MPEG4 Advanced Simple Profile Decoder on C64x+

User's Guide

Literature Number: SPRUGT2
November 2009

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 Decoder implementation on the C64x+ 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) standard. XDM is an extension of the 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 C64x+ 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. 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.
- **Chapter 5 – Frequently Asked Questions,** provides answers to few frequently asked questions related to using this decoder.

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 Interface Standard (also known as XDAIS) specification.
- *Technical Overview of eXpressDSP - Compliant Algorithms for DSP Software Producers* (literature number SPRA579) describes how to make algorithms compliant with the TMS320 DSP Algorithm Standard which is part of TI's eXpressDSP technology initiative.
- *Using the TMS320 DSP Algorithm Standard in a Static DSP System* (literature number SPRA577) describes how an eXpressDSP-compliant algorithm may be used effectively in a static system with limited memory.
- *DMA Guide for eXpressDSP-Compliant Algorithm Producers and Consumers* (literature number SPRA445) describes the DMA architecture specified by the TMS320 DSP Algorithm Standard (XDAIS). It also describes two sets of APIs used for accessing DMA resources: the IDMA2 abstract interface and the ACPY2 library.
- *eXpressDSP Digital Media (XDM) Standard API Reference* (literature number SPRUEC8)

The following documents describe TMS320 devices and related support tools:

- *Design and Implementation of an eXpressDSP-Compliant DMA Manager for C6X1X* (literature number SPRA789) describes a C6x1x-optimized (C6211, C6711) ACPY2 library implementation and DMA Resource Manager.
- *TMS320C64x+ Megamodule* (literature number SPRAA68) describes the enhancements made to the internal memory and describes the new features which have been added to support the internal memory architecture's performance and protection.
- *TMS320C64x+ DSP Megamodule Reference Guide* (literature number SPRU871) describes the C64x+ megamodule peripherals.
- *TMS320C64x to TMS320C64x+ CPU Migration Guide* (literature number SPRAA84) describes migration from the Texas Instruments TMS320C64x™ digital signal processor (DSP) to the TMS320C64x+™ DSP.
- *TMS320C6000 Optimizing Compiler v 6.0 Beta User's Guide* (literature number SPRU187N) explains how to use compiler tools

such as compiler, assembly optimizer, standalone simulator, library-build utility, and C++ name demangler.

- *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide* (literature number SPRU732) describes the CPU architecture, pipeline, instruction set, and interrupts of the C64x and C64x+ DSPs

Related Documentation

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

- MPEG4 standard specified by Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG
- H.263 Standard (ITU-T Series H 02/98)

Abbreviations

The following abbreviations are used in this document.

Table 1-1. List of Abbreviations

Abbreviation	Description
API	Application Programming Interface
CBR	Constant Bit Rate
CPB	Constrained Parameters Bit-streams
DCT	Discrete Cosine Transform
DMA	Direct Memory Access
DMAN3	DMA Manager
DSS	Direct Satellite System
DTV	Digital Television
DVB	Digital Video Broadcast
DVD	Digital Versatile Disc
EVM	Evaluation Module
fps	Frames per second
IDCT	Inverse Discrete Cosine Transform
Kbps	Kilo bits per second
MPEG	Motion Picture Expert Group
XDAIS	eXpressDSP Algorithm Interface Standard
XDM	eXpressDSP Digital Media

Abbreviation	Description
ASP	Advanced Simple Profile

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 C64x+) and version number. The version number of the codec is included in the title of the release notes that accompanies this codec.

Trademarks

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 and XDM. It also provides an overview of TI's implementation of the MPEG4 Advanced Simple Profile Decoder on the C64x+ platform and its supported features.

Topic **Page**

1.1 Overview of XDAIS and XDM

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).

1.1.1 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. In order 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's 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).

1.1.2 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, or H.264) 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

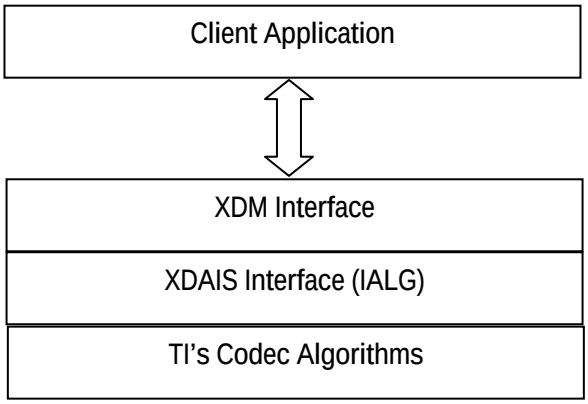
(For example audio, video, image, and speech). The XDM standard defines the following two APIs:

- `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.

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 Overview of MPEG4 Advanced Simple Profile Decoder

MPEG4 is a popular video algorithm defined by Motion Picture Expert Group (MPEG) for video conferencing applications.

Frames in a video sequence is categorized as I-frames, P-frames and B-frames. I-frames, or intra-frames, are encoded without reference to any other frame in the sequence, in the same manner as a still image. In contrast, P-frames, or predicted inter-frames, depend on information from a previous frame for its encoding. The input to the decoder can be MPEG4 or

H.263 encoded bit-stream. B-frames are predicted from the past and future reference.

The decoder first receives the bit-stream corresponding to the I-frame and decodes it. P-frames are reconstructed by adding the prediction information obtained by applying the motion vectors on the previously decoded frame with the residual information for the current frame. This process is repeated until the entire bit-stream is decoded.

The output of the decoder is a YUV sequence, which can be of format 420 planar and 422 interleaved.

From this point onwards, all references to MPEG4 Decoder means MPEG4 Advanced Simple Profile Decoder only.

1.3 Supported Services and Features

This user guide accompanies TI's implementation of MPEG4 Decoder on the C64x+ platform. This version of the codec has the following supported features:

- Supports MPEG4 Advanced Simple Profile, level 0, 1, 2, 3, 4 and 5
- Supports MPEG4 visual simple profile, level 0, 1, 2, 3, and 4A
- Supports H.263 profile 3 and level 10, 20, 30, 45, 50, 60, and 70
- Supports H.263 Annex-IJKT
- Supports post-processing filter, de-blocking, and de-ringing
- Supports spatial and temporal error concealment only for I and P progressive frames
- Contains optimized I and P flow to decode frames up to WVGA (854 x 480 and 480 x 854) and D1 (720x576) resolution at 30 fps
- Outputs are available in YUV 420 planar and 422 interleaved little endian formats
- Supports half pel and quarter pel interpolation
- Supports display width feature
- Supports single object
- Supports streams that are non-multiple of 16
- eXpressDSP Digital Media (XDM 1.0 IVIDDEC2) interface compliant
- Supports Frame level byte-swap. If it is enabled, algorithm will do byte-swap conversion at frame level dynamically. Also, encoded bytes per frame information need not be provided as input to the application.
- Supports Global Motion Compensation (GMC) 0 and 1 warp supported for progressive frame

This version of the codec does not support the following feature:

Global Motion Compensation (GMC) 2 and 3 warp

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

This section describes the hardware and software requirements for the normal functioning of the codec component.

