H.264 Baseline Profile Decoder on DM6467

User's Guide

Literature Number: SPRUET5
July 2008

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
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Clocks and Timers	www.ti.com/clocks	Digital Control	www.ti.com/digitalcontrol
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video & Imaging	www.ti.com/video
		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) H.264 Baseline Profile Decoder implementation on the DM6467 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 DM6467 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.

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 Inteface 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.
- *DaVinci Technology - Digital Video Innovation Product Bulletin (Rev. A)* (literature number SPRT378A)
- *The DaVinci Effect: Achieving Digital Video Without Complexity White Paper* (literature number SPRY079)
- *DaVinci Benchmarks Product Bulletin* (literature number SPRT379)
- *DaVinci Technology for Digital Video White Paper* (literature number SPRY067)
- *The Future of Digital Video White Paper* (literature number SPRY066)

Related Documentation

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

- *ISO/IEC 14496-10:2005 (E) Rec.- Information technology – Coding of audio-visual objects – H.264 (E) ITU-T Recommendation*

Abbreviations

The following abbreviations are used in this document.

Table 1-1. List of Abbreviations

Abbreviation	Description
ASO	Arbitrary Slice Ordering
AVC	Advanced Video Coding
BIOS	TI's simple RTOS for DSPs
CABAC	Context Adaptive Binary Arithmetic Coding
CAVLC	Context Adaptive Variable Length Coding
CSL	Chip Support Library
D1	720x480 or 720x576 resolutions in progressive scan
DCT	Discrete Cosine Transform
DMA	Direct Memory Access
DMAN	DMA Manager
EVM	Evaluation Module
FMO	Flexible Macroblock Ordering
HDTV	High Definition Television
I_PCM	Intra-frame Pulse Code Modulation
IDR	Instantaneous Decoding Refresh
IRES	Interface standard to request and receive handles to resources
ITU-T	International Telecommunication Union
JM	Joint Menu
JVT	Joint Video Team
MB	Macro Block
MBAFF	Macro Block Adaptive Field Frame
MPEG	Moving Pictures Experts Group

Abbreviation	Description
MV	Motion Vector
NAL	Network Adaptation Layer
NTSC	National Television Standards Committee
PicAFF	Picture Adaptive Field Frame
RMAN	Resource Manager
RTOS	Real Time Operating System
VGA	Video Graphics Array (640 x 480 resolution)
VOP	Video Object Plane
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 (H.264 Baseline Profile Decoder on DM6467) 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, the DAVINCI Logo, DAVINCI, DSP/BIOS, eXpressDSP, TMS320, TMS320C64x, TMS320C6000, TMS320DM6467, and TMS320C64x+ are trademarks of Texas Instruments.

All trademarks are the property of their respective owners.

This page is intentionally left blank

Contents

Read This First ..... **iii**
About This Manual iii
Intended Audience iii
How to Use This Manual iii
Related Documentation From Texas Instruments iv
Related Documentation v
Abbreviations vi
Text Conventions vii
Product Support vii
Trademarks vii
Contents ..... **ix**
Figures ..... **xi**
Tables ..... **xiii**
Introduction **1-1**
1.1 Overview of XDAIS and XDM 1-2
1.1.1 XDAIS Overview 1-2
1.1.2 XDM Overview 1-3
1.2 Overview of H.264 Baseline Profile Decoder 1-4
1.3 Supported Services and Features 1-6
Installation Overview **2-1**
2.1 System Requirements 2-2
2.1.1 Hardware 2-2
2.1.2 Software 2-2
2.2 Installing the Component 2-2
2.3 Before Building the Sample Test Application 2-5
2.3.1 Installing DSP/BIOS 2-5
2.3.2 Installing Codec Engine (CE) 2-5
2.3.3 Installing HDVICP Library (HDVICP) 2-6
2.4 Building and Running the Sample Test Application 2-7
2.5 Configuration Files 2-7
2.5.1 Generic Configuration File 2-8
2.5.2 Decoder Configuration File 2-8
2.6 Uninstalling the Component 2-9
2.7 Evaluation Version 2-9
Sample Usage **3-1**
3.1 Overview of the Test Application 3-2
3.1.1 Parameter Setup 3-3
3.1.2 Algorithm Instance Creation and Initialization 3-3
3.1.3 Process Call 3-4
3.1.4 Algorithm Instance Deletion 3-6
3.2 Frame Buffer Management by Application 3-7
3.2.1 Frame Buffer Input and Output 3-7
3.2.2 Frame Buffer Management by Application 3-9
3.3 Handshaking Between Application and Algorithm 3-10

3.4 Sample Test Application....3-12
API Reference....4-1
4.1 Symbolic Constants and Enumerated Data Types....4-2
4.1.1 H.264 Decoder Enumerated Data Types....4-6
4.2 Data Structures....4-8
4.2.1 Common XDM Data Structures....4-8
4.2.2 H264 Decoder Data Structures....4-25
4.3 Interface Functions....4-28
4.3.1 Creation APIs....4-28
4.3.2 Initialization API....4-31
4.3.3 Control API....4-32
4.3.4 Data Processing API....4-33
4.3.5 Termination API....4-35

Figures

Figure 1-1. Flow diagram of the H.264 Decoder 1-5
Figure 2-1. Component Directory Structure 2-3
Figure 3-1. Test Application Sample Implementation 3-2
Figure 3-2. Process call with Host release 3-5
Figure 3-3. Frame Buffer Pointer Implementation 3-8
Figure 3-4. Interaction of Frame Buffers Between Application and Framework 3-9
Figure 3-5. Interaction Between Application and Codec 3-10

This page is intentionally left blank

Tables

Table 1-1. List of Abbreviations ..... **vi**
Table 2-1. Component Directories **2-4**
Table 3-1. Process() Implementation **3-12**
Table 4-1. List of Enumerated Data Types **4-2**
Table 4-2. H264 Decoder Enumerated Data Types **4-6**
Table 4-3. H264 Decoder Error Codes. **4-18**

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 H.264 Baseline Profile Decoder on the DM6467 platform and its supported features.

Topic **Page**

1.1 Overview of XDAIS and XDM	**1-2**
1.2 Overview of H.264 Baseline Profile Decoder	**1-4**
1.3 Supported Services and Features	**1-6**

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

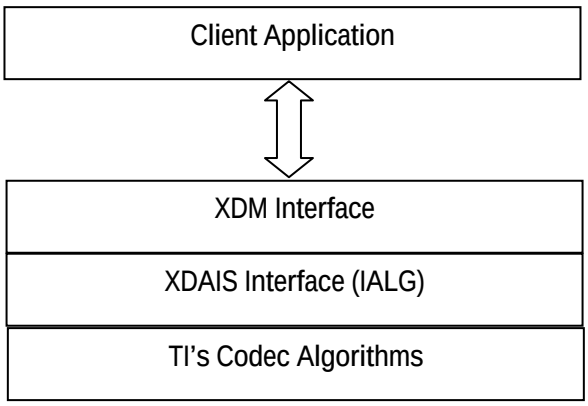
In the multimedia application space, you have the choice of integrating any codec into your multimedia system. For example, if you are building a video decoder system, you can use any of the available video decoders (such as MPEG4, H.263, or H.264) in your system. To enable easy integration with the 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 H.264 Baseline Profile Decoder

H.264 (from ITU-T, also called as H.264/AVC) is a popular video coding algorithm enabling high quality multimedia services on a limited bandwidth network. H.264 standard defines several profiles and levels that 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 that limits all decoders conforming to that profile may support. Each level specifies a set of limits on the values that may be taken by the syntax elements in that profile.

Some important H.264 profiles and their special features are:

- Baseline Profile:
  - Only I and P type slices are present
  - Only frame mode (progressive) picture types are present
  - Only CAVLC is supported
- Main Profile:
  - Only I, P, and B type slices are present
  - Frame and field picture modes (in progressive and interlaced modes) picture types are present
  - Both CAVLC and CABAC are supported
  - ASO is not supported
  - FMO is not supported
- High Profile:
  - Only I, P, and B type slices are present
  - Frame and field picture modes (in progressive and interlaced modes) picture types are present
  - Both CAVLC and CABAC are supported
  - ASO is not supported
  - FMO is not supported
  - 8x8 transform supported
  - Scaling matrices supported

The input to the decoder is a H.264 encoded bit-stream in the byte-stream syntax. The byte-stream consists of a sequence of byte-stream NAL unit syntax structures. Each byte-stream NAL unit syntax structure contains one start code prefix of size four bytes and value 0x00000001, followed by one NAL unit syntax structure. The encoded frame data is a group of slices, each of which is encapsulated in NAL units. The slice consists of the following:

- Intra coded data: Spatial prediction mode and prediction error data that is subjected to DCT and later quantized.
- Inter coded data: Motion information and residual error data (differential data between two frames) that is subjected to DCT and later quantized.

The first frame received by the decoder is IDR (Instantaneous Decode Refresh) picture frame. The decoder reconstructs the frame by spatial intra-prediction specified by the mode and by adding the prediction error. The subsequent frames may be intra or inter coded.

