array of names is supplied. To ensure the array is sufficiently sized the client should: first call this function with the compNames field NULL to determine the number of component names second call this function with the compNames field pointing to an array of names allocated according to the number returned by the first call.

The core should return from this call within 5 msec.

Parameters:

[in] *role* This is generic standard component name consisting only of component class name and the type within that class (e.g. 'audio_decoder.aac').
[inout] pNumComps This is used both as input and output.

If compNames is NULL, the input is ignored and the output specifies how many components support the given role.

If compNames is not NULL, on input it bounds the size of the input structure and on output, it specifies the number of components string names listed within the compNames parameter.

Parameters:

[inout] compNames If NULL this field is ignored. If non-NULL this points to an array of 128-byte strings which accepts a list of the names of all physical components that implement the specified standard component name. Each name is NULL terminated. numComps indicates the number of names.

OMX_GetHandle (pHandle, cComponentName, pAppData, pCallBacks)

The OMX_GetHandle method will locate the component specified by the component name given, load that component into memory and then invoke the component's methods to create an instance of the component.

The core should return from this call within 20 msec.

Parameters:

[out] *pHandle* pointer to an OMX_HANDLETYPE pointer to be filled in by this method.
[in] *cComponentName* pointer to a null terminated string with the component name. The names of the components are strings less than 127 bytes in length plus the trailing null for a maximum size of 128 bytes. An example of a valid component name is "OMX.TI.AUDIO.DSP.MIXER\0". Names are assigned by the vendor, but shall start with "OMX." and then have the Vendor designation next.
[in] *pAppData* pointer to an application defined value that will be returned during callbacks so that the application can identify the source of the callback.
[in] *pCallBacks* pointer to a **OMX_CALLBACKTYPE** structure that will be passed to the component to initialize it with.

Returns:

OMX_ERRORTYPE If the command successfully executes, the return code will be OMX_ErrorNone. Otherwise the appropriate OMX error will be returned.

OMX_GetRolesOfComponent (compName, pNumRoles, roles)

[This API is not implemented in this release].

The OMX_GetRolesOfComponent method will return the number of roles supported by the given component and (if the roles field is non-NULL) the names of those roles. The call will fail if an insufficiently sized array of names is supplied. To ensure the array is sufficiently sized the client should: first call this function with the roles field NULL to determine the number of role names second call this function with the roles field pointing to an array of names allocated according to the number returned by the first call.

The core should return from this call within 5 msec.

Parameters:

[in] *compName* This is the name of the component being queried about.
[inout] pNumRoles This is used both as input and output.

If roles is NULL, the input is ignored and the output specifies how many roles the component supports.

If compNames is not NULL, on input it bounds the size of the input structure and on output, it specifies the number of roles string names listed within the roles parameter.

Parameters:

[out] *roles* If NULL this field is ignored. If non-NULL this points to an array of 128-byte strings which accepts a list of the names of all standard components roles implemented on the specified component name. numComps indicates the number of names.

OMX_Init (void)

The OMX_Init method is used to initialize the OMX core. It shall be the first call made into OMX and it should only be executed one time without an interviening OMX_Deinit call.

The core should return from this call within 20 msec.

Returns:

OMX_ERRORTYPE If the command successfully executes, the return code will be OMX_ErrorNone. Otherwise the appropriate OMX error will be returned.

OMX_SetupTunnel (hOutput, nPortOutput, hInput, nPortInput)

The OMX_SetupTunnel method will handle the necessary calls to the components to setup the specified tunnel the two components. NOTE: This is an actual method (not a define macro). This method will make calls into the component ComponentTunnelRequest method to do the actual tunnel connection.

The ComponentTunnelRequest method on both components will be called. This method shall not be called unless the component is in the OMX_StateLoaded state except when the ports used for the tunnel are disabled. In this case, the component may be in the OMX_StateExecuting, OMX_StatePause, or OMX_StateIdle states.

The core should return from this call within 20 msec.

Parameters:

[in] *hOutput* Handle of the component to be accessed. Also this is the handle of the component whose port, specified in the nPortOutput parameter will be used the source for the tunnel. This is the component handle returned by the call to

the OMX_GetHandle function. There is a requirement that hOutput be the source for the data when tunelling (i.e. nPortOutput is an output port). If 0x0, the component specified in hInput will have it's port specified in nPortInput setup for communication with the application / IL client.
[in] *nPortOutput* nPortOutput is used to select the source port on component to be used in the tunnel.
[in] *hInput* This is the component to setup the tunnel with. This is the handle of the component whose port, specified in the nPortInput parameter will be used the destination for the tunnel. This is the component handle returned by the call to the OMX_GetHandle function. There is a requirement that hInput be the destination for the data when tunelling (i.e. nPortInut is an input port). If 0x0, the component specified in hOutput will have it's port specified in nPortPOutput setup for communication with the application / IL client.
[in] *nPortInput* nPortInput is used to select the destination port on component to be used in the tunnel.

Returns:

OMX_ERRORTYPE If the command successfully executes, the return code will be OMX_ErrorNone. Otherwise the appropriate OMX error will be returned. When OMX_ErrorNotImplemented is returned, one or both components is a non-interop component and does not support tunneling.

On failure, the ports of both components are setup for communication with the application / IL Client.

[This API is not implemented in this release, as Non tunneled mode does not require this].

ComponentTunnelRequest(hComp, nPort, hTunneledComp, nTunneledPort, pTunnelSetup)

The ComponentTunnelRequest method will interact with another OMX component to determine if tunneling is possible and to setup the tunneling. The return codes for this method can be used to determine if tunneling is not possible, or if tunneling is not supported.

Base profile components (i.e. non-interop) do not support this method and should return OMX_ErrorNotImplemented

The interop profile component MUST support tunneling to another interop profile component with a compatible port parameters. A component may also support proprietary communication.

If proprietary communication is supported the negotiation of proprietary communication is done outside of OMX in a vendor specific way. It is only required that the proper result be returned and the details of how the setup is done is left to the component implementation.

When this method is invoked when nPort in an output port, the component will: 1. Populate the pTunnelSetup structure with the output port's requirements and constraints for the tunnel.

When this method is invoked when nPort in an input port, the component will: 1. Query the necessary parameters from the output port to determine if the ports are compatible for tunneling 2. If the ports are compatible, the component should store the tunnel step provided by the output port 3. Determine which port (either input or output) is the buffer supplier, and call OMX_SetParameter on the output port to indicate this selection.

The component will return from this call within 5 msec.

Parameters:

[in] *hComp* Handle of the component to be accessed. This is the component handle returned by the call to the OMX_GetHandle method.
[in] *nPort* nPort is used to select the port on the component to be used for tunneling.
[in] *hTunneledComp* Handle of the component to tunnel with. This is the component handle returned by the call to the OMX_GetHandle method. When this parameter is 0x0 the component should setup the port for communication with the application / IL Client.
[in] *nPortOutput* nPortOutput is used indicate the port the component should tunnel with.
[in] *pTunnelSetup* Pointer to the tunnel setup structure. When nPort is an output port the component should populate the fields of this structure. When When nPort is an input port the component should review the setup provided by the component with the output port.

Returns:

OMX_ERRORTYPE If the command successfully executes, the return code will be OMX_ErrorNone. Otherwise the appropriate OMX error will be returned.

[This API is not implemented in this release, as Non tunneled mode does not require this].

OpenMax Component and Core APIs

OMX_GetComponentVersion(hComponent, pComponentName, pComponentVersion, pSpecVersion, pComponentUUID)

GetComponentVersion will return information about the component. This is a blocking call. This macro will go directly from the application to the component (via a core macro). The component will return from this call within 5 msec (timing not verified in this version).

Parameters:

[in] *hComponent* handle of component to execute the command
[out] *pComponentName* pointer to an empty string of length 128 bytes. The component will write its name into this string. The name will be terminated by a single zero byte. The name of a component will be 127 bytes or less to leave room for the trailing zero byte. An example of a valid component name is "OMX.ABC.ChannelMixer\0".
[out] *pComponentVersion* pointer to an OMX Version structure that the component will fill in. The component will fill in a value that indicates the component version. NOTE: the component version is NOT the same as the OMX Specification version (found in all structures). The component version is defined by the vendor of the component and its value is entirely up to the component vendor.
[out] *pSpecVersion* pointer to an OMX Version structure that the component will fill in. The SpecVersion is the version of the specification that the component was built against. Please note that this value may or may not match the structure's version. For example, if the component was built against the 2.0 specification, but the application (which creates the structure is built against the 1.0 specification the versions would be different.

[out] *pComponentUUID* pointer to the UUID of the component which will be filled in by the component. The UUID is a unique identifier that is set at RUN time for the component and is unique to each instantion of the component.

Returns:

OMX_ERRORTYPE If the command successfully executes, the return code will be OMX_ErrorNone. Otherwise the appropriate OMX error will be returned.

OMX_GetExtensionIndex(hComponent, cParameterName, pIndexType)

Not implemented in this release.

The OMX_GetExtensionIndex macro will invoke a component to translate a vendor specific configuration or parameter string into an OMX structure index. There is no requirement for the vendor to support this command for the indexes already found in the OMX_INDEXTYPE enumeration (this is done to save space in small components). The component shall support all vendor supplied extension indexes not found in the master OMX_INDEXTYPE enumeration. This is a blocking call.

The component should return from this call within 5 msec.

Parameters:

[in] *hComponent* Handle of the component to be accessed. This is the component handle returned by the call to the GetHandle function.
[in] *cParameterName* OMX_STRING that shall be less than 128 characters long including the trailing null byte. This is the string that will get translated by the component into a configuration index.
[out] *pIndexType* a pointer to a OMX_INDEXTYPE to receive the index value.

Returns:

OMX_ERRORTYPE If the command successfully executes, the return code will be OMX_ErrorNone. Otherwise the appropriate OMX error will be returned.

OMX_GetParameter(hComponent, nParamIndex, pComponentParameterStructure)

The OMX_GetParameter macro will get one of the current parameter settings from the component. This macro cannot only be invoked when the component is in the OMX_StateInvalid state. The nParamIndex parameter is used to indicate which structure is being requested from the component. The application shall allocate the correct structure and shall fill in the structure size and version information before invoking this macro. When the parameter applies to a port, the caller shall fill in the appropriate nPortIndex value indicating the port on which the parameter applies. If the component has not had any settings changed, then the component should return a set of valid DEFAULT parameters for the component. This is a blocking call.

The component should return from this call within 20 msec (timing not verified in this version).

The parameters that have been implemented are listed later in this chapter in the section ‘Parameters Implemented’. The parameters are divided into (i) common parameters that have been implemented for all components and (ii) additional

parameters if any for each component are specified later against the name of the component.

Parameters:

[in] *hComponent* Handle of the component to be accessed. This is the component handle returned by the call to the OMX_GetHandle function.
[in] *nParamIndex* Index of the structure to be filled. This value is from the OMX_INDEXTYPE enumeration.
[in,out] *pComponentParameterStructure* Pointer to application allocated structure to be filled by the component.

Returns:

OMX_ERRORTYPE If the command successfully executes, the return code will be OMX_ErrorNone. Otherwise the appropriate OMX error will be returned.

OMX_GetState(hComponent, pState)

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 5 msec (timing not verified in this version).

The components provided in the current release must be transitioned from states Loaded à Idle à Executing during the setup and from Executing à Idle à Loaded during tear-down.
This release only implements OMX_StateLoaded, OMX_StateIdle, and OMX_StateExecuting. Other states are not implemented. In case of invalid/corrupt parameters, or unavailable resources, the behavior of this release is not guaranteed/ tested.

Parameters:

[in] *hComponent* Handle of the component to be accessed. This is the component handle returned by the call to the OMX_GetHandle function.
[out] *pState* pointer to the location to receive the state. The value returned is one of the OMX_STATETYPE members

Returns:

OMX_ERRORTYPE If the command successfully executes, the return code will be OMX_ErrorNone. Otherwise the appropriate OMX error will be returned.