2.1.1 Hardware

This codec has been built and tested on OMAP3530 EVM with XDS560 JTAG emulator.

This codec can be used on any of TI's C64x+ based platforms such as DM644x, DM643x, OMAP35xx and their derivatives.

2.1.2 Software

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

- **Development Environment:** This project is developed using Code Composer Studio version 3.2.40.8.
- **Code Generation Tools:** This project is compiled, assembled, archived, and linked using the code generation tools version 6.0.8.

2.2 Installing the Component

The codec component is released as a compressed archive. To install the codec, extract the contents of the zip file onto your local hard disk. The zip file extraction creates a top-level directory called 100_V_MPEG4_D_02_01, under which another directory named C64XPLUS_ASP_002 is created.

Figure 2-1 shows the sub-directories created in the C64XPLUS_ASP_002 directory.

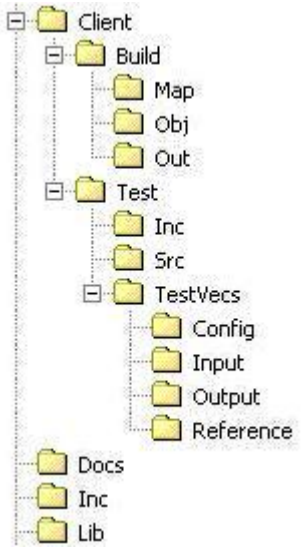


Figure 2-1. Component Directory Structure

Note: If you are installing an evaluation version of this codec, the directory name will be 100E_V_MPEG4_D_02_01.

Table 2-1 provides a description of the sub-directories created in the C64XPLUS_ASP_002 directory.

Table 2-1. Component Directories

Sub-Directory	Description
\Inc	Contains XDM related header files which allow interface to the codec library
\Lib	Contains the codec library file
\Docs	Contains user guide and datasheet
\Client\Build	Contains the sample test application project (.pjt) file
\Client\Build\Map	Contains the memory map generated on compilation of the code
\Client\Build\Obj	Contains the intermediate .asm and/or .obj file generated on compilation of the code
\Client\Build\Out	Contains the final application executable (.out) file generated by the sample test application
\Client\Test\Src	Contains application C files
\Client\Test\Inc	Contains header files needed for the application code
\Client\Test\TestVecs\Input	Contains input test vectors
\Client\Test\TestVecs\Output	Contains output generated by the codec
\Client\Test\TestVecs\Reference	Contains read-only reference output to be used for verifying against codec output
\Client\Test\TestVecs\Config	Contains configuration parameter files

2.3 Before Building the Sample Test Application

This codec is accompanied by a sample test application. To run the sample test application, you need DSP/BIOS and TI Framework Components (FC).

This version of the codec has been validated with DSP/BIOS version 5.31.02,and Framework Component (FC) version 2.20.01.

2.3.1 Installing DSP/BIOS

You can download DSP/BIOS from the TI external website:

https://www-a.ti.com/downloads/sds_support/targetcontent/bios/index.html

Install DSP/BIOS at the same location where you have installed Code Composer Studio. For example:

<install directory>\CCStudio_v3.2

The sample test application uses the following DSP/BIOS files:

- Header file, bcache.h available in the <install directory>\CCStudio_v3.2\<bios_directory>\packages \ti\bios\include directory.
- Library file, biosDM420.a64P available in the <install directory>\CCStudio_v3.2\<bios_directory>\packages \ti\bios\lib directory.

2.3.2 Installing Framework Component (FC)

You can download FC from the TI external website:

https://www-a.ti.com/downloads/sds_support/targetcontent/FC/index.html

Extract the FC zip file to the same location where you have installed Code Composer Studio. For example:

<install directory>\CCStudio_v3.2

The test application uses the following DMAN3 files:

- Library file, dman3.a64P available in the <install directory>\CCStudio_v3.2\<fc_directory>\packages \ti\sdo\fc\dman3 directory.
- Header file, dman3.h available in the <install directory>\CCStudio_v3.2\<fc_directory>\packages \ti\sdo\fc\dman3 directory.
- Header file, idma3.h available in the <install directory>\CCStudio_v3.2\<fc_directory>\fctools\packages \ti\xdais directory.

2.4 Building and Running the Sample Test application

This codec is accompanied by a sample test application. This application runs in TI's Code Composer Studio development environment. To build and run the sample application in Code Composer Studio, follow these steps:

1) Verify that you have installed TI's Code Composer Studio version 3.2.40.8 and code generation tools version 6.0.8.

2) Verify that the codec object library, m4h3dec_ti.l64P exists in the \Lib sub-directory.

3) Open the test application project file, TestAppDecoder.pjt in Code Composer Studio. This file is available in the \Client\Build sub-directory.

4) Select **Project > Build** to build the sample test application. This creates an executable file, TestAppDecoder.out in the \Client\Build\Out sub-directory.

5) Select **File > Load**, browse to the \Client\Build\Out sub-directory, select the codec executable created in step 4, and load it into Code Composer Studio in preparation for execution.

6) Select **Debug > Run** to execute the sample test application.

   The sample test application takes the input files stored in the \Client\Test\TestVecs\Input sub-directory, runs the codec, and uses the reference files stored in the \Client\Test\TestVecs\Reference sub-directory to verify that the codec is functioning as expected.

7) On successful completion, the application displays one of the following messages for each frame:

   - "Decoder compliance test passed/failed"(for compliance check mode)
   - "Decoder output dump completed" (for output dump mode)

2.5 Configuration Files

This codec is shipped along with:

- Generic configuration file (Testvecs.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.

2.5.1 Generic Configuration File

The sample test application shipped along with the codec uses the configuration file, Testvecs.cfg for determining the input and reference files for running the codec and checking for compliance. The Testvecs.cfg file is available in the \Client\Test\TestVecs\Config sub-directory.