In case of inter coding, the decoder reconstructs the bit-stream by adding the residual error data to the previously decoded image, at the location specified by the motion information. This process is repeated until the entire bit-stream is decoded. The output of the decoder is a YUV sequence, which is of 420 interleaved format (Y is a single plane and the Chroma data – cb and cr are interleaved to form the other plane).

Figure 1-1 depicts the working of the decoder.

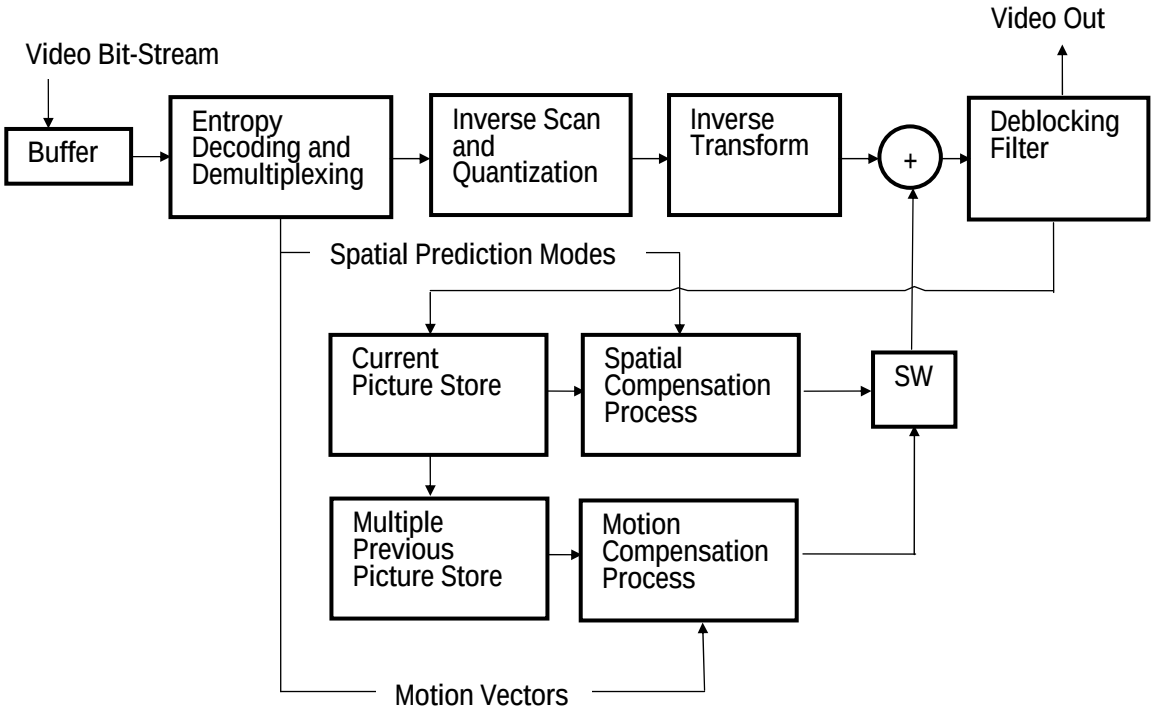


Figure 1-1. Flow diagram of the H.264 Decoder

From this point onwards, all references to H.264 Decoder means H.264 Baseline Profile Decoder only.

1.3 Supported Services and Features

This user guide accompanies TI's implementation of H.264 Decoder on the DM6467 platform.

This version of the codec has the following supported features:

- eXpressDSP Digital Media (XDM 1.2 IVIDDEC2) compliant
- Supports up to level 4 features of the Baseline Profile (BP)
- Supports progressive type picture decoding
- Supports multiple slices and multiple reference frames
- Supports CAVLC decoding
- Supports all intra-prediction and inter-prediction modes
- Supports up to 16 MV per MB
- Supports frame based decoding
- Supports frame width of the range of 16 to 1920 pixels
- Tested for compliance with JM version 10.1 reference decoder
- Long term reference frame and adaptive reference picture marking
- Reference Picture List Reordering
- PCM Macroblock decoding
- Gaps in frame_num
- Supports Error Resilience and Error Concealment for Baseline Profile

This version of the decoder does not support the following features as per the Baseline Profile feature set:

- I_PCM (Intra-frame pulse code modulation) decoding
- Supplemental Enhancement Info (SEI) and Video Usability Information (VUI)
- ASO/FMO functionality
- Decoding of pictures with width less than or equal to 64 (<=64)

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	**2-2**
2.2 Installing the Component	**2-2**
2.3 Before Building the Sample Test Application	**2-5**
2.4 Building and Running the Sample Test Application	**2-7**
2.5 Configuration Files	**2-7**
2.6 Uninstalling the Component	**2-9**
2.7 Evaluation Version	**2-9**

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 the DM6467 EVM only.

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 3.3.49.
- **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 200_V_H264_D_1_00, under which another directory named DM6467_BP_001 is created.

Figure 2-1 shows the sub-directories created in the DM6467_BP_001 directory.

Note:

The source folders under H264Decoder (algSrc) will not be present in the case of a library based (object code) release.

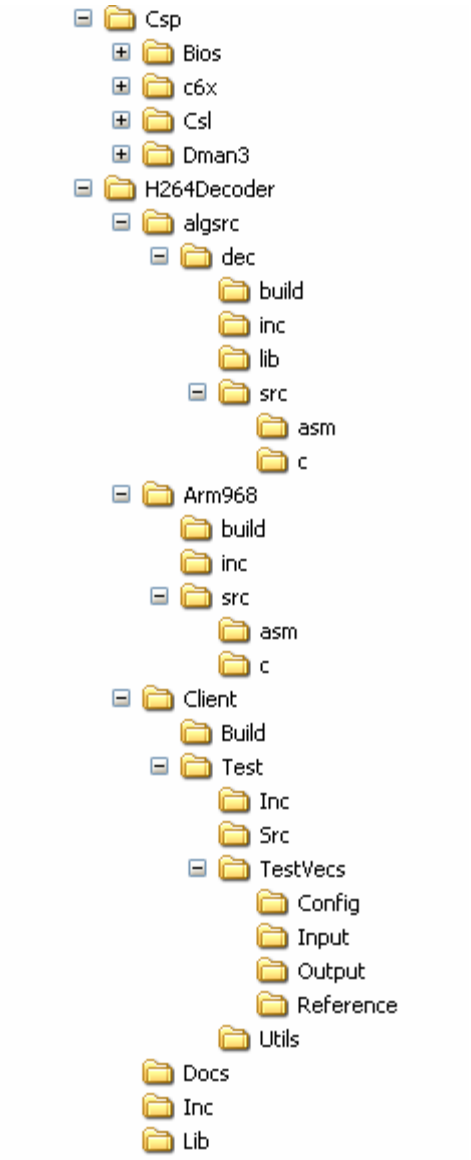


Figure 2-1. Component Directory Structure

Table 2-1provides a description of the sub-directories created in the DM6467_BP_001directory.

Note:
If you are installing an evaluation version of this codec, the directory name will be 200_V_H264_D_1_00E

Table 2-1. Component Directories

Sub-Directory	Description
Csp\Bios	Contains BIOS specific files
Csp\Csl	Contains CSL files
Csp\C6x	Contains CSL files
Csp\Dman3	Contains DMAN3 related files
\H264Decoder\algsrc\dec\build	Contains the H.264 Decoder project
\H264Decoder\algsrc\dec\inc	Contains the H.264 Decoder include files
\H264Decoder\algsrc\dec\lib	Contains libraries used by H.264 Decoder
\H264Decoder\algsrc\dec\src\c	Contains all “C” files for H.264 Decoder
\H264Decoder\algsrc\dec\src\asm	Contains all assembly files for the H.264 Decoder
\H264Decoder\Arm968\build	Contains ARM968 project
\ H264Decoder\Arm968\inc	Contains ARM968 project include files
\ H264Decoder\Arm968\src\asm	Contains ARM968 project asm files
\ H264Decoder\Arm968\src\c	Contains ARM968 project C files
\H264Decoder\Client\Build	Contains a sample test application project and linker command file
\H264Decoder\Client\Test\Inc	Contains include files for the application project
\H264Decoder\Client\Test\Src	Contains source files for the application project

Sub-Directory	Description
\H264Decoder\Client\Test\TestVecs\Config	Contains configuration files for the decoder application
\H264Decoder\Client\Test\TestVecs\Input	Contains input decoder bit-streams for the H.264 decoder
\H264Decoder\Client\Test\TestVecs\Output	Contains output decoded files
\H264Decoder\Client\Test\TestVecs\Reference	Contains reference files
\H264Decoder\Client\Test\Utils	Contains utilities like file compare and YUV display executables
\H264Decoder\Docs\	Contains user guide and datasheet
\H264Decoder\Inc\	Contains include files required by the application and the decoder library
\H264Decoder\Lib	Contains H.264 Decoder library 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, TI Codec Engine (CE) and HDVICP API. This version of the codec is validated on DSP/BIOS version 5.31, Codec Engine (CE) version 2.10.01 (or newer) and HDVICP library version 1.01.000.