OMX_SendCommand(hComponent, Cmd, nParam, pCmdData)

Send a command to the component. This call is a non-blocking call. The component should check the parameters and then queue the command to the component thread to be executed. The component thread shall send the EventHandler() callback at the conclusion of the command. This macro will go directly from the application to the component (via a core macro). The component will return from this call within 5 msec (timing not verified in this version).

When the command is "OMX_CommandStateSet" the component will queue a state transition to the new state idenfied in nParam.

When the command is "OMX_CommandFlush", to flush a port's buffer queues, the command will force the component to return all buffers NOT CURRENTLY BEING PROCESSED to the application, in the order in which the buffers were received.

When the command is "OMX_CommandPortDisable" or "OMX_CommandPortEnable", the component's port (given by the value of nParam) will be stopped or restarted.

When the command "OMX_CommandMarkBuffer" is used to mark a buffer, the pCmdData will point to a **OMX_MARKTYPE** structure containing the component handle of the component to examine the buffer chain for the mark. nParam1 contains the index of the port on which the buffer mark is applied. See Specification text for more details. Implemented in this release except for OMX_CommandMarkBuffer and OMX_CommandFlush.

Parameters:

[in] *hComponent* handle of component to execute the command
[in] *Cmd* Command for the component to execute
[in] *nParam* Parameter for the command to be executed. When Cmd has the value OMX_CommandStateSet, value is a member of OMX_STATETYPE. When Cmd has the value OMX_CommandFlush, value of nParam indicates which port(s) to flush. -1 is used to flush all ports a single port index will only flush that port. When Cmd has the value "OMX_CommandPortDisable" or "OMX_CommandPortEnable", the component's port is given by the value of nParam. When Cmd has the value "OMX_CommandMarkBuffer" the components pot is given by the value of nParam.
[in] *pCmdData* Parameter pointing to the **OMX_MARKTYPE** structure when Cmd has the value "OMX_CommandMarkBuffer".

Returns:

OMX_ERRORTYPE If the command successfully executes, the return code will be OMX_ErrorNone. Otherwise the appropriate OMX error will be returned.

OMX_SetConfig(hComponent, nConfigIndex, pComponentConfigStructure)

The OMX_SetConfig macro will send one of the configuration structures to a component. Each structure shall be sent one at a time, each in a separate invocation of the macro. This macro can be invoked anytime after the component has been loaded. The application shall allocate the correct structure and shall fill in the structure size and version information (as well as the actual data) before invoking this macro. The application is free to dispose of this structure after the call as the component is required to copy any data it shall retain. This is a blocking call.

The component should return from this call within 5 msec.

Parameters:

[in] *hComponent* Handle of the component to be accessed. This is the component handle returned by the call to the OMX_GetHandle function.
[in] *nConfigIndex* Index of the structure to be sent. This value is from the OMX_INDEXTYPE enumeration above.
[in] *pComponentConfigStructure* pointer to application allocated structure to be used for initialization by the component.

Returns:

OMX_ERRORTYPE If the command successfully executes, the return code will be OMX_ErrorNone. Otherwise the appropriate OMX error will be returned.

OMX_SetParameter(hComponent, nParamIndex, pComponentParameterStructure)

The OMX_SetParameter macro will send an initialization parameter structure to a component. Each structure shall be sent one at a time, in a separate invocation of the macro. This macro can only be invoked when the component is in the OMX_StateLoaded state, or the port is disabled (when the parameter applies to a port). The nParamIndex parameter is used to indicate which structure is being passed to the component. The application shall allocate the correct structure and shall fill in the structure size and version information (as well as the actual data) before invoking this macro. The application is free to dispose of this structure after the call as the component is required to copy any data it shall retain. This is a blocking call.

The component should return from this call within 20 msec (timing not verified in this release).

The parameters that have been implemented are listed later in this chapter in the section 'Parameters Implemented'. The parameters are divided into (i) common parameters that have been implemented for all components and (ii) additional parameters if any for each component are specified later against the name of the component.

Parameters:

[in] *hComponent* Handle of the component to be accessed. This is the component handle returned by the call to the OMX_GetHandle function.
[in] *nIndex* Index of the structure to be sent. This value is from the OMX_INDEXTYPE enumeration.
[in] *pComponentParameterStructure* pointer to application allocated structure to be used for initialization by the component.

Returns:

OMX_ERRORTYPE If the command successfully executes, the return code will be OMX_ErrorNone. Otherwise the appropriate OMX error will be returned.

OMX_EmptyThisBuffer(hComponent, pBuffer)

The OMX_EmptyThisBuffer macro will send a buffer full of data to an input port of a component. The buffer will be emptied by the component and returned to the application via the EmptyBufferDone call back. This is a non-blocking call in that the component will record the buffer and return immediately and then empty the buffer, later, at the proper time. As expected, this macro may be invoked only while the component is in the OMX_StateExecuting. If nPortIndex does not specify an input port, the component shall return an error.

The component should return from this call within 5 msec.

Parameters:

[in] *hComponent* Handle of the component to be accessed. This is the component handle returned by the call to the OMX_GetHandle function.
[in] *pBuffer* pointer to an **OMX_BUFFERHEADERTYPE** structure allocated with UseBuffer or AllocateBuffer.

Returns:

OMX_ERRORTYPE If the command successfully executes, the return code will be OMX_ErrorNone. Otherwise the appropriate OMX error will be returned.

OMX_FillThisBuffer(hComponent, pBuffer)

The OMX_FillThisBuffer macro will send an empty buffer to an output port of a component. The buffer will be filled by the component and returned to the application via the FillBufferDone call back. This is a non-blocking call in that the component will record the buffer and return immediately and then fill the buffer, later, at the proper time. As expected, this macro may be invoked only while the component is in the OMX_ExecutingState. If nPortIndex does not specify an output port, the component shall return an error.

The component should return from this call within 5 msec.

Parameters:

[in] *hComponent* Handle of the component to be accessed. This is the component handle returned by the call to the OMX_GetHandle function.
[in] *pBuffer* pointer to an **OMX_BUFFERHEADERTYPE** structure allocated with UseBuffer or AllocateBuffer.

Returns:

OMX_ERRORTYPE If the command successfully executes, the return code will be OMX_ErrorNone. Otherwise the appropriate OMX error will be returned.

OMX_FreeBuffer(hComponent, nPortIndex, pBuffer)

The OMX_FreeBuffer macro will release a buffer header from the component which was allocated using either OMX_AllocateBuffer or OMX_UseBuffer. If the component allocated the buffer (see the OMX_UseBuffer macro) then the component shall free the buffer and buffer header. This is a blocking call.

The component should return from this call within 20 msec.

Parameters:

[in] *hComponent* Handle of the component to be accessed. This is the component handle returned by the call to the OMX_GetHandle function.
[in] *nPortIndex* nPortIndex is used to select the port on the component the buffer will be used with.
[in] *pBuffer* pointer to an **OMX_BUFFERHEADERTYPE** structure allocated with UseBuffer or AllocateBuffer.

Returns:

OMX_ERRORTYPE If the command successfully executes, the return code will be OMX_ErrorNone. Otherwise the appropriate OMX error will be returned.

OMX_UseBuffer(hComponent, ppBufferHdr, nPortIndex, pAppPrivate, nSizeBytes, pBuffer)

The OMX_UseBuffer macro will request that the component use a buffer (and allocate its own buffer header) already allocated by another component, or by the IL Client. This is a blocking call.

The component should return from this call within 20 msec (timing not verified in this release).

Parameters:

[in] *hComponent* Handle of the component to be accessed. This is the component handle returned by the call to the OMX_GetHandle function.
[out] *ppBuffer* pointer to an **OMX_BUFFERHEADERTYPE** structure used to receive the pointer to the buffer header

Returns:

OMX_ERRORTYPE If the command successfully executes, the return code will be OMX_ErrorNone. Otherwise the appropriate OMX error will be returned.

OMX_UseEGLImage(hComponent, ppBufferHdr, nPortIndex, pAppPrivate, eglImage)

Not implemented in this release

The OMX_UseEGLImage macro will request that the component use a EGLImage provided by EGL (and allocate its own buffer header) This is a blocking call.

The component should return from this call within 20 msec.

Parameters:

[in] *hComponent* Handle of the component to be accessed. This is the component handle returned by the call to the OMX_GetHandle function.
[out] *ppBuffer* pointer to an **OMX_BUFFERHEADERTYPE** structure used to receive the pointer to the buffer header. Note that the memory location used for this buffer is NOT visible to the IL Client.
[in] *nPortIndex* nPortIndex is used to select the port on the component 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.
[in] *pAppPrivate* pAppPrivate is used to initialize the pAppPrivate member of the buffer header structure.
[in] *eglImage* eglImage contains the handle of the EGLImage to use as a buffer on the specified port. The component is expected to validate properties of the EGLImage against the configuration of the port to ensure the component can use the EGLImage as a buffer.

Returns:

OMX_ERRORTYPE If the command successfully executes, the return code will be OMX_ErrorNone. Otherwise the appropriate OMX error will be returned.

Additional Implementation Notes

Additional implementation notes for this release:

- OMX_COMPONENTREGISTERTYPE [Sec 3.1.2.1]

  In our implementation, we statically register components with the core.

- OMX_PRIORITYMGMTTYPE [Sec 3.1.2.5]

In this release, we do not have different priorities for different components.

- OMX_RESOURCECONCEALMENTTYPE [Sec 3.1.2.6]

  Not implemented in this release.

- OMX_CALLBACKTYPE [Sec 3.1.2.9]

  In the current release, the three callback functions (EventHandler, FillBufferDone and EmptyBufferDone) have been implemented in the IL client.. For EventHandler, only OMX_EventCmdComplete and OMX_EventError have been implemented and tested.

CONFIG/SETPARMETER API Implementation

Common Get/Set Param/Config across all components

Parameters	***Description***
OMX_IndexParamAudioInit	See Sec 8.3 of Specifications
OMX_IndexParamImageInit	See Sec 8.3 of Specifications
OMX_IndexParamVideoInit	See Sec 8.3 of Specifications
OMX_IndexParamOtherInit	See Sec 8.3 of Specifications
OMX_IndexParamPortDefinition	See Sec 8.3 of Specifications
OMX_IndexParamCompBufferSupplier	See Sec 8.3 of Specifications
OMX_IndexParamPriorityMgmt	See Sec 8.3 of Specifications

Table 4-1: Common parameters implemented in GetParams for all components (See Sec 8.1 of specifications for explanation)

Parameters	**Description**
OMX_IndexParamPortDefinition	See Sec 8.3 of Specifications
OMX_IndexParamCompBufferSupplier	See Sec 8.3 of Specifications
OMX_IndexParamPriorityMgmt	See Sec 8.3 of Specifications

Table 4-2: Common Parameters implemented in standard SetParams for all components (See Sec 8.3 of specifications for explanation)

Parameters	Description
OMX_CommandStateSet	See Sec 3.1.1.1 of specifications
OMX_CommandPortDisable	See Sec 3.1.1.1 of specifications
OMX_CommandPortEnable	See Sec 3.1.1.1 of specifications
OMX_CommandMarkBuffer	See Sec 3.1.1.1 of specifications
OMX_CommandFlush	See Sec 3.1.1.1 of specifications

Table 4-3: Common commands implemented in SendCommand for all components (See Sec 8.3 of specifications for explanation)

Common Parameters implemented in SetParams for all components (See Sec 8.3 of specifications for explanation)

1. OMX_PARAM_COMPPORT_NOTIFYTYPE OMX_TI_IndexParamCompPortNotifyType:

Description: The OMX component's ports could be configured with various notification types. This will decide the component port latency, where the data processing is based on DataNotify rather than the periodic task. This will be per port configuration and the Default configuration is Notify Never.

