MPEG4 Advanced Simple Profile Decoder on DM365

User's Guide

Literature Number: SPRUGR3
March 2010

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI 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. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI 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 and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265

Read This First

About This Manual

This document describes how to install and work with Texas Instruments' (TI) MPEG4 Advanced Simple Profile Decoder implementation on the DM365 platform. It also provides a detailed Application Programming Interface (API) reference and information on the sample application that accompanies this component.

TI's codec implementations are based on the eXpressDSP Digital Media (XDM) and IRES standards. XDM and IRES are extensions of eXpressDSP Algorithm Interface Standard (XDAIS).

Intended Audience

This document is intended for system engineers who want to integrate TI's codecs with other software to build a multimedia system based on the DM365 platform.

This document assumes that you are fluent in the C language, have a good working knowledge of Digital Signal Processing (DSP), digital signal processors, and DSP applications. Good knowledge of eXpressDSP Algorithm Interface Standard (XDAIS) and eXpressDSP Digital Media (XDM) standard will be helpful.

How to Use This Manual

This document includes the following chapters:

- **Chapter 1 – Introduction**, provides a brief introduction to the XDAIS and XDM standards, Framework Components (FC), and software architecture. It also provides an overview of the codec and lists its supported features.
- **Chapter 2 – Installation Overview**, describes how to install, build, and run the codec.
- **Chapter 3 – Sample Usage**, describes the sample usage of the codec.
- **Chapter 4 – API Reference**, describes the data structures and interface functions used in the codec.

Related Documentation From Texas Instruments

The following documents describe TI's DSP algorithm standards such as, XDAIS and XDM. To obtain a copy of any of these TI documents, visit the Texas Instruments website at www.ti.com.

- *TMS320 DSP Algorithm Standard Rules and Guidelines* (literature number SPRU352) defines a set of requirements for DSP algorithms that, if followed, allow system integrators to quickly assemble production-quality systems from one or more such algorithms.
- *TMS320 DSP Algorithm Standard API Reference* (literature number SPRU360) describes all the APIs that are defined by the TMS320 DSP Algorithm Interoperability Standard (also known as XDAIS) specification.
- *Using IRES and RMAN Framework Components for C64x+* (literature number SPRAAI5) provides an overview of the IRES interface, with some concrete resource types and resource managers that illustrate the definition, management and use of new types of resources.

Related Documentation

You can use the following documents to supplement this user guide:

- *ISO/IEC 14496-2:2003(E), 'Information Technology – Coding Of Audio-Visual Objects – Part 2: Visual'*

Abbreviations

The following abbreviations are used in this document.

Table 1-1. List of Abbreviations

Abbreviation	Description
BIOS	TI's simple RTOS for DSPs
CMEM	Generic memory manager in Linux
CSL	Chip Support library
D1	720x480 or 720x576 resolutions in progressive scan
DCT	Discrete Cosine Transform
DMA	Direct Memory Access
EDMA	Enhanced Direct Memory Access
Full HD	1920x1088 or 1920x1080 resolution
GMC	Global Motion Compensation
HDTV	High Definition Television

Abbreviation	Description
HDVICP	High Definition Video and Imaging Coprocessor sub-system
ITU-T	International Telecommunication Union
IMX	Imaging Multimedia Extension
MB	Macro Block
MBAFF	Macro Block Adaptive Field Frame
MPEG	Motion Pictures Expert Group
MV	Motion Vector
NTSC	National Television Standards Committee
PMP	Portable Media Player
RMAN	Resource Manager
RTOS	Real Time Operating System
SW	Switch
UUID	Universal Unique Identifier
VGA	Video Graphics Array
VUI	Video Usability Information
XDAIS	eXpressDSP Algorithm Interface Standard
XDM	eXpressDSP Digital Media
YUV	Color space in luminance and chrominance form

Text Conventions

The following conventions are used in this document:

- Text inside back-quotes (‘‘) represents pseudo-code.
- Program source code, function and macro names, parameters, and command line commands are shown in a `mono-spaced` font.

Product Support

When contacting TI for support on this codec, quote the product name (MPEG4 Advanced Simple Profile Decoder on DM365) and version number. The version number of the codec is included in the Title of the Release Notes that accompanies this codec.

Trademarks

Code Composer Studio, DSP/BIOS, eXpressDSP, TMS320, TMS320C64x, TMS320C6000, TMS320DM644x, and TMS320C64x+ are trademarks of Texas Instruments.

All trademarks are the property of their respective owners.

Contents

Figures

This page is intentionally left blank

Tables

This page is intentionally left blank

Chapter 1

Introduction

This chapter provides a brief introduction to XDAIS, XDM, and DM365 software architecture. It also provides an overview of TI's implementation of the MPEG4 Advanced Simple Profile Decoder on the DM365 platform and its supported features.

Topic **Page**

1.1 Software Architecture

DM365 codec provides XDM compliant API to the application for easy integration and management. The details of the interface are provided in the subsequent sections.

DM365 is a digital multi-media system on-chip primarily used for video security, video conferencing, PMP and other related application.

DM365 codec are OS agonistic and interacts with kernel through the Framework Component (FC) APIs. FC acts as a software interface between OS and the codec. FC manages resources and memory by interacting with kernel through predefined APIs.

Following diagram shows the software architecture.

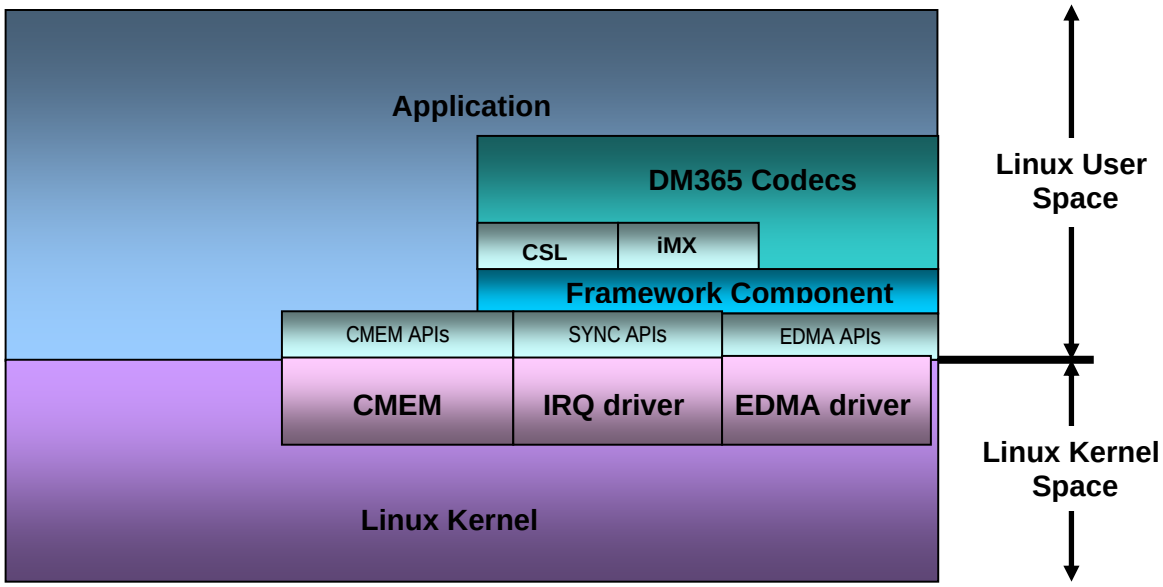


Figure 1-1. Software Architecture.

1.2 Overview of XDAIS, XDM, and Framework Component Tools

TI's multimedia codec implementations are based on the eXpressDSP Digital Media (XDM) standard. XDM is an extension of the eXpressDSP Algorithm Interface Standard (XDAIS). IRES is a TMS320 DSP Algorithm Standard (xDAIS) interface for management and utilization of special resource types such as hardware accelerators, certain types of memory and DMA. RMAN is a generic Resource Manager that manages software component's logical resources based on their IRES interface configuration. Both IRES and RMAN are Framework Component modules.

1.2.1 XDAIS Overview

An eXpressDSP-compliant algorithm is a module that implements the abstract interface IALG. The IALG API takes the memory management function away from the algorithm and places it in the hosting framework. Thus, an interaction occurs between the algorithm and the framework. This

interaction allows the client application to allocate memory for the algorithm and also share memory between algorithms. It also allows the memory to be moved around while an algorithm is operating in the system. To facilitate these functionalities, the IALG interface defines the following APIs:

- `algAlloc()`
- `algInit()`
- `algActivate()`
- `algDeactivate()`
- `algFree()`

The `algAlloc()` API allows the algorithm to communicate its memory requirements to the client application. The `algInit()` API allows the algorithm to initialize the memory allocated by the client application. The `algFree()` API allows the algorithm to communicate the memory to be freed when an instance is no longer required.

Once an algorithm instance object is created, it can be used to process data in real-time. The `algActivate()` API provides a notification to the algorithm instance that one or more algorithm processing methods is about to be run zero or more times in succession. After the processing methods have been run, the client application calls the `algDeactivate()` API prior to reusing any of the instance scratch memory.

The IALG interface also defines three more optional APIs `algControl()`, `algNumAlloc()` and `algMoved()`. For more details on these APIs, see *TMS320 DSP Algorithm Standard API Reference* (literature number SPRU360).

1.2.2 XDM Overview

In the multimedia application space, you have the choice of integrating any codec into your multimedia system. For example, if you are building a video decoder system, you can use any of the available video decoders (such as MPEG4, H.263, and so on) in your system. To enable easy integration with the client application, it is important that all codecs with similar functionality use similar APIs. XDM was primarily defined as an extension to XDAIS to ensure uniformity across different classes of codecs (for example audio, video, image, and speech). The XDM standard defines the following two APIs:

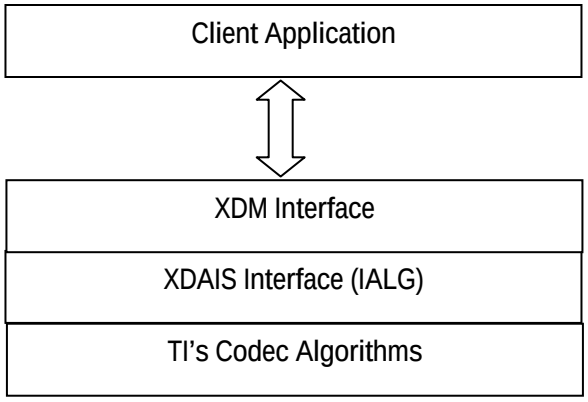
- `control()`
- `process()`

The `control()` API provides a standard way to control an algorithm instance and receive status information from the algorithm in real-time. The `control()` API replaces the `algControl()` API defined as part of the IALG interface. The `process()` API does the basic processing (encode/decode) of data. This API represents a blocking call for the encoder and the decoder, that is, with the usage of this API, the control is returned to the calling application only after encode or decode of one unit (frame) is completed. Since in case of DM365, the main encode or decode is carried out by the hardware accelerators, the host processor from which

the `process()` call is made can be used by the application in parallel with the encode or the decode operation. To enable this, the framework provides flexibility to the application to pend the decoder task when the frame level computation is happening on coprocessor.

Apart from defining standardized APIs for multimedia codecs, XDM also standardizes the generic parameters that the client application must pass to these APIs. The client application can define additional implementation specific parameters using extended data structures.

The following figure depicts the XDM interface to the client application.



As depicted in the figure, XDM is an extension to XDAIS and forms an interface between the client application and the codec component. XDM insulates the client application from component-level changes. Since TI's multimedia algorithms are XDM compliant, it provides you with the flexibility to use any TI algorithm without changing the client application code. For example, if you have developed a client application using an XDM-compliant MPEG4 video decoder, then you can easily replace MPEG4 with another XDM-compliant video decoder, say H.263, with minimal changes to the client application.

For more details, see *eXpressDSP Digital Media (XDM) Standard API Reference* (literature number SPRUEC8).

1.2.3 Framework Component

As discussed earlier, Framework Component acts like a middle layer between the codec and OS and also serves as a resource manager. The following block diagram shows the FC components and their interfacing structure.

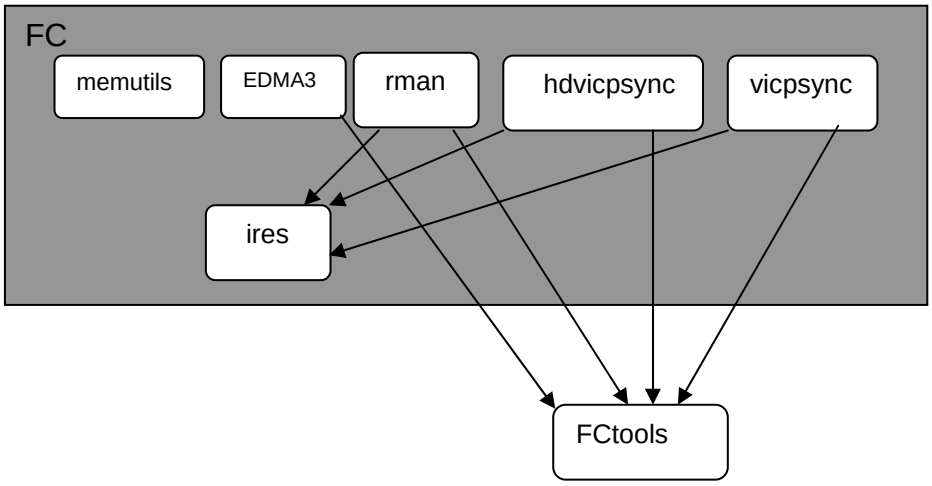


Figure 1-2. Framework Component Interfacing Structure.

Each component is explained in detail in the following sections.

1.2.3.1 IRES and RMAN Overview

IRES is a generic, resource-agnostic, extendible resource query, initialization and activation interface. The application framework defines, implements and supports concrete resource interfaces in the form of IRES extensions. Each algorithm implements the generic IRES interface, to request one or more concrete IRES resources. IRES defines standard interface functions that the framework uses to query, initialize, activate/deactivate and reallocate concrete IRES resources. To create an algorithm instance within an application framework, the algorithm and the application framework must agree on the concrete IRES resource types that are being requested. The framework calls the IRES interface functions, in addition to the IALG functions, to perform IRES resource initialization, activation and deactivation.

The IRES interface introduces support for a new standard protocol for cooperative preemption, in addition to the IALG-style non-cooperative sharing of scratch resources. Co-operative preemption allows activated algorithms to yield to higher priority tasks sharing common scratch resources. Framework components includes the following modules and interfaces to support algorithms requesting IRES-based resources:

- **IRES** - This is the standard interface allowing the client application to query and provide the algorithm with its requested IRES resources.
- **RMAN** - This is the generic IRES-based resource manager. It manages and grants concrete IRES resources to algorithms and applications. RMAN uses a new standard interface, the IRESMAN, to support run-time registration of concrete IRES resource managers.

Client applications call the algorithm's IRES interface functions to query its concrete IRES resource requirements. If the requested IRES resource type matches a concrete IRES resource interface supported by the application

framework, and if the resource is available, the client grants the algorithm logical IRES resource handles representing the allotted resources. Each handle provides the algorithm with access to the resource as defined by the concrete IRES resource interface.