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 CodeComposer Studio. For example:<install directory>\CCStudio_v3.3

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

- Header file, bcache.h available in the <install directory>\CCStudio_v3.3\<bios_directory>\packages \ti\bios\include directory.
- Library file, biosDM420.a64P available in the <install directory>\CCStudio_v3.3\<bios_directory>\packages\ti\bios\lib directory.

2.3.2 Installing Codec Engine (CE)

You can download Codec Engine version 2.10.01 (or newer) from the TI external website:

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

The codec uses framework components and XDAIS version that are a part of CE 2.10.01 (or newer).

1) Extract the CE zip file to the location where you have installed Code Composer Studio. For example: <install directory>\CCStudio_v3.3.

   The test application uses the following RMAN files:

   - Library file, rmand.a64P available in the <install directory>\ CCStudio_v3.3\<ce_directory>\cetools\packages\ti\sdo\fc\rman directory.
   - Header file, rman.h available in the <install directory>\CCStudio_v3.3 \<ce_directory>\cetools\packages\ti\sdo\fc\rman directory.
   - Header file, ires.h available in the <install directory>\CCStudio_v3.3\ <ce_directory>\cetools\packages\ti\xdais directory.

2) Set system environment variable named `FC_INSTALL_DIR` pointing to <install directory>\CCStudio_v3.3\<ce_directory>\cetools.

3) Set system environment variable named `XDAIS_INSTALL_DIR` pointing to <install directory>\CCStudio_v3.3\<ce_directory>\ cetools\packages\ti\xdais.

4) Set system environment variable `EDMA3LLD_INSTALL_DIR` pointing to <install directory>\CCStudio_v3.3\<ce_directory>\cetools\ packages\ti\sdo\edma3.

5) Set system environment variable `CCS_PATH` pointing to <install directory>\CCStudio_v3.3.

2.3.3 Installing HDVICP Library (HDVICP)

Download HDVICP Library version 1.01.000.

1) Extract the HDVICP Library (Source or Object Release) zip file to any location, For example: <hdvicp_install directory>

2) Set system environment variable named `HDVICP_API` pointing to the Hdvicp_api directory in the <hdvicp_install directory>

   `HDVICP_API` = <hdvicp_install_directory>\200_V_HDVICP_ X_1_01\DM6467_X_001\Hdvicp_api

   The codec uses the following HDVICP Library files:

   - Library file, hdvicp_ti_api_c64Plus.lib available in the <`HDVICP_API` >\lib directory.
   - Header files, available in the <`HDVICP_API`>\inc directory.

2.4 Building and Running the Sample Test Application

The sample test application that accompanies this codec component runs in TI's Code Composer Studio development environment. To build and run the sample test application in Code Composer Studio, follow these steps:

1) Extract the .zip file.

2) Verify that you have installed TI's Code Composer Studio version 3.3.49 and code generation tools version 6.0.8. Start the Code Composer Studio to view the Parallel Debug Manager (PDM) window.

3) Double-click on ARM926 to open PDM window, load the GEL file davincihd_arm.gel, and click on **Debug >Connect**.

4) Double-click on C6400PLUS to open PDM window, load the GEL file davincihd_dsp.gel, and click on **Debug >Connect**.

5) Open the testappdecoder.pjt in C6400PLUS window. This file is located in the \H264Decoder\Client\Build sub-directory. Select the configuration **Release**

6) Select **Project > Build** to build the sample test application.

7) This in turn, builds the dependent project – H264Decoder.pjt –stored at location \H264Decoder\algsrc\dec\build and creates the final executable file, TestAppDecoder.out in the H264Decoder\Client\Build\Out sub-directory.

8) Select **File > Load**, browse to the H264Decoder\Client\Build\Out sub-directory, select the codec executable created in step 6, and load it into C6400PLUS window in preparation for execution.

9) Select **Debug > Run** to execute the sample test application on DSP. The sample test application takes the input files stored in the \H264Decoder\Client\Test\TestVecs\Input sub-directory, runs the codec, and dumps the output files in \H264Decoder\Client\Test\TestVecs\Output sub-directory. The user can use the reference files stored in \H264Decoder\Client\Test\TestVecs\Reference sub-directory to compare and check for compliance.

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.
  - 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` is the output .yuv file name
- `Reference` is the .crc file name.

A sample Testvecs.cfg file is as shown:

```
1
..\..\Test\TestVecs\Config\Testparams.cfg
..\..\Test\TestVecs\Input\foreman_ipb_cabac.264
..\..\Test\TestVecs\Output\foreman_ipb_cabac_c.yuv
..\..\Test\TestVecs\Reference\foreman_ipb_cabac_c.crc

0
..\..\Test\TestVecs\Config\Testparams.cfg
..\..\Test\TestVecs\Input\foreman_ipb_cabac.264
..\..\Test\TestVecs\Output\foreman_ipb_cabac_test.yuv
..\..\Test\TestVecs\Output\foreman_ipb_cabac_c.crc
```

In compliance mode of operation, the decoder tests for the CRC of each decoded frame with the CRC of the corresponding reference frame. If these CRCs do not match, the application dumps the current frame and halts further decoding.