OMX_NOTIFY_TYPE: Notification Type

```
typedef enum OMX_NOTIFY_TYPE {
        OMX_NOTIFY_TYPE_NONE   = 0x0,
        /** Notify Never */
        OMX_NOTIFY_TYPE_ALWAYS = 0x1,
        /** Always Notify */
        OMX_NOTIFY_TYPE_ONCE   = 0x2,
        /** Notify Once, Not supported and is for future use */
        OMX_NOTIFY_TYPE_32_BIT = 0x10000
}OMX_NOTIFY_TYPE;

typedef struct OMX_PARAM_COMPPORT_NOTIFYTYPE {
```

```
    OMX_U32 nSize;
    /**< Size of the structure in bytes */
    OMX_VERSIONTYPE nVersion;
    /**< OMX specification version information */
    OMX_U32 nPortIndex;
    /**< Index of the port */
    OMX_NOTIFY_TYPE eNotifyType;
    /**< Notification Type */
} OMX_PARAM_COMPPORT_NOTIFYTYPE;
```

2. OMX_PARAM_BUFFER_MEMORYTYPE

OMX_TI_IndexParamBuffMemType:

Description: Type of Memory from where the component buffers are getting allocated. By default it will be normal non tiled memory [In this release, only memory type OMX_BUFFER_MEMORY_DEFAULT is supported]

OMX_BUFFER_MEMORY_TYPE: Buffer allocation type. New types has been added required for tiled support

```
typedef enum OMX_BUFFER_MEMORY_TYPE {
    OMX_BUFFER_MEMORY_DEFAULT          = 0x1,
    /** Default Normal(Non-tiled) 1D Memory */
    OMX_BUFFER_MEMORY_TILED_8BIT       = 0x2,
    /** 8-bit Tiled memory */
    OMX_BUFFER_MEMORY_TILED_16BIT      = 0x3,
    /** 16-bit Tiled memory */
    OMX_BUFFER_MEMORY_TILED_32BIT      = 0x4,
    /** 32-bit Tiled memory */
    OMX_BUFFER_MEMORY_TILED_PAGE       = 0x5,
    /** Page Tiled memory */
    OMX_BUFFER_MEMORY_CUSTOM           = 0xA,
    /** Custom buffer allocation which will be specified by derived component,
    not tested */
    OMX_BUFFER_MEMORY_32_BIT           = 0x10000
} OMX_BUFFER_MEMORY_TYPE;
```

```
typedef struct OMX_PARAM_BUFFER_MEMORYTYPE {
        OMX_U32 nSize;
        /**< Size of the structure in bytes */
        OMX_VERSIONTYPE nVersion;
        /**< OMX specification version information */
        OMX_U32 nPortIndex;
        /**< Index of the port */
        OMX_BUFFER_MEMORY_TYPE eBufMemoryType;
        /**< Type of the Memory to be allocated */
} OMX_PARAM_BUFFER_MEMORYTYPE;
```

Common Parameters implemented in SetConfig for all components (See Sec 8.3 of specifications for explanation)

1. **OMX_CONFIG_DOMXPROXYCOMPINFO**

   Index : OMX_TI_IndexConfigGetDomxCompInfo

   Decryption: To get the domx related component info such stub & skel handles etc

```
typedef struct OMX_CONFIG_DOMXPROXYCOMPINFO
{
        OMX_U32 nSize;
        /**< Size of the structure in bytes */
        OMX_VERSIONTYPE nVersion;
        /**< OMX specification version information */
        OMX_HANDLETYPE hCompRealHandle;
        /**< Real Component handle - valid only on remote core */
        OMX_PTR pRpcStubHandle;
        /**< Rpc Stub Handle for the OmxProxy */
        OMX_U32 nRpcSkelPtr;
        /**< Rpc Skel Handle for the OmxProxy - Valid only on remote core */
        OMX_S8 cComponentName[OMX_MAX_STRINGNAME_SIZE];
        /**< Component name */
        OMX_S8 cComponentRcmSvrName[OMX_MAX_STRINGNAME_SIZE];
```

```
        /**< Component rcmsvr name */
} OMX_CONFIG_DOMXPROXYCOMPINFO;
```

2. OMX_CONFIG_VIDCHANNEL_RESOLUTION

OMX_TI_IndexConfigVidChResolution

Decryption: API to Configure the resolution related parameters of a specific channel (port).
Assumption - Any driver channel of VFPC module will have a max of two input ports and a max of two output ports

```
typedef struct OMX_CONFIG_VIDCHANNEL_RESOLUTION {

        OMX_U32 nSize;

        /**< Size of the structure in bytes */

        OMX_VERSIONTYPE nVersion;

        /**< OMX specification version information */

        OMX_U32 nPortIndex;

        /**< Index of the port */

        OMX_DIRTYPE eDir;

        /**< OMX_DirInput - Input, OMX_DirOutput - Output */

        OMX_U32 nChId;

        /**< Channel ID */

        OMX_U32 Frm0Width;

        /**< Width of first Frame */

        OMX_U32 Frm0Height;

        /**< Height of first Frame */

        OMX_U32 Frm0Pitch;

        /**< Pitch of first Frame */

        OMX_U32 Frm1Width;

        /**< Width of Second Frame */

        OMX_U32 Frm1Height;

        /**< Height of Second Frame */

        OMX_U32 Frm1Pitch;

        /**< Pitch of Second Frame */

        OMX_U32 FrmStartX;

        /**< Horizontal start offset */

        OMX_U32 FrmStartY;
```

```
        /**< Vertical start offset */
        OMX_U32 FrmCropWidth;
        /**< Crop Width */
        OMX_U32 FrmCropHeight;
        /**< Crop Height */
} OMX_CONFIG_VIDCHANNEL_RESOLUTION;
```

3. OMX_CONFIG_ALG_ENABLE OMX_TI_IndexConfigAlgEnable

Decryption: Flag to enable/disable the Algorithm/IP

```
typedef struct OMX_CONFIG_ALG_ENABLE {
        OMX_U32 nSize;
        /**< Size of the structure in bytes */
        OMX_VERSIONTYPE nVersion;
        /**< OMX specification version information */
        OMX_U32 nPortIndex;
        /**< Index of the port */
        OMX_U32 nChId;
        /**< Channel ID */
        OMX_BOOL bAlgBypass;
        /**< Algorithm/IP enable/disable flag */
} OMX_CONFIG_ALG_ENABLE;
```

4. OMX_CONFIG_SUBSAMPLING_FACTOR OMX_TI_IndexConfigSubSamplingFactor

Decryption: To set the Sub sampling factor of a specific port. This is used to drop the output frame rate of the component in the order of 1/2, 1/3, 1/4 etc. Please not that validated only for 1/2(reduce to half) configuration. The sub sampling factor shall be an integer. For example: 1 – no drop, 2 – alternative frame drops etc.

```
typedef struct OMX_CONFIG_SUBSAMPLING_FACTOR {
        OMX_U32 nSize;
        /**< Size of the structure in bytes */
        OMX_VERSIONTYPE nVersion;
        /**< OMX specification version information */
        OMX_U32 nPortIndex;
```

```
    /**< Index of the port */
    OMX_U32 nSubSamplingFactor;
    /**< Video frame rate sub sampling factor */
} OMX_CONFIG_SUBSAMPLING_FACTOR;
```

Chapter 4

Multimedia Sample Applications

This chapter explains OpenMax components sample applications available on the DM814x OpenMax Multimedia Sub-system. It explains the sample API sequence and component chaining.

4.1 Sample Demonstration Applications

In this SDK package, a sample demonstration of multimedia application is provided, which runs on EVM with EIO card attached to it. This is provided as ICONs on matrix launcher of SDK, where user can click on multimedia demo to see the application running. This Application is running on Host A8 processor as executables. These executables make use of OMX components running on media controller.

The sample demonstations are:

1. Decode_Display (as described in section 4.2.2)
2. Capture_Encode (as described in section 4.2.3)

4.2 Sample IL Clients / Applications for Development and OpenMax usage flow

4.2.1 Decode example

In this OMX release, a sample IL client program is provided at omx\demo\decode folder. This sample program shows the OpenMax APIs and its usage in context of Video decoder component. This application is built for cortex A8 processor running Linux.

Sample IL client is intended for decoding of a H264 or MPEG2 elementary bit stream. This example shows the flow of OpenMax APIs.

Omx_init initializes the DOMX required by OMX apis to be executed on media controller. This does the memory initialization. And setsup the shared regions as well. [Please note by default media controller firmware would not be loaded by this app., So care must be taken to load the firmware (with the utilities provided in SDK) before running the application]

§ **Component instantiation**

After doing the OMX init, decode component is created by calling the

OMX_GetHandle(&pHandle, (OMX_STRING) "OMX.TI.DUCATI.VIDDEC", pAppData, pAppData->pCb);

Component name is unique identifier for every component. In earlier section all components names have been described. Component expects callback functions (to IL client) to be provided during GetHandle call.

§ **Parameter settings**

After getting the handle component parameters are set by calling

OMX_SetParameter (pHandle, OMX_IndexParamPortDefinition, &pInPortDef)

This Api can take different indexes, as provided in header files. In this example d decoder width / height / framerate / buffer count etc is set by using OMX_IndexParamPortDefinition index. This is OpenMax standard index, structure of this index is available in header files provided in this SDK.

§ **OpenMax Port Enable**

After setting the parameters, ports of components are enabled by

OMX_SendCommand (pAppData->pHandle, OMX_CommandPortEnable, OMX_VIDDEC_INPUT_PORT, NULL);

OMX_SendCommand (pAppData->pHandle, OMX_CommandPortEnable, OMX_VIDDEC_OUTPUT_PORT, NULL);

[By default VDEC components ports are enabled, so it is optional that user enables the ports by calling this API]After enabling the ports, component state is changed from loaded (after GetHandle component is in loaded state) to IDLE state. This requires all buffers to be allocated before component can be moved to IDLE state. This is accomplished by OMX_AllocateBuffer.

§ **OpenMax Buffer Allocation**

API:

OMX_AllocateBuffer (pHandle, &pAppData->pInBuff[i], pAppData->pInPortDef->nPortIndex, pAppData, pAppData->pInPortDef->nBufferSize);

In this release, component on media controller allocates the buffers and provides the buffer header to IL client. Buffer header contains information about buffer pointer and associated data structure. [This release does not support buffer allocation done on IL client and supplied to component]. Component allocated buffer can be used by other component by using OMX_UseBuffer API.

§ **Data processing**

After buffers are allocated, component is moved to execute state, and component is ready to process buffers. IL client provides the buffer by calling following APIs **EmptyThisBuffer**(pHandle, pAppData->pInBuff[i]);

FillThisBuffer(pHandle,pAppData->pOutBuff[i]);

In this example a stream is read from file and parser provides the frames in each buffer. Il client provides this stream data by using EmptyThisBuffer call. Output buffers to components are provided by using FillThisBuffer APIs.

Component informs the IL Client by calling the callbacks provided during getHandle(), namely **FillBufferDone** and **EmptyBufferDone**.

After processing few frames in this sample application, component is moved back to idle and loaded state. Finally component handle is deleted by using OMX_FreeHandle() API.

- **Building the Application**

For Building the app, SDK needs to be installed on linux host machine. In OMX /package/makerules folder env.mk file is available, which sets the path for tools to build. PATH can also be specified in command line for build. “Make decode” would create the host binary in DEST_ROOT folder

- **Running the application**

For running the application following steps are required –

- Run the application

```
./decode_a8host_debug.xv5T -i sample.264 -w 1920
-h 1080 -o sample.yuv -c h264
```

4.2.2 Encode example

In this OMX release, a sample IL client program is provided at omx\demo\encode folder. This sample program shows the OpenMax APIs and its usage in context of Video decoder component. This application is built for cortex A8 processor running Linux.