The format of the Testvecs.cfg file is:

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

where:

- `X` may be set as:
  - 1 - for compliance checking, no output file is created
  - 0 - for writing the output to the output file
- `Config` is the Decoder configuration file. For details, see Section 2.5.2.
- `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.cfg file is as shown:

```
1
..\..\Test\TestVecs\Config\Testparams.cfg
..\..\Test\TestVecs\Input\davincieffect_qcif_256kbps.m4v
..\..\Test\TestVecs\Reference\davincieffect_qcif_256kbps_420.yuv
0
..\..\Test\TestVecs\Config\Testparams.cfg
..\..\Test\TestVecs\Input\davincieffect_qcif_256kbps.m4v
..\..\Test\TestVecs\Output\ davincieffect_qcif_256kbps.yuv
```

2.5.2 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 is as shown:

```
# New Input File Format is as follows
# <ParameterName> = <ParameterValue> # Comment
############################################################
# Parameters
############################################################
ImageWidth     = 864     # Image width in Pels, must be multiple of 16
ImageHeight    = 480     # Image height in Pels, must be multiple of 16
ChromaFormat    = 1      # 1 => XMI_YUV_420P, 3 => XMI_YUV_422IBE,
                         # 4 => XMI_YUV_422ILE
FramesToDecode  = 5      # Number of frames to be decoded
DeblockFlag  = 0
DeringFlag   = 0
ConcealFlag     = 0    # concealment is not supported for B frames
FrameLevelByteSwapFlag     = 1      # Enables byteswap at frame level
```

Any field in the `IVIDDEC2_Params` structure (see Section 4.2.1.8) can be set in the Testparams.cfg file using the syntax shown above. If you specify additional fields in the Testparams.cfg file, ensure to modify the test application appropriately to handle these fields.

2.6 Standards Conformance and User-Defined Inputs

To check the conformance of the codec for the default input file shipped along with the codec, follow the steps as described in Section 2.4.

To check the conformance of the codec for other input files of your choice, follow these steps:

1) Copy the input files to the \Client\Test\TestVecs\Inputs sub-directory.

2) Copy the reference files to the \Client\Test\TestVecs\Reference sub-directory.

3) Edit the configuration file, Testvecs.cfg available in the \Client\Test\TestVecs\Config sub-directory. For details on the format of the Testvecs.cfg file, see Section 2.5.1.

4) Execute the sample test application. On successful completion, the application displays one of the following message for each frame:

   - o “Decoder compliance test passed/failed” (for compliance check mode)
   - o “Decoder output dump completed” (for output dump mode)

If you have chosen the option to write to an output file (x is 0), you may use any standard file comparison utility to compare the codec output with the reference output and check for conformance.

2.7 Uninstalling the Component

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

2.8 Evaluation Version

If you are using an evaluation version of this codec a Texas Instruments logo will be visible in the output.

Note:

Compliance test succeeds only for the example input file provided with evaluation package, due to the presence of Texas instruments logo in the input file. It should not be checked for other inputs.

This page is intentionally left blank

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 `IVIDDEC2` extended class of the MPEG4 Decoder library. The main test application files are TestAppDecoder.c and TestAppDecoder.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.

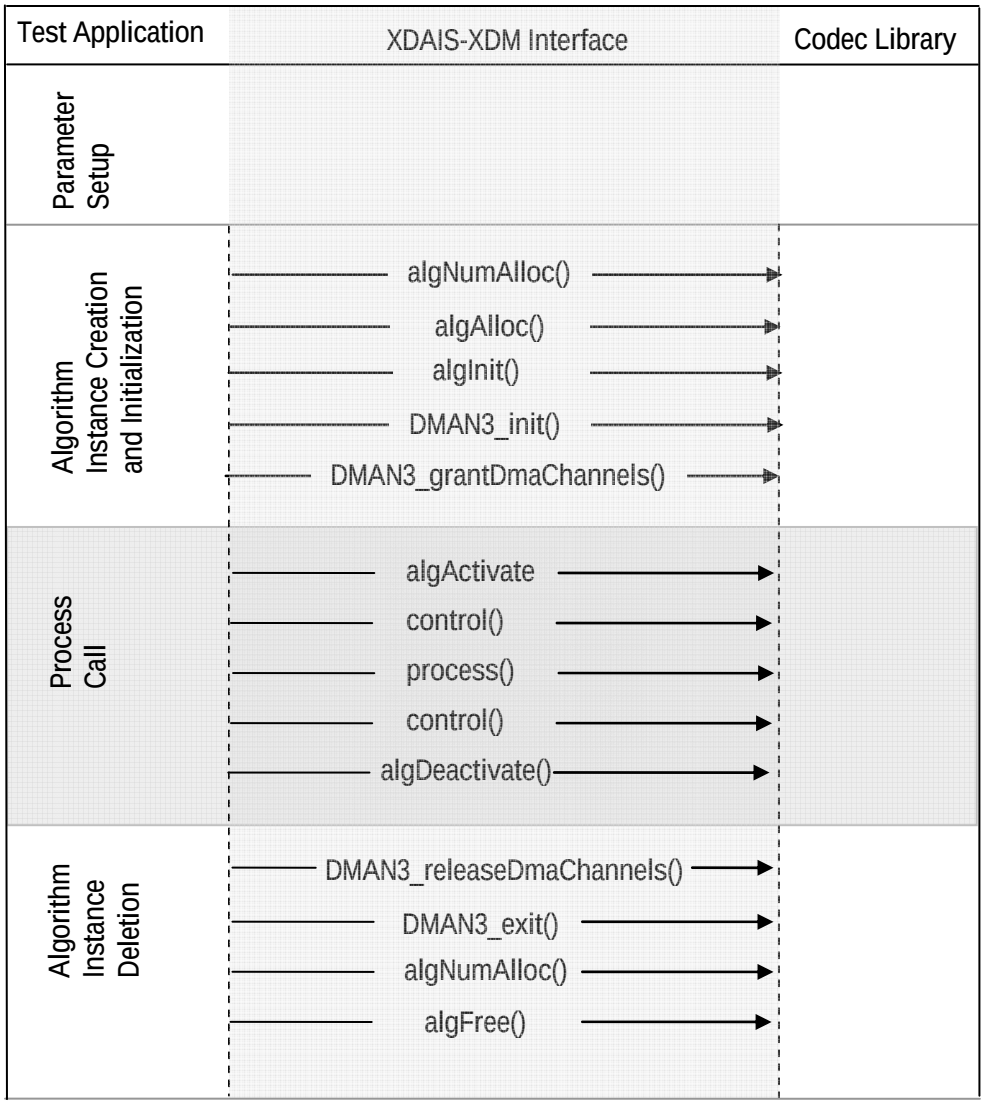


Figure 3-1. Test Application Sample Implementation

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.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.

   For more details on the configuration files, see Section 2.5.

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

4) Initializes the various DMAN3 parameters.