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    =  1920  # Image width in Pels, must be
multiple of 16
ImageHeight   =  1088  # Image height in Pels, must be
multiple of 16
ChromaFormat  =  1    # 1 => XDM_YUV_420P
FramesToDecode = 20   # Number of frames to be decoded
```

Note:

`ChromaFormat` supported in this codec is 420 interleaved. As the current XDM does not have an e-num defined for 420i, 420p is used as shown in the code snippet. This may change in future versions of XDM.

2.6 Uninstalling the Component

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

2.7 Evaluation Version

If you are using an evaluation version of this codec, there will be a limit of decoding 54000 frames.

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	3-2
3.2 Frame Buffer Management by Application	3-7
3.3 Handshaking Between Application and Algorithm	3-10
3.4 Sample Test Application	3-12

3.1 Overview of the Test Application

The test application exercises the `IVIDDEC2` base class of the H.264 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. The DMA initialization shown in this figure is handled inside the codec currently.

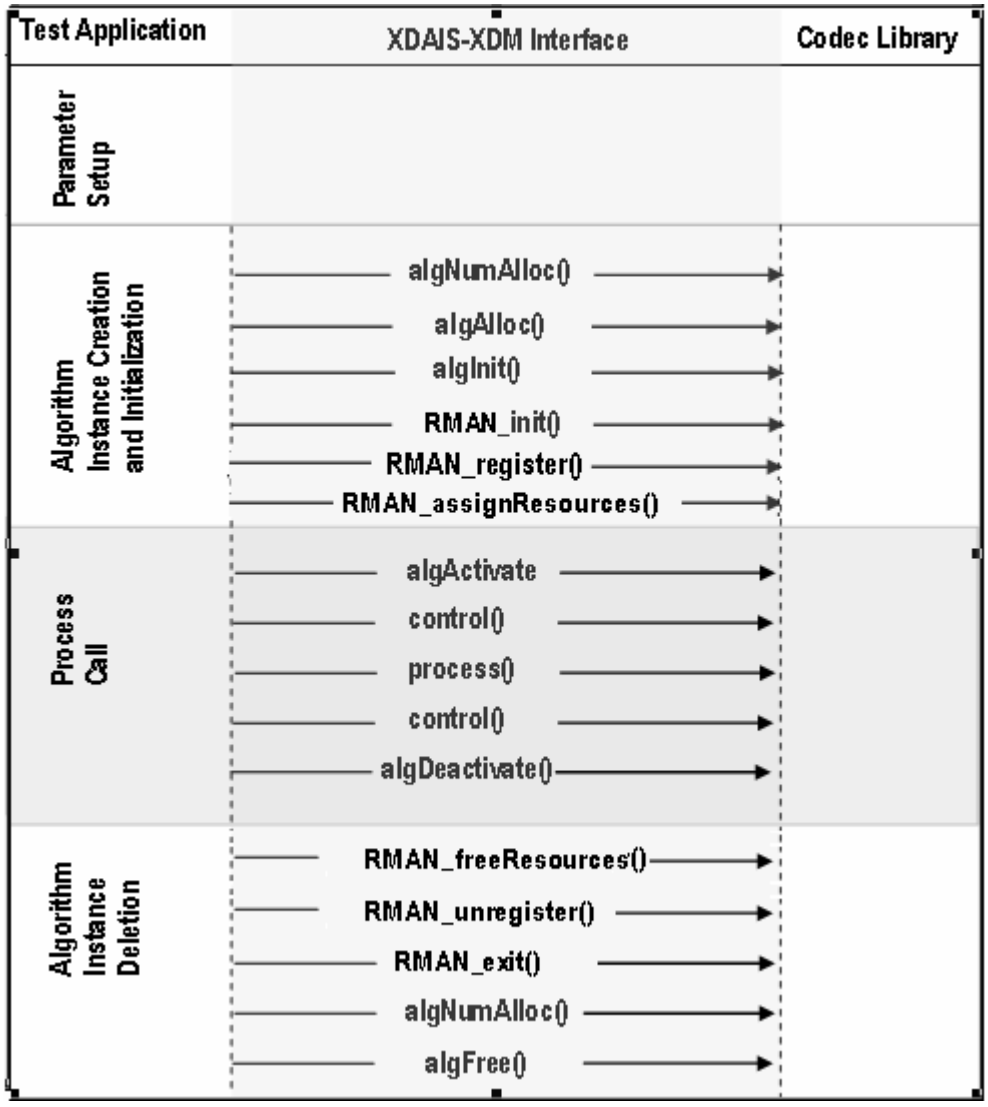


Figure 3-1. Test Application Sample Implementation

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

After successful completion of 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 and HDVICP Resource allocation for the algorithm. This requires initialization of Resource Manager Module (RMAN) and grant of DMA and HDVICP resources. This is implemented by calling RMAN interface functions in the following sequence:

1) `RMAN_init()` - To initialize the RMAN module.

2) `RMAN_register()` - To register the HDVICP protocol/resource manager with the generic resource manager.

3) `RMAN_assignResources()` - To register resources to the algorithm as requested HDVICP protocol/resource manager with the generic resource manager.

Note:

RMAN function implementations are provided in rmand.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) Implements the process call based on the non-blocking mode of operation explained in step 4. The behavior of the algorithm can be controlled using various dynamic parameters (see Section 4.2.1.6). The inputs to the `process()` functions are input and output buffer descriptors, pointer to the `IVIDDEC2_InArgs` and `IVIDDEC2_OutArgs` structures.

4) On the call to the `process()` function for encoding/decoding a single frame of data, the software triggers the start of encode/decode. After triggering the start of the encode/decode frame, the video task can be put to `SEM-pend` state using semaphores. On receipt of interrupt signal at the end of frame encode/decode, the application releases the semaphore and resume the video task, which does any book-keeping operations by the codec and updates the output parameter of `IVIDDEC2_OutArgs` structure.

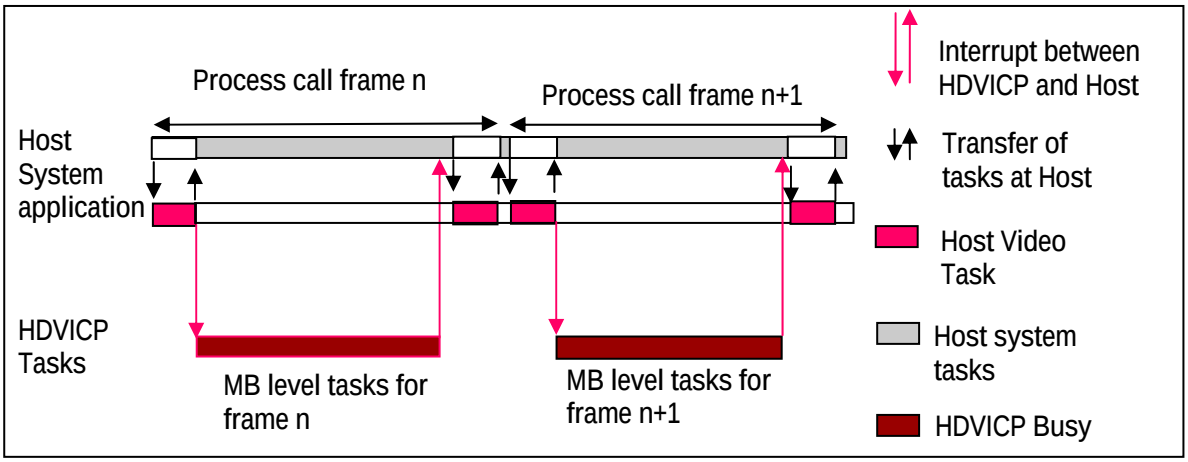


Figure 3-2. Process call with Host release

Note:

- The process call returns control to the application after the initial setup related tasks are completed.
- Application can schedule a different task to use the freed up Host resource.
- All service requests from HDVICP are handled through interrupts.
- Application resumes the suspended process call after last service request for HDVICP 0/1 has been handled.
- Application can now complete concluding portions of the process call and return.

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 picture 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. The test application does a cache invalidate for the valid input buffers before `process()` and a cache write back invalidate for output buffers after a `control()` call with `GET_STATUS` command.

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 sequence:

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

2) `algFree()` - To query the algorithm for memory, to free when removing an instance.

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.

After successful execution of the algorithm, the test application frees up the DMA and HDVICP resource allocated for the algorithm. This is implemented by calling RMAN interface functions in the following sequence:

1) `RMAN_unregister()` - To unregister the HDVICP protocol/resource manager with the generic resource manager.

2) `RMAN_exit()` - To delete the generic IRES RMAN and release the memory.

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.