IRES interface definition and function calling sequence is depicted in the following figure. For more details, see *Using IRES and RMAN Framework Components for C64x+* (literature number SPRAAI5).

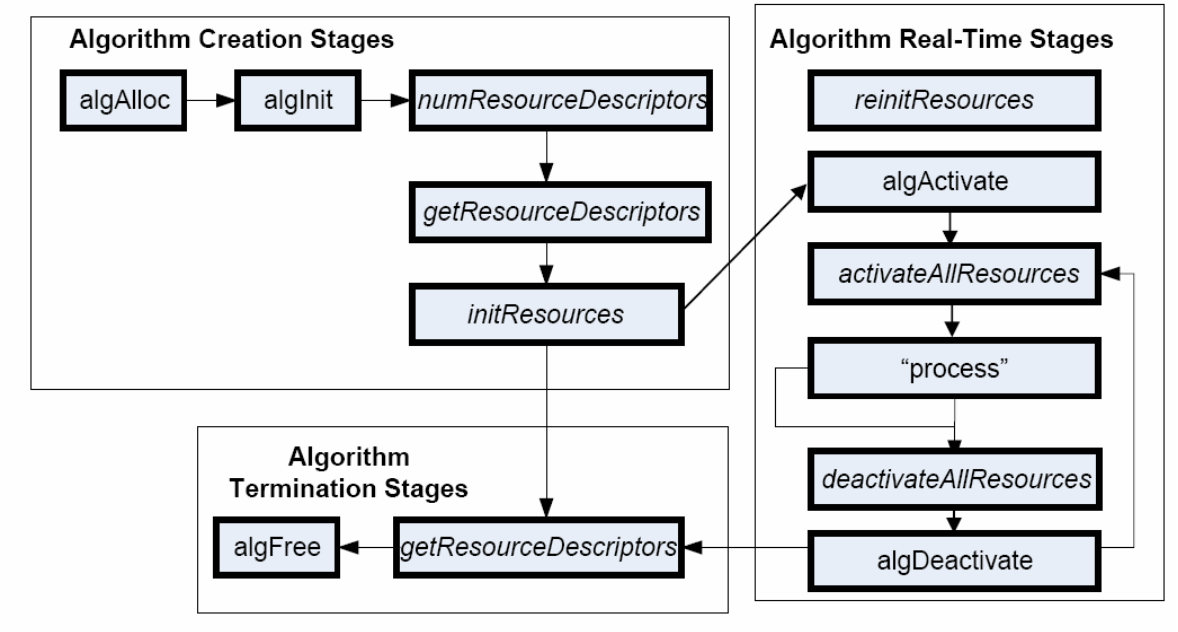


Figure 1-3. IRES Interface Definition and Function-calling Sequence.

For more details, see *Using IRES and RMAN Framework Components for C64x+* (literature number SPRAAI5).

1.2.3.2 HDVICP

The IRES HDVICP Resource Interface, IRES_HDVICP, allows algorithms to request and receive handles representing Hardware Accelerator resource, HDVICP, on supported hardware platforms. Algorithms can request and acquire one of the co-processors using a single IRES request descriptor. IRES_HDVICP is an example of a very simple resource type definition, which operates at the granularity of the entire processor and does not publish any details about the resource that is being acquired other than the ID of the processor. The algorithm manages internals of the resource based on the ID.

1.2.3.3 EDMA3

The IRES EDMA3 Resource Interface, IRES_EDMA3CHAN, allows algorithms to request and receive handles representing EDMA3 resources associated with a single EDMA3 channel. This is a very low-level resource definition.

Note:

The existing XDAIS IDMA3 and IDMA2 interfaces can be used to request logical DMA channels, but the IRES EDMA3CHAN interface provides the ability to request resources with finer precision than with IDMA2 or IDMA3.

1.2.3.4 VICP

The Imaging Coprocessor provides an integrated platform for the imaging hardware accelerators required to achieve the performance goals for the targeted device.

1.2.3.5 HDVICP Sync

Synchronization is necessary in a coprocessor system. HDVICP sync provides framework support for synchronization between codec and HDVICP coprocessor usage. This module is used by frameworks or applications, which have XDIAS algorithms that use HDVICP hardware accelerators.

1.2.3.6 Memutils

This for generic APIs to perform cache and memory related operations:

- `cacheInv` – Invalidates a range of cache
- `cacheWb` – Writes back a range of cache
- `cacheWbInv` – Writes back and invalidates cache
- `getPhysicalAddr` – Obtains physical (hardware specific) address

1.3 Overview of MPEG4 Advanced Simple Profile Decoder

MPEG4 (from ISO-IEC) is a popular video coding algorithm enabling high quality multimedia services on a limited bandwidth network. The standard defines several profiles and levels, which specify restrictions on the bit-stream, and hence limits the capabilities needed to decode the bit-streams. Each profile specifies a subset of algorithmic features and limits all decoders conforming to that profile. Each level specifies limits on the values that may be taken by the syntax elements in the profile.

Some important features of MPEG4 Advanced Simple Profile are:

- Advanced Simple Profile:

Some important MPEG4 profiles and their special features are:

- I, P and B type VOPs/Packets are present
- Only frame mode (progressive) picture types are present
- Data Partioned and Reversible Variable Length Codes are supported
- Video Packets are supported
- 1MV , 4MV and UMV supported

- AC/DC prediction supported
- Motion Compensation pixel accuracy up to Quarter-pel.
- Both MPEG4 and H.263 style quantization is supported.

The input to the decoder is a MPEG4 encoded bit stream in the byte-stream syntax. All frames in a video sequence are categorized as I-frames, P-frames and B-frames. I-frames called as intra-frames are decoded without reference to any other frame in the sequence, same as a still image would be decoded. In contrast, B-frames and P-frames called as predicted frames or inter-frames depend on information from a previous frame for their decoding. The video frames that are close in time are similar.

The output of the decoder is a YUV sequence, which can be of format 420 with the chroma components interleaved in little endian.

The following figure depicts the working of the decoder.

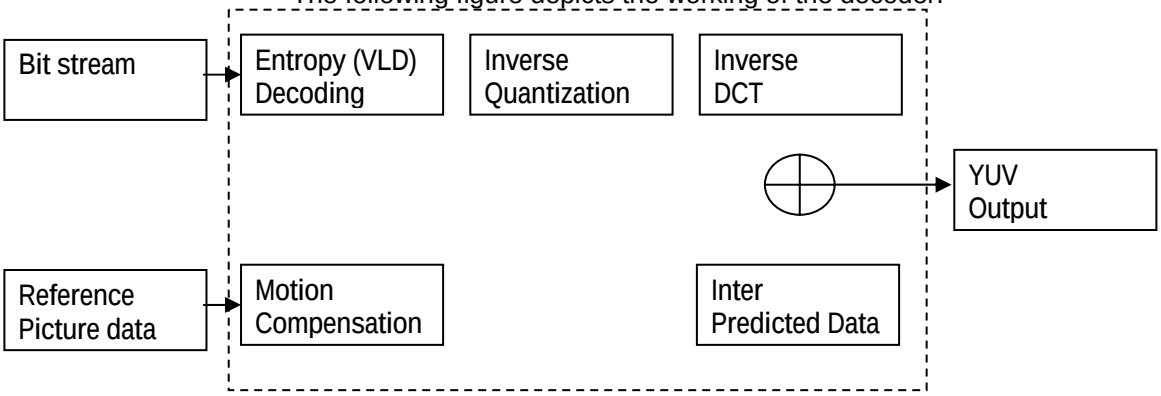


Figure 1-4. Block Diagram of MPEG4 Decoder

From this point onwards, all references to MPEG4 decoder mean MPEG4 Advanced Simple Profile decoder only.

1.4 Supported Services and Features

This user guide accompanies TI's implementation of MPEG4 Decoder on the DM365 platform.

This version of the codec has the following supported features of the standard:

- eXpressDSP Digital Media (XDM 1.2 IVIDDEC2) compliant
- Supports up to Level 5 of the Advanced Simple Profile (limited by the Full HD resolution 1920x1088)
- H.263 baseline profile (profile0), levels 10, 20, 30 and 45 supported
- Supports progressive frame type picture decoding
- Supports multiple packet decoding
- Supports short video header (H.263)
- Supports all AC/DC prediction
- Supports 4MV and UMV modes
- Motion compensation pixel accuracy up to Quarter-pel
- Supports both MPEG4 and H.263 style quantization
- Supports Error Resilience(ER) tools (DP, RVLC, HEC and re-sync marker) up to D1 resolution
- Supports resolutions up to 1080P (1920 x 1088).
- Supports a minimum resolution of 48 x 48
- Outputs are available in YUV420 interleaved little endian format
- Uses configurable frame display delay for out of order display

- Performs basic error concealment on erroneous frames and reports the type of error occurred.

This version of the decoder does not support the following features:

- Global Motion Compensation Toolset
- Dynamic change in resolution

Chapter 2

Installation Overview

This chapter provides a brief description on the system requirements and instructions for installing the codec component. It also provides information on building and running the sample test application.

Topic **Page**

2.1 System Requirements for Linux

This section describes the hardware and software requirements for the normal functioning of the codec component in Linux. For details about the version of the tools and software, see Release Note.

2.1.1 Hardware

- DM365 EVM (Set all the bits of SW4 and SW5 to low(0) position)RS232 cable and network cable.

2.1.2 Software

The following are the software requirements for the normal functioning of the codec:

- **Build Environment:** This project is built using Linux with MVL ARM.
- **ARM Tool Chain**: This project is compiled and linked using MVL ARM tool chain.

2.2 Installing the Component for Linux

The codec component is released as a compressed archive. To install the codec, extract the contents of the tar file onto your local hard disk. The tar file extraction creates a directory called dm365_mpeg4dec_xx_xx_xx_xx_production. Figure 2-1 shows the sub-directories created in this directory.

Note:

xx_xx_xx_xx in the directory name is the version of the codec. For example, If the version of the codec is 02.00.01.00, then the directory created on extraction of tar file is dm365_mpeg4dec _02_00_01_00_production.

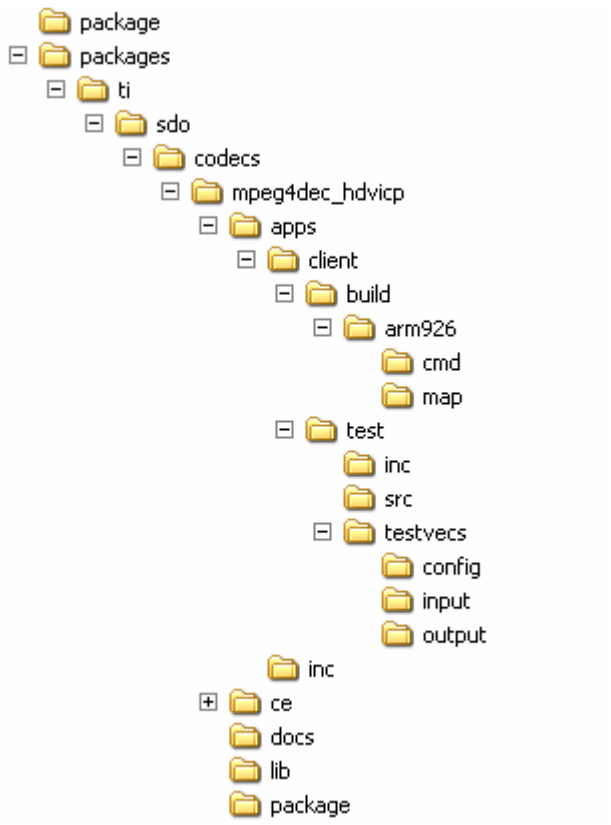


Figure 2-1. Component Directory Structure for Linux.

Table 2-2 provides a description of the sub-directories created in the dm365_mpeg4dec_xx_xx_xx_xx_production directory.

Table 2-2. Component Directories for Linux.

Sub-Directory	Description
\package	Contains files related while building the package
\packages\ti\sdo\codecs\mpeg4dec_hdvicp\lib	Contains the codec library files on host
\packages\ti\sdo\codecs\mpeg4dec_hdvicp\docs	Contains user guide and release notes
\packages\ti\sdo\codecs\mpeg4dec_hdvicp\apps\client\build\arm926	Contains the make file to built sample test application
\packages\ti\sdo\codecs\mpeg4dec_hdvicp\apps\client\build\arm926\cmd	Contains a template (.xdt) file to used to generate linker command file
\packages\ti\sdo\codecs\mpeg4dec_hdvicp\apps\client\build\arm926\map	Contains the memory map generated on compilation of the code
\packages\ti\sdo\codecs\mpeg4dec_hdvicp\apps\client\test\src	Contains application C files

Sub-Directory	Description
\packages\ti\sdo\codecs\mpeg4dec_hdvicp\apps\client\test\inc	Contains header files needed for the application code
\packages\ti\sdo\codecs\mpeg4dec_hdvicp\apps\client\test\testvecs\input	Contains input test vectors
\packages\ti\sdo\codecs\mpeg4dec_hdvicp\apps\client\test\testvecs\output	Contains output generated by the codec
\packages\ti\sdo\codecs\mpeg4dec_hdvicp\apps\client\test\testvecs\reference	Contains read-only reference output to be used for verifying against codec output
\packages\ti\sdo\codecs\mpeg4dec_hdvicp\apps\client\test\testvecs\config	Contains configuration parameter files

2.3 Building and Running the Sample Test Application on LINUX

To build the sample test application in Linux environment, follow these steps:

1) Verify that dma library, dma_ti_dm365.a, exists in the packages\ti\sdo\codecs\mpeg4dec_hdvicp\lib.

2) Verify that codec object library, mpeg4vdec_ti_arm926.a, exists in the \packages\ti\sdo\codecs\mpeg4dec_hdvicp\lib.

3) Ensure that you have installed the LSP, MontaVista Arm tool chain, XDC, Framework components releases with version numbers as mentioned in the release notes.

4) For installing framework component, unzip the content at some location and set the path of the base folder in FC_INSTALL_DIR environment variable.

5) Verify that the release package top-level folder is mapped to the target file system and is accessible from EVM.

6) In the folder \packages\ti\sdo\codecs\mpeg4dec_hdvicp\client\build\arm926, change the paths in the file rules.make according to your setup.

7) Open the command prompt at the sub-directory \packages\ti\sdo\codecs\mpeg4dec_hdvicp\client\build\arm926 and type the command **make**. This generates an executable file mpeg4vdec-r in the same directory.

8) To run the executable generated from the above steps, branch to the directory where the executable is present and type ./mpeg4vdec-r in the command window.

2.4 Configuration Files

This codec is shipped along with:

- Generic configuration file (testvecs_linux.cfg) – specifies input and reference files for the sample test application.
- Decoder configuration file (testparams.cfg) – specifies the configuration parameters used by the test application to configure the Decoder.

2.4.1 Generic Configuration File

The sample test application shipped along with the codec uses the configuration file, testvecs_linux.cfg, for determining the input and reference files for running the codec and checking for compliance. The testvecs_linux.cfg files are available in the \client\test\testvecs\config sub-directory.