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

After successful completion of the these 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 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. This requires initialization of DMA Manager Module and grant of DMA resources. This is implemented by calling DMAN3 interface functions in the following sequence:

1) `DMAN3_init()` - To initialize the DMAN module.

2) `DMAN3_grantDmaChannels()` - To grant the DMA resources to the algorithm instance.

Note:

DMAN3 function implementations are provided in dman3.a64P library.

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) Calls the `process()` function to encode/decode a single frame of data. The behavior of the algorithm can be controlled using various dynamic parameters (see Section 4.2.1.9). The inputs to the process function are input and output buffer descriptors, pointer to the `IVIDDEC2_InArgs` and `IVIDDEC2_OutArgs` structures.

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 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 every time before the next call. The do-while loop breaks off either when an error condition occurs or when the input buffer exhausts. It also protects the `process()` call from file operations by placing appropriate calls for cache operations as well. The test application does a cache invalidate for the valid input buffers before `process()` and a cache write back invalidate for output buffers after `process()`.

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 encoding/decoding is complete, the test application releases the DMA channels granted by the DMA Manager interface and deletes the current algorithm instance. The following APIs are called in sequence:

1) `DMAN3_releaseDmaChannels()` - To remove logical channel resources from an algorithm instance.

2) `DMAN3_exit()` - To free DMAN3 memory resources.

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

4) `algFree()` - To query the algorithm to get the memory record information.

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 1.0, decoder does not ask for frame buffer at the time of `alg_create()`. It uses buffer from `XDM1_BufDesc *outBufs`, which the decoder gets during each decode process call. The framework needs to ensure that it does not overwrite to buffers, which are locked by the codec.

```
m4h3VDEC_create();
m4h3VDEC_control(XDM_GETBUFINFO); /* Returns default PAL D1
size */
do{
m4h3VDEC_decode(); //call the decode API
m4h3VDEC_control(XDM_GETBUFINFO); /* updates the memory re-
quired as per the size parsed in stream header */
}
while(all frames)
```

Note:

- Application can take the information returned by the control function with the `XDM_GETBUFINFO` command and change the size of the buffer passed in the next process call.
- Application can also re-use the extra buffer space of the first frame, if the above control call returns a smaller size than the one returned in the first call.

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

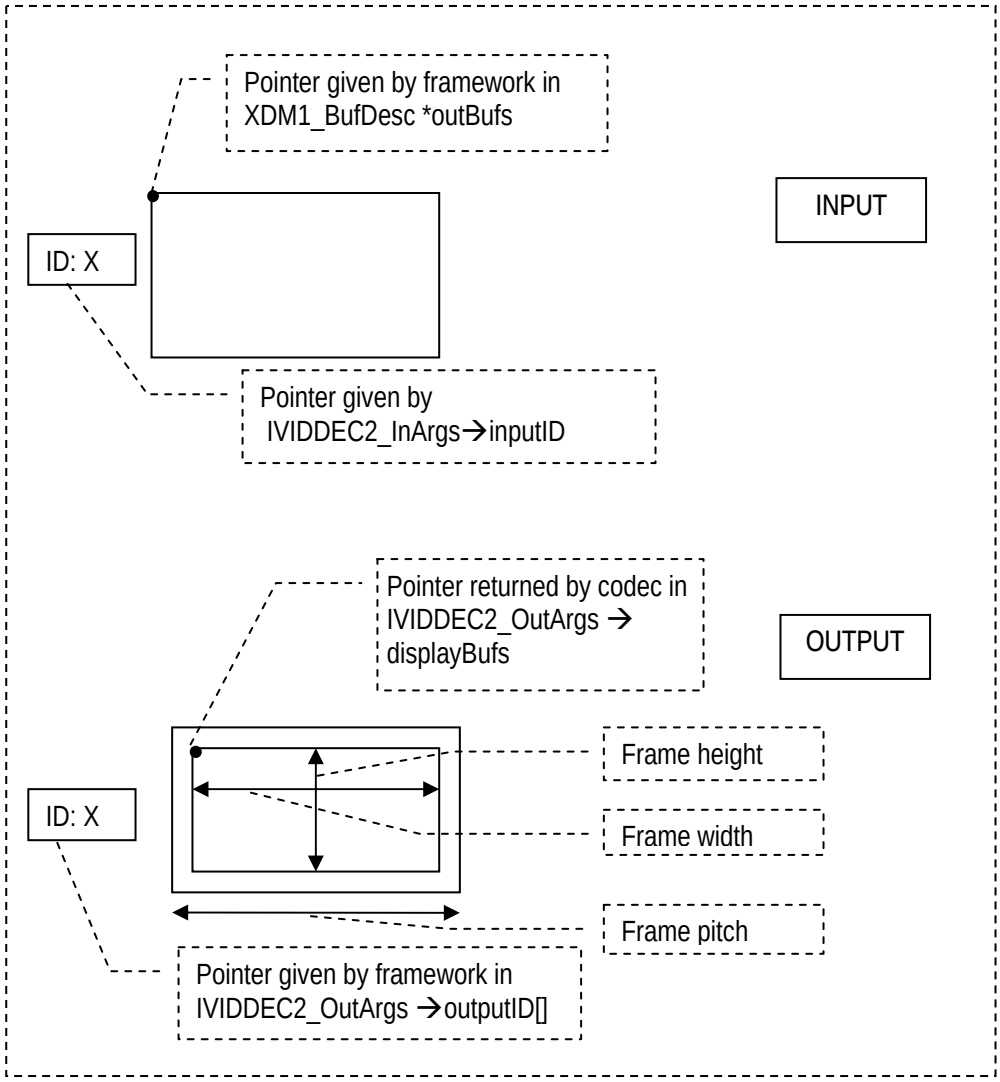


Figure 3-3. Frame Buffer Pointer Implementation

Note:

- Frame pointer returned by codec in `displayBufs` will point to actual start location of picture
- Frame height and width will return the actual height and width (after removing cropping and padded width)
- Frame pitch indicates the offset between the pixels at the same horizontal coordinate on two consecutive lines.