```
H264VDEC_create();
H264VDEC_control(XDM_GETBUFINFO); /* Returns default 1080p
HD size */
do{
H264VDEC_decode(); //call the decode API
H264VDEC_control(XDM_GETBUFINFO); /* updates the memory
required as per the size parsed in stream header */
}
while(all frames)
```

Note:

- Application can take the information retured by the control function with the `XDM_GETBUFINFO` command and change the size of the buffer passed in the next process call.
- Application can re-use the extra buffer space of the 1st frame, if the above control call returns a small size than that was provided.

The frame pointer given by the application and that returned by the algorithm may be different. `BufferID` (`InputID/outputID`) provides the 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.

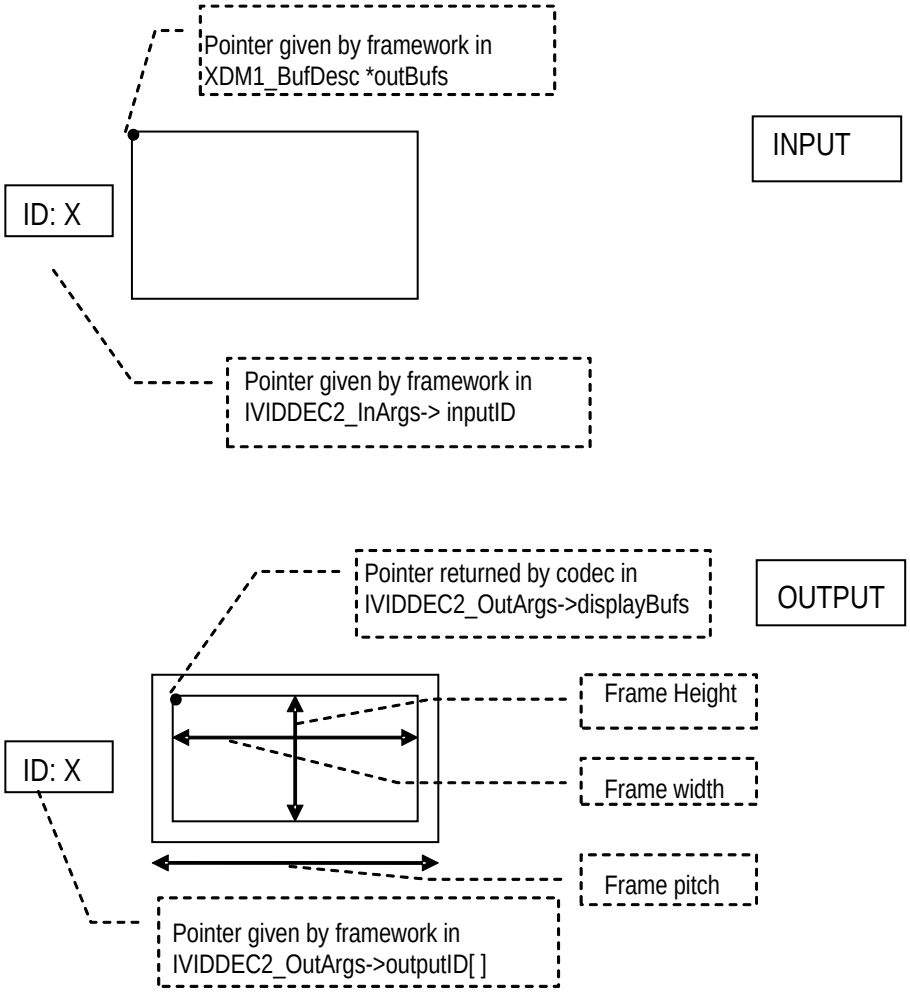


Figure 3-3. Frame Buffer Pointer Implementation

Note:

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

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

Note:

`BufferID` returned in `IVIDDEC2_OutArgs ->outputID[]` is just for display purpose. Application should not consider it free unless it is a 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.

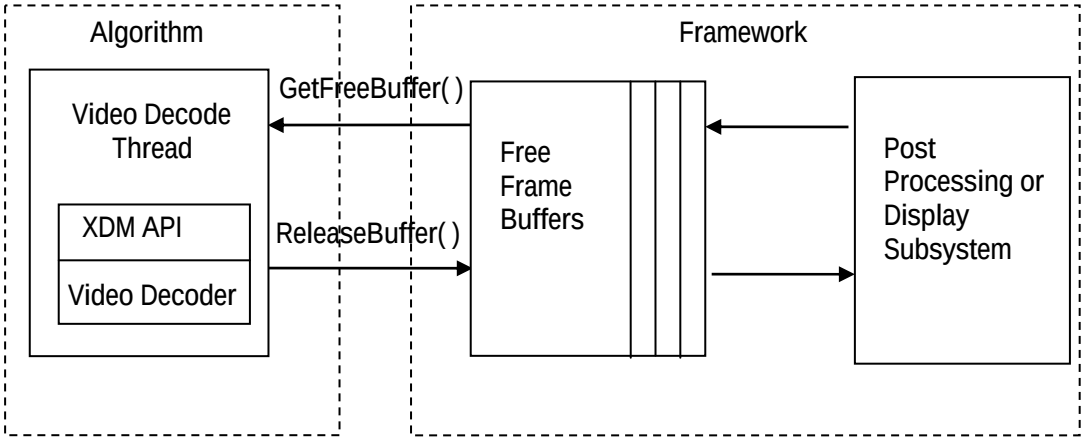


Figure 3-4. Interaction of Frame Buffers Between Application and Framework

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

- `BUFFMGR_Init() - 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.
- `BUFFMGR_ReInit()` - `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.
- `BUFFMGR_GetFreeBuffer()` - `BUFFMGR_GetFreeBuffer` function searches for a free buffer in the global buffer array and returns the address of that element. Incase none of the elements are free, then it returns `NULL`.

- `BUFFMGR_ReleaseBuffer()` - `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 moves until it encounters a buffer ID as zero or it hits the `MAX_BUFF_ELEMENTS`.
- `BUFFMGR_DeInit()`- `BUFFMGR_DeInit` function releases all memory allocated by buffer manager.

3.3 Handshaking Between Application and Algorithm

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.

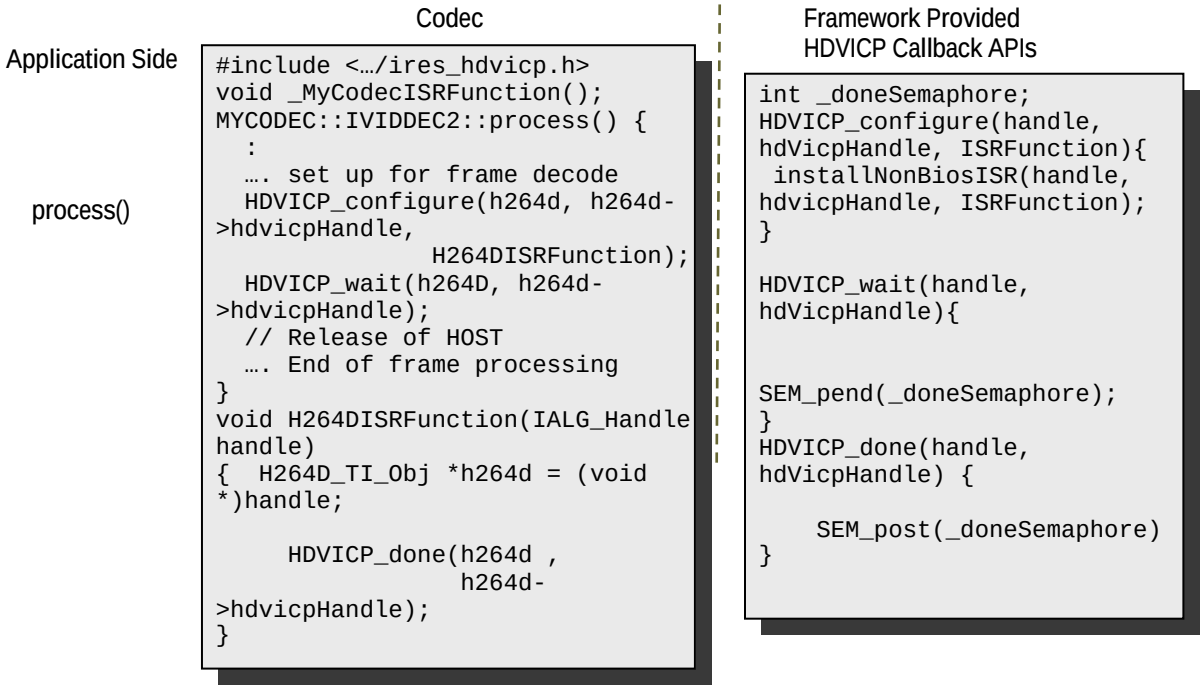


Figure 3-5. Interaction Between Application and Codec

Note:

- Process call architecture to share Host resource among multiple threads.
- ISR ownership is with the Host layer resource manager – outside the codec.
- The actual codec routine to be executed during ISR is provided by the codec.
- OS/System related calls (`SEM_pend, SEM_post`) also outside the codec.
- Codec implementation is OS independent.

The functions to be implemented by the application are:

- `HDVICP_configure(IALG_Handle handle, void *hdvicpHandle, void (*ISRfunctionptr)(IALG_Handle handle))`

This function is called by the algorithm to register its ISR function, which the application needs to call when it receives interrupts pertaining to the video task.

- `HDVICP_wait (void *hdvicpHandle)`

This function is called by the algorithm to move the video task to `SEM-pend` state.

- `HDVICP_done (void *hdvicpHandle)`

This function is called by the algorithm to release the video task from `SEM-pend` state. In the sample test application, these functions are implemented in hdvicp_framework.c file. The application can implement it in a way considering the underlying system.

3.4 Sample Test Application

The test application exercises the `IVIDDEC2` base class of the H.264 Decoder.

Table 3-1. Process() Implementation.

```
/*Main Function acting as a client for Video Decode Call*/

  BUFFMGR_Init();


  TestApp_SetInitParams(&params.viddecParams);

  /*---------------- Decoder creation -----------------*/
  handle = (IALG_Handle) H264VDEC_create();

     /* Get Buffer information              */
  H264VDEC_control(handle, XDM_GETBUFINFO);


  /* Do-While Loop for Decode Call  for a given stream  */
    do
    {
/* Read the bitstream 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 via
control() */

     H264VDEC_control(handle, XDM_SETPARAMS);

/*-----------------------------------------------------*/
/* Start the process : To start decoding a frame      */
/* This will always follow a H264VDEC_decode_end call */
/*-----------------------------------------------------*/
      retVal = H264VDEC_decode
           (
             handle,
             (XDM1_BufDesc *)&inputBufDesc,
             (XDM_BufDesc *)&outputBufDesc,
             (IVIDDEC2_InArgs *)&inArgs,
             (IVIDDEC2_OutArgs *)&outArgs
           );

     /* Get the statatus of the decoder using comtrol */
     H264VDEC_control(handle, IH264VDEC_GETSTATUS);

       /* Get Buffer information :      */
     H264VDEC_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	4-2
4.2 Data Structures	4-8
4.3 Interface Functions	4-28

4.1 Symbolic Constants and Enumerated Data Types

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.

Table 4-1. List of Enumerated Data Types

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
IVIDEO_FrameType	IVIDEO_NA_FRAME	Frame type not available
	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 a 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

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	IVIDEO_MBAFF_IDR_FRAME	Intra coded MBAFF frame that can be used for refreshing video content.
	IVIDEO_FRAMETYPE_DEFAULT	Default set to IVIDEO_I_FRAME
IVIDEO_ContentType	IVIDEO_CONTENTTYPE_NA	Content type is not applicable
	IVIDEO_PROGRESSIVE	Progressive video content
	IVIDEO_INTERLACED	Interlaced video content
	IVIDEO_CONTENTTYPE_DEFAULT	Default set to IVIDEO_PROGRESSIVE
IVIDEO_FrameSkip	IVIDEO_NO_SKIP	Do not skip the current frame. Default Value
	IVIDEO_SKIP_P	Skip forward inter coded frame. Not supported in this version of H264 Decoder.
	IVIDEO_SKIP_B	Skip bi-directional inter coded frame. Not supported in this version of H264 Decoder.
	IVIDEO_SKIP_I	Skip intra coded frame. Not supported in this version of H264 Decoder.
	IVIDEO_SKIP_IP	Skip I and P frame/field(s) Not supported in this version of H264 Decoder.
	IVIDEO_SKIP_IB	Skip I and B frame/field(s). Not supported in this version of H264 decoder.
	IVIDEO_SKIP_PB	Skip P and B frame/field(s). Not supported in this version of H264 Decoder.
	IVIDEO_SKIP_IPB	Skip I/P/B/BI frames Not supported in this version of H264 Decoder.
	IVIDEO_SKIP_IDR	Skip IDR Frame Not supported in this version of H264 Decoder.
	IVIDEO_SKIP_DEFAULT	Default set to IVIDEO_NO_SKIP
IVIDEO_OutputFrameStatus	IVIDEO_FRAME_NOERROR	The output buffer is available.
	IVIDEO_FRAME_NOTAVAILABLE	The codec does not have any output buffers.