The format of the testvecs_linux.cfg file is:

```
X
Config
Input
Output/Reference
```

where:

- `X` may be set as:
  - 1 - Compliance checking, no output file is created
  - 0 - Writing the output to the output file
- `Config` is the Decoder configuration file. For details, see Section 2.4.2.
- `Input` is the input file name (use complete path).
- `Output/Reference` is the output file name (if `X` is 0) or reference file name (if `X` is 1).

A sample testvecs_linux.cfg file is as shown.

```
1 /* X: Value "1" does Test Compliance and Value "0"
Dumps the Output */
../../test/testvecs/config/testparams.cfg
../../test/testvecs/input/colorful_toys_cif_5frms_420p.m4v
../../test/testvecs/reference/colorful_toys_cif_5frms_420p.chksum
```

2.4.2 Decoder Configuration File

The decoder configuration file, testparams.cfg contains the configuration parameters required for the decoder. The testparams.cfg file is available in the \client\test\testvecs\config sub-directory.

A sample testparams.cfg file for 720p stream is as shown.

```
# New Input File Format is as follows
# <ParameterName> = <ParameterValue> # Comment
############################################################
# Parameters
############################################################
ImageWidth              = 1280    # Image width in Pixels,
                                     must be multiple of 16
ImageHeight             = 720     # Image height in Pixels,
                                     must be multiple of 16
FramesToDecode          = 8000    # Number of frames to be
                                     decoded
displayDelay             = 1       # 0 -> No delay (Decode
                                        order),
                                     1-> (Default value)
                                     Display order for
                                     streams with  B frames
```

To check the functionality of the codec for the other inputs (other than the input provided with the release), change the configuration file accordingly with the corresponding input test vector.

2.5 Uninstalling the Component

To uninstall the component, delete the codec directory from your hard disk.

Chapter 3

Sample Usage

This chapter provides a detailed description of the sample test application that accompanies this codec component.

Topic **Page**

3.1 Overview of the Test Application

The test application exercises the extended class (extended over `IVIDDEC2`) structure `IMP4HDVICPDEC_Obj` of the MPEG4 Decoder library. The main test application files are mpeg4vdec_ti_arm926testapp.c andmpeg4vdec_ti_arm926testapp.h. These files are available in the \client\test\src and \client\test\inc sub-directories respectively.

Figure 3-1 depicts the sequence of APIs exercised in the sample test application.

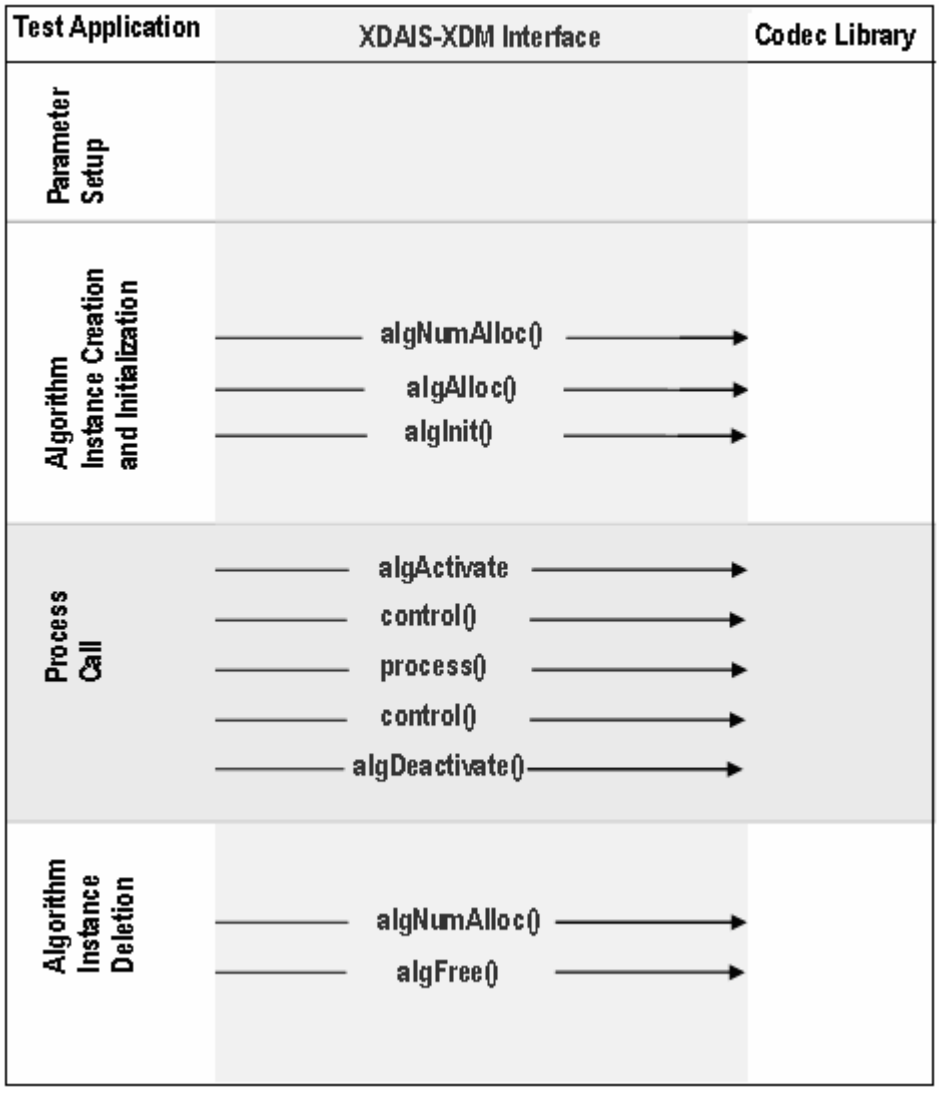


Figure 3-1. Test Application Sample Implementation

The test application is divided into four logical blocks:

- Parameter setup
- Algorithm instance creation and initialization
- Process call
- Algorithm instance deletion

3.1.1 Parameter Setup

Each codec component requires various codec configuration parameters to be set at initialization. For example, a video codec requires parameters such as video height, video width, and so on. The test application obtains the required parameters from the Decoder configuration files.

In this logical block, the test application does the following:

1) Opens the generic configuration file, testvecs_linux.cfg and reads the compliance checking parameter, Decoder configuration file name (testparams.cfg), input file name, and output/reference file name.

2) Opens the Decoder configuration file, (testparams.cfg) and reads the various configuration parameters required for the algorithm.

3) Sets the `IVIDDEC2_Params` structure based on the values it reads from the testparams.cfg file.

4) Reads the input bit-stream into the application input buffer.

After successful completion of the above steps, the test application does the algorithm instance creation and initialization.

3.1.2 Algorithm Instance Creation and Initialization

In this logical block, the test application accepts the various initialization parameters and returns an algorithm instance pointer. The following APIs are called in a sequence:

1) `algNumAlloc()` - To query the algorithm about the number of memory records it requires.

2) `algAlloc()` - To query the algorithm about the memory requirement to be filled in the memory records.

3) `algInit()` - To initialize the algorithm with the memory structures provided by the application.

A sample implementation of the create function that calls `algNumAlloc()`, `algAlloc()`, and `algInit()` in sequence is provided in the `ALG_create()` function implemented in the alg_create.c file.

After successful creation of the algorithm instance, the test application does DMA resource allocation for the algorithm.

Note:

DMAN3 function and IDMA3 interface is not implemented in DM365 codecs. Instead, it uses a DMA resource header file that gives the framework the flexibility to change DMA resource to codec.

3.1.3 Process Call

After algorithm instance creation and initialization, the test application does the following:

1) Sets the dynamic parameters (if they change during run-time) by calling the `control()` function with the `XDM_SETPARAMS` command.

2) Sets the input and output buffer descriptors required for the `process()` function call. The input and output buffer descriptors are obtained by calling the `control()` function with the `XDM_GETBUFINFO` command.

3) Implements the process call based on the mode of operation – blocking or non-blocking. These different modes of operation are explained below. The behavior of the algorithm can be controlled using various dynamic parameters (see Section 4.2.1.9). The input to the `process()` functions are input and output buffer descriptors, pointer to the `IVIDDEC2_InArgs` and `IVIDDEC2_OutArgs` structures.

4) Call the `process()` function to encode/decode a single frame of data. After triggering the start of the encode/decode frame start, the video task can be moved to SEM-pend state using semaphores. On receipt of interrupt signal for the end of frame encode/decode, the application should release the semaphore and resume the video task that does any book-keeping operations by the codec and updates the output parameters of the `IVIDDEC2_OutArgs` structure.

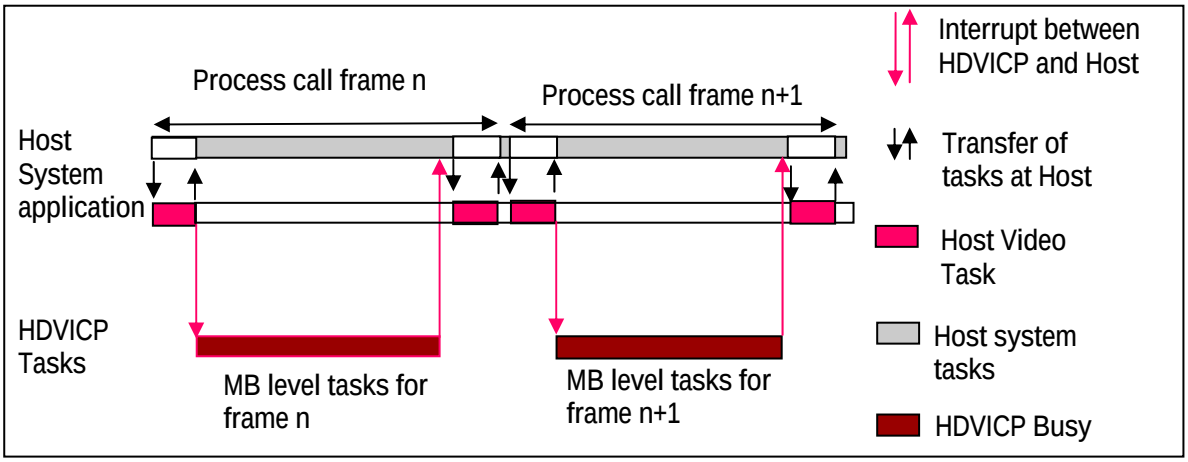


Figure 3-2. Process Call with Host Release

Note:

- The process call returns the control to the application after the initial setup related tasks are completed.
- Application can schedule a different task to use free Host resource.
- All service requests from HDVICP are handled through interrupts.
- Application resumes the suspended process call after last service request for HDVICP has been handled.
- Application can now complete concluding portions of the process call.

The `control()` and `process()` functions should be called only within the scope of the `algActivate()` and `algDeactivate()` XDAIS functions which activate and deactivate the algorithm instance respectively. Once an algorithm is activated, there could be any ordering of `control()` and `process()` functions. The following APIs are called in a sequence:

1) `algActivate()` - To activate the algorithm instance.

2) `control()` (optional) - To query the algorithm on status or setting of dynamic parameters and so on, using the six available control commands.

3) `process()` - To call the Decoder with appropriate input/output buffer and arguments information.

4) `control()` (optional) - To query the algorithm on status or setting of dynamic parameters and so on, using the six available control commands.

5) `algDeactivate()` - To deactivate the algorithm instance.

The do-while loop encapsulates frame level `process()` call and updates the input buffer pointer before the next call. The do-while loop breaks off either when an error condition occurs or when the input buffer exhausts.

In the sample test application, after calling `algDeactivate()`, the output data is either dumped to a file or compared with a reference file.

3.1.4 Algorithm Instance Deletion

Once decoding/encoding is complete, the test application deletes the current algorithm instance. The following APIs are called in a sequence:

1) `algNumAlloc()` - To query the algorithm about the number of memory records it used.

2) `algFree()` - To query the algorithm to get the memory record information, which can be used by the application for freeing them up.

A sample implementation of the delete function that calls `algNumAlloc()` and `algFree()` in sequence is provided in the `ALG_delete()` function implemented in the alg_create.c file.

3.2 Frame Buffer Management by Application

3.2.1 Frame Buffer Input and Output

With the new XDM 1.2, decoder does not ask for frame buffer at the time of `alg_create()`. It uses buffer from `XDM1_BufDesc *outBufs`, which it reads during each decode process call. Hence, there is no distinction between DPB and display buffers. The framework needs to ensure that it does not overwrite the buffers that are locked by the codec.

```
mp4VDEC_create();
mp4VDEC_control(XDM_GETBUFINFO); /* Returns maximum number of
                                    buffers with size based on
                                    resolution configured */
do{
mp4VDEC_decode();              /* call the decode API
}
while(all frames)
```

The frame pointer given by the application and that returned by the algorithm may be different. `BufferID` (`InputID/outputID`) provides a unique ID to keep a record of the buffer given to the algorithm and released by the algorithm. The following figure explains the frame pointer usage.

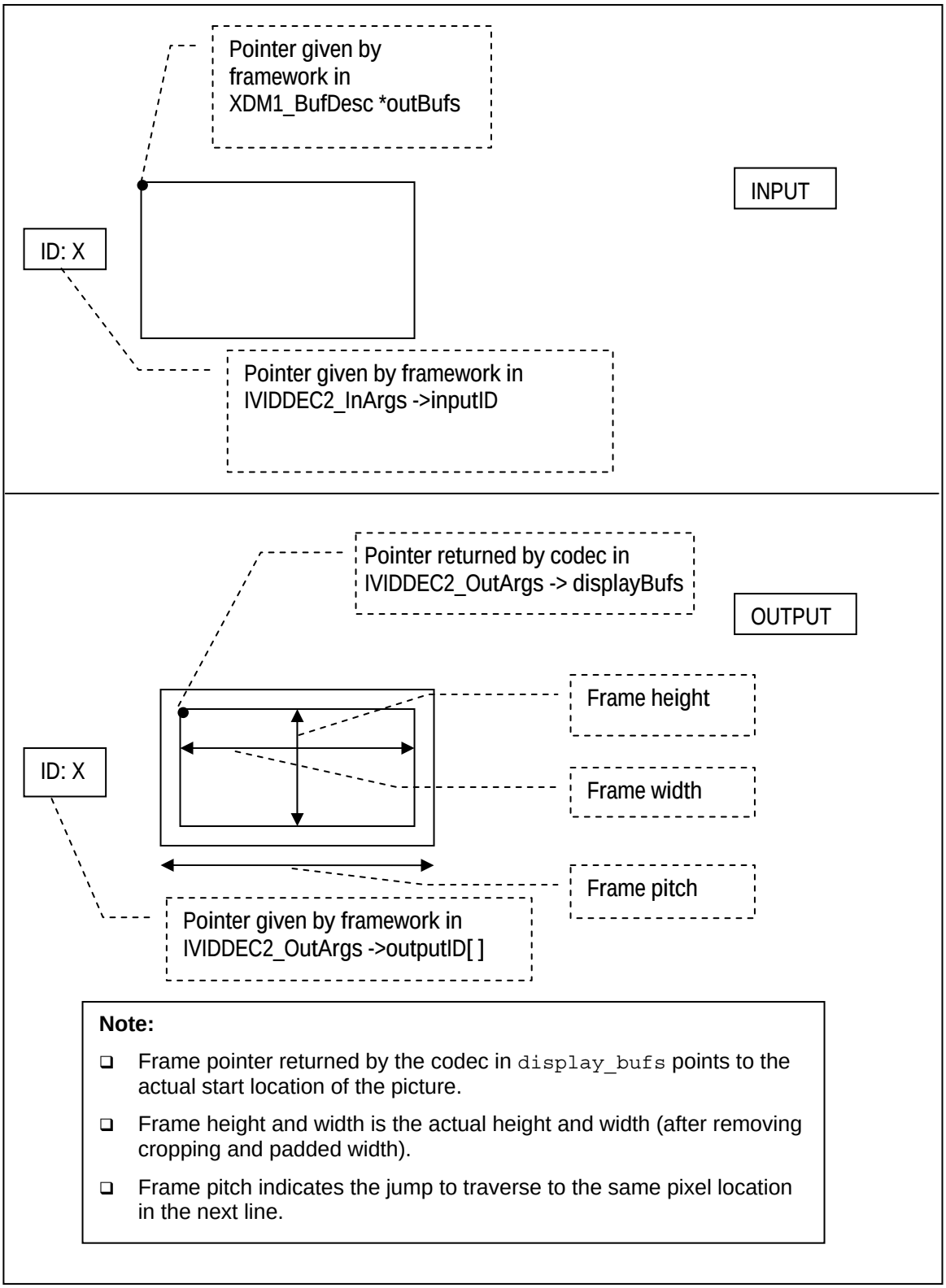


Figure 3-3. Frame Buffer Pointer Implementation

As explained , the buffer pointer cannot be used as a unique identifier to keep a record of the frame buffers. Any buffer given to the algorithm should be considered locked by the algorithm, unless the buffer is returned to the application through `IVIDDEC2_OutArgs->freeBufID[]`.