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

Note:

`BufferID` given back in `IVIDDEC2_OutArgs ->outputID[]` are just for display purpose. Application should not consider it free unless it comes as part of `IVIDDEC2_OutArgs->freeBufID[]`

3.2.2 Frame Buffer Management by Application

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

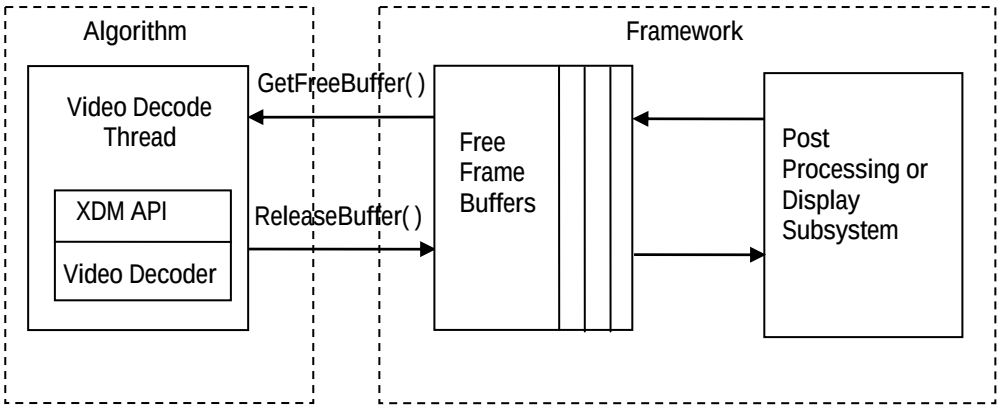


Figure 3-2. Interaction of Frame Buffers Between Application and Framework.

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

- `BUFFMGR_Init()` - This function is called by the test application to initialize the global buffer element array to default and to allocate one frame of memory data for the first process call output buffers depending on the maximum width and maximum height supplied in params.

- BUFFMGR_ReInit() - This function allocates the remaining luma and chroma buffers required by the stream based on the actual picture width and height obtained after first process call.
- BUFFMGR_GetFreeBuffer() - This function searches for a free buffer in global buffer array and returns the address of that element. If none of the elements are free, then it returns NULL.
- BUFFMGR_ReleaseBuffer() - This function takes an array of buffer-IDs that are released by the test-app. 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.
- BUFFMGR_DeInit()- This function releases all memory allocated by buffer manager.

3.3 Sample Test Application

The test application exercises the IVIDDEC2 extended class of the MPEG4 Decoder. The process() implementation is as shown.

```
/* Main Function acting as a client for Video Decode
Call*/
BUFFMGR_Init();

TestApp_SetInitParams(&params.viddecParams);

/*---------------- Decoder creation -----------------*/
handle = (IALG_Handle) m4h3VDEC_create();

/* Get Buffer information */
 m4h3VDEC_control(handle, XDM_GETBUFINFO);

/* Do-While Loop for Decode Call for a given stream */
do
{
/* Read the bit-stream in the Application Input Buffer */
validBytes = ReadByteStream(inFile);

/* Get free buffer from buffer pool */
buffEle = BUFFMGR_GetFreeBuffer();

/* Optional: Set Run-time parameters in the Algorithm
through control() */
m4h3VDEC_control(handle, XDM_SETPARAMS);
/*-------------------------------------------------*/
/* Start the process : To start decoding a frame */
/* This will always follow a m4h3VDEC_decode_end call */
/*-------------------------------------------------*/
retVal = m4h3VDEC_decode ( handle, (XDM1_BufDesc
*)&inputBufDesc, (XDM_BufDesc *)&outputBufDesc,
(IVIDDEC2_InArgs *)&inArgs, (IVIDDEC2_OutArgs *)&outArgs
);

/* Get the status of the decoder using control call */
m4h3VDEC_control(handle, Im4h3VDEC_GETSTATUS);

/* Get Buffer information : */
m4h3VDEC_control(handle, XDM_GETBUFINFO);
```

```
/* Optional: Reinit the buffer manager in case the
/* frame size is different*/
BUFFMGR_ReInit();

/* Always release buffers - which are released from
/* the algorithm side -back to the buffer manager */
BUFFMGR_ReleaseBuffer((XDAS_UInt32 *)outArgs.freeBufID);
}

while(1); /* end of Do-While loop - which decodes
frames*/

ALG_delete (handle); BUFFMGR_DeInit();
```

Note:
This sample test application does not depict the actual function parameter or control code. It shows the basic flow of the code.

This page is intentionally left blank

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.

Table 4-1. List of Enumerated Data Types

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
IVIDEO_FrameType	IVIDEO_I_FRAME	Intra coded frame
	IVIDEO_P_FRAME	Forward inter coded frame
	IVIDEO_B_FRAME	Bi-directional inter coded frame
	IVIDEO_IDR_FRAME	Intra coded frame that can be used for refreshing video content.
	IVIDEO_II_FRAME	Interlaced Frame, both fields are I frames.
	IVIDEO_IP_FRAME	Interlaced Frame, first field is an I frame, second field is a P frame.
	IVIDEO_IB_FRAME	Interlaced Frame, first field is an I frame, second field is a B frame.
	IVIDEO_PI_FRAME	Interlaced Frame, first field is a P frame, second field is an I frame.
	IVIDEO_PP_FRAME	Interlaced Frame, both fields are P frames.
	IVIDEO_PB_FRAME	Interlaced Frame, first field is a P frame, second field is a B frame.
	IVIDEO_BI_FRAME	Interlaced Frame, first field is a B frame, second field is an I frame.
	IVIDEO_BP_FRAME	Interlaced Frame, first field is a B frame, second 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	Default value is set to IVIDEO_I_FRAME

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
IVIDEO_ContentType	IVIDEO_CONTENTTYPE_NA	-1, Frame type is not available.
	IVIDEO_PROGRESSIVE	0, Progressive frame.
	IVIDEO_PROGRESSIVE_FRAME	Is equal to IVIDEO_PROGRESSIVE.
	IVIDEO_INTERLACED	1, Interlaced frame.
	IVIDEO_INTERLACED_FRAME	IVIDEO_INTERLACED.
	IVIDEO_INTERLACED_TOPFIELD	2, Interlaced picture, top field.
	IVIDEO_INTERLACED_BOTTOMFIELD	3, Interlaced picture, bottom field.
IVIDEO_FrameSkip	IVIDEO_NO_SKIP	Do not skip the current frame.
	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)
	IVIDEO_SKIP_IB	Skip I and B frame/field(s)
	IVIDEO_SKIP_PB	Skip P and B frame/field(s)
	IVIDEO_SKIP_IPB	Skip I/P/B/BI frames
	IVIDEO_SKIP_IDR	Skip IDR Frame
	IVIDEO_SKIP_DEFAULT	Default value is set to IVIDEO_NO_SKIP
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.
XDM_ChromaFormat	XDM_CHROMA_NA	-1, Chroma format not applicable.
	XDM_YUV_420P	YUV 4:2:0 planar

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_YUV_422P	YUV 4:2:2 planar. Not supported in this version of MPEG4 Decoder.
	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)
	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_CHROMAFORMAT_DEFAULT	Default value is set to XDM_YUV_422ILE
XDM_CmdId	XDM_GETSTATUS	Query algorithm instance to fill Status structure
	XDM_SETPARAMS	Set run-time dynamic parameters through the DynamicParams structure.
	XDM_RESET	Reset the algorithm
	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. The result will be returned in the @c data field of the respective _Status structure.
XDM_DecMode	XDM_DECODE_AU	Decode entire access unit

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_PARSE_HEADER	Decode only header
XDM_ErrorBit	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 encoding) ❑ 0 - Recoverable error

Note:

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

- Bit 16-32: Reserved
- Bit 8: Reserved
- Bit 0-7: Codec and implementation specific

The algorithm can set multiple bits to 1 depending on the error condition.