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	IVIDEO_FRAME_ERROR	The output buffer is available and corrupted.
	IVIDEO_OUTPUTFRAMESTATUS_DEFAULT	Default set to IVIDEO_FRAME_NOERROR
XDM_DataFormat	XDM_BYTE	Big endian stream
	XDM_LE_16	16-bit little endian stream. Not supported in this version of H264 Decoder.
	XDM_LE_32	32-bit little endian stream. Not supported in this version of H264 Decoder.
XDM_ChromaFormat	XDM_CHROMA_NA	Chroma format not applicable
	XDM_YUV_420P	YUV 4:2:0 planar Not supported in this version of H264 Decoder.
	XDM_YUV_422P	YUV 4:2:2 planar. Not supported in this version of H264 Decoder.
	XDM_YUV_422IBE	YUV 4:2:2 interleaved (big endian). Not supported in this version of H264 Decoder.
	XDM_YUV_422ILE	YUV 4:2:2 interleaved (little endian) Not supported in this version of H264 Decoder.
	XDM_YUV_444P	YUV 4:4:4 planar. Not supported in this version of H264 Decoder.
	XDM_YUV_411P	YUV 4:1:1 planar. Not supported in this version of H264 Decoder.
	XDM_GRAY	Gray format. Not supported in this version of H264 Decoder.
	XDM_RGB	RGB color format. Not supported in this version of H264 Decoder.
	XDM_CHROMAFORMAT_DEFAULT	Default set to XDM_YUV_422ILE
XDM_CmdId	XDM_GETSTATUS	Query algorithm instance to fill Status structure

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_SETPARAMS	Set run-time dynamic parameters via the DynamicParams structure Not supported in this version of H264 Decoder.
	XDM_RESET	Reset the algorithm. Not supported in this version of H264 Decoder.
	XDM_SETDEFAULT	Initialize all fields in Params structure to default values specified in the library. Not supported in this version of H264 Decoder.
	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. Not supported in this version of H264 Decoder.
XDM_AccessMode	XDM_ACCESSMODE_READ	The algorithm read from the buffer using the CPU.
	XDM_ACCESSMODE_WRITE	The algorithm wrote from the buffer using the CPU
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

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_UNSUPPORTEDPARAM	Bit 14 ❑ 1 - Unsupported input parameter or configuration ❑ 0 - Ignore
	XDM_FATALERROR	Bit 15 ❑ 1 - Fatal error ❑ 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.

4.1.1 H.264 Decoder Enumerated Data Types

This section describes the H.264 Decoder specific data structures.

Table 4-2. H264 Decoder Enumerated Data Types

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
eLevelNum_t	Level1	0: Level 1
	Level11	1: Level 1.1
	Level12	2: Level 1.2
	Level13	3: Level 1.3
	Level1b	4: Level 1b
	Level2	5: Level 2
	Level21	6: Level 2.1
	Level22	7: Level 2.2
	Level3	8: Level 3
	Level31	9: Level 3.1
	Level32	10: Level 3.2
	Level4	11: Level 4

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	Level41	12: Level 4.1
	Level42	13: Level 4.2
	Level5	14: Level 5
	Level51	15: Level 5.1
	MAXLEVELID	16: Maximum Level ID which is not supported.
eProfile_t	Profile_Baseline	0: Decode if Baseline profile only
	Profile_Main	1: Decode if Main profile only
	Profile_High	2: Decode if High Profile only
	Profile_Any	3: Decode any of the above profiles (Baseline, Main, High)

4.2 Data Structures

This section describes the XDM defined data structures, which 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	Datatype	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	Datatype	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	Datatype	Input/ Output	Description
`*buf`	`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	Datatype	Input/ Output	Description
`*buf`	`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 by 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	Datatype	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 H264 Baseline 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 = 1920 * 1088
  UV buffer = 1920 * 544

These are the maximum buffer sizes but they can be reconfigured 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	Datatype	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 used 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	Video frame types. See IVIDEO_FrameType enumeration for details.
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	Video output buffer status. See IVIDEO_OutputFrameStatus enumeration for details.
repeatFrame	XDAS_Int32	Input	Number of times the display process needs to repeat the displayed progressive frame
contentType	XDAS_Int32	Input	Content type of the buffer. See IVIDEO_ContentType enumeration for details.
chromaFormat	XDAS_Int32	Input	Chroma formats. See XDM_ChromaFormat for details.

Note:

`IVIDEO_MAX_YUV_BUFFERS:`

- Max YUV buffers - one each for Y, U, and V.
- `repeatFirstFieldFlag` and `repeatFrame` are not supported and their default value is 0.

4.2.1.7 IVIDDEC2_Fxns

Description

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

Fields

Field	Datatype	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	Datatype	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 Default is 1088
maxWidth	XDAS_Int32	Input	Maximum video width to be supported in pixels Default is 1920
maxFrameRate	XDAS_Int32	Input	Maximum frame rate in fps * 1000 to be supported. Default is 30000
maxBitRate	XDAS_Int32	Input	Maximum bit rate to be supported in bits per second. For example, if bit rate is 10 Mbps, set this field to 10485760. Default is 10000000
dataEndianness	XDAS_Int32	Input	Endianness of input data. See XDM_DataFormat enumeration for details. Default is XDM_BYTE
forceChromaFormat	XDAS_Int32	Input	Sets the output to the specified format. (Only 420 format supported currently). For example, if the output should be in YUV 4:2:2 interleaved (little endian) format, set this field to XDM_YUV_422ILE. Default is XDM_YUV_420P See XDM_ChromaFormat enumeration for details.

Note:

- H264 Decoder does not use the maxFrameRate and maxBitRate fields for creating the algorithm instance. In the current implementation, maxFrameRate is set to 1000 * 30, and maxBitRate is set to 10000000.
- Maximum video height and width supported are 1088 pixels and 1920 pixels respectively (for 1080p).
- dataEndianness field should be set to XDM_BYTE.

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	Datatype	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 (Not Supported)
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.
frameSkipMode	XDAS_Int32	Input	Frame skip mode. See IVIDEO_FrameSkip enumeration for details.
frameOrder	XDAS_Int32	Input	Frame Display Order. See IVIDDEC2_FrameOrder enumeration for details.
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:

- H264 Decoder does not support decodeHeader field. It is set to the default value as zero.
- H.264 Decoder does not use displayWidth field. It is set to the default value as zero.
- frameOrder is a set to the enum value of IVIDDEC2_DISPLAY_ORDER.

- Frame skip is not supported. Set the `frameSkipMode` field to `IVIDEO_NO_SKIP`.
- H264 Decoder does not support `newFrameFlag` and `mbDataFlag` in this version. Their values are set as zero.

4.2.1.10 IVIDDEC2_InArgs

Description

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

Fields

Field	Datatype	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:

H264 Decoder copies the `inputID` value to the `outputID` value of `IVIDDEC2_OutArgs` structure after factoring in a display delay of 16.

4.2.1.11 IVIDDEC2_Status

Description

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

Fields

Field	Datatype	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. Table 4-3 lists all the codec specific codes generated by H264 Decoder. These are populated in the 8 LSBs of this field. If concealment is applied the decoder sets the XDM_APPLIEDCONCEALMENT bit-field.
data	XDM_SingleBufDesc	Output	Buffer information structure for information passing buffer.
maxNumDisplayBufs	XDAS_Int32	Output	The maximum number of buffers required by the codec.
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:

- Algorithm sets the `frameRate` and `bitRate` fields to default values.
- H264 Decoder shall not be using the buffer descriptor meant for passing additional information between the application and the decoder.

Table 4-3. H264 Decoder Error Codes.

IVIDDEC2_Status.extendedError (8 LSBs)	Error Code	Description
H264D_ERR_NOERROR	0x00	No error
H264D_ERR_SEM_NALU_FORBIDDENBIT	0x01	Semantic error in `forbidden_bit` value in NAL unit
H264D_ERR_SEM_NALU_NALREFIDC	0x02	Semantic error in `nal_ref_idc` value in NAL unit
H264D_ERR_SEM_NALU_NALUTYP	0x03	Semantic Error in `nal_unit_type` value in NAL unit
H264D_ERR_SEM_SPS_POCTYPE	0x04	Semantic error in `pic_order_cnt_type` value in SPS
H264D_ERR_SEM_SPS_MAXPOCLSB	0x05	Semantic error in `log2_max_pic_order_cnt_lsb_minus4` value in SPS
H264D_ERR_SEM_SPS_NUMREFFRAMESINPOCCYCLE	0x06	Semantic error in `num_ref_frames_in_pic_order_cnt_cycle` value in SPS
H264D_ERR_SEM_SPS_DIRECT8X8FLAG	0x07	Semantic error in `direct_8x8_inference_flag` in SPS
H264D_ERR_SEM_SPS_ACTIVESPS_MISMATCH	0x08	Mismatch between active SPS and new SPS with the same `seq_parameter_set_id`
H264D_ERR_SEM_SPS_SEQID	0x09	Semantic error in `seq_parameter_set_id` value in SPS
H264D_ERR_SEM_SPS_REF_FRAMES_BEYOND_LIMIT	0x0A	Semantic error in `num_ref_frames` value in SPS
H264D_ERR_MINOR_SPS_FIELDERROR	0x0B	Semantic error in `bit_depth_luma_minus8` and/or `bit_depth_chroma_minus8` fields in SPS
H264D_ERR_MINOR_SPS_MAXFRAMENUM	0x0C	Semantic error in `log2_max_frame_num_minus4` Value in SPS
H264D_ERR_SEM_SPS_FRAMECROPPING_PARAMS	0x0D	Semantic error in frame cropping parameters in SPS