> **Note**:
>
> `BufferID` returned in `IVIDDEC2_OutArgs ->outputID[]` is for display purpose. Application should not consider it free, unless it comes as part of `IVIDDEC2_OutArgs->freeBufID[]`.

3.2.2 Frame Buffer Memory Optimizations

The `control()` API function (with `XDM_GETBUFINFO` command) requests the buffers of the size, based on the configured resolution. However, the actual memory required may be less compared to the one requested through `XDM_GETBUFINFO` for a given stream to be decoded. This is because stream properties are not known to the decoder at that moment. However, after first successful frame decode, `XDM_GETBUFINFO` can return the correct memory requirements. Hence, to save on memory requirements, two `XDM_GETBUFINFO` `control()` API calls can be made. After first call, application may choose to allocate only one buffer of the requested resolution. Rest of the buffers can be allocated for exact resolution after second call.

```
mp4VDEC_create();
mp4VDEC_control(XDM_GETBUFINFO); /* Returns maximum number of
                                    buffers with size based on
                                    resolution configured */
/* Allocate memory for 1 frame buffer */
frame_num = 0;

do
{
    mp4VDEC_decode();              /* Call the decode API */

    if no header related error
    {
        frame_num++;
    }

    if(frame_num == 1)
    {
        mp4VDEC_control(XDM_GETBUFINFO); /* Returns exact
                                            size buffers */

        /* Allocate memory for rest of frame buffers */
    }
}
while(all_frames)
```

The first frame decode is said to be successful, if there is no header related error. If the first frame decode is unsuccessful, the frame buffer is freed by the decoder and can be reused by the application to continue decoding for next frame. The following are the header related errors:

1) Specified by extended error `XDM_UNSUPPORTEDINPUT` bit

2) Specified by extended error `XDM_CORRUPTEDHEADER` bit

3) Specified by extended error code `IMPEG4VDEC_EX_ERR_HEADER_NOT_FOUND`

> **Note**:
>
> Application can choose to re-use the extra buffer space of the 1st frame, if the second control call returns a smaller size.

However, the number of buffers requested may be same in both the control calls.

3.2.3 Frame Buffer Management by Application

The application framework can efficiently manage frame buffers by maintaining a pool of free frames, from which it gives the decoder empty frames on request.

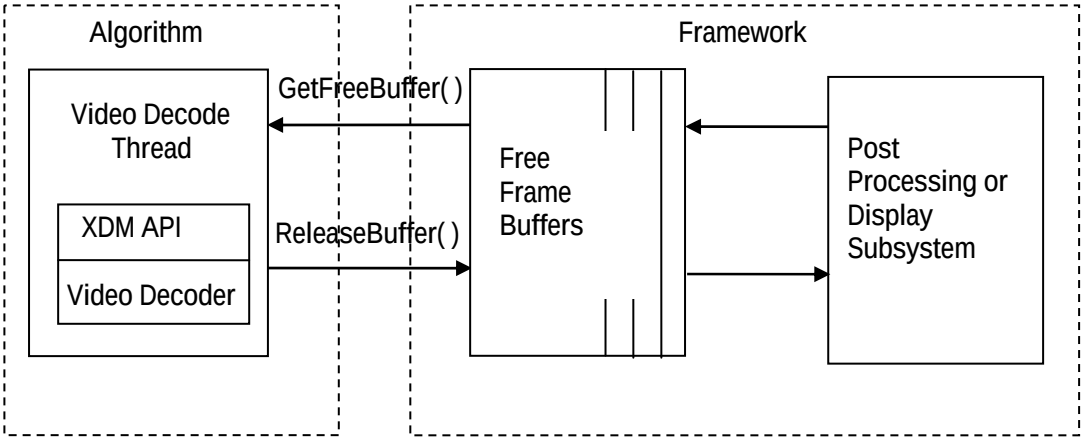


Figure 3-4. Interaction of Frame Buffers between Application and Framework

The sample application also provides a prototype for managing frame buffers. It implements the following functions, which are defined in the buffermanager.c file provided along with the test application.

1) `BUFFMGR_Init()`

   The `BUFFMGR_Init` function is called by the test application to initialize the global buffer element array to default and to allocate the required number of memory data for reference and output buffers. The maximum required DPB size is defined by the supported profile and level.

2) `BUFFMGR_ReInit()`

   The `BUFFMGR_ReInit` function allocates global luma and chroma buffers and allocates entire space to the first element. This element will be used in the first frame decode. After the picture height and width

and its luma and chroma buffer requirements are obtained, the global luma and chroma buffers are re-initialized to other elements in the buffer array.

3) `BUFFMGR_GetFreeBuffer()`

The `BUFFMGR_GetFreeBuffer` function searches for a free buffer in global buffer array and returns the address of that element. Incase, if none of the elements are free, then it returns `NULL`.

4) `BUFFMGR_ReleaseBuffer()`

The `BUFFMGR_ReleaseBuffer` function takes an array of buffer-IDs, which are released by the test-application. “0" is not a valid buffer ID, hence this function keeps moving until it encounters a buffer ID as zero or it hits the `MAX_BUFF_ELEMENTS`.

5) `BUFFMGR_DeInit()`

The `BUFFMGR_DeInit` function releases all memory allocated by buffer manager.

3.3 Handshaking Between Application and Algorithm

3.3.1 Resource Level Interaction

Following diagram explains about the resource level interaction of the application with framework component and codecs. Application uses XDM for interacting with codecs. Similarly, it uses RMAN to grant resources to the codec.

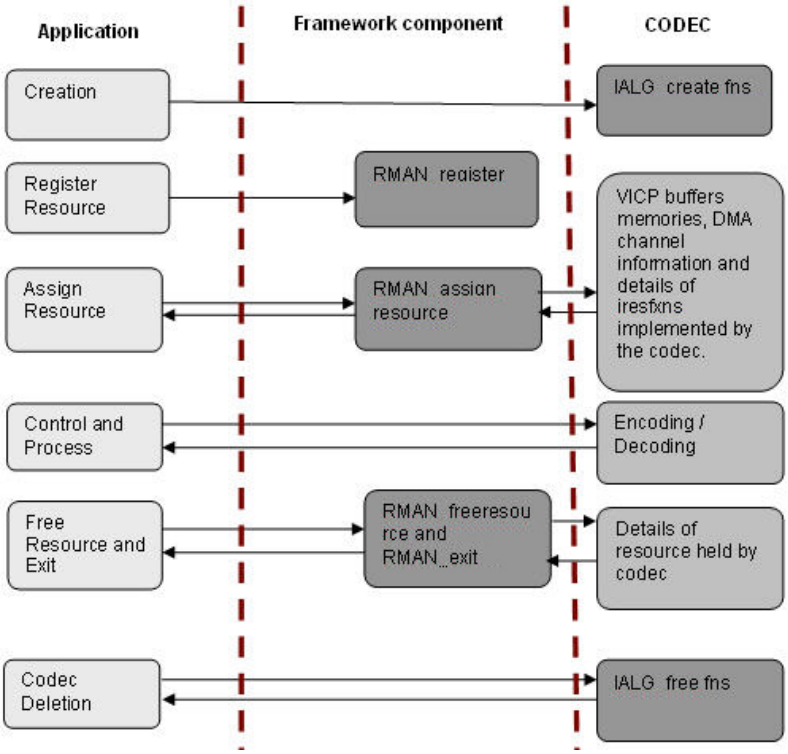


Figure 3-5. Process Call with Host Release.

3.3.2 Handshaking Between Application and Algorithms

Application provides the algorithm with its implementation of functions for the video task to move to SEM-pend state, when the execution happens in the co-processor. The algorithm calls these application functions to move the video task to SEM-pend state.

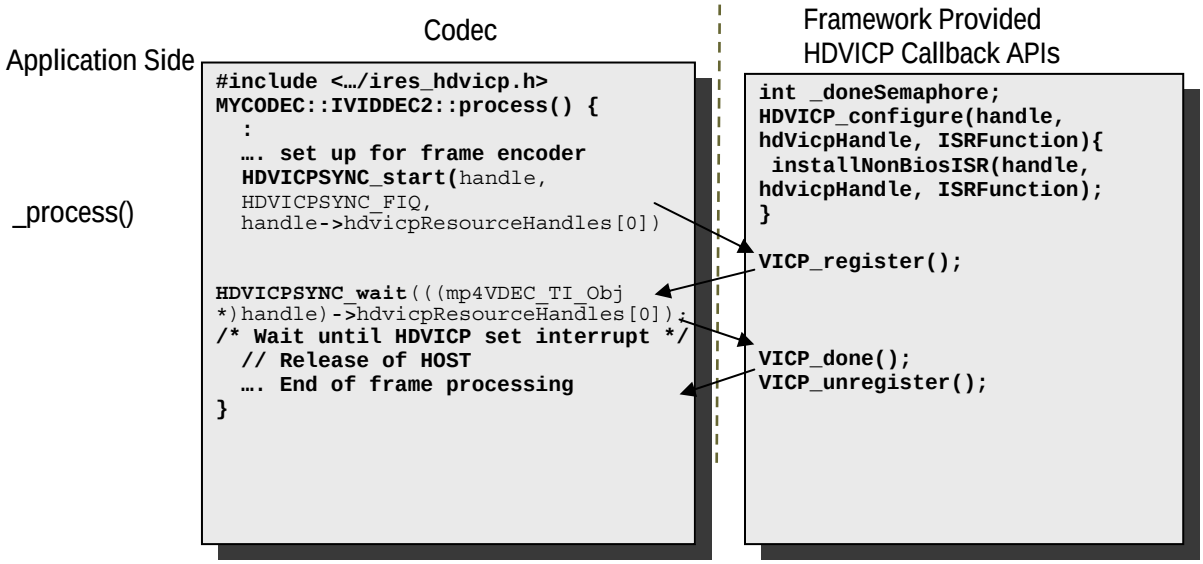


Figure 3-6. Interaction Between Application and Codec.

Note:

- Process call architecture shares Host resource among multiple threads.
- ISR ownership is with the FC manager – outside the codec.
- Codec implementation is OS independent.

The functions to be implemented by the application are:

1) `HDVICPSYNC_start(IALG_Handle handle, HDVICPSYNC_InterruptType intType, IRES_HDVICP_Handle hdvicpHandle)`

   This function is called by the algorithm to register the interrupt with the OS. This function also configures the Framework Component interrupt synchronization routine.

2) `HDVICPSYNC_wait (IRES_HDVICP_Handle hdvicpHandle)`

   This function is a FC call back function use to pend on a semaphore. Whenever the codec has completed the work on Host processor (after transfer of frame level encode/decode to HDVICP) and needs to relive the CPU for other tasks, it calls this function.

   This function of FC implements a semaphore, which goes into pend state and then the OS switches the task to another non-codec task.

Interrupts from HDVICP to Host ARM926 is used to inform when the frame processing is done. HDVICP sends interrupt which maps to `INT No 10` of ARM926 INTC. After receiving this interrupt, the semaphore on which the codec task was waiting gets released and the execution resumes after the `HDVICPSYNC_wait()` function.

The following figure explains the interrupt interaction between application and codec.

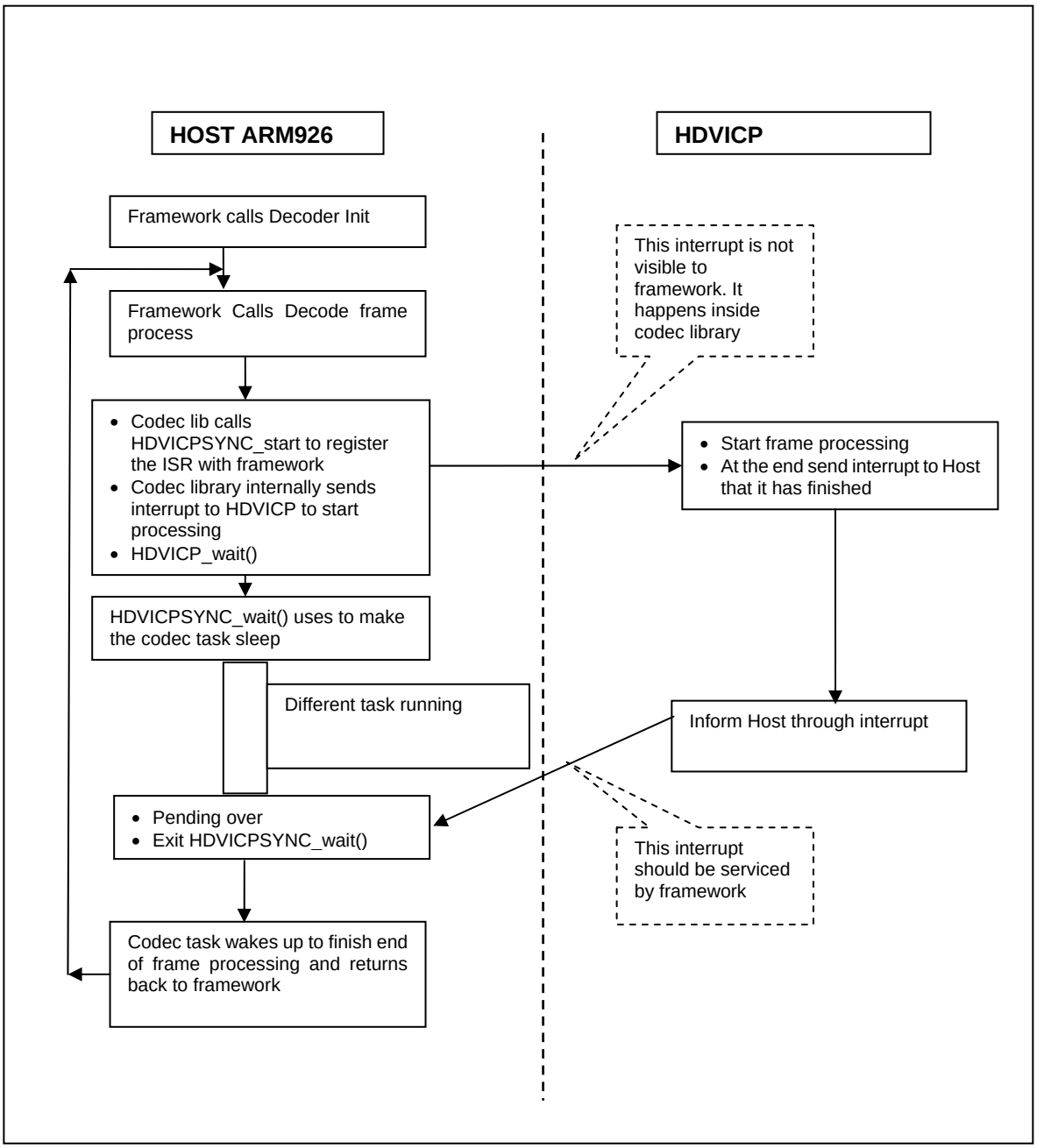


Figure 3-7. Interrupt Between Codec and Application.