The MPEG4 Decoder specific error status messages are listed in Table 4-2. The value column indicates the decimal value of the last 8-bits reserved for codec specific error statuses.

Table 4-2. MPEG4 Decoder Error Status

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
M4H3DEC_ERROR	M4H3DEC_ERROR_Failure	1	Decoder failed due to some error in processing
	M4H3DEC_ERROR_nonMpegStream	2	Not a valid MPEG4 stream
	M4H3DEC_ERROR_nonVideoMpegStream	3	Video content not present in the stream
	M4H3DEC_ERROR_unSupportedProfile	4	Profile not supported
	M4H3DEC_ERROR_invalidBitstreamAddress	5	Corrupted input to the decoder
	M4H3DEC_ERROR_invalidVideoObjectLayerStartCode	6	Incorrect video object layer start code
	M4H3DEC_ERROR_nullBitstreamAddress	7	Corrupted input to the decoder
	M4H3DEC_ERROR_insufficientData	8	Incorrect data, ask the client to feed in the correct data
	M4H3DEC_ERROR_unsupportedVOL_verid	9	Video object layer version ID not supported
	M4H3DEC_ERROR_invalidAspectRatio	10	Aspect ratio not correct
	M4H3DEC_ERROR_invalidChromaFormat	11	Chroma format not correct
	M4H3DEC_ERROR_unsupportedVOLShape	12	Any shape other than rectangular not supported
	M4H3DEC_ERROR_invalidVOPTimeIncrementResolution	13	Incorrect VOP time increment
	M4H3DEC_ERROR_unsupportedFeatureOBMC	15	OBMC not supported
	M4H3DEC_ERROR_unsupportedVideoDataPrecision	16	Only 8 bits per pixel images supported
	M4H3DEC_ERROR_unsupportedObjectType	17	Only simple object type supported
	M4H3DEC_ERROR_unsupportedFeatureScalability	19	Scalability not supported
	M4H3DEC_ERROR_invalidCallingOrder,	20	Incorrect input bit-stream

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
	M4H3DEC_ERROR_invalidVideoObjectSC,	21	Video object start code not correct
	M4H3DEC_ERROR_invalidVOPSC	22	Not a video object plane start code
	M4H3DEC_ERROR_invalidQuant	23	Quant value not correct
	M4H3DEC_ERROR_invalidFcode	24	Fcode value not correct
	M4H3DEC_ERROR_invalidMBnumInVPH	25	Invalid number of macro blocks in video packet
	M4H3DEC_ERROR_endOfSequence	26	End of sequence reached
	M4H3DEC_ERROR_invalidGOBnum	27	GOB number not valid
	M4H3DEC_ERROR_corruptedHeader	28	Input stream has corrupted header
	M4H3DEC_ERROR_corruptedBitStream	29	Input is a corrupted bit-stream
	M4H3DEC_ERROR_unsupportedFeatureSprite	31	Sprite not supported
	M4H3DEC_ERROR_exceededResolution	33	Resolution exceeded the valid values
	M4H3DEC_ERROR_unsupportedFeatureIntraDcVlcThreshold	34	Non-zero value of intra DC VLC threshold is not supported
	M4H3DEC_ERROR_invalidValue	35	Invalid value of some parameter decoded
	M4H3DEC_ERROR_stuffingInMB	36	Incorrect input given to decoder, caught as stuffing bit
	M4H3DEC_ERROR_numMbRowsInVpExceeded	37	Input bit-stream has exceeded the maximum number of macro block rows in video packet
	M4H3DEC_TI_ERROR_cannotdecodempeg4	38	Used for H.263 build only, mpeg4 inputs cannot be decoded
	M4H3DEC_TI_ERROR_incorrectWidthHeight	39	Width and height of the input sequence is not supported by the current build
	M4H3DEC_TI_ERROR_insufficientMemory	40	Internal memory allocated is insufficient for frame decoding
	M4H3DEC_TI_ERROR_consumedAllInputBits	41	Decoder has consumed all the input bits
	M4H3DEC_TI_ERROR_noRefBufferToFlush	42	Decoder has no output buffer to flush

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
	M4H3DEC_TI_ERROR_missingFirstIFrame	43	First I frame is missing
	M4H3DEC_TI_ERROR_invalidDmvLength	44	Invalid Dmv Length
	M4H3DEC_TI_ERROR_unsupportedNumberOfWarpingPointGMC	45	Number of warping points in GMC are unsupported
	M4H3DEC_TI_ERROR_invalidWarpPoint	46	Invalid Warp point
	M4H3DEC_TI_ERROR_unsupportedFeatureInterlaceSprite	47	Interlace Sprite feature is unsupported
	M4H3DEC_TI_ERROR_unsupportedFeatureDPSprite	48	DP Sprite feature is unsupported
	M4H3DEC_TI_ERROR_invalidToolsSimpleObject	49	Invalid Tools Simple Object
	M4H3DEC_TI_ERROR_nullPtr	50	NULL Buffer pointer is passed
	M4H3DEC_TI_ERROR_bufSizeError	51	Invalid value for the size parameter
	M4H3DEC_TI_ERROR_unsupportedFeature	52	Unsupported feature

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 descriptor

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
`*buf`	`XDAS_Int8`	Input	Pointer to the buffer
`bufSize`	`XDAS_Int32`	Input	Size of 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
*buf	XDAS_Int8	Input	Pointer to the buffer
bufSize	XDAS_Int32	Input	Size of buffer in bytes
accessMask	XDAS_Int32	Output	If the buffer was not accessed by the algorithm processor (for example, it was filled by DMA or other hardware accelerator that does not write through the algorithm's CPU), then bits in this mask should not 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 Decoder, the buffer details are:

- Number of input buffer required is 1
- Number of output buffer required is 1 for YUV 422ILE, if `mbDataFlag` is set 0
- Number of output buffer required is 2 for YUV 422ILE, if `mbDataFlag` is set 1
- Number of output buffer required is 3 for YUV 420P, if `mbDataFlag` is set 0
- Number of output buffer required is 4 for YUV 420P, if `mbDataFlag` is set 1
- The output buffer address needs to be 64 bit aligned.
- There is no restriction on input buffer size except that it should contain atleast one frame of encoded data.
- Padding of 128 bytes of zeroes should be done by the application at the end of the input buffer for decoding error streams. This is recommended to avoid decoder from accessing beyond `numBytes` provided, due to corrupted bit-stream.
- The output buffer sizes (in bytes) for worst case HDTV_1080I format are:
  - For YUV 422ILE:
    Buffer = 1920 * 1080 * 2
  - For YUV 420P:
    Y buffer = 1920 * 1080
    U buffer = 960 * 544
    V buffer = 960 * 544

These are the maximum buffer sizes but you can reconfigure depending on the format of the bit-stream.