Sample IL client is intended for encode of a YUV 420 data into H264 or MPEG4 or H263 elementary bit stream. This example shows the flow of OpenMax APIs.

Omx_init initializes the DOMX required by OMX apis to be executed on media controller. This does the memory initialization. And setsup the shared regions as well. [Please note by default media controller firmware would not be loaded by this app., So care must be taken to load the firmware (with the utilities provided in SDK) before running the application]

§ **Component instantiation**

After doing the OMX init, decode component is created by calling the

```
OMX_GetHandle(&pHandle, (OMX_STRING) "OMX.TI.DUCATI.VIDENC",
pAppData, pAppData->pCb);
```

Component name is unique identifier for every component. In earlier section all components names have been described. Component expects callback functions (to IL client) to be provided during GetHandle call.

§ **Parameter settings**

After getting the handle component parameters are set by calling

```
OMX_SetParameter (pHandle, OMX_IndexParamPortDefinition,
&pInPortDef)
```

This Api can take different indexes, as provided in header files. In this example d decoder width / height / framerate / buffer count etc is set by using OMX_IndexParamPortDefinition

index. This is OpenMax standard index, structure of this index is available in header files provided in this SDK.

§ **OpenMax Port Enable**

After setting the parameters, ports of components are enabled by

```
OMX_SendCommand ( pAppData->pHandle, OMX_CommandPortEnable, OMX_VIDENC_INPUT_PORT, NULL );

OMX_SendCommand ( pAppData->pHandle, OMX_CommandPortEnable, OMX_VIDENC_OUTPUT_PORT, NULL );
```

[By default VENC components ports are enabled, so it is optional that user enables the ports by calling this API]After enabling the ports, component state is changed from loaded (after GetHandle component is in loaded state) to IDLE state. This requires all buffers to be allocated before component can be moved to IDLE state. This is accomplished by OMX_AllocateBuffer.

§ **OpenMax Buffer Allocation**

API:

```
OMX_AllocateBuffer (pHandle, &pAppData->pInBuff[i], pAppData->pInPortDef->nPortIndex, pAppData, pAppData->pInPortDef->nBufferSize);
```

In this release, component on media controller allocates the buffers and provides the buffer header to IL client. Buffer header contains information about buffer pointer and associated data structure. [This release does not support buffer allocation done on IL client and supplied to component]. Component allocated buffer can be used by other component by using OMX_UseBuffer API.

§ **Data processing**

After buffers are allocated, component is moved to execute state, and component is ready to process buffers. IL client provides the buffer by calling following APIs **EmptyThisBuffer**(pHandle, pAppData->pInBuff[i]);

FillThisBuffer(pHandle,pAppData->pOutBuff[i]);

In this example a stream is read from file and parser provides the frames in each buffer. Il client provides this stream data by using EmptyThisBuffer call. Output buffers to components are provided by using FillThisBuffer APIs.

Component informs the IL Client by calling the callbacks provided during getHandle(), namely **FillBufferDone** and **EmptyBufferDone**.

After processing few frames in this sample application, component is moved back to idle and loaded state. Finally component handle is deleted by using OMX_FreeHandle() API.

- **Building the Application**

For Building the app, SDK needs to be installed on linux host machine. To build the examples components must be pre-built. In the top level SDK folder, “make components” would build the components required for examples. “make examples” would create the decode app binary in folder component-sources/omx_05_xx_yy_bb/rebuilt-binaries/decode folder.

- **Running the application**

For running the application following steps are required (By default in the init scripts of Linux in /etc/init/rc5.d folder, firmware and module will be getting loaded)

- Run the application

```
./encode_a8host_debug.xv5T  -i  sample.yuv  -w  1920
-h 1080 –f 30 –b 1000000 -o sample.h264 -c h264
```

4.2.3 Decode_display example

This example uses three OMX components VDEC, VFPC, VFDC for creating a simple application, which can decode an H264 elementary stream and scale and display it.
This application is an IL Client running on A8 processor with Linux operating system. Decoder component runs on media controller HDVICP2, while scalar and display component runs on media controller HDVPSS part of DM8148. OpenMax VDEC component takes a buffer, which contains single frame of H264 elementary stream, and after decoding gives the buffer to A8. Elementary stream chunking logic is implemented as H264 stream parser, running on A8.

Data flow is implemented in IL client as different threads, processing buffers from neighboring component. For details please refer to section IL Client design.

The application can be configured via cmd line arguments for display to be configured in IL Client to display video on LCD display or the on-chip HDMI port. The cmd line arguments points to the LCD when triggred from the matrix launcher.

If configured to use the LCD display, the scalar is configured to output a frame which will fit LCD display. When using the On-chip HDMI as the display device, the scalar configured such that no scaling occurs.

This application takes width, height, frame rate, display Id and file name as input argument. Frame rate control allows decoder to run at frame rate specified as argument. Display is running at 60 frames / second. The max frame rate supported is 30 frames per second.\. So the display is of decoded stream's video frame size. Scalar parameters can be adjusted for scaling the decoded stream. For details on parameters, please refer to VFPC section in OMX components chapter.

Figure 2 depicts the data flow between A8, decode, scalar and display components.

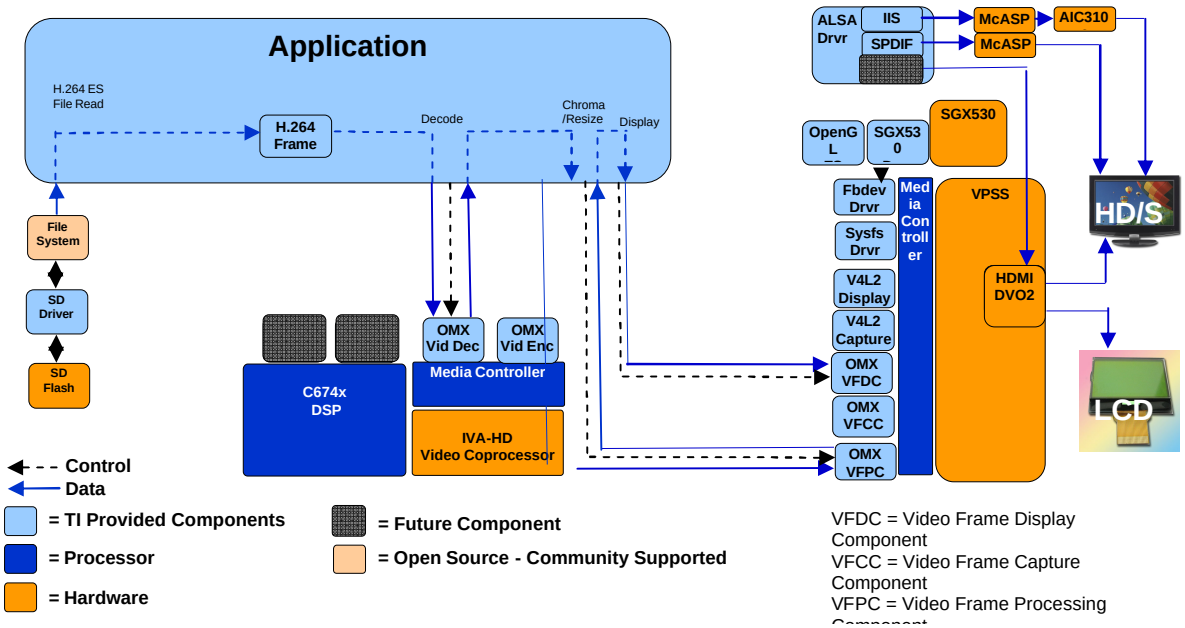


Figure 1 decode and display components

- **Building application**

For Building the app, SDK needs to be installed on linux host machine. In OMX /package/makerules folder env.mk file is available, which sets the path for tools to build. PATH can also be specified in command line for build.
Make decode_display would create the host binary in DEST_ROOT folder.

- **Running the application for display on LCD**

For running the application following steps are required –

- Run the application

```
./decode_display_a8host_debug.xv5T   -w   1920   -h
1080 -f 30 -i titv.264 -g 0 -d 1
```

- **Running the application for display on on-chip HDMI**

For running the application following steps are required –

- Run the application

```
./decode_display_a8host_debug.xv5T   -w   1920   -h
1080 -f 30 -i titv.264 -g 0 -d 0
```

Note:

1. As for the on-chip HDMI, the default mode is 1080p-60 configuring the display0 is not necessary. For more details on how to configure the display0, please refer to the PSP VPSS user guide.

4.2.4 display example

This example demonstrates, how to use SD/HDMI/LCD display. This example generates the color bar and provides that as input to display. User should see color bar on display.

- **Buidling the Application**

For Building the app, SDK needs to be installed on linux host machine. To build the examples components must be pre-built. In the top level SDK folder, “make components” would build the components required for examples. “make examples” would create the display app binary in folder component-sources/omx_05_02_00_xx/rebuilt-binaries/display folder

- **Running the application**

For running the application following steps are required (By default in the init scripts of Linux in /etc/init/rc5.d folder, firmware and module will be getting loaded)

./display_a8host_debug.xv5T -d 0/1/2 (LCD for DM8148 only)

4.2.5 Decode_mosaicdisplay example

Please refer following link for the details.

http://processors.wiki.ti.com/index.php/OMX_EZSDK_Examples#Decode_MosaicDisplay

4.2.6 Capture_encode example

This example uses four OMX components VENC, VFPC, VFDC, VFCC for creating a simple application, which can capture and encode it to an H264 elementary stream and also display it. In this application, capture component which is running on HDVPSS part of media controller captures the 1080p60/720p60 input from TVP7002 decoder on Catalog EIO board. Aleternate frames captured from the capture device are dropped to get 30fps output. This data is fed to DEI component, which produces two outputs. One output is given to display component while other is fed to encoder component running on HDVICP2. Since capture is being done with 420 progressive formats, DEI algorithm is not turned ON. For interlace capture, DEI can be used for de-interlacing. This application takes mode, frame rate, bitrate and file name as input argument. Application can be configured to output to the LCD display on the EIO board or the on-Chip HDMI. When triggered via Matrix Launcher, the display device chosen is the LCD.
. Encoder bit rate can be varied through arguments passed in the application. More parameters for encoding can be changed in IL client though OMX APIs.