3.4 Cache Management by Application

3.4.1 Cache Usage by Codec Algorithm

The codec source code and data, which runs on Host ARM926, can be placed in DDR (Double Data Rate). The host of DM365 has MMU and cache that the application can enable for better performance. Since the codec also uses DMA, there can be inherent cache coherency problems when application turns on the cache.

3.4.2 Cache and Memory Related Call Back Functions for Linux

To resolve the cache coherency and virtual to physical address issues, FC provides memory until library. These following functions can be used by codecs to resolve the cache coherency issues in Linux:

- `cacheInvalidate`
- `cacheWb`
- `cacheWbInv`
- `getPhysicalAddr`

3.4.2.1 cacheInvalidate

Cache invalidation deletes the entries of cache. This API Invalidate a range of cache.

```
Void MEMUTILS_cacheInv (Ptr  addr, Int  sizeInBytes)
```

3.4.2.2 cacheWb

This API writes back cache to the cache source when it is necessary.

```
Void MEMUTILS_cacheWb (Ptr  addr, Int  sizeInBytes)
```

3.4.2.3 cacheWbInv

This API writes back cache to the cache source when it is necessary and deletes cache contents.

```
Void MEMUTILS_cacheWbInv (Ptr  addr, Int  sizeInBytes)
```

3.4.2.4 getPhysicalAddr

This API obtains the physical address.

```
Void* MEMUTILS_getPhysicalAddr (Ptr  addr))
```

3.5 Sample Test Application

The test application exercises the `IVIDDEC2` base class of the MPEG4 Decoder.

Table 3-1. Process() Implementation

```
/* Main Function acting as a client for Video decode Call*/
/* Acquiring and intializing the resources needed to run
the decoder */
iresStatus = (IRES_Status) RMAN_init();
iresStatus = (IRES_Status)
RMAN_register(&IRESMAN_EDMA3CHAN,
(IRESMAN_Params *)&configParams);
iresStatus = (IRES_Status) RMAN_register(&IRESMAN_HDVICP,
(IRESMAN_Params *)&configParams);
iresStatus = (IRES_Status)RMAN_register(&IRESMAN_ADDRSPACE,
(IRESMAN_Params *)&configParams);

/*---------------- Decoder creation ----------------*/
handle = mp4VDEC_create(&fxns, &params)

/*Getting instance of algorithms that implements IALG and
IRES functions*/
iErrorFlag = RMAN_assignResources((IALG_Handle)handle,
                            &mp4VDEC_TI_IRES, /*
IRES_Fxns* */
                              1 /* scratchId */);
/* Get Buffer information  */
iErrorFlag = mp4VDEC_control(
            handle, // Instance Handle
            XDM_GETBUFINFO, // Command
            NULL, // Pointer to Dynamicparam structure
            &status// Pointer to the status structure
        );
/* Allocate memory for input and output frame buffers */
do {

/* Read the bitstream in the Application Input Buffer */
TestApp_ReadByteStream(inFile);
    /* Get a free frame buffer for decoder */
    BUFFMGR_GetFreeBuffer();

 iErrorFlag = mp4VDEC_decode (
                handle,   // Instance Handle    - Input
                &inobj,   // Input Buffers      - Input
                &outobj,  // Output Buffers     - Output
                &inargs,  // Input Parameters   - Input
                &outargs  // Output Parameters  - Output
            );
/* Get the status of the Decoder using control */
mp4VDEC_control(
               handle,         // Instance Handle
               XDM_GETSTATUS,  // Command - GET STATUS
               NULL, // Input
               &status         // Output
                );
    /* Display decoder output frame buffer, if any */

    /* Free, if frame buffer is released by decoder */
    BUFFMGR_ReleaseBuffer((XDAS_UInt32*)
                          outArgs.viddecOutArgs.freeBufID);

     }
/* end of Do-While loop - which Decodes frames  */
/* Free assigned resources */
RMAN_freeResources((IALG_Handle)(handle),
                    &MPEG4VDEC_TI_IRES, /* IRES_Fxns* */
```

```
                    );

/* Delete the decoder Object handle*/
mp4VDEC_delete(handle);

/* Free input and output frame buffers memory */

/* Unregister protocal*/
RMAN_unregister(&IRESMAN_EDMA3CHAN);
RMAN_unregister(&IRESMAN_HDVICP);
RMAN_unregister(&IRESMAN_ADDRSPACE);

RMAN_exit();
```

Note:

This sample test application does not depict the actual function parameter or control code. It provides an outline of the basic flow of the code.

3.6 Error Reporting and Inconsistencies Within Error Codes

While decoding the encoded bit-stream, any error that occurs is reported by the decoder algorithm in multiple ways. The definition and the interpretation of error codes by the decoder is provided as possible enumeration values for `XDM_ErrorBit` and codec specific error codes.

The user/application can access these process returned error codes in three methods:

- Error code as returned in `extendedError` element of `IVIDDEC2_Status` structure in `control()` API function with `XDM_GETSTATUS` command.
- Error code returned in `decodedBufs` element of `IVIDDEC2_OutArgs` structure in `process()` API function.
- Error code as returned in `displayBufs` element of `IVIDDEC2_OutArgs` structure in `process()` API function.

Since `extendedError` element of `IVIDDEC2_Status` structure and `decodedBufs` element of `IVIDDEC2_OutArgs` structure are obtained after frame decode, these two error codes are always in sync. However, the error code in `displayBufs` element of `IVIDDEC2_OutArgs` structure is obtained when the given frame is flushed or displayed from DPB. The `display_delay` parameter of `IVIDDEC2_Params` structure controls when a frame is displayed from DPB. If the `display_delay` parameter is set to a lower value (less than 16), then it is possible that a picture is displayed from DPB before its actual turn.

This timing difference between the error codes can result in `ERR_ORDER` bit of the error code to be set in `displayBufs` element, while it is not set in `decodedBufs` and `extendedError` elements.

From LSB side first 8 bits(bit 0 to bit 7) in `extendedError` are codec specific error codes set by the decoder.Table 3-2 can be used to interpret the Error code. In case of multiple errors in `process()` API function, only one codec specific error code is reported as there is no provision to update multiple codec specific error codes.

Table 3-2 List of Codec Specific Error Codes.

Error Code	Error Name	Error Description
0x1	IMPEG4VDEC_EX_ERR_INV_IMPL_ID	Invalid implementation Id seen in handle structure
0x2	IMPEG4VDEC_EX_ERR_INV_CONTXT	Invalid codec context in the handle structure
0x3	IMPEG4VDEC_EX_ERR_INV_CODEC_ID	Invalid codec ID in the handle

Error Code	Error Name	Error Description
0x4	IMPEG4VDEC_EX_ERR_ALGO_INACTIVE	`algActivate` not called before control
0x5	IMPEG4VDEC_EX_ERR_ERR_RESOURCE_INIT_NOT_DONE	process or control called before `initResources` was successfully completed
0x6	IMPEG4VDEC_NOTALL_RES_ACTIVE	Not all resources were activated before calling process
0xA	IMPEG4VDEC_EX_ERR_INV_STATUS_SIZE	Size of status structure is not valid
0xB	IMPEG4VDEC_EX_ERR_INV_HANDLE	Invalid handle passed
0xC	IMPEG4VDEC_EX_ERR_INV_DYN_PARAMS	Invalid dynamic params structure passed
0xD	IMPEG4VDEC_EX_ERR_VER_ERR_INSUFF_BUFSIZE	Insufficient `bufSize` passed for `GETVERSION` call - 128 bytes is the `minBufSize`
0xE	IMPEG4VDEC_EX_ERR_INV_DBLK_VAL	Invalid value passed for `outloopDeblocking` param
0xF	IMPEG4VDEC_EX_ERR_INV_DRNG_VAL	Invalid value passed for `outloopDeringing` param
0x10	IMPEG4VDEC_EX_ERR_INV_DEC_HEADR_MODE	Invalid value passed for decode header param
0x11	IMPEG4VDEC_EX_ERR_ERR_FRAME_PITCH	Invalid value passed for `displayWidth` param, greater than `maxWidth`, non-multiple of 32 and so on
0x12	IMPEG4VDEC_EX_ERR_ERR_FRAME_ORDER	Invalid value passed for `decodeOrder` param
0x13	IMPEG4VDEC_EX_ERR_INV_FRM_SKIP_MODE	Invalid value passed for `frmaeSkipMode` param
0x14	IMPEG4VDEC_EX_ERR_INV_RESET_HDVICP	Invalid value passed for `resetHDVICPeveryFrame` param
0x15	IMPEG4VDEC_EX_ERR_DEBLK_MORE_THAN_CIF	Deblocking and Deringing enabled for a stream larger than CIF
0x16	IMPEG4VDEC_EX_ERR_INV_MBDATAFLAG_PARAM	`MBData` Flag is not supported
0x17	IMPEG4VDEC_EX_ERR_INV_NEWFRAMEFLAG_PARAM	New frame flag is not supported
0x19	IMPEG4VDEC_EX_ERR_DEBLK_RUNTIME_CHANGE_NOT_SUPPORTED	Run-time configuration of de-blocking not supported

Error Code	Error Name	Error Description
0x1A	IMPEG4VDEC_EX_ERR_DERING_RUNTIME_CHANGE_NOT_SUPPORTED	Run-time configuration of de-Ringing not supported
0x1B	IMPEG4VDEC_EX_ERR_DISP_WD_RUNTIME_CHANGE_NOT_SUPPORTED	Run-time configuration of `dsiplayWidth` not supported
0x1C	IMPEG4VDEC_EX_ERR_FRM_ORDER_RUNTIME_CHANGE_NOT_SUPPORTED	Run-time configuration of `frameOrder` not supported
0x1E	IMPEG4VDEC_EX_ERR_NULL_ARGS	Null arguments to process
0x1F	IMPEG4VDEC_EX_ERR_INVALID_ARG_SIZES	Invalid size of `inargs` or `outargs`
0x20	IMPEG4VDEC_EX_ERR_INVALID_BITS_BUF	Invalid size
0x21	IMPEG4VDEC_EX_ERR_INVALID_YUV_BUF	Invalid size
0x22	IMPEG4VDEC_EX_ERR_INVALID_NUM_BYTES	Invalid value for `inargs->numBytes`
0x28	IMPEG4VDEC_EX_ERR_UNSUPP_PROFILE	Unsupported profile found while decoding sequence header
0x29	IMPEG4VDEC_EX_ERR_VISUAL_OBJ_NOT_SUPP	Unsupported visual object found
0x2A	IMPEG4VDEC_EX_ERR_CHROMA_FORMAT_NOT_SUPP	Unsupported Chroma Format
0x2B	IMPEG4VDEC_EX_ERR_VOL_SHAPE_INVALID	Invalid VOL shape found
0x2C	IMPEG4VDEC_EX_ERR_OBMC_NOT_SUPPORTED	OBMC is not supported. The flag will be ignored and normal motion compensation is done
0x2D	IMPEG4VDEC_EX_ERR_GUC_ESTIMATION_METHOD	If video`_object_layer_verid` > 1, estimation_method is not supported
0x2E	IMPEG4VDEC_EX_ERR_GOV_HDR_FOLL_BY_NONI	GOV starts with non-I frame
0x2F	IMPEG4VDEC_EX_ERR_VOL_VERID_INVALID	Invalid `vol_verifier`
0x30	IMPEG4VDEC_EX_ERR_ASPECT_RATIO_INFO_INV	Invalid ASPECT ration info
0x31	IMPEG4VDEC_EX_ERR_VOP_TIME_INCRE_ZERO	VOP Time increment set to zero

Error Code	Error Name	Error Description
0x32	IMPEG4VDEC_EX_ERR_FIX_VOP_TIME_INCRE_ZERO	Fixed VOP increment is set to zero
0x33	IMPEG4VDEC_EX_ERR_SPRITE_CODING_UNSUPP	Sprite enabled stream found
0x34	IMPEG4VDEC_EX_ERR_UNALIGNED_DISP_BUF	Display buffer base addresses are not aligned
0x35	IMPEG4VDEC_EX_ERR_B_PRESENT_IN_LOW_DELAY	B frame present when `display_delay` was set to 0, that is, the decode order dump mode
0x36	IMPEG4VDEC_EX_ERR_SPRITE_ENABLED_FRAME	Sprite enabled frame. This will be treated as not-coded VOP
0x3C	IMPEG4VDEC_EX_ERR_NOT_8_BIT_UNSUPP	`not_8_bit` flat set
0x3D	IMPEG4VDEC_EX_ERR_REDUCED_RESLN_VOP_ENBL	Reduced resolution mode not supported
0x3E	IMPEG4VDEC_EX_ERR_SCALABLE_CODING_UNSUPP	Scalability not supported
0x40	IMPEG4VDEC_EX_ERR_FAULT_VOP_HDR	Error in decoding VOP header
0x41	IMPEG4VDEC_EX_ERR_PTYPE_UNSUP_BASE_PROF	Extended PTYPE not supported in short video header stream
0x42	IMPEG4VDEC_EX_ERR_H263_UNSUPP_DIMENSIONS	Unsupported dimensions seen in `short_video` header
0x43	IMPEG4VDEC_EX_ERR_H263_UNSUPP_SRCFRMT_DIM	Invalid value for source format in `short_video_header`
0x44	IMPEG4VDEC_EX_ERR_INV_H263_OR_MPEG4_STR	Not a valid H263 or MPEG4 stream
0x45	IMPEG4VDEC_EX_ERR_B_S_VOP_UNSUPP_SMPL_PRO	SVOP seen in simple profile complexity estimatation header decode
0x48	IMPEG4VDEC_EX_ERR_HEADER_NOT_FOUND	Header not found in the given buffer
0x49	IMPEG4VDEC_EX_ERR_NEWPRED_NOT_SUPP	NEW Pred not supported
0x50	IMPEG4VDEC_EX_ERR_PACK_HDR	Error while decoding video packet header
0x51	IMPEG4VDEC_EX_ERR_MB_HDR	Error while decoding MB header
0x52	IMPEG4VDEC_EX_ERR_HEIGHT_SMALLER	Height smaller than 48

Error Code	Error Name	Error Description
0x53	IMPEG4VDEC_EX_ERR_WIDTH_SMALLER	Width smaller than 48
0x54	IMPEG4VDEC_EX_ERR_VOL_WIDTH_INVALID	Width decoded from the stream is invalid- either greater than max width or equal to zero
0x55	IMPEG4VDEC_EX_ERR_VOL_HEIGHT_INVALID	Height decoded from the stream is invalid- either greater than max width or equal to zero
0x56	IMPEG4VDEC_EX_ERR_VOP_NOTCODED_DEC_START	Stream starts with a not-coded VOP
0x57	IMPEG4VDEC_EX_ERR_BLKDATA	Error while decoding block data by ECD
0x58	IMPEG4VDEC_EX_ERR_WIDTH_NON_MULT2	Width non-multiple of 2 - Benign error
0x59	IMPEG4VDEC_EX_ERR_HEIGHT_NON_MULT2	Height non-multiple of 2 - Benign error

Chapter 4

API Reference

This chapter provides a detailed description of the data structures and interfaces functions used in the codec component.