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	Input	Number of buffers
frameWidth	XDAS_Int32	Input	Width of the video frame
frameHeight	XDAS_Int32	Input	Height of the video frame
framePitch	XDAS_Int32	Input	Frame pitch use to store the frame
bufDesc[IVIDEO_MAX_YUV_BUFFERS]	XDM1_SingleBufDesc	Input	Pointer to the vector containing buffer addresses
extendedError	XDAS_Int32	Input	Extended error field
frameType	XDAS_Int32	Input	Indicates the decoded frame type as IVIDEO_FrameType enumerator type
topFieldFirstFlag	XDAS_Int32	Input	Flag to indicate when the application should display the top field first
repeatFirstFieldFlag	XDAS_Int32	Input	Flag to indicate when the first field should be repeated
frameStatus	XDAS_Int32	Input	Frame status of IVIDEO_Output
repeatFrame	XDAS_Int32	Input	Number of times the display process needs to repeat the display progressive frame
contentType	XDAS_Int32	Input	Content type of the buffer
chromaFormat	XDAS_Int32	Input	XDM_Chroma buffer

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.

Fields

Field	Data type	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes.
maxHeight	XDAS_Int32	Input	Maximum video height to be supported in pixels ❑ Min : 1 ❑ Max: : 1088 ❑ Default value is 1088
maxWidth	XDAS_Int32	Input	Maximum video width to be supported in pixels ❑ Min : 1 ❑ Max: : 1920 ❑ Default value is 1920
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 second. For example, if bit-rate is 10 Mbps, set

Field	Data type	Input/ Output	Description
			this field to 10485760.
dataEndianness	XDAS_Int32	Input	Endianness of input data. See XDM_DataFormat enumeration for details. Default value is XDM_BYTE.
forceChromaFormat	XDAS_Int32	Input	Sets the output to the specified format. For example, if the output should be in YUV 4:2:2 interleaved (little endian) format, set this field to XDM_YUV_422ILE. See XDM_ChromaFormat enumeration for details. Default value is XDM_YUV_422ILE.

Note:

- MPEG4 Decoder does not use the maxFrameRate and maxBitRate fields for creating the algorithm instance.
- Maximum video height and width supported are 1088 pixels and 1920 pixels respectively (for HDTV_1080I format).

4.2.1.9 IVIDDEC2_DynamicParams

Description

This structure defines the run-time 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.

Fields

Field	Data type	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes.
decodeHeader	XDAS_Int32	Input	Number of access units to decode: - 0 (XDM_DECODE_AU) - Decode entire frame including all the headers - 1 (XDM_PARSE_HEADER) - Decode only one NAL unit Default value is 0.

Field	Data type	Input/ Output	Description
displayWidth	XDAS_Int32	Input	If the field is set to: ❑ 0 - Uses decoded image width as pitch ❑ If any other value greater than the decoded image width is given, then this value in pixels is used as pitch. ❑ Min : 0 ❑ Max : No restriction ❑ Default value is 0.
frameSkipMode	XDAS_Int32	Input	Frame skip mode. See IVIDEO_FrameSkip enumeration for details.
frameOrder	XDAS_Int32	Input	Frame display order ❑ 0 (IVIDDEC2_DISPLAY_ORDER) - The decoder provides decoded output in the actual order of displaying the output buffer. ❑ 1 (IVIDDEC2_DECODE_ORDER) - The decoder provides decoded output in the order of decoding. Default value : 0
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.
mbDataFlag	XDAS_Int32	Input	Flag to indicate that the algorithm should generate MB Data in addition to decoding the data

Note:

In this version of MPEG4 decoder, display width can not be changed dynamically while executing a stream.
In case mbDataFlag is set to 1, the decoder fills the last buffer among the buffers supplied within outBufs->bufs[] with the decoded MB data generated by the Decoder module. Application performs cache invalidation if the data is intended to be used by other processor.

In this version of MPEG-4, decoder reports MB wise error status as follows:

- 0 means no error
- non-zero value (between 1 to 4) means error

To enable the MB wise status reporting, mbDataFlag should be enabled. For each MB, decoder uses one byte to report status.

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 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:

For B-frames, MPEG4 Decoder copies the current inputID value to the outputID value of IVIDDEC2_OutArgs structure. However, for I and P frames, inputID value of the previous reference frame is copied to the outputID value of IVIDDEC2_OutArgs structure.

MPEG4 Decoder copies the inputID value to the outputID value of IVIDDEC2_OutArgs structure after factoring in a display delay of 1.

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 XDM_ErrorBit enumeration for details.

Field	Data type	Input/ Output	Description
data	XDM_SingleBufDesc	Output	Buffer Information structure for Information passing buffer
maxNumDisplayBufs	XDAS_Int32	Output	The maximum number of buffers required by the coded
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	Output chroma format. See XDM_ChromaFormat enumeration for details.
bufInfo	XDM_AlgBufInfo	Output	Input and output buffer information. See XDM_AlgBufInfo data structure for details.

Note:

In case GET_STATUS, data points to the buffer containing the quant values for all the macro blocks of the frame, minimum buffer size required is 3600*2. In case GET_VERSION, data points to the version, minimum buffer size required is 10.

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[XDM_MAX_IO_BUFFERS]	XDAS_Int32	Output	Output ID corresponding to displayBufs. A value of zero (0) indicates an invalid ID. The first zero entry

Field	Data type	Input/ Output	Description
			in an array will indicate end of valid output IDs within the array. Hence, the application can stop 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 will be incomplete. However, the algorithm will provide incomplete decoded frame data in case application may choose to use it for error recovery purposes.
displayBufs[XDM_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.
free-BufID[IVIDDEC2_FREE_BUFF_SIZE]	XDAS_Int32	Output	This is an array of inputIDs corresponding to the frames that have been unlocked in the current process call .
outBufsInUseFlag	XDAS_Int32	Output	Flag to indicate that the outBufs provided with the process() call are in use. No outBufs are required to be supplied with the next process() call.