IVIDDEC2_Status.extendedError (8 LSBs)	Error Code	Description
H264D_ERR_SEM_PPS_PPSID	0x0E	Semantic error in pic_parameter_set_id value in PPS
H264D_ERR_SEM_PPS_SEQID	0x0F	Semantic error in seq_parameter_set_id value in PPS
H264D_ERR_SEM_PPS_NUMREFIDXACTIVEL0	0x10	Semantic error in num_ref_idx_l0_active_minus1 value in PPS
H264D_ERR_SEM_PPS_NUMREFIDXACTIVEL1	0x11	Semantic error in num_ref_idx_l1_active_minus1 value in PPS
H264D_ERR_SEM_PPS_INITDQP	0x12	Semantic error in pic_init_qp_minus26 value in PPS
H264D_ERR_SEM_PPS_INITDQS	0x13	Semantic error in pic_init_qs_minus26 value in PPS
H264D_ERR_SEM_PPS_QPINDEXOFFSET	0x14	Semantic error in chroma_qp_index_offset and/or second_chroma_qp_index_offset value in PPS
H264D_ERR_SEM_PPS_ACTIVEPPS_MISMATCH	0x15	Mismatch between active PPS and new PPS with the same pic_parameter_set_id
H264D_ERR_SEM_PPS_BIPREDIDC_INVALID	0x16	Semantic error in weighted_bipred_idc value in PPS
H264D_ERR_SEM_SLCHDR_DELTAPICCNTBOT	0x17	Semantic error in delta_pic_order_cnt_bottom value in slice header
H264D_ERR_SEM_SLCHDR_PICPARAMSETID	0x18	Semantic error in pic_parameter_set_id Value in slice header
H264D_ERR_SEM_SLCHDR_SLCTYP	0x19	Semantic error in slice_type Value in slice header
H264D_ERR_SEM_SLCHDR_FIRSTMBINSLC	0x1A	Semantic error in first_mb_in_slice value in slice header
H264D_ERR_SEM_SLCHDR_IDRPICID	0x1B	Semantic error in idr_pic_id value in slice header
H264D_ERR_SEM_SLCHDR_REDUNDANTPICCNT	0x1C	Semantic error in redundant_pic_cnt value in slice header
H264D_ERR_SEM_SLCHDR_NUMREFIDXACTIVEL0	0x1D	Semantic error in num_ref_idx_l0_active_minus1 value in slice header
H264D_ERR_SEM_SLCHDR_NUMREFIDXACTIVEL1	0x1E	Semantic error in num_ref_idx_l1_active_minus1 value in slice header

IVIDDEC2_Status.extendedError (8 LSBs)	Error Code	Description
H264D_ERR_SEM_SLCHDR_CABACINITIDC	0x1F	Semantic error in cabac_init_idc value in slice header
H264D_ERR_SEM_SLCHDR_SLCQPDELTA	0x20	Semantic error in slice_qp_delta value in slice header
H264D_ERR_SEM_SLCHDR_SLCQSDELTA	0x21	Semantic error in slice_qs_delta value in slice header
H264D_ERR_SEM_SLCHDR_SLCALPHAC0OFFSET	0x22	Semantic error in slice_alpha_c0_ offset_div2 value in slice header
H264D_ERR_SEM_SLCHDR_SLCBETAOFFSET	0x23	Semantic error in slice_beta_ offset_div2 value in slice header
H264D_ERR_SEM_SLCHDR_DISABLEDEBLOCKFILTERIDC	0x24	Semantic error in disable_deblocking _filter_idc value in slice header
H264D_ERR_MINOR_SLCHDR_FRAMENUM	0x25	Semantic error in frame_num value in slice header
H264D_ERR_SEM_SLCHDR_ILLEGAL_PRED_WEIGHT	0x26	Semantic error in prediction weight table values in slice header
H264D_ERR_SEM_SLCHDR_SPS_CHANGE_IN_NONIDR	0x27	Semantic error detected due to change of SPS when current NAL unit type is not IDR
H264D_ERR_SEM_SLCHDR_POCLSB	0x28	Semantic error in pic_order_cnt_lsb value in slice header
H264D_ERR_SEM_RPLR_PICNUMSIDC	0x29	Error in reordering_of_pic_nums_idc Value in reference picture list reordering semantics
H264D_ERR_SEM_RPLR_ABSDIFFPICNUMMINUS1	0x2A	Error in abs_diff_pic_num_minus1 Value in reference picture list reordering semantics
H264D_ERR_SEM_SLCHDR_SPS_LTREFFLAG	0x2B	Error in long_term_reference_flag Value in decoder reference picture marking semantics
H264D_ERR_SEM_SLCHDR_PICINVAR	0x2C	Invalid change detected in slice header fields for the same picture
H264D_ERR_SEM_SLCHDR_LESS_ENTRIES_IN_LIST_0_1	0x2D	Number of entries in List 0/1 less than the signaled by the bit-stream
H264D_ERR_SEM_SLCHDR_MMCO_COMMANDS	0x2E	Limit reached on the number of MMCO commands detected in the bit-stream
H264D_ERR_SEM_SPS_CPBCNT	0x2F	Invalid value of cpb_cnt detected while parsing VUI parameters
H264D_ERR_NALU_SPS_INSUFFICIENT_BITS	0x30	Insufficient bits in NAL unit of type SPS

IVIDDEC2_Status.extendedError (8 LSBs)	Error Code	Description
H264D_ERR_NALU_PPS_INSUFFICIENT_BITS	0x31	Insufficient bits in NAL unit of type PPS
H264D_ERR_NALU_SLC_INSUFFICIENT_BITS	0x32	Insufficient bits in NAL unit of type Slice
H264D_ERR_SEM_SPS_INVLD_LEVEL	0x33	Invalid value of `level_idc` parameter detected in SPS
H264D_ERR_COPROCESSOR_STREAM_BUFFER_ERROR	0x40	Stream Buffer error given by HDVICP (ECD)–indicates corrupted data in bit-stream
H264D_ERR_COPROCESSOR_LESS_MBS_DECODED	0x41	HDVICP decodes lesser MBs than required for the slice – indicates less MB data available or Slice Lost
H264D_ERR_IMPL_NULL_REFFRAME	0x42	Reference frame not available for P or B slice
H264D_ERR_COPROCESSOR_EOSLC_DETECTION_FAILURE	0x43	HDVICP failed to detect End of Slice
H264D_ERR_IMPL_FRAMELOSS_DETECTED	0x44	Frame loss detected
H264D_ERR_IMPL_NULL_COL_PIC	0x45	Collocated Picture pointer is NULL
H264D_ERR_IMPL_MMCOPICMGMT_CORRUPTED_BITSTREAM	0x46	Error detected in the picture management function
H264D_ERR_IMPL_MMCOSLDWNDW_CORRUPTED_BITSTREAM	0x47	Error detected in the sliding window buffer management function
H264D_ERR_IMPL_PICSIZE_BEYOND_ALLOCATED_MEM	0x50	Change to higher resolution detected , however allocated buffer size is insufficient for decoding
H264D_ERR_IMPL_PPSUNAVAIL	0x60	Valid PPS not available
H264D_ERR_IMPL_SPSUNAVAIL	0x61	Valid SPS not available
H264D_ERR_SEM_SPS_FRAMESIZE_BEYOND_LEVEL	0x70	Picture size greater than allowed by the level
H264D_ERR_SEM_SPS_UNSUPPORTED_CHROMAFORMATIDC	0x71	`Chromaformat` 4:2:2 or 4:4:4 detected but not supported by the implementation
H264D_ERR_SEM_PPS_UNSUPPORTED_FMO	0x72	FMO detected but not supported by the implementation