Topic **Page**

4.1 Symbolic Constants and Enumerated Data Types

This section summarizes all the symbolic constants specified as either #define macros and/or enumerated C data types. For each symbolic constant, the semantics or interpretation of the same is also provided.

4.1.1 Common XDM Constants and Enumerated Data Types

This section summarizes all the common XDM constants and enumerated data types.

Table 4-1. List of Enumerated Data Types

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	IVIDDEC2_MAX_IO_BUFFERS	20 – used to assign maximum number of buffers used in IVIDDEC2_OutArgs and IVIDDEC2_Status structures.
IVIDEO_FrameType	IVIDEO_I_FRAME	Intra coded progressive frame. This also represents the frame type, when returning after decode of intra coded first field of an interlaced frame
	IVIDEO_P_FRAME	Forward inter coded frame. This also represents the frame type, when returning after decode of forward inter coded first field of an interlaced frame
	IVIDEO_B_FRAME	Bi-directional inter coded frame. This also represents the frame type, when returning after decode of bi-directional inter coded first field of an interlaced frame
	IVIDEO_IDR_FRAME	Intra coded frame that can be used for refreshing video content. This also represents the frame type, when returning after decode of IDR coded first field of an interlaced frame. Not applicable to MPEG4
	IVIDEO_II_FRAME	Interlaced frame, top field is an I or IDR, bottom field is again I or IDR frame. Not applicable to MPEG4
	IVIDEO_IP_FRAME	Interlaced frame, top field is an I or IDR frame, bottom field is a P frame. Not applicable to MPEG4
	IVIDEO_IB_FRAME	Interlaced frame, top field is an I or IDR frame, bottom field is a B frame. Not applicable to MPEG4

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	IVIDEO_PI_FRAME	Interlaced frame, top field is a P frame, bottom field is an I or IDR frame.
	IVIDEO_PP_FRAME	Interlaced frame, both fields are P frames.
	IVIDEO_PB_FRAME	Interlaced frame, top field is a P frame, bottom field is a B frame.
	IVIDEO_BI_FRAME	Interlaced frame, top field is a B frame, bottom field is an I or IDR frame.
	IVIDEO_BP_FRAME	Interlaced frame, top field is a B frame, bottom field is a P frame.
	IVIDEO_BB_FRAME	Interlaced frame, both fields are B frames.
	IVIDEO_MBAFF_I_FRAME	Intra coded MBAFF frame.
	IVIDEO_MBAFF_P_FRAME	Forward inter coded MBAFF frame.
	IVIDEO_MBAFF_B_FRAME	Bi-directional inter coded MBAFF frame.
	IVIDEO_MBAFF_IDR_FRAME	Intra coded MBAFF frame that can be used for refreshing video content.
	IVIDEO_FRAMETYPE_DEFAULT	By default, it is set to IVIDEO_I_FRAME.
IVIDEO_OutputFrameStatus	IVIDEO_FRAME_NOERROR	The output buffer is available.
	IVIDEO_FRAME_NOTAVAILABLE	The codec does not have any output buffers.
	IVIDEO_FRAME_ERROR	The output buffer is available and corrupted.
	IVIDEO_OUTPUTFRAMESTATUS_DEFAULT	By default, it is set to IVIDEO_FRAME_NOERROR.
IVIDEO_ContentType	IVIDEO_CONTENTTYPE_NA	Content type is not applicable
	IVIDEO_PROGRESSIVE	Progressive video content

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	IVIDEO_INTERLACED	Interlaced video content.
IVIDEO_FrameSkip	IVIDEO_NO_SKIP	Do not skip the current frame. This is the default value.
	IVIDEO_SKIP_P	Skip forward inter coded frame. Not supported in this version of MPEG4 Decoder.
	IVIDEO_SKIP_B	Skip bi-directional inter coded frame. Not supported in this version of MPEG4 Decoder.
	IVIDEO_SKIP_I	Skip intra coded frame. Not supported in this version of MPEG4 Decoder.
	IVIDEO_SKIP_IP	Skip I and P frame/field(s). Not supported in this version of MPEG4 Decoder.
	IVIDEO_SKIP_IB	Skip I and B frame/field(s). Not supported in this version of MPEG4 Decoder.
	IVIDEO_SKIP_PB	Skip P and B frame/field(s). Not supported in this version of MPEG4 Decoder.
	IVIDEO_SKIP_IPB	Skip I/P/B/BI frames. Not supported in this version of MPEG4 Decoder.
	IVIDEO_SKIP_IDR	Skip IDR Frame. Not supported in this version of MPEG4 Decoder.
XDM_DataFormat	XDM_BYTE	Big endian stream
	XDM_LE_16	16-bit little endian stream. Not supported in this version of MPEG4 Decoder.
	XDM_LE_32	32-bit little endian stream. Not supported in this version of MPEG4 Decoder.
XDM_ChromaFormat	XDM_YUV_420P	YUV 4:2:0 planar. Not supported in this version of MPEG4 Decoder.
	XDM_YUV_422P	YUV 4:2:2 planar. Not supported in this version of MPEG4 Decoder.

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_YUV_422IBE	YUV 4:2:2 interleaved (big endian). Not supported in this version of MPEG4 Decoder.
	XDM_YUV_422ILE	YUV 4:2:2 interleaved (little endian). Not supported in this version of MPEG4 Decoder.
	XDM_YUV_444P	YUV 4:4:4 planar. Not supported in this version of MPEG4 Decoder.
	XDM_YUV_411P	YUV 4:1:1 planar. Not supported in this version of MPEG4 Decoder.
	XDM_GRAY	Gray format. Not supported in this version of MPEG4 Decoder.
	XDM_RGB	RGB color format. Not supported in this version of MPEG4 Decoder.
	XDM_YUV_420SP	YUV 420 semiplaner (Luma 1st plane, * CbCr interleaved 2nd plane)
	XDM_ARGB8888	Alpha plane Not supported in this version of MPEG4 Decoder
	XDM_RGB555	RGB 555 color format Not supported in this version of MPEG4 Decoder
	XDM_RGB565	RGB 556 color format Not supported in this version of MPEG4 Decoder
	XDM_YUV_444ILE	YUV 4:4:4 interleaved (little endian) Not supported in this version of MPEG4 Decoder
XDM_CmdId	XDM_GETSTATUS	Query algorithm instance to fill Status structure.
	XDM_SETPARAMS	Set dynamic parameters via the DynamicParams structure. Most of the parameters in the structure are allowed to change only before the first process call in this version of the decoder.

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_RESET	Reset the algorithm. All fields in the internal data structures are reset and all internal buffers are flushed.
	XDM_SETDEFAULT	Initialize all fields in Params structure to default values specified in the library.
	XDM_FLUSH	Handle end of stream conditions. This command forces algorithm instance to output data without additional input.
	XDM_GETBUFINFO	Query algorithm instance regarding the properties of input and output buffers
	XDM_GETVERSION	Query the algorithm's version.
XDM_AccessMode	XDM_ACCESSMODE_READ	The algorithm reads from the buffer using the CPU.
	XDM_ACCESSMODE_WRITE	The algorithm writes from the buffer using the CPU.
	XDM_APPLIEDCONCEALMENT	Bit 9 ❑ 1 - Applied concealment ❑ 0 - Ignore
	XDM_INSUFFICIENTDATA	Bit 10 ❑ 1 - Insufficient data ❑ 0 - Ignore
	XDM_CORRUPTEDDATA	Bit 11 ❑ 1 - Data problem/corruption ❑ 0 - Ignore
	XDM_CORRUPTEDHEADER	Bit 12 ❑ 1 - Header problem/corruption ❑ 0 - Ignore
	XDM_UNSUPPORTEDINPUT	Bit 13 ❑ 1 - Unsupported feature/parameter in input ❑ 0 - Ignore
	XDM_UNSUPPORTEDPARAM	Bit 14 ❑ 1 - Unsupported input parameter or configuration ❑ 0 - Ignore
	XDM_FATALERROR	Bit 15 ❑ 1 - Fatal error (stop decoding) ❑ 0 - Recoverable error

Note: The remaining bits that are not mentioned in `XDM_ErrorBit` are interpreted as:

- Bit 16-32: Reserved
- Bit 0 - 7: Codec and implementation specific errors as described in section 3.6

4.2 Data Structures

This section describes the XDM defined data structures that are common across codec classes. These XDM data structures can be extended to define any implementation specific parameters for a codec component.

4.2.1 Common XDM Data Structures

This section includes the following common XDM data structures:

- XDM_BufDesc
- XDM1_BufDesc
- XDM_SingleBufDesc
- XDM1_SingleBufDesc
- XDM_AlgBufInfo
- IVIDEO1_BufDesc
- IVIDDEC2_Fxns
- IVIDDEC2_Params
- IVIDDEC2_DynamicParams
- IVIDDEC2_InArgs
- IVIDDEC2_Status
- IVIDDEC2_OutArgs

4.2.1.1 XDM_BufDesc

‖ Description

This structure defines the buffer descriptor for input and output buffers.

‖ Fields

Field	Data type	Input/ Output	Description
**bufs	XDAS_Int8	Input	Pointer to the vector containing buffer addresses
numBufs	XDAS_Int32	Input	Number of buffers
*bufSizes	XDAS_Int32	Input	Size of each buffer in bytes

4.2.1.2 XDM1_BufDesc

Description

This structure defines the buffer descriptor for input and output buffers.

Fields

Field	Data type	Input/ Output	Description
numBufs	XDAS_Int32	Input	Number of buffers
descs[XDM_MAX_IO_BUFFERS]	XDM1_SingleBufDesc	Input	Array of buffer descriptors.

4.2.1.3 XDM_SingleBufDesc

Description

This structure defines the buffer descriptor for single input and output buffers.

Fields

Field	Data type	Input/ Output	Description
*bufs	XDAS_Int8	Input	Pointer to the buffer
bufSize	XDAS_Int32	Input	Size of the buffer in bytes

4.2.1.4 XDM1_SingleBufDesc

Description

This structure defines the buffer descriptor for single input and output buffers.

Fields

Field	Data type	Input/ Output	Description
*bufs	XDAS_Int8	Input	Pointer to the buffer
bufSize	XDAS_Int32	Input	Size of the buffer in bytes
accessMask	XDAS_Int32	Output	If the buffer was not accessed by the algorithm processor (For example, it was filled through DMA or other hardware accelerator that does not write through the algorithm CPU), then no bits in this mask should be set. .

4.2.1.5 XDM_AlgBufInfo

‖ Description

This structure defines the buffer information descriptor for input and output buffers. This structure is filled when you invoke the `control()` function with the `XDM_GETBUFINFO` command.

‖ Fields

Field	Data type	Input/ Output	Description
`minNumInBufs`	`XDAS_Int32`	Output	Number of input buffers
`minNumOutBufs`	`XDAS_Int32`	Output	Number of output buffers
`minInBufSize[XDM_MAX_IO_BUFFERS]`	`XDAS_Int32`	Output	Size in bytes required for each input buffer
`minOutBufSize[XDM_MAX_IO_BUFFERS]`	`XDAS_Int32`	Output	Size in bytes required for each output buffer

Note:

For MPEG4 Advanced Simple Profile Decoder, the buffer details are:

- Number of input buffer required is 1.
- Number of output buffer required is 2 for YUV420 interleaved.
- There is no restriction on input buffer size except that it should contain atleast one frame of encoded data.
- The output buffer sizes (in bytes) for worst case 1080p format are:

For YUV 420 interleaved:
`Y buffer = ((maxWidth + 48 + alignment)*(maxHeight+ 96)) = 1984 * 1184  UV buffer = ((maxWidth + 48 + alignment)*(maxHeight+ 96))/2 = 1984 * 592`. These are the maximum buffer sizes, you can reconfigure depending on the format of the output bit-stream. In the current implementation, output format of 420 with planar Y and interleaved UV only is supported.

4.2.1.6 IVIDEO1_BufDesc

‖ Description

This structure defines the buffer descriptor for input and output buffers.

‖ Fields

Field	Data type	Input/ Output	Description
`numBufs`	`XDAS_Int32`	Output	Number of buffers
`frameWidth`	`XDAS_Int32`	Output	Width of the video frame

Field	Data type	Input/ Output	Description
frameHeight	XDAS_Int32	Output	Height of the video frame
framePitch	XDAS_Int32	Output	Frame pitch used to store the frame
bufDesc[IVIDEO_MAX_YUV_BUFFERS]	XDM1_SingleBufDesc	Output	Pointer to the vector containing buffer addresses
extendedError	XDAS_Int32	Output	Extended error information
frameType	XDAS_Int32	Output	Type of the video frame. This takes one of the values from enumerated datatype IVIDEO_FrameType as described in Table 4-1.
topFieldFirstFlag	XDAS_Int32	Output	Flag to indicate when the application should display the top field first
repeatFirstFieldFlag	XDAS_Int32	Output	Flag to indicate when the first field should be repeated
frameStatus	XDAS_Int32	Output	Flag to indicate the status of the output frame. This takes one of the values from enumerated datatype IVIDEO_OutputFrameStatus as described in Table 4-1.
repeatFrame	XDAS_Int32	Output	Number of times the display process needs to repeat the displayed progressive frame
contentType	XDAS_Int32	Output	Content type of the buffer IVIDEO_ContentType
chromaFormat	XDAS_Int32	Output	Only supported value is XDM_YUV_420SP

4.2.1.7 IVIDDEC2_Fxns

Description

This structure contains pointers to all the XDAIS and XDM interface functions.