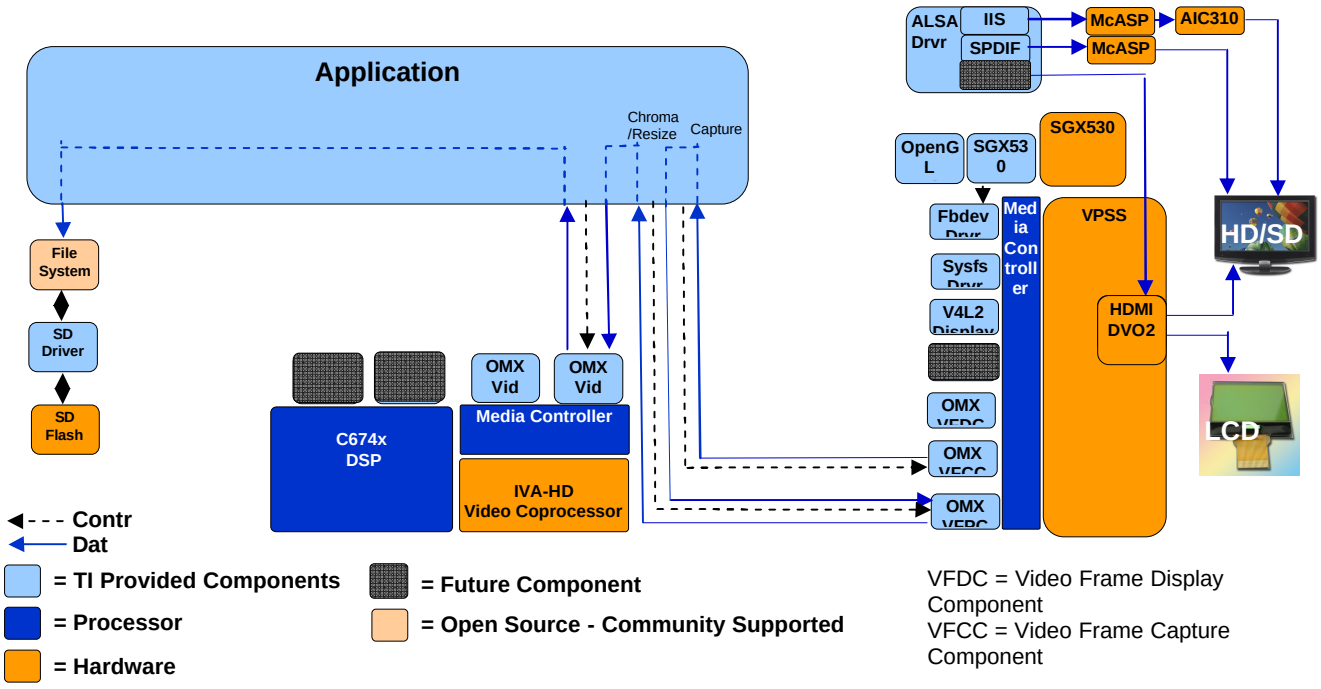


Figure 2 capture and encode/display components

- **Buidling the Application**

For Building the app, SDK needs to be installed on linux host machine. In OMX /package/makerules folder env.mk file is available, which sets the path for tools to build. PATH can also be specified in command line for build. Make decode_display would create the host binary in DEST_ROOT folder

- **Running the application for display on LCD**

For running the application following steps are required –

- Run the application

```
./capture_encode_a8host_debug.xv5T -o sample.h264 -m
1080p -f 30 -b 1000000 -n 1000 –d 1
```

- **Running the application for display on on-chip HDMI**

For running the application following steps are required –

- Run the application

```
./capture_encode_a8host_debug.xv5T -o sample.h264
-m 1080p -f 30 -b 1000000 -n 1000
```

Note:

1. To enable 720p60 capture and display on-chip HDMI, the following additional commands will be required:

```
echo 0 > /sys/devices/platform/vpss/display0/enabled

echo 720p-60 > /sys/devices/platform/vpss/display0/mode

echo 1 > /sys/devices/platform/vpss/display0/enabled
```

This should be done after HDMI and VPSS kernel drivers have been inserted.

The application command line changes to this:

```
./capture_encode_a8host_debug.xv5T -o sample.h264 -m
720p -f 30 -b 1000000 -n 1000
```

This will configure the display for the 720p-60 mode. If this is not done, then the captured frame will be displayed on the left-corner of the screen.

2. For display on LCD, only the –m value needs to changed as the LCD display supports only 1 set of timing parameters.

4.2.7 IL Client design details for decode_display and capture_encode examples

This section would describe in brief the APIs used in these examples. More Details of component and parameters are descried in OMX USER guide present in SDK.

OpenMax components typically require following OMX API sequence –

1. Create a component

OMX API : OMX_GetHandle()

This API is required to be called for each component. This API takes the component name as parameters and creates a particular OMX component.

2. Set the parameters

OMX API: OMX_SetParameter() / OMX_SetConfig()

This API takes the specific component handle, an index and structure pointer corresponding to parameters supported by the component. Index and corresponding structures are defined in OMX interface header files.

3. Enable the ports

OMX API: OMX_SendCommand()

Before allocating the buffers, ports must be enabled for each component. In the release, encoder/decoder ports are enabled by default so it is not mandatory for encoder/decoder. This API would take parameter as port index and command index as OMX_CommandPortEnable.

4. Change State to IDLE

OMX API: OMX_SendCommand()

This API takes specific component handle, and specific state to be changed. As per OMX standard buffers are allocated during loaded to idle state transition. In the SDK, conventionally output port of a component provides the buffers for input port of connected component. So in these examples buffers are allocated on output port, and same buffer is supplied to input port of other component. All data buffers in OMX is specified by standard OMX buffer header. Each component allocated buffer header of each buffer, which is used in the component.

5. Allocate buffers / buffer headers

OMX APIs: OMX_AllocateBuffer() / OMX_UseBuffer()

After moving the component to idle state, buffers allocation APIs are invoked for buffer allocation. In the SDK, all buffers are allocated by media comtroller in response to OMX_AllocateBuffer API. As described above buffer are allocated on output port of a component, and informed to connect component by means of OMX_UseBuffer() API. If a component's input port is not connected to any other component, OMX_AllocateBuffer() API is called on input port as well. In the decode example, as the input port of decode is not connected to any other component, OMX_AllocateBuffer() APIs is invoked on input port as well**.** In response to AllocateBuffer / USeBuffer APIs component returns a buffer header corresponding to that buffer. IL Client uses these buffer headers to keep track of buffers.

6. Change State to EXECUTE

OMX API: OMX_SendCommand()

This API is same as changing state to idle. This would take parameter as state to be changed as 'execute'. After transitioning to execute state component is ready to take the buffers from IL client.

Following flow diagram in figure 4 depicts the OMX component creation and state changes.

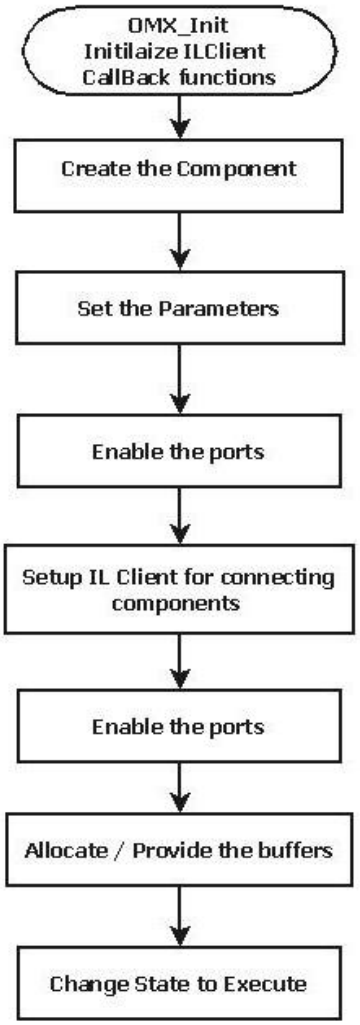


Figure 3 OpenMax component creating and state change flow

7. Start sending / receiving data to/from Components.

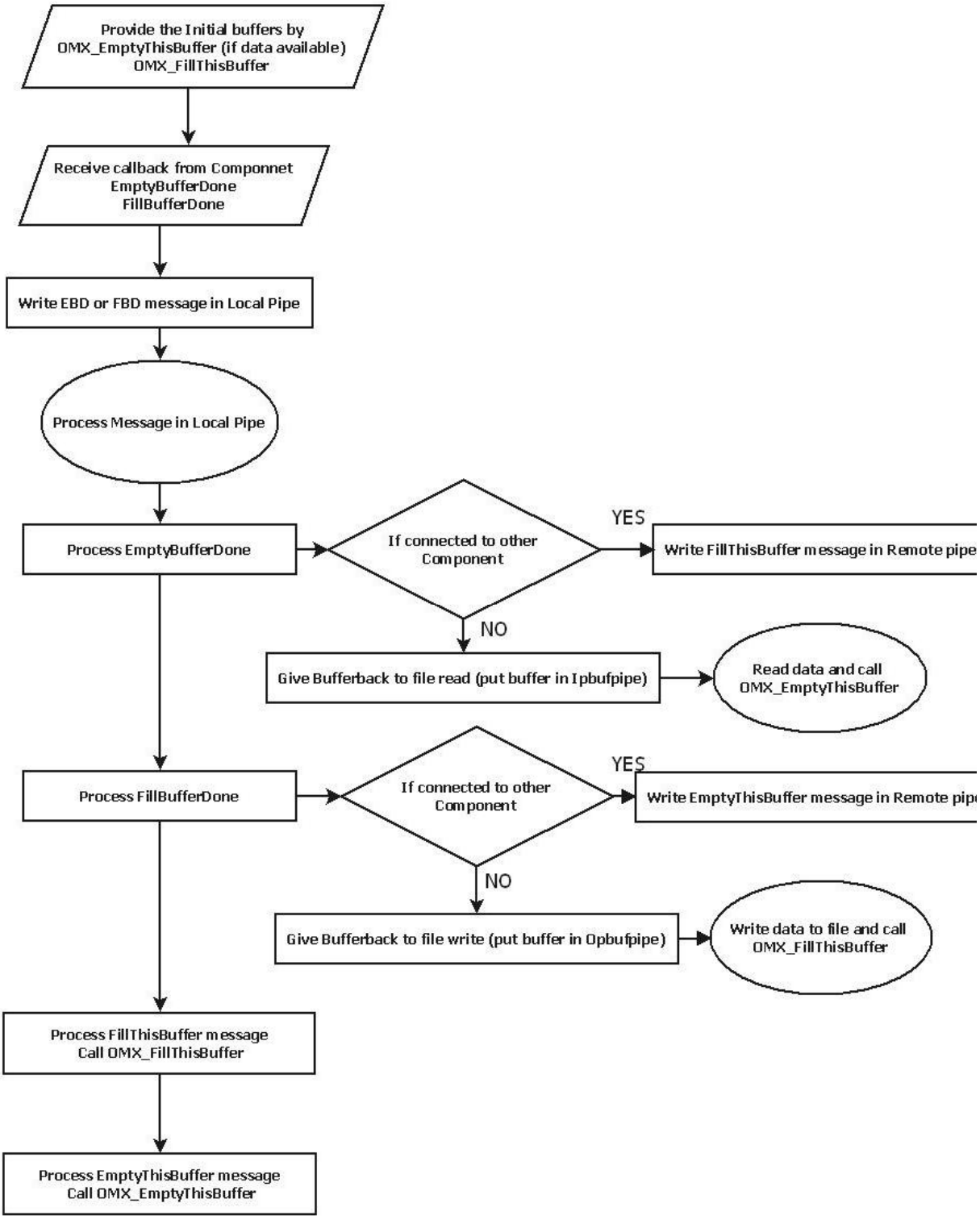


Figure 4 Data flow in IL Client

This is accomplished by following 4 OMX APIs.

i. OMX API : OMX_EmptyThisBuffer()

IL Client provides a filled data buffers (for e.g. bit-stream data) to the component by calling this API. This API takes specific component handle and a buffer header to the component. This API is called at the input port of a component.

ii. OMX API: OMX_FillThisBuffer()

IL Client provide the empty buffer at the output port of a component, by calling this API. This API also takes specific component handle and buffer header for output port of that component.

iii. OMX_EmptyBufferDone : callback by component

This is callback function, which is invoked by the component. This is implemented as an event handler function in IL Client and provided to component during GetHandle() API. Based on notification from component, IL Client can take decision of refilling this component or providing to other component.

iv. OMX_FillBufferDone: callback by component

This is also callback function, implemented as event handler by IL Client. This is invoked by component, whenever output data is ready to be sent out. This callback is same as EmptyBufferDone callback, and component handle and application private pointer is supplied to distinguish between different components.

In the examples, IL client creates a thread (*IL_ClientConnInConnOutTask*) for each component, which can send/receive buffers from other component. As described above, this thread is responsible for calling / acting on the above 4 APIs. This thread is designed to read a message from a pipe called as 'local pipe' and based on message use one of the above mentioned 4 APIs. Local pipe of each component is populated by two ways.

i. By component's own callback functions
ii. By connected component callback function

Since initially there will not be any messages in the 'local pipe', thread provides the initial buffers to the component as one time initialization. As component processes the buffers, it starts writing into local pipe by callback functions. When a callback (from component) is received as EmptyBufferDone, a message as "EBD" is written into local pipe. Similarly when a callback (from component) is received as FillBufferDone(), a message as " FBD" is written into local pipe. The thread, which reads the local pipe checks these messages, and takes appropriate action, based on the connection status of the port on which it received these callbacks.

