H.264 Baseline Profile Decoder on C64x+

User's Guide

Literature Number: SPRUEA1C
December 2008

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

Preface

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 C64x+ platform. It also provides a detailed Application Programming Interface (API) reference and information on the sample application that accompanies this component.

TI's codec implementations are based on the eXpressDSP Digital Media (XDM) standard. XDM is an extension of the eXpressDSP Algorithm Interface Standard (XDAIS).

Intended Audience

This document is intended for system engineers who want to integrate TI's codecs with other software to build a multimedia system based on the C64x+ platform.

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

How to Use This Manual

This document includes the following chapters:

- **Chapter 1 - Introduction**, provides a brief introduction to the XDAIS and XDM standards. It also provides an overview of the codec and lists its supported features.
- **Chapter 2 - Installation Overview**, describes how to install, build, and run the codec.
- **Chapter 3 - Sample Usage**, describes the sample usage of the codec.
- **Chapter 4 - API Reference**, describes the data structures and interface functions used in the codec.
- **Appendix A – Revision History**, highlights the changes made to the SPRUEA1B codec specific user guide to make it SPRUEA1C.

Related Documentation From Texas Instruments

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

- *TMS320 DSP Algorithm Standard Rules and Guidelines* (literature number SPRU352) defines a set of requirements for DSP algorithms that, if followed, allow system integrators to quickly assemble production-quality systems from one or more such algorithms.
- *TMS320 DSP Algorithm Standard API Reference* (literature number SPRU360) describes all the APIs that are defined by the TMS320 DSP Algorithm Interface Standard (also known as XDAIS) specification.
- *Technical Overview of eXpressDSP - Compliant Algorithms for DSP Software Producers* (literature number SPRA579) describes how to make algorithms compliant with the TMS320 DSP Algorithm Standard which is part of TI's eXpressDSP technology initiative.
- *Using the TMS320 DSP Algorithm Standard in a Static DSP System* (literature number SPRA577) describes how an eXpressDSP-compliant algorithm may be used effectively in a static system with limited memory.
- *DMA Guide for eXpressDSP-Compliant Algorithm Producers and Consumers* (literature number SPRA445) describes the DMA architecture specified by the TMS320 DSP Algorithm Standard (XDAIS). It also describes two sets of APIs used for accessing DMA resources: the IDMA2 abstract interface and the ACPY2 library.
- *eXpressDSP Digital Media (XDM) Standard API Reference.(* literature number SPRUEC8)

The following documents describe TMS320 devices and related support tools:

- *Design and Implementation of an eXpressDSP-Compliant DMA Manager for C6X1X* (literature SPRA789) describes a C6x1x-optimized (C6211, C6711) ACPY2 library implementation and DMA Resource Manager.
- *TMS320c64x+ Megamodule* (literature 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 SPRU871) describes the C64x+ megamodule peripherals.
- *TMS320C64x to TMS320C64x+ CPU Migration Guide* (literature 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 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.
- *TMS320DM6446 Digital Media System-on-Chip* (literature number SPRS283)
- *TMS320DM6446 Digital Media System-on-Chip Errata (Silicon Revision 1.0)* (literature number SPRZ241) describes the known exceptions to the functional specifications for the TMS320DM6446 Digital Media System-on-Chip (DMSoC).
- TMS320DM6443 Digital Media System-on-Chip (literature number SPRS282)
- *TMS320DM6443 Digital Media System-on-Chip Errata (Silicon Revision 1.0)* (literature number SPRZ240) describes the known exceptions to the functional specifications for the TMS320DM6443 Digital Media System-on-Chip (DMSoC).
- *TMS320DM644x DMSoC DSP Subsystem Reference Guide* (literature number SPRUE15) describes the digital signal processor (DSP) subsystem in the TMS320DM644x Digital Media System-on-Chip (DMSoC).
- *TMS320DM644x DMSoC ARM Subsystem Reference Guide* (literature number SPRUE14) describes the ARM subsystem in the TMS320DM644x Digital Media System on a Chip (DMSoC).
- *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. 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
DMAN3	DMA Manager
DPB	Decoded Picture Buffer
EVM	Evaluation Module
FMO	Flexible Macroblock Ordering
HDTV	High Definition Television
HRD	Hypothetical Reference Decoder
I_PCM	Intra-frame pulse code modulation
IDR	Instantaneous Decoding Refresh
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
MV	Motion Vector
NAL	Network Adaptation Layer

Abbreviation	Description
NTSC	National Television Standards Committee
PicAFF	Picture Adaptive Field Frame
POC	Picture Order Count
RTOS	Real Time Operating System
SEI	Supplemental Enhancement Information
VCL	Video Coded Layer
VGA	Video Graphics Array (640 x 480 resolution)
VUI	Video Usability Information
XDAIS	eXpressDSP Algorithm Interface Standard
XDM	eXpressDSP Digital Media
YUV	Color space in luminance and chrominance form

Text Conventions

The following conventions are used in this document:

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

Product Support

When contacting TI for support on this codec, quote the product name (H.264 Baseline Profile Decoder on C64x+) and version number. The version number of the codec is included in the Title of the Release Notes that accompanies this codec.

Trademarks

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

All trademarks are the property of their respective owners.

This page is intentionally left blank

Contents

Figures

This page is intentionally left blank

Tables

This page is intentionally left blank

Chapter 1

Introduction

This chapter provides a brief introduction to XDAIS and XDM. It also provides an overview of TI's implementation of the H.264 Baseline Profile Decoder on the C64x+ platform and its supported features.

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

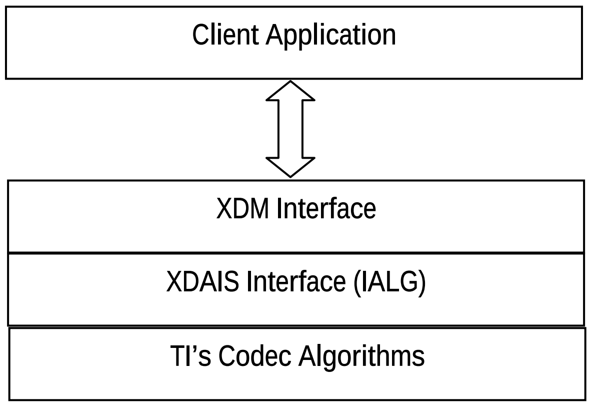
In the multimedia application space, you have the choice of integrating any codec into your multimedia system. For example, if you are building a video decoder system, you can use any of the available video decoders (such as MPEG4, H.263, or H.264) in your system. To enable easy integration with the 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, which specify restrictions on the bit stream, and hence limits the capabilities needed to decode the bit streams. Each profile specifies a subset of algorithmic features and limits that 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 the 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
  - ASO/FMO and redundant slices for error concealment 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

The H.264 Baseline Profile Decoder is a completely programmable single-chip solution. 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, which is subjected to DCT and later quantized.
- Inter coded data: Motion information and residual error data (differential data between two frames), which 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 can be of format 420 planar and 422 interleaved in little endian.

Figure 1-1 depicts the working of the decoder.