Fields

Field	Data type	Input/ Output	Description
ialg	IALG_Fxns	Input	Structure containing pointers to all the XDAIS interface functions. For more details, see *TMS320 DSP Algorithm Standard API Reference* (literature number SPRU360).
*process	XDAS_Int32	Input	Pointer to the process() function
*control	XDAS_Int32	Input	Pointer to the control() function

4.2.1.8 IVIDDEC2_Params

Description

This structure defines the creation parameters for an algorithm instance object. Set this data structure to NULL, if you are not sure of the values to be specified for these parameters. Default values will be used if the data structure points to NULL.

Fields

Field	Data type	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes. Default value = 196
maxHeight	XDAS_Int32	Input	Maximum video height to be supported in pixels Default value = 1088 The minimum height supported by this implementation is 48 pixels (for luma).
maxWidth	XDAS_Int32	Input	Maximum video width to be supported in pixels Default value = 1920 The minimum width of the picture supported in this implementation is 48 pixels (for luma).
maxFrameRate	XDAS_Int32	Input	Maximum frame rate in fps * 1000 to be supported.
maxBitRate	XDAS_Int32	Input	Maximum bit-rate to be supported in bits per

Field	Data type	Input/ Output	Description
			second. For example, if bit-rate is 10 Mbps, set this field to 10485760.
dataEndianness	XDAS_Int32	Input	Endianness of input data. See XDM_DataFormat enumeration for details. Default value = XDM_BYTE
forceChromaFormat	XDAS_Int32	Input	Only supported value is XDM_YUV_420SP

Note:

- MPEG4 Decoder does not use the maxFrameRate and maxBitRate fields for creating the algorithm instance.
- Maximum video width supported is 1920 pixels (for 1080p format).
- The picture size constraint is on width (max 1920 pixels) and number of luma MBs (max 8160). This implies that even rotated pictures like 1088x1920 are also supported, along with 1920x1088 pictures.
- Maximum video height supported is 1920 pixels, provided it adheres to the maximum MB criteria of 8160. This additional requirement is imposed to ensure sufficiency of DPB buffers including padding requirements in the reference frames.
- dataEndianness field should be set to XDM_BYTE.
- Minimum supported width is 48 and height is 48.

4.2.1.9 IVIDDEC2_DynamicParams

‖ Description

This structure defines the run-time parameters for an algorithm instance object. However, the decoder does not use a few of the parameters. Few others are honored only before the first frame decode and hence run-time change for those parameters is not allowed. This structure must be initialized, while calling control() API with SETPARAMS command.

‖ Fields

Field	Data type	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes.

Field	Data type	Input/ Output	Description
decodeHeader	XDAS_Int32	Input	Number of access units to decode: ❑ 0 (XDM_DECODE_AU) - Decode entire frame including all the headers that occur before the frame.. In case header is not decoded after the instance creation, decoder will continue to ignore all the data, unless it finds the header. If decoder cannot find the header in the stream, it returns error. Once the header is successfully decoded, the decoder expects one full frame of data. ❑ 1 (XDM_PARSE_HEADER) - Decode only header Decoder looks for the header. Once header is decoded, it returns XDM_EOK. In case decoder cannot find header, it returns XDM_EFAIL. After the successful return of XDM_PARSE_HEADER, decodeheader should be changed to XDM_DECODE_AU to resume normal decoding process. In case stream contains multiple headers of varying resolutions, it is recommended to use XDM_DECODE_AU or else correct information during control() API SETPARAMS is not guaranteed.
displayWidth	XDAS_Int32	Input	Display buffer pitch: ❑ 0 – Default display buffer pitch. Configuring with the default value before first frame decode means decoder should derive the minimum possible display buffer pitch based on the decoded image width. ❑ Any non-zero value width be considered as display buffer pitch. However, after decoding the frame, if the configured displayWidth is found to be insufficient for the given stream, the process call is returned with fatal error. This can be used to configure only before the first frame decode. Run time change in displayWidth is not allowed. When calling control() API with SETPARAMS command after first frame decode, it must have the same value as the one used before first frame decode or must be set to default 0 value. This needs to be an integral multiple of 32 pixels and must be greater than the sum of padded area and the width of decoded image. The combined total padded area is 32 pixels and it must be less than maxWidth
displayHeight	XDAS_Int32	Input	This field is reserved. No use case assigned to it.
frameSkipMode	XDAS_Int32	Input	Frame skip mode. This is not supported by the current version of decoder and it expects frameSkipMode to be set to IVIDEO_NO_SKIP

Field	Data type	Input/ Output	Description
frameOrder	XDAS_Int32	Input	Frame display order. ❑ 0 (IVIDDEC2_DISPLAY_ORDER) – Decoder provides decoded output in actual order of display. When IVIDDEC2_DISPLAY_ORDER is configured, the display delay set through displayDelay in section 4.2.2.1 IMP4HDVICPDEC_Params is honoured. So, in case displayDelay is set less than the default value of 16, the correct display order in not guaranteed. ❑ 1 (IVIDDEC2_DECODE_ORDER) – Decoder provides decoded output in order of decoding. This sets the display delay to 0, that is, the frame buffer is given back for display as soon as it is decoded. The display delay set through displayDelay in section 4.2.2.1 IMP4HDVICPDEC_Params gets overwriten internally. This can be used to configure only before the first frame decode. Run time change in frameOrder is not allowed. When calling control() API with SETPARAMS command after first frame decode, it must have the same value as the one used before first frame decode.
newFrameFlag	XDAS_Int32	Input	Flag to indicate that, the algorithm should start a new frame. Valid values are XDAS_TRUE and XDAS_FALSE. This is useful for error recovery, For example when the end of frame cannot be detected by the codec but is known to the application. This logic is not implemented by the current version of decoder and it thus expects newFrameFlag to be set to XDAS_FALSE.
mbDataFlag	XDAS_Int32	Input	Flag to indicate that the algorithm should generate MB Data in addition to decoding the data. This is not supported by the current version of decoder and it thus expects mbDataFlag to be set to XDAS_FALSE.

Field	Data type	Input/ Output	Description
resetHDVICPeveryFrame	XDAS_Int8	Input	Flag to reset HDVICP at the start of every frame being decoded. This is useful for multi-channel and multi-format encoding/decoding. ❑ 1 – ON ❑ 0 – OFF Default value = 1. MPEG4 decoder has two libraries, which dynamically gets loaded into HDVICP depending on the picture type. These three libraries correspond to different picture types namely ❑ 1. I and P Pictures ❑ 2. B Pictures If this flag is set, then decoder assumes that the memory of HDVICP was overwritten by some other codec or by other instance of same codec with different picture type. For example, Application will set this flag to 1 if running another instance of different codec like MPEG4 encoder or if running another MPEG4 decoder instance with different picture type. However, the application can set this flag to 0 for better performance if it runs multiple instances of MPEG4 decoder with same picture type.

4.2.1.10 IVIDDEC2_InArgs

Description

This structure defines the run-time input arguments for an algorithm instance object.

Fields

Field	Data type	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes.
numBytes	XDAS_Int32	Input	Size of input data (in bytes) provided to the algorithm for decoding.
inputID	XDAS_Int32	Input	Application passes this ID to the algorithm and decoder will attach this ID to the corresponding output frames. This is useful in case of re-ordering (For example, B frames). If there is no re-ordering, outputID field in the IVIDDEC2_OutArgs data structure will be same as inputID field.

Note:

- MPEG4 Decoder returns a failure if `inputID` is 0
- MPEG4 decoder always expects `numBytes` to be equal to 1 frame of data.if `numBytes` is less than 4, a failure is returned.

4.2.1.11 IVIDDEC2_Status

‖ Description

This structure defines parameters that describe the status of an algorithm instance object.

‖ Fields

Field	Data type	Input/ Output	Description
`size`	`XDAS_Int32`	Input	Size of the basic or extended (if being used) data structure in bytes.
`extendedError`	`XDAS_Int32`	Output	Extended error code. See section 3.6 for details.
`data`	`XDM_SingleBufDesc`	Input/Output	Buffer information structure for information passing buffer.
`maxNumDisplayBufs`	`XDAS_Int32`	Output	The maximum number of buffers that is required by the codec. The maximum number of buffers can be `IVIDDEC2_MAX_IO_BUFFERS`. In this decoder implementation it is always set to 3, .
`outputHeight`	`XDAS_Int32`	Output	Output height in pixels
`outputWidth`	`XDAS_Int32`	Output	Output width in pixels
`frameRate`	`XDAS_Int32`	Output	Average frame rate in fps * 1000. The average frame rate for all video decoders is 30 fps.
`bitRate`	`XDAS_Int32`	Output	Average bit-rate in bits per second
`contentType`	`XDAS_Int32`	Output	Video content. See `IVIDEO_ContentType` enumeration for details.
`outputChromaFormat`	`XDAS_Int32`	Output	Only supported value is `XDM_YUV_420SP`.
`bufInfo`	`XDM_AlgBufInfo`	Output	Input and output buffer information. See `XDM_AlgBufInfo` data structure for details.

Note:

- Algorithm sets the `frameRate` and `bitRate` fields to zero.
- MPEG4 Decoder does not use the buffer descriptor meant for passing additional information between the application and the decoder.
- Fields which are not updated may return an uninitialized value when read.

4.2.1.12 IVIDDEC2_OutArgs

Description

This structure defines the run-time output arguments for an algorithm instance object.

Fields

Field	Data type	Input/ Output	Description
`size`	`XDAS_Int32`	Input	Size of the basic or extended (if being used) data structure in bytes.
`bytesConsumed`	`XDAS_Int32`	Output	Bytes consumed per decode call
`outputID[IVIDDEC2_MAX_IO_BUFFERS]`	`XDAS_Int32`	Output	Output ID corresponding to `displayBufs` A value of zero (0) indicates an invalid ID. The first zero entry in array indicates end of valid `outputIDs` within the array. Hence the application stops reading the array when it encounters the first zero entry
`decodedBufs`	`IVIDEO1_BufDesc`	Output	The decoder fills this structure with buffer pointers to the decoded frame. Related information fields for the decoded frame are also populated When frame decoding is not complete, as indicated by `outBufsInUseFlag`, the frame data in this structure is incomplete. However, the algorithm provides incomplete decoded frame data in case the application chooses to use it for error recovery
`displayBufs[IVIDDEC2_MAX_IO_BUFFERS]`	`IVIDEO1_BufDesc`	Output	Array containing display frames corresponding to valid ID entries in the `outputID[]` array.
`outputMbDataID`	`XDAS_Int32`	Output	Output ID corresponding with the MB Data
`mbDataBuf`	`XDM1_SingleBufDesc`	Output	The decoder populates the last buffer among the buffers supplied within `outBufs->bufs[]` with the decoded MB data generated by the decoder module
`freeBufID[IVIDDEC2_MAX_IO_BUFFERS]`	`XDAS_Int32`	Output	This is an array of `inputIDs` corresponding to the frames that have been unlocked in the current process call

Field	Data type	Input/ Output	Description
outBufsInUseFlag	XDAS_Int32	Output	Flag to indicate that the outBufs provided with the process() call are in use. outBufs are not required for the next process() call

Note:

- decodedBufs, OutputMbDataID and mbDataBuf is not given as output in this version of the decoder. These fields may return an uninitialized value when read.
- After decoding first field of an interlaced frame, outBufsInUseFlag is set to 1.

4.2.2 MPEG4 Decoder Data Structures

This section includes the following MPEG4 decoder specific data structures:

- IMP4HDVICPDEC_Params
- IMP4HDVICPDEC_DynamicParams
- IMP4HDVICPDEC_InArgs
- IMP4HDVICPDEC_Status
- IMP4HDVICPDEC_OutArgs

4.2.2.1 IMP4HDVICPDEC_Params

Description

This structure defines the creation parameters and any other implementation specific parameters for the MPEG4 Decoder instance object. The creation parameters are defined in the XDM data structure, IVIDDEC2_Params.

Fields

Field	Data type	Input/ Output	Description
viddecParams	IVIDDEC2_Params	Input	See IVIDDEC2_Params data structure for details.
displayDelay	XDAS_Int32	Input	Display delay before which the decoder starts to output frames for display. Delay in the display can also be controlled by frameOrder. See section 4.2.1.9 - Default value: 1 (when base class is used). - Valid range: [0, 1]
hdvicpHandle	Void *	Input	HDVICP related handle object. Default value: NULL (when base class is used)
disableHDVICPeveryFrame	XDAS_Int8	Input	Flag to indicate if the co-processor needs to be disable and enabled at end of every frame. This is for power optimization Not supported in this version of decoder.

4.2.2.2 IMP4HDVICPDEC_DynamicParams

Description

This structure defines the run-time parameters and any other implementation specific parameters for the MPEG4 Decoder instance object. The run-time parameters are defined in the XDM data structure, IVIDDEC2_DynamicParams.

‖ Fields

Field	Data type	Input/ Output	Description
viddecDynamicParams	IVIDDEC2_DynamicParams	Input	See IVIDDEC2_DynamicParams data structure for details.
outloopDeblocking	XDAS_Int32	Input	To enable de-blocking in the post processing stage. Default is 0. Supported only till CIF resolution
outloopDeRinging	Void *	Input	To enable de-ringing in the post processing stage. Default is 0 Supported only till CIF resolution
resetHDVICPeveryFrame	XDAS_Int8	Input	Flag to reset HDVICP at the start of every frame being decoded. This is useful for multi-channel and multi-format encoding/decoding. ❑ 1 – ON ❑ 0 – OFF Default value = 1. MPEG4 decoder has two libraries, which dynamically gets loaded into HDVICP depending on the picture type. These two libraries correspond to different picture types namely ❑ 1. I and P Pictures ❑ 2. B Pictures If this flag is set, then decoder assumes that the memory of HDVICP was over written by some other codec or by other instance of same codec with different picture type. For example, Application will set this flag to 1 if running another instance of different codec like MPEG4 encoder or if running another MPEG4 decoder instance with different picture type. However, the application can set this flag to 0 for better performance if it runs multiple instances of MPEG4 decoder with same picture type.

4.2.2.3 IMP4HDVICPDEC_InArgs

Description

This structure defines the run-time input arguments for the MPEG4 Decoder instance object.

Fields

Field	Data type	Input/ Output	Description
viddecInArgs	IVIDDEC2_InArgs	Input	See IVIDDEC2_InArgs data structure for details.

4.2.2.4 IMP4HDVICPDEC_Status

Description

This structure defines parameters that describe the status of the MPEG4 Decoder and any other implementation specific parameters. The status parameters are defined in the XDM data structure, IVIDDEC2_Status.

Fields

Field	Data type	Input/ Output	Description
viddecStatus	IVIDDEC2_Status	Output	See IVIDDEC2_Status data structure for details.

4.2.2.5 IMP4HDVICPDEC_OutArgs

Description

This structure defines the run-time output arguments for the MPEG4 Decoder instance object.