If the port is connected to another component, it takes following action

i. **Message "EBD"** – this message informs the thread that, a buffer has been consumed at the input port a component, and it is ready to be recycled. IL Client checks the connection status of this port (status maintained in IL Client data structure), and if it is connected to another component, it informs other component that this buffer can be use at output port of the connected component. To accomplish this, a message as FillThisBuffer() / FTB is written into connected component's local pipe. IL client maintains the local pipe of connected component as "remote pipe" variable in each component.

ii. **Message "FBD"** – This message informs the thread that, a buffer has been produced by the component and can be consumed by other component. If this component is connected to other component, IL client writes a message as " EmptyThisBuffer / ETB into other component's local pipe. (referred as remote pipe in the component, where callback is received)

So “local pipe” for each component, takes the message for ETB/FTB/EBD/FBD and acts on its component by calling OMX APIs.

For terminal component ports such as Input port of decoder / output port of encoder, which is not connected to any other component, but does file read /write, a separate thread is created, which reads/writes into a file. These threads in IL Clients are referenced as *IL_ClientInputBitStreamReadTask()* and *IL_ClientOutputBitStreamWriteTask()*.

a. ***IL_ClientInputBitStreamReadTask()*** – This thread reads a H264 elementary stream parses it , and provides a single frame of data in a buffers to the component. To keep track and recycling of these buffers a pipe called ipBufPipe is used to hold the buffer headers. Callback function “IL_ClientCbEmptyBufferDone” checks for port status, and if it is not connected it writes the buffer header into ipBufPipe of the decoder component.

b. ***ClientOutputBitStreamWriteTask()*** . This thread takes the output buffers from encoder and wite the bitstream in a file. IL_ClientCbFillBufferDone() callback, checks for the port connection status and if it is not connected to any other component (as the case for encoder), it writes the buffer headers into opBufPipe.

In both the above cases, initial buffers are provided to component by above threads as one time initialization process. Above described methodology for buffer communication is summarized in Figure 6 for decode_display examples, and in Figure 7 for capture_encode example.

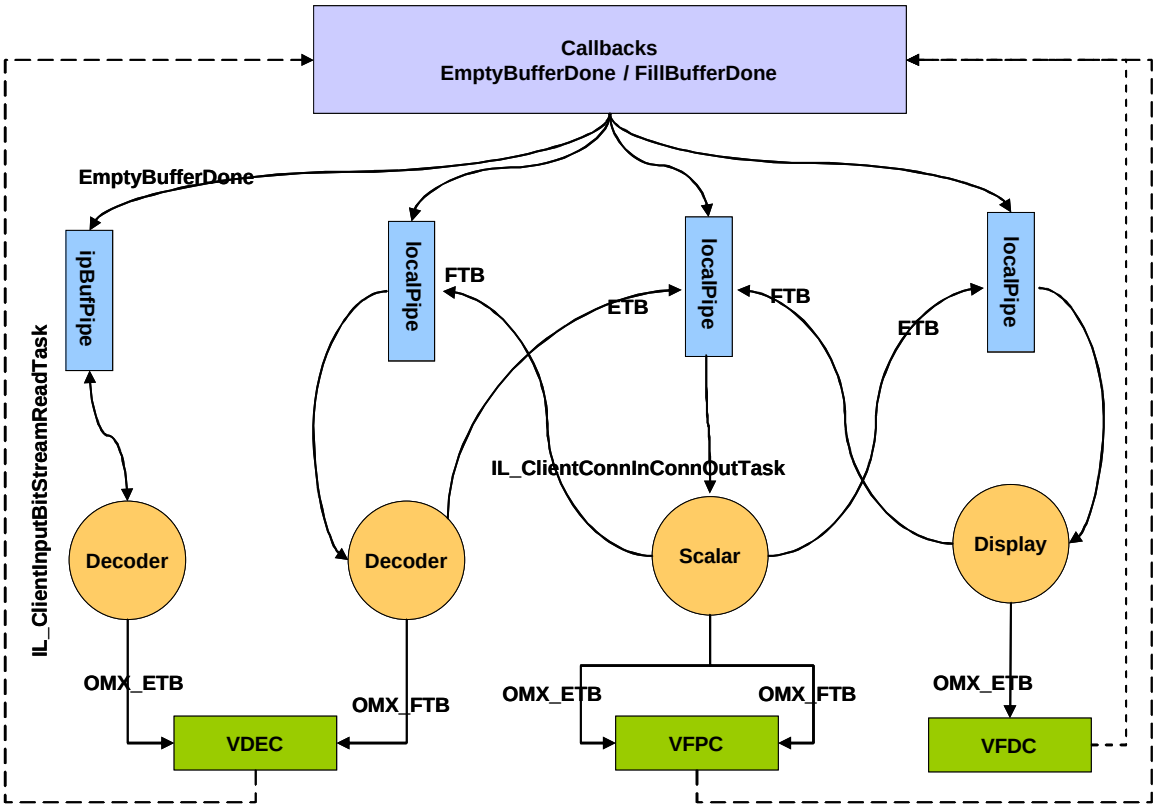


Figure 5 Data Flow diagram for Decode – Display sample application

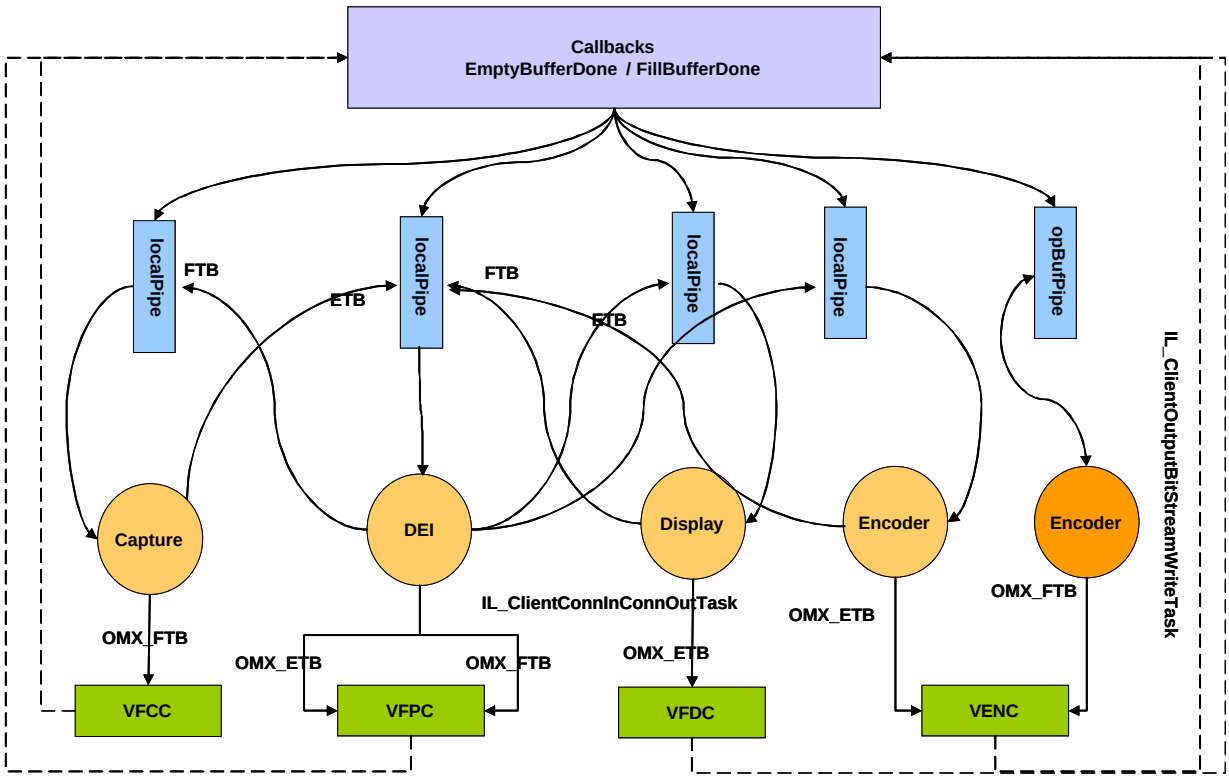


Figure 6. Data Flow diagram for Capture – Encode sample application

- **Tear down sequence -**

For terminating the application, OMX component state machines are changed to loaded state before deleting the component, so that all buffers are freed up. Figure 8 depicts the tear-down sequence.

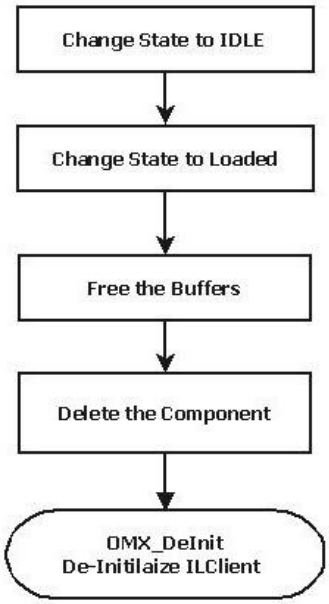


Figure 7 Tear down sequence

Chapter 5

DSP OpenMax example

This section describes an example OpenMax component on DSP. It also briefly explains the sample API sequence.

5.1.1 MP3 Decoder Integration

Integration details for MP3 decoder are available on Wiki at
http://processors.wiki.ti.com/index.php/MP3_Decoder_Integration_in_EZSDK

5.1.2 Audio Decode example

In this OMX release, a sample IL client program is provided at omx\demos\adec_snt folder. This sample program shows the OpenMax APIs and its usage in context of Audio decoder component. This application is built for cortex A8 processor running Linux.

Sample IL client is intended for decoding of a MP3 or AAC elementary bit stream. **Currently the decoded output does not playout, instead decoded output is written back to a file in 16bit linear format (Intel format). Playing out the PCM is scoped for future EZSDK release.**

This example shows the flow of OpenMax APIs.

Omx_init initializes the DOMX required by OMX apis to be executed on media controller. This does the memory initialization. And sets up the shared regions as well. [Please note by default media controller firmware would not be loaded by this app., So care must be taken to load the firmware (with the utilities provided in SDK) before running the application]

§ **Component instantiation**

After doing the OMX init, decode component is created by calling the

```
OMX_GetHandle(&pHandle, (OMX_STRING) "OMX.TI.DSP.AUDDEC", pAppData, pAppData->pCb);
```

Component name is unique identifier for every component. In earlier section all components names have been described. Component expects callback functions (to IL client) to be provided during GetHandle call.

§ **Parameter settings**

After getting the handle component parameters are set by calling

```
OMX_SetParameter (pHandle, OMX_IndexParamPortDefinition, &pInPortDef)
```

This Api can take different indexes, as provided in header files. In this example decoder audio format / buffer size etc is set by using OMX_IndexParamPortDefinition index. This is OpenMax standard index, structure of this index is available in header files provided in this SDK.

§ **OpenMax Port Enable**

After setting the parameters, ports of components are enabled by

```
OMX_SendCommand ( pAppData->pHandle, OMX_CommandPortEnable, OMX_AUDDEC_INPUT_PORT, NULL );

OMX_SendCommand ( pAppData->pHandle, OMX_CommandPortEnable, OMX_AUDDEC_OUTPUT_PORT, NULL );
```

[By default ADEC components ports are enabled, so it is optional that user enables the ports by calling this API]After enabling the ports, component state is changed from loaded (after GetHandle component is in loaded state) to IDLE state. This requires all buffers to be allocated before component can be moved to IDLE state. This is accomplished by OMX_AllocateBuffer.