IVIDDEC2_Status.extendedError (8 LSBs)	Error Code	Description
H264D_ERR_IMPL_NOTSUPPORTED_ASOFMO	0x73	ASO/FMO detected but not supported by the implementation
H264D_ERR_MINOR_UNSUPPORTED_PROFILE	0x74	Profile not supported by the implementation or blocked by the decoder using IH264VDEC_Params. presetProfileIdc
H264D_ERR_MINOR_LEVEL_BEYOND_SUPPORTED_LEVEL	0x75	Level greater than supported level or as specifed by IH264VDEC_Params. presetLevelIdc
H264D_PROF_CONSTRAINT_BASELINE_BSLICE	0x76	B slice detected for Baseline Profile
H264D_PROF_CONSTRAINT_BASELINE_CABAC	0x77	CABAC bit-stream detected for Baseline Profile
H264D_PROF_CONSTRAINT_BASELINE_INTERLACED	0x78	Interlaced content detected for Baseline Profile
H264D_PROF_CONSTRAINT_BASELINE_WEIGHTEDPRED	0x79	Weighted prediction detected for Baseline Profile
H264D_PROF_CONSTRAINT_BASELINEMAIN_TRANS8X8	0x7A	transform_8x8_mode_flag enabled for Baseline Profile
H264D_PROF_CONSTRAINT_MAINHIGH_FMO	0x7B	FMO detected for Main or High Profile
H264D_PROF_CONSTRAINT_MAINHIGH_ASO	0x7C	ASO detected for Main or High Profile
H264D_ERR_IDR_EXPECTED	0x80	The decoder sets this error code, if one of the errors from 0x50 to 0x7C (inclusive) occurred and the decoder is called again but without an IDR. The decoder expects an IDR to resume decoding. It rejects all frames until the next IDR is received. **Note:** All errors in the range 0x50 to 0x7C (inclusive) are marked as FATAL by the decoder.
H264D_ERR_COPROCESSOR_UNDEF_ECD_CMD	0x90	Bad command error given by HDVICP (ECD)
H264D_ERR_COPROCESSOR_UNDEF_BS_CMD	0x91	Bad command error given by HDVICP (BS)
H264D_ERR_COPROCESSOR_UNDEF_MC_CMD	0x92	Bad command error given by HDVICP (MC)
H264D_ERR_COPROCESSOR_UNDEF_CALC_CMD	0x93	Bad command error given by HDVICP (CALC)
H264D_ERR_COPROCESSOR_UNDEF_LPF_CMD	0x94	Bad command error given by HDVICP (LPF)
H264D_ERR_COPROCESSOR_RING_BUFFER_ERROR	0x95	Ring Buffer error given by HDVICP (ECD)–indicates corrupted data in bit-stream

IVIDDEC2_Status.extendedError (8 LSBs)	Error Code	Description
H264D_ERR_COPROCESSOR_DMA_NULL_POINTER	0x96	Null pointer passed as src/dst parameter for DMA
H264D_ERR_XDM_API_BADPARAMETERS	0x97	Bad/illegal parameters passed to decode process API
H264D_ERR_MC_COMMAND_INVOKED_ISLICE	0x98	MC command preparation is invoked for I slice
H264D_ERR_SEM_NALU_STARTCODEPREFIX	0x99	Unable to find start code prefix
H264D_ERR_FATAL_ERROR_OCCURRED	0x9A	The decoder sets this error code, if one of the errors from 0x90 to 0x99 (inclusive) occurred and the decoder is called again but not with FLUSH command. The decoder expects the application to flush out the remaining frames in its picture buffer using the XDM_FLUSH command. The application closes this instance of the decoder after it has flushed out all frames. **Note:** All errors in the range 0x90 to 0x99 (inclusive) are marked as FATAL by the decoder. Ideally, these errors should never occur.

Note:

- Errors from 0x01 to 0x47 are treated as Non-Fatal – the decoder continues processing of the bit-stream.
- Errors from 0x50 to 0x9A are treated as Fatal.

4.2.1.12 IVIDDEC2_OutArgs

Description

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

Fields

Field	Datatype	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 in array will indicate end of valid outputIDs within the array. Hence, the application

Field	Datatype	Input/ Output	Description
			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.
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.
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:

- OutputMbDataID and mbDataBuf ID are not used in this version of the decoder, as dumping of MB data is not supported.
- XDM_MAX_IO_BUFFERS - Maximum number of I/O buffers set to 20.

4.2.2 H264 Decoder Data Structures

This section includes the following H264 Decoder specific data structures:

- IH264VDEC_Params
- IH264VDEC_DynamicParams
- IH264VDEC_InArgs
- IH264VDEC_Status
- IH264VDEC_OutArgs

4.2.2.1 IH264VDEC _Params

‖ Description

This structure defines the creation parameters and any other implementation specific parameters for an H264 Decoder instance object. The creation parameters are defined in the XDM data structure, IVIDDEC2_Params.

‖ Fields

Field	Datatype	Input/ Output	Description
viddecParams	IVIDDEC2_Params	Input	See IVIDDEC2_Params data structure for details.
displayDelay	XDAS_Int32	Input	Display delay will decide the initial delay before which the decode call starts.
presetLevelIdc	eLevelNum_t	Input	The maximum H.264 level supported by the decoder.
presetProfileIdc	eProfile_t	Input	The profile decoded by the H264 Decoder, else an error is returned

Note:

- The default value of displayDelay is 16.
- The default value of presetLevelIdc is Level4.
- The default value of presetProfileIdc is Profile Any.

4.2.2.2 IH264VDEC_DynamicParams

Description

This structure defines the run-time parameters and any other implementation specific parameters for an H.264 instance object. The run-time parameters are defined in the XDM data structure, `IVIDDEC2_DynamicParams`.

Fields

Field	Datatype	Input/ Output	Description
`viddecDynamicParams`	`IVIDDEC2_DynamicParams`	Input	See `IVIDDEC2_DynamicParams` data structure for details.

4.2.2.3 IH264VDEC_InArgs

Description

This structure defines the run-time input arguments for an H264 instance object.

Fields

Field	Datatype	Input/ Output	Description
`viddecInArgs`	`IVIDDEC2_InArgs`	Input	See `IVIDDEC2_InArgs` data structure for details.

4.2.2.4 IH264VDEC_Status

Description

This structure defines parameters that describe the status of the H264 Decoder and any other implementation specific parameters. The `status` parameters are defined in the XDM data structure, `IVIDDEC2_Status`.

Fields

Field	Datatype	Input/ Output	Description
`viddecStatus`	`IVIDDEC2_Status`	Output	See `IVIDDEC2_Status` data structure for details

4.2.2.5 IH264VDEC_OutArgs

Description

This structure defines the run-time output arguments for the H264 Decoder instance object.

Fields

Field	Datatype	Input/ Output	Description
viddecOutArgs	IVIDDEC2_OutArgs	Output	See IVIDDEC2_OutArgs data structure for details.

4.3 Interface Functions

This section describes the application programming interfaces used in the H264 Decoder. The H264 Decoder 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.

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.

‖ 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. Since the client 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.

‖ See Also

`algNumAlloc(), algFree()`

4.3.2 Initialization API

Initialization API is used to initialize an instance of the H264 Decoder. The initialization parameters are defined in the `IVIDDEC2_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; /* handle to the algorithm instance*/

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. All fields in the `params` structure must be set as described in `IALG_Params` structure (see Data Structures section for details).

For more details, see TMS320 DSP Algorithm Standard API Reference.

‖ See Also

```
algAlloc(), algMoved()
```

4.3.3 Control API

Control API is used for controlling the functioning of H264 Decoder during run-time. This is done by changing the status of the controllable parameters of the decoder during run-time. These controllable parameters are defined in the `IVIDDEC2_DynamicParams` data structure (see Data Structures section for details).

‖ Name

`control()` – change run-time parameters of the H264 Decoder and query the decoder status

‖ Synopsis

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

‖ Arguments

```
IVIDDEC2_Handle handle; /* handle to the H264 decoder
instance */

IVIDDEC2_Cmd id; /* H264 decoder specific control
commands*/

IVIDDEC2_DynamicParams *params /* H264 decoder run-time
parameters */

IVIDDEC2_Status *status /* H264 decoder instance status
parameters */
```

‖ Return Value

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

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

‖ Description

This function changes the run-time parameters of H264 Decoder and queries the status of decoder. `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 the H264 Decoder instance object.

The second argument is a command ID. See `IVIDDEC2_Cmd` in enumeration table for details.

The third and fourth arguments are pointers to the `IVIDDEC2_DynamicParams` and `IVIDDEC2_Status` data structures respectively.

‖ See Also

`algInit()`

4.3.4 Data Processing API

Data processing API is used for processing the input data using the H264 Decoder.

‖ 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 video 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; /* handle to the H264 decoder
instance */

XDM1_BufDesc *inBufs; /* pointer to input buffer descriptor
data structure */

XDM_BufDesc *outBufs; /* pointer to output buffer descriptor
data structure */

IVIDDEC2_InArgs *inargs /* pointer to the H264 decoder
runtime input arguments data structure */

IVIDDEC2_OutArgs *outargs /* pointer to the H264 decoder
runtime output arguments data structure */
```

‖ Return Value

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

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

‖ Description

This function does the basic H264 video decoding. The first argument to process() is a handle to the H264 Decoder instance object.

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 the H264 Decoder instance object.

> **Note:**
>
> Prior to each decode call, ensure that all fields are set as described in XDM1_BufDesc, XDM_BufDesc, and IVIDDEC2_InArgs structures.

The last argument is a pointer to the IVIDDEC2_OutArgs data structure that defines the run-time output arguments for the H264 Decoder instance object.

The algorithm may also modify the output buffer pointers. The return value is IALG_EOK for success or IALG_EFAIL in case of failure. The extendedError field of the IVIDDEC2_Status structure contains error conditions flagged by the algorithm. This structure can be populated by a calling Control API using XDM_GETSTATUS command.

‖ See Also

control()

‖ 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.

‖ See Also

`algActivate()`

4.3.5 Termination API

Termination API is used to terminate the H264 Decoder 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()`