Fields

Field	Data type	Input/ Output	Description
viddecOutArgs	IVIDDEC2_OutArgs	Output	See IVIDDEC2_OutArgs data structure for details.
pixelRange	XDAS_UInt8	Output	❑ 0 - Range of Y from 16 to 235, Cb and Cr from 16 to 240 ❑ Default value: 1 - Range of Y from 0 to 255, Cb and Cr from 0 to 255.
parWidth	XDAS_UInt16	Output	Pixel aspect ratio width Default value: 1

Field	Data type	Input/ Output	Description
parHeight	XDAS_UInt16	Output	Pixel aspect ratio height Default value: 1
numErrMbs	XDAS_UInt16	Output	Number of erroneous MBs in the output. Note: Due to the nature of VLD processing, even unintended bit-combinations may be a valid code resulting in successful decoding. This affects reporting the exact position of error in the input stream.

4.3 Interface Functions

This section describes the Application Programming Interfaces (APIs) used in the MPEG4 Decoder. The APIs are logically grouped into the following categories:

- **Creation** – `algNumAlloc()`, `algAlloc()`
- **Initialization –** `algInit()`
- **Control** – `control()`
- **Data processing** – `algActivate()`, `process()`, `algDeactivate()`
- **Termination** – `algFree()`

You must call these APIs in the following sequence:

1) `algNumAlloc()`
2) `algAlloc()`
3) `algInit()`
4) `algActivate()`
5) `process()`
6) `algDeactivate()`
7) `algFree()`

`control()` can be called any time after calling the `algInit()` API.

`algNumAlloc()`, `algAlloc()`, `algInit()`, `algActivate()`, `algDeactivate()`, and `algFree()` are standard XDAIS APIs. This document includes only a brief description for the standard XDAIS APIs. For more details, see *TMS320 DSP Algorithm Standard API Reference* (SPRU360).

4.3.1 Creation APIs

Creation APIs are used to create an instance of the component. The term creation could mean allocating system resources, typically memory.

‖ Name

`algNumAlloc()` – determine the number of buffers that an algorithm requires

‖ Synopsis

```
XDAS_Int32 algNumAlloc(Void);
```

‖ Arguments

```
Void
```

‖ Return Value

```
XDAS_Int32; /* number of buffers required */
```

‖ Description

`algNumAlloc()` returns the number of buffers that the `algAlloc()` method requires. This operation allows you to allocate sufficient space to call the `algAlloc()` method.

`algNumAlloc()` may be called at any time and can be called repeatedly without any side effects. It always returns the same result. The `algNumAlloc()` API is optional.

For more details, see *TMS320 DSP Algorithm Standard API Reference* (literature number SPRU360).

‖ See Also

`algAlloc()`

‖ Name

`algAlloc()` – determine the attributes of all buffers that an algorithm requires

‖ Synopsis

```
XDAS_Int32 algAlloc(const IALG_Params *params, IALG_Fxns
**Fxns, IALG_MemRec memTab[]);
```

‖ Arguments

```
IALG_Params *params; /* algorithm specific attributes */

IALG_Fxns **Fxns;/* output parent algorithm functions */

IALG_MemRec memTab[]; /* output array of memory records */
```

‖ Return Value

```
XDAS_Int32 /* number of buffers required */

IALG_EFAIL /* Status indicating failure */
```

‖ Description

`algAlloc()` returns a table of memory records that describe the size, alignment, type, and memory space of all buffers required by an algorithm. If successful, this function returns a positive non-zero value indicating the number of records initialized.

The first argument to `algAlloc()` is a pointer to a structure that defines the creation parameters. This pointer may be `NULL`; however, in this case, `algAlloc()` must assume default creation parameters and must not fail. If the size of this argument is not equal to the size of `IVIDDEC2_Params` or the size of `IMP4HDVICPDEC_Params` then the default values of `IMP4HDVICPDEC_Params` structure are used.

The second argument to `algAlloc()` is an output parameter. `algAlloc()` may return a pointer to its parent IALG functions. If an algorithm does not require a parent object to be created, this pointer must be set to `NULL`.

The third argument is a pointer to a memory space of size `nbufs * sizeof(IALG_MemRec)` where, `nbufs` is the number of buffers returned by `algNumAlloc()` and `IALG_MemRec` is the buffer-descriptor structure defined in ialg.h. This version of the decoder returns failure if this is `NULL`.

After calling this function, `memTab[]` is filled up with the memory requirements of an algorithm.

For more details, see *TMS320 DSP Algorithm Standard API Reference* (literature number SPRU360).

‖ See Also

`algNumAlloc(), algFree()`

4.3.2 Initialization API

Initialization API is used to initialize an instance of the algorithm. The initialization parameters are defined in the `Params` structure (see 4.2 section for details).

‖ Name

`algInit()` – initialize an algorithm instance

‖ Synopsis

```
XDAS_Int32 algInit(IALG_Handle handle, IALG_MemRec
memTab[], IALG_Handle parent, IALG_Params *params);
```

‖ Arguments

```
IALG_Handle handle; /* algorithm instance handle*/

IALG_memRec memTab[]; /* array of allocated buffers */

IALG_Handle parent; /* handle to the parent instance */

IALG_Params *params; /* algorithm initialization
parameters */
```

‖ Return Value

```
IALG_EOK; /* status indicating success */

IALG_EFAIL; /* status indicating failure */
```

‖ Description

`algInit()` performs all initialization necessary to complete the run-time creation of an algorithm instance object. After a successful return `from algInit()`, the instance object is ready to be used to process data.

The first argument to `algInit()` is a handle to an algorithm instance. This value is initialized to the base field of `memTab[0]`. This version of the decoder returns failure if this is `NULL`.

The second argument is a table of memory records that describe the base address, size, alignment, type, and memory space of all buffers allocated for an algorithm instance. The number of initialized records is identical to the number returned by a prior call to `algAlloc()`. This version of the decoder returns failure if this is `NULL`.

The third argument is a handle to the parent instance object. If there is no parent object, this parameter must be set to `NULL`.

The last argument is a pointer to a structure that defines the algorithm initialization parameters. If any of the input fields in this parameter structure are invalid as defined by the table in section 4.2.2.1, this function uses default values wherever possible and raises a warning flag - `XDM_UNSUPPORTEDPARAM` through `extended_error` field of status structure during immediate `control()` API `(XDM_GETSTATUS)` call.

For more details, see *TMS320 DSP Algorithm Standard API Reference* (literature number SPRU360).

‖ See Also

`algAlloc(), algMoved()`

4.3.3 Control API

Control API is used to control the functioning and querying of the status of the algorithm instance during run-time. XDM_FLUSH and XDM_RESET control commands are supported in this implementation. XDM_GETBUFINFO, XDM_GETSTATUS, XDM_SETPARAMS, XDM_SETDEFAULT and XDM_GETVERSION are few other commands that are implemented.

‖ Name

control() – change run-time parameters and query the status

‖ Synopsis

```
XDAS_Int32 (*control) (IVIDDEC2_Handle handle,
IVIDDEC2_Cmd id, IVIDDEC2_DynamicParams *params,
IVIDDEC2_Status *status);
```

‖ Arguments

```
IVIDDEC2_Handle handle; /* algorithm instance handle */

IVIDDEC2_Cmd id; /* algorithm specific control commands*/

IVIDDEC2_DynamicParams *params /* algorithm run-time
parameters */

IVIDDEC2_Status *status /* algorithm instance status
parameters */
```

‖ Return Value

```
IALG_EOK; /* status indicating success */

IALG_EFAIL; /* status indicating failure */
```

‖ Description

This function changes the run-time parameters of an algorithm instance and queries the algorithm status. control() must only be called after a successful call to algInit() and must never be called after a call to algFree().

The first argument to control() is a handle to an algorithm instance.

The second argument is an algorithm specific control command. See XDM_CmdId enumeration for details.

The third and fourth arguments are pointers to the IVIDDEC2_DynamicParams and IVIDDEC2_Status data structures respectively. DynamicParams structure is used in the control() API only for XDM_SETPARAMS command, hence no validation check is performed on the contents of this structure and this structure pointer can be NULL as well for supported commands like XDM_GETBUFINFO , XDM_FLUSH, XDM_RESET, XDM_GETSTATUS, XDM_SETDEFAULT and GETVERSION.

> **Note:**
>
> If extended data structures are used, then the fourth argument must be a pointer to the extended Status data structure. Also, ensure that the size field is set to the size of the extended data structure. Depending on the value set for the size field, the algorithm uses either basic or extended parameters. However, in this implementation the status structure does not have any extended structure members.

‖ Preconditions

The following conditions must be true prior to calling this function; otherwise, its operation is undefined.

- `control()` can be called only after successful return from `algInit()` and `algActivate()`.
- `handle` must be a valid handle for the algorithm instance object.

‖ Postconditions

The following conditions are true immediately after returning from this function.

- If the control operation is successful, the return value from this operation is equal to `IALG_EOK`; otherwise it is equal to either `IALG_EFAIL` or an algorithm specific return value.
- If the control command is not supported/recognized, the return value from this operation is equal to `IVIDDEC2_EUNSUPPORTED`
- If the control operation is not successful for a supported command, the return value from this operation is equal to `IALG_EFAIL`.

> **Note:**
>
> If the operation is not successful for `IMP4HDVICPDEC_SETPARAMS` control command due to invalid dynamic parameter settings that are not supported by the decoder, it returns `IALG_EFAIL with XDM_UNSUPPORTEDPARAM` bit set in `extendedError` element of `IVIDDEC2_Status` structure.

‖ Example

See test application file, mpeg4vdec_ti_arm926testapp.c available in the \client\test\src sub-directory.

‖ See Also

`algInit()`, `algActivate()`, `process()`

4.3.4 Data Processing API

Data processing API is used for processing the input data.

‖ Name

`algActivate()` – initialize scratch memory buffers prior to processing.

‖ Synopsis

```
Void algActivate(IALG_Handle handle);
```

‖ Arguments

```
IALG_Handle handle; /* algorithm instance handle */
```

‖ Return Value

```
Void
```

‖ Description

`algActivate()` initializes any of the instance scratch buffers using the persistent memory that is part of the algorithm instance object.

The first (and only) argument to `algActivate()` is an algorithm instance handle. This handle is used by the algorithm to identify various buffers that must be initialized prior to calling any of the algorithm processing methods.

For more details, see *TMS320 DSP Algorithm Standard API Reference.* (literature number SPRU360).

‖ See Also

`algDeactivate()`

‖ Name

`process()` – basic encoding/decoding blocking call

‖ Synopsis

```
XDAS_Int32 (*process)(IVIDDEC2_Handle handle, XDM1_BufDesc
*inBufs, XDM_BufDesc *outBufs, IVIDDEC2_InArgs *inargs,
IVIDDEC2_OutArgs *outargs);
```

‖ Arguments

```
IVIDDEC2_Handle handle; /* algorithm instance handle */

XDM1_BufDesc *inBufs; /* algorithm input buffer descriptor
*/

XDM_BufDesc *outBufs; /* algorithm output buffer descriptor
*/

IVIDDEC2_InArgs *inargs /* algorithm runtime input
arguments */

IVIDDEC2_OutArgs *outargs /* algorithm runtime output
arguments */
```

‖ Return Value

```
IALG_EOK; /* status indicating success */

IALG_EFAIL; /* status indicating failure */
```

‖ Description

This function is a blocking call implementation for encoding/decoding process for the current frame.

The first argument to `process()` is a handle to an algorithm instance.

The second and third arguments are pointers to the input and output buffer descriptor data structures respectively (see `XDM_BufDesc` data structure for details). When the flush mode is 0, the number of input buffers supported is 1 and the number of output buffers supported is 2. If it is not equal to the above-mentioned values, this version of the decoder returns failure. In the flush mode (flush mode is 1) these fields are ignored.

The fourth argument is a pointer to the `IVIDDEC2_InArgs` data structure that defines the run-time input arguments for an algorithm instance object.

The last argument is a pointer to the `IVIDDEC2_OutArgs` data structure that defines the run - time output arguments for an algorithm instance object.

> **Note:**
>
> The `process()` API can be called with base or extended `InArgs` and `OutArgs` data structures. If you are using extended data structures, the fourth and fifth arguments must be pointers to the extended `InArgs` and `OutArgs` data structures respectively. Also, ensure that the `size` field is set to the size of the extended data structure. Depending on the value set for the `size` field, the algorithm uses either basic or extended parameters.

‖ Preconditions

The following conditions must be true prior to calling this function; otherwise, its operation is undefined.

- `process()` can only be called after a successful return from `algInit()` and `algActivate()`.
- `handle` must be a valid handle for the algorithm instance object.
- Buffer descriptor for input and output buffers must be valid when not in flush mode.
- Input buffers must have valid input data when not in flush mode.

‖ Postconditions

The following conditions are true immediately after returning from this function.

- If the process operation is successful, the return value from this operation is equal to `IALG_EOK`; otherwise it is equal to either `IALG_EFAIL` or an algorithm specific return value.
- After successful return from `process()` function, `algDeactivate()` can be called.

‖ Example

See test application file, mpeg4vdec_ti_arm926testapp.c available in the \client\test\src sub-directory.

‖ See Also

`algInit()`, `algDeactivate()`, `control()`

> **Note:**
>
> A video encoder or decoder cannot be pre-empted by any other video encoder or decoder instance. That is, you cannot perform task switching while encode/decode of a particular frame is in progress.

‖ Name

`algDeactivate()` – save all persistent data to non-scratch memory

‖ Synopsis

```
Void algDeactivate(IALG_Handle handle);
```

‖ Arguments

```
IALG_Handle handle; /* algorithm instance handle */
```

‖ Return Value

```
Void
```

‖ Description

`algDeactivate()` saves any persistent information to non-scratch buffers using the persistent memory that is part of the algorithm instance object.

The first (and only) argument to `algDeactivate()` is an algorithm instance handle. This handle is used by the algorithm to identify various buffers that must be saved prior to next cycle of `algActivate()` and processing.

For more details, see *TMS320 DSP Algorithm Standard API Reference* (literature number SPRU360).

‖ See Also

`algActivate()`

4.3.5 Termination API

Termination API is used to terminate the algorithm instance and free up the memory space that it uses.

‖ Name

`algFree()` – determine the addresses of all memory buffers used by the algorithm

‖ Synopsis

```
XDAS_Int32 algFree(IALG_Handle handle, IALG_MemRec
memTab[]);
```

‖ Arguments

```
IALG_Handle handle; /* handle to the algorithm instance */

IALG_MemRec memTab[]; /* output array of memory records */
```

‖ Return Value

```
XDAS_Int32; /* Number of buffers used by the algorithm */
```

‖ Description

`algFree()` determines the addresses of all memory buffers used by the algorithm. The primary aim of doing so is to free up these memory regions after closing an instance of the algorithm.

The first argument to `algFree()` is a handle to the algorithm instance.

The second argument is a table of memory records that describe the base address, size, alignment, type, and memory space of all buffers previously allocated for the algorithm instance.

> **Note:**
>
> In the current implementation `algFree()` API additionally resets HDVICP hardware co-processor and also releases DMA resources held by it. Thus, its important that this function is used only to release the resource at the end and not in between `process()`/`control()` API functions.

For more details, see *TMS320 DSP Algorithm Standard API Reference* (literature number SPRU360).

‖ See Also

`algAlloc()`