§ **OpenMax Buffer Allocation**

API:

```
OMX_AllocateBuffer (pHandle, &pAppData->pInBuff[i], pAppData->pInPortDef->nPortIndex, pAppData, pAppData->pInPortDef->nBufferSize);
```

In this release, component on media controller allocates the buffers and provides the buffer header to IL client. Buffer header contains information about buffer pointer and associated data structure. [This release does not support buffer allocation done on IL client and supplied to component]. Component allocated buffer can be used by other component by using OMX_UseBuffer API.

§ **Data processing**

After buffers are allocated, component is moved to execute state, and component is ready to process buffers. IL client provides the buffer by calling following APIs **EmptyThisBuffer**(pHandle, pAppData->pInBuff[i]);

FillThisBuffer(pHandle,pAppData->pOutBuff[i]);

In this example, bitstream data is read from file and data is copied into input buffer. IL client provides this stream data by using EmptyThisBuffer call. Output buffers to components are provided by using FillThisBuffer APIs.

Component informs the IL Client by calling the callbacks provided during getHandle(), namely **FillBufferDone** and **EmptyBufferDone**.

After processing all the frames in the input file, component is moved back to idle and loaded state. Finally component handle is deleted by using OMX_FreeHandle() API.

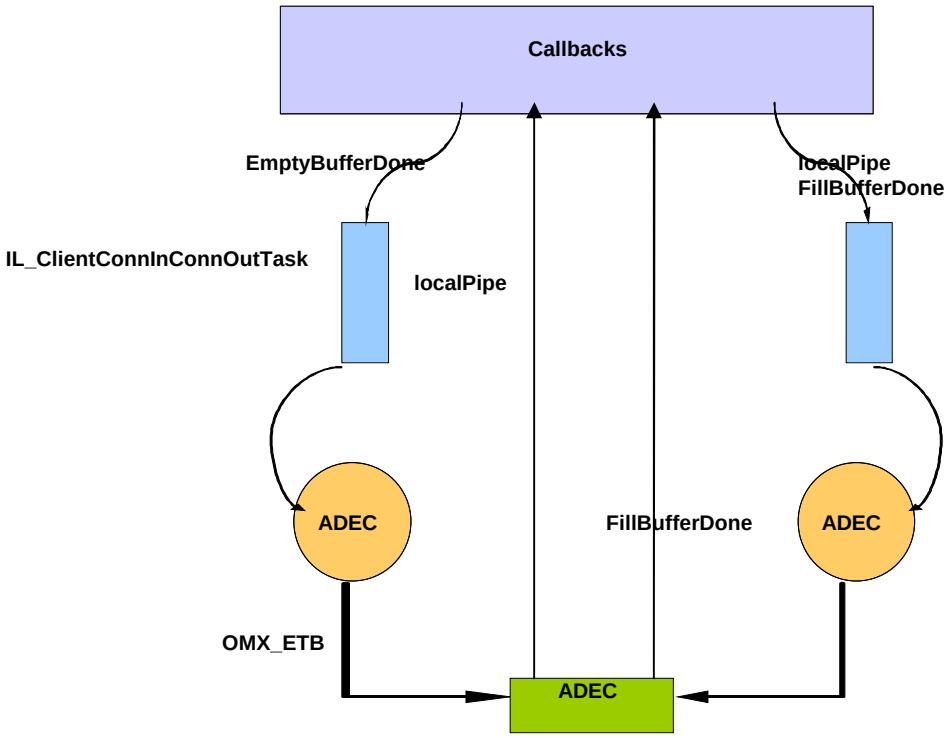


Figure 4. Data Flow diagram for ADEC SNT Audio application

- **Building the Application**

For Building the app, SDK needs to be installed on linux host machine. To build the examples, components must be pre-built. In the top level SDK folder, “make omx” would build the dsp executable and the IL Client.

- **Running the application**

For running the application following steps are required (By default in the init scripts of Linux in /etc/init/rc5.d folder, firmware and module will be getting loaded, so following is required only if it is disabled in rc5.d scripts)

–

- Insert syslink

```
insmod syslink.ko
```

- Load the Firmware using firmware_loader utility provided in SDK. (filesystem\user\bin)

i. `firmware_loader 0 /usr/share/ti/ti-media-controller-utils/dm814x_c6xdsp.xe674 start`

ii. `firmware_loader 1 /usr/share/ti/ti-media-controller-utils/dm814x_hdvicp.xem3 start`

iii. `firmware_loader 2 /usr/share/ti/ti-media-controller-utils/dm814x_hdvpss.xem3 start`

- Run the application: The first argument is the input file name to be decoded; the second argument is the decoded output file name. The third argument defines the codec type [mp3, aaclc]. Fourth says format (check help of example), and 5th defines sampling rate

./adec_snt_a8host_debug.xv5T –i sample.aac –o output.pcm –c aaclc -r 1 -s 48000

5.1.3 Audio Encode example

In this OMX release, a sample IL client program is provided at omx\demos\audio_encode folder. This sample program shows the OpenMax APIs and its usage in context of Audio encoder component. This application is built for cortex A8 processor running Linux.

Sample IL client is intended for encoding of a PCM elementary bit stream into aaclc encode**.** [**Please note by default AACLC encode is not present in dsp binary, it needs to be integrated as described in next section, before testing the IL Client**.]

This example shows the flow of OpenMax APIs.

Omx_init initializes the DOMX required by OMX apis to be executed on media controller. This does the memory initialization. And sets up the shared regions as well. [Please note by default media controller firmware would not be loaded by this app., So care must be taken to load the firmware (with the utilities provided in SDK) before running the application]

§ **Component instantiation**

After doing the OMX init, decode component is created by calling the

```
OMX_GetHandle(&pHandle, (OMX_STRING) "OMX.TI.DSP.AUDENC", pAppData, pAppData->pCb);
```

Component name is unique identifier for every component. In earlier section all components names have been described. Component expects callback functions (to IL client) to be provided during GetHandle call.

§ **Parameter settings**

After getting the handle component parameters are set by calling

```
OMX_SetParameter (pHandle, OMX_IndexParamPortDefinition, &pInPortDef)
```

This Api can take different indexes, as provided in header files. In this example decoder audio format / buffer size etc is set by using OMX_IndexParamPortDefinition index. This is OpenMax standard index, structure of this index is available in header files provided in this SDK.

§ **OpenMax Port Enable**

After setting the parameters, ports of components are enabled by

```
OMX_SendCommand ( pAppData->pHandle, OMX_CommandPortEnable, OMX_AUDENC_INPUT_PORT, NULL );

OMX_SendCommand ( pAppData->pHandle, OMX_CommandPortEnable, OMX_AUDENC_OUTPUT_PORT, NULL );
```

[By default AENC components ports are enabled, so it is optional that user enables the ports by calling this API]After enabling the ports, component state is changed from loaded (after GetHandle component is in loaded state) to IDLE state. This requires all buffers to be allocated before component can be moved to IDLE state. This is accomplished by OMX_AllocateBuffer.

§ **OpenMax Buffer Allocation**

API:

```
OMX_AllocateBuffer (pHandle, &pAppData->pInBuff[i], pAppData->pInPortDef->nPortIndex, pAppData, pAppData->pInPortDef->nBufferSize);
```

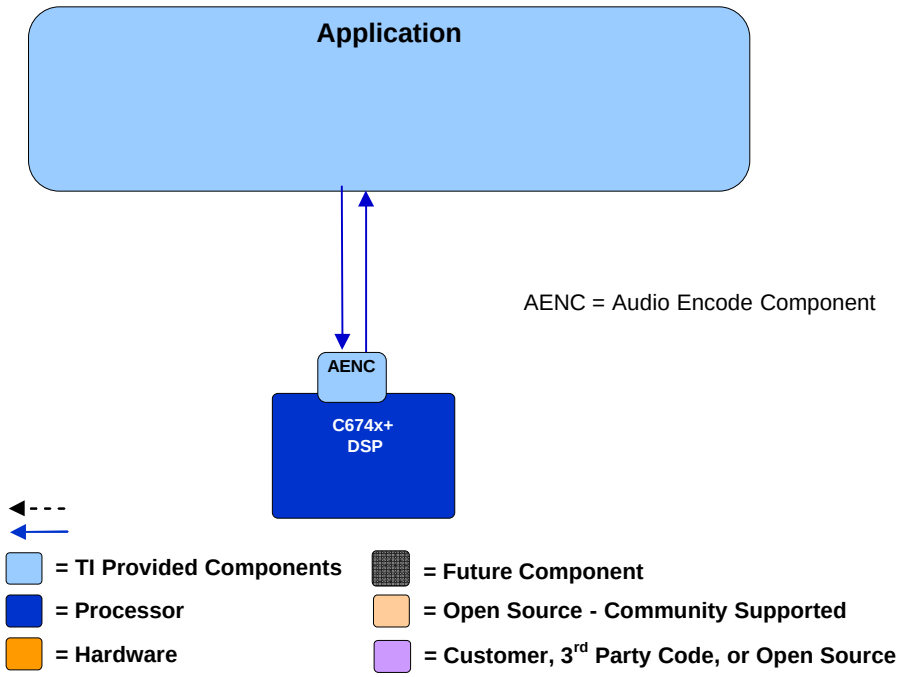
In this release, component on media controller allocates the buffers and provides the buffer header to IL client. Buffer header contains information about buffer pointer and associated data structure. [This release does not support buffer allocation done on IL client and supplied to component]. Component allocated buffer can be used by other component by using OMX_UseBuffer API.

§ **Data processing**

After buffers are allocated, component is moved to execute state, and component is ready to process buffers. IL client provides the buffer by calling following APIs **EmptyThisBuffer**(pHandle, pAppData->pInBuff[i]);

FillThisBuffer(pHandle,pAppData->pOutBuff[i]);

In this example, bitstream data is read from file and data is copied into input buffer. IL client provides this stream data by using EmptyThisBuffer call. Output buffers to components are provided by using FillThisBuffer APIs.



Component informs the IL Client by calling the callbacks provided during getHandle(), namely **FillBufferDone** and **EmptyBufferDone**.

After processing all the frames in the input file, component is moved back to idle and loaded state. Finally component handle is deleted by using OMX_FreeHandle() API.

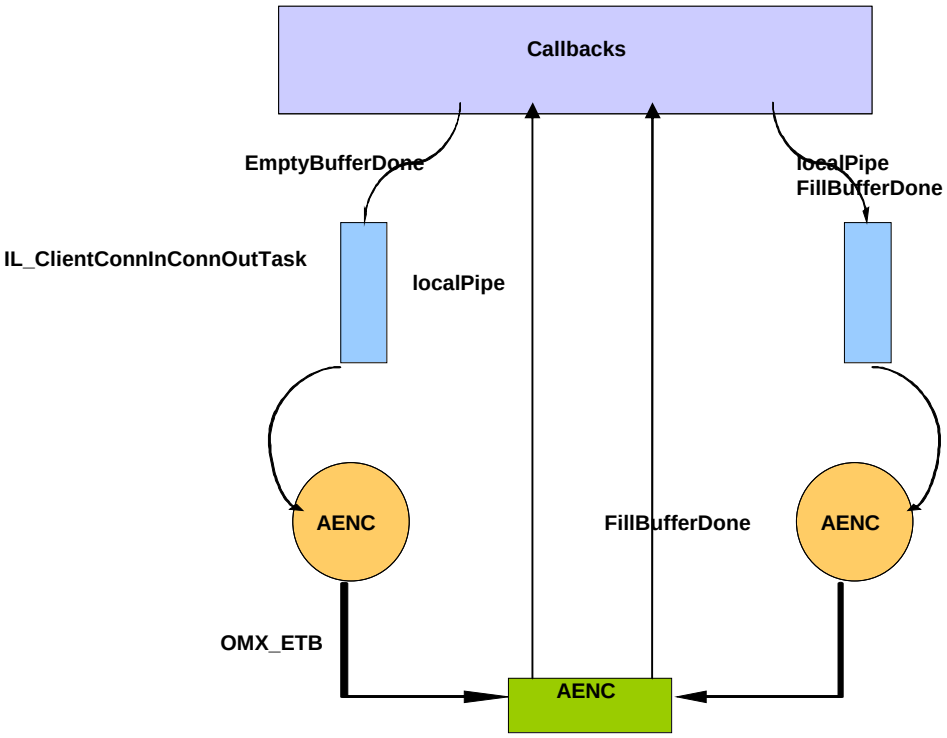


Figure 4. Data Flow diagram for Audio_Encode Audio application

- **Building the Application**

For Building the app, SDK needs to be installed on linux host machine. To build the examples, components must be pre-built. In the top level SDK folder, “make omx” would build the dsp executable and the IL Client.