Note:

For B-frames, MPEG4 Decoder copies the current inputID value to the outputID value of IVIDDEC2_OutArgs structure. However, for I and P frames, inputID value of the previous reference frame is copied to the outputID value of IVIDDEC2_OutArgs structure.

4.2.2 MPEG4 Decoder Data Structures

This section includes the following MPEG4 Decoder specific extended data structures:

- IM4H3DEC_Params
- IM4H3DEC_DynamicParams
- IM4H3DEC_InArgs
- IM4H3DEC_Status
- IM4H3DEC_OutArgs

4.2.2.1 IM4H3DEC_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.

4.2.2.2 IM4H3DEC_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
postDeblock	XDAS_Int32	Input	De-blocking post process ❑ 0 - Off (default) ❑ 1 - On

Field	Data type	Input/ Output	Description
postDering	XDAS_Int32	Input	De-ringing post process ❑ 0 - Off (default) ❑ 1 - On
errorConceal	XDAS_Int32	Input	Spatial and temporal error concealment ❑ 0 - Off (default) ❑ 1 - On
FrameLevelByteSwap	XDAS_UInt32	Input	Performs byte swap only for the number of bytes of the frame to be decoded
useHighPrecIdctQp1	XDAS_UInt32	Input	Selects IDCT algorithm used ❑ 0: Use Default Idct ❑ 1: Use High precision Idct for QP 1

4.2.2.3 IM4H3DEC_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 IM4H3DEC_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

Field	Data type	Input/ Output	Description
vopTimeIncrementResolution	XDAS_Int32	Output	This is a 16-bit unsigned integer that indicates the number of evenly spaced subintervals, called ticks, within one modulo time. One modulo time represents the fixed interval of one second.
vopTimeIncrement	XDAS_Int32	Output	This value represents the number of ticks between two successive VOPs in the display order.
brokenLink	XDAS_Int32	Output	Broken link flag in GOV header
cloedGOV	XDAS_Int32	Output	Closed GOV flag in GOV header
actualHeight	XDAS_Int32	Output	Decoded height in pixels
actualWidth	XDAS_Int32	Output	Decoded width in pixels

Note:

- Default values of extended parameters are used when size field of viddecOutArgs is set to BASE mode.
- closed_gov and brokenlink flags are used when the encoded stream is edited. When brokenlink flag is set to 1, it implies that the references for B frames in the current GOV may not be present and output of B frame may not be correct. In such a case, the application may avoid displaying that B frame.

4.2.2.5 IM4H3DEC_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
profile	XDAS_Int32	Output	MPEG4 profile - 0: Simple Profile - 1: Advance Simple Profile

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* (literature number 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
**parentFxns, IALG_MemRec memTab[]);
```

‖ Arguments

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

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

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

‖ Return Value

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

‖ 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.

The second argument to `algAlloc()` is an output parameter. `algAlloc()` may return a pointer to its parent's 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.

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 Data Structures 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]`.

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()`.

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 parameter pointer is NULL the algorithm assumes default creation parameters and returns success.

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 for controlling the functioning of the algorithm instance during run-time. This is done by changing the status of the controllable parameters of the algorithm during run-time. These controllable parameters are defined in the `Status` data structure (see Data Structures section for details).

‖ 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's 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.

> **Note:**
>
> If you are using extended data structures, the third and fourth arguments must be pointers to the extended `DynamicParams` and `Status` 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.

- `control()` can only be called after a successful return from `algInit()` and `algActivate()`.
- If algorithm uses DMA resources, `control()` can only be called after a successful return from `DMAN3_init()`.
- `handle` must be a valid handle for the algorithm's 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 recognized, the return value from this operation is not equal to `IALG_EOK`.

‖ Example

See test application file, TestAppDecoder.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's scratch buffers using the persistent memory that is part of the algorithm's 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's processing methods.

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

‖ See Also

`algDeactivate()`

‖ Name

`process()` – basic encoding/decoding 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 does the basic encoding/decoding. 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 `XDM1_BufDesc` and `XDM_BufDesc` data structure for details).

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:
>
> 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()`.

- If algorithm uses DMA resources, `process()` can only be called after a successful return from `DMAN3_init()`.
- `handle` must be a valid handle for the algorithm's instance object.
- Buffer descriptor for input and output buffers must be valid.
- Input buffers must have valid input data.

‖ 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, TestAppDecoder.c available in the \Client\Test\Src sub-directory.

‖ See Also

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

Note:

A video encoder or decoder cannot be preempted by any other encoder or decoder instance. That is, you cannot perform task switching while encode/decode of a particular frame is in progress. Pre-emption can happen only at frame boundaries and after `algDeactivate()` is called.

‖ 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's 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.

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

‖ See Also

`algAlloc()`

This page is intentionally left blank

Chapter 5

Frequently Asked Questions

This chapter provides answers to few frequently asked questions related to using this decoder.

Table 5-1. FAQs for MPEG4 Decoder on OMAP3530.

Question	Answer
What is the input/output format for MPEG4 codec?	Input format supported Big-endian stream and 32 bit little-endian stream. Output can be configured as 420 planar or 422 ILE.
Does this decoder support Advance Simple Profile?	MPEG4 Advance Simple Profile can be supported with limited functionality.
Does the decoder support any post processing?	MPEG4 decoder supports post processing, De-blocking and De-ringing. This is configurable dynamically by setting `postDeblock` and `postDering` flag of the extended parameter.
Does the decoder support non-multiple of 16 height and width?	The decoder supports decoding of non-multiple of 16 frame sizes. The output buffer in this case can be of the actual frame size. The height and width value used to create decoder instance should be multiple of 16
What is decoder behavior for corrupted bit stream?	In case of erroneous bit stream, the decoder attempts error concealment for that frame. This error concealed frame is sent to output and an error code for the specific error is also signaled.
What is decoder behavior incase of profile/level that are not supported?	If profile/ level indicated in bit stream is not supported, the decoder still proceeds and attempts decoding. If decoding is not possible then only error is reported.
Is it possible to know the MB error status?	In case of erroneous bit stream algorithm can populate the MB error status to the application supplied buffer.
Is it possible to know the picture height and width without actually decoding a frame?	It is possible to know the frame height and width without actually decoding a frame. The dynamic input setting `XDM_PARSE_HEADER` can be used to parse the header information only, from which frame height and frame width is known.