- **Running the application**

For running the application following steps are required (By default in the init scripts of Linux in /etc/init/rc5.d folder, firmware and module will be getting loaded, so following is required only if it is disabled in rc5.d scripts)

–

- Insert syslink

```
modprobe syslink
```

- Load the Firmware using firmware_loader utility provided in SDK. (filesystem\user\bin)

i. `firmware_loader 0 /usr/share/ti/ti-media-controller-utils/dm814x_c6xdsp.xe674 start`

ii. `firmware_loader 1 /usr/share/ti/ti-media-controller-utils/dm814x_hdvicp.xem3 start`

iii. `firmware_loader 2 /usr/share/ti/ti-media-controller-utils/dm814x_hdvpss.xem3 start`

- Run the application: The first argument is the input file name to be decoded; the second argument is the decoded output file name. The third argument defines the codec type [aaclc], fourth is number of channels, fifth is bit rate, sixth is sampling rate and last is algorithm, as in following.

```
./audio_encode_a8host_debug.xv5T -i input.pcm -o output.aac -c aaclc -n 2
-b 192000 -s 44100 -f ADTS
```

5.1.4 Integrating AACLC encode

Integration details for AACLC encoder are available on Wiki at

http://processors.wiki.ti.com/index.php/OMX_AAC_LC_Encoder_Integration_in_EZSDK

5.1.5 VLPB example

This section describes an example OpenMax component on DSP. It also briefly explains the sample API sequence.

About OpenMax Video LooPBack Component (VLPB):

VLPB stands for Video Loop Back Component. The component name is "OMX.TI.C67X.VLPB". As is clear from the name, the component runs on the DSP (C67x). This component has 16 input ports and 16 output ports. It copies a buffer on its input port to a buffer on the output port. There is a one to one correspondence between the input and output ports. Therefore, essentially this a copy component running on the DSP

About the OpenMax IL-Client (c6xtest):

In this SDK, a sample application 'C' program is provided at omx\demos\c6xtest folder. This sample program shows the OpenMax APIs and its usage. This application is built for cortex A8 processor running Linux.

This example uses the OpenMax component VLPB for creating a simple application, which can copy a buffer from one memory location to another. This application does not take any additional parameters as input argument. In this example, only one input and one output port of the VLPB component is used.

This examples initially does the platform_init. This does the memory initialization. And sets up the shared regions. Omx_init loads the dsp firmware (if enabled), and initializes the DOMX required by OMX apis to be executed on dsp. [Please note by default dsp firmware would not be loaded by this app., So care must be taken to load the firmware (with the utilities provided in SDK) before running the application]

In case user is interested in loading the binaries by this application itself, c6xtest application must be rebuilt by changing the setting the following variable to '1' in app_cfg.h file (in same folder as c6xtest source files)

#define DOMX_CORE_DOPROCINIT (**1**)

After doing the OMX init, vlpb component is created by calling the

OMX_GetHandle (&pAppData->pVlpbHandle, (OMX_STRING) "OMX.TI.C67X.VLPB", pAppData->vlpbILComp, &pAppData->pCb);;

Component name is a unique identifier for every component. Component expects callback functions (to IL client) to be provided during GetHandle call. Asfter GetHandle call, the component is in LOADED state.

After getting the handle component parameters are set by calling

OMX_SetParameter (pHandle, OMX_IndexParamPortDefinition, &pInPortDef)

This Api can take different indexes, as provided in header files. In this example width / height / buffer count etc is set by using OMX_IndexParamPortDefinition index. This is OpenMax standard index, structure of this index is available in header files provided in this SDK.

After setting the parameters, ports of components are enabled by

OMX_SendCommand (pHandle, OMX_CommandPortEnable, OMX_VLPB_INPUT_PORT_START_INDEX, NULL);

OMX_SendCommand (pHandle, OMX_CommandPortEnable, OMX_VLPB_OUTPUT_PORT_START_INDEX, NULL);

[By default components ports are disabled, so it is required that user enables the ports by calling this API]

After enabling the ports, component state is changed from LOADED to IDLE state. This requires all buffers to be allocated before component can be moved to IDLE state. This is accomplished by

OMX_AllocateBuffer (pHandle, &pAppData->pInBuff[i], pAppData->pInPortDef->nPortIndex, pAppData, pAppData->pInPortDef->nBufferSize);

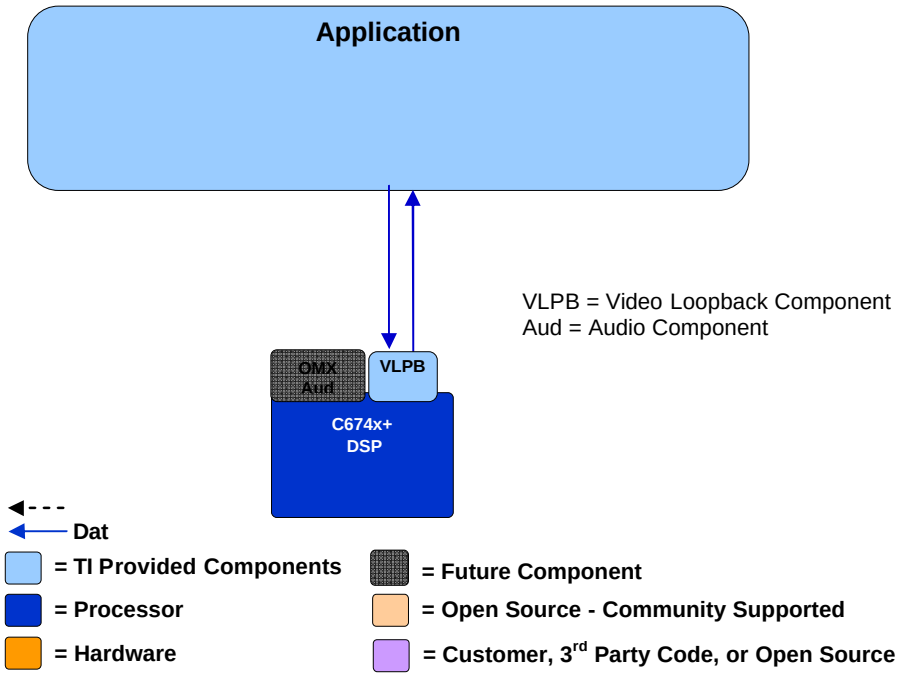
In this release, component on dsp allocates the buffers and provides the buffer header to IL client. Buffer header contains information about buffer pointer and associated data structure. [This release does not support buffer allocation done on IL client and supplied to component]. Component allocated buffer can be used by other component by using OMX_UseBuffer API.

After buffers are allocated, component is moved to execute state, and component is ready to process buffers. IL client provides the buffer by calling following APIs

OMX_EmptyThisBuffer(pHandle, pAppData->pInBuff[i]);

OMX_FillThisBuffer(pHandle,pAppData->pOutBuff[i]);

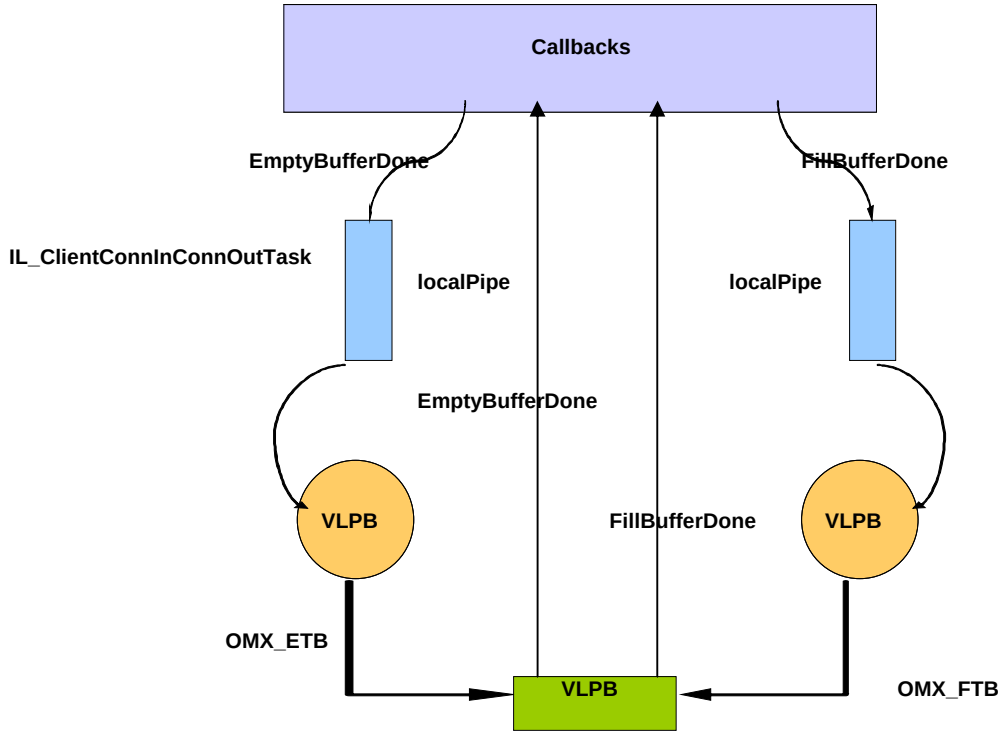
In this example a buffer of size **IL_CLIENT_VLPB_BUFFER_SIZE** is filled with **IL_CLIENT_VLPB_PATTERN** and is passed as the input buffer. IL client provides this buffer by using OMX_EmptyThisBuffer call. Output buffers to components are provided by using OMX_FillThisBuffer APIs. Component informs the IL Client by calling the callbacks provided during GetHandle(), namely **FillBufferDone** and **EmptyBufferDone**. After processing **IL_CLIENT_VLPB_MAX_FRAMES** frames in this sample application, component is moved back to idle and loaded state. Finally component handle is deleted by using OMX_FreeHandle() API. The constants **IL_CLIENT_VLPB_xxx** are defined in ilclient_utils.h



IL Client Design:

The IL Client design is similar to those documented in Chapter 4. The flow diagram is given below

Figure 3. Data Flow diagram for c6xtest – DSP sample application



- **Building the Application**

For Building the app, SDK needs to be installed on Linux host machine. To build the examples, components must be pre-built. In the top level SDK folder, “make components” would build the components required for examples. “make omx” would create the c6xtest app (Linux A8) binary in component-sources/omx_05_02_00_0x/bin/c6xtest/bin/ti814x-evm/c6xtest folder. It will also rebuild the dsp firmware in component-sources/omx_05_02_00_0x/bin/dm81xx/bin/ti814x-evm folder.

- **Running the application**

For running the application following steps are required.

- Insert syslink

```
insmod syslink.ko
```

Above step would be done by default in init scripts of SDK, so they will not be required if no change is done in init scripts.

- Load the Firmware using firmware_loader utility provided in SDK. (filesystem\user\bin)

```
i. firmware_loader 0 /usr/share/ti/ti-media-controller-utils/dm814x_c6xdsp.xe674 start
```

v Run the application

```
./c6xtest_a8host_debug.xv5T
```

Chapter 6

References

[1] Khronos OpenMax Overview http://www.khronos.org/openmax/

[2] OpenMax IL v1.1.2 specifications, http://www.khronos.org/files/openmax_il_spec_1_1_2.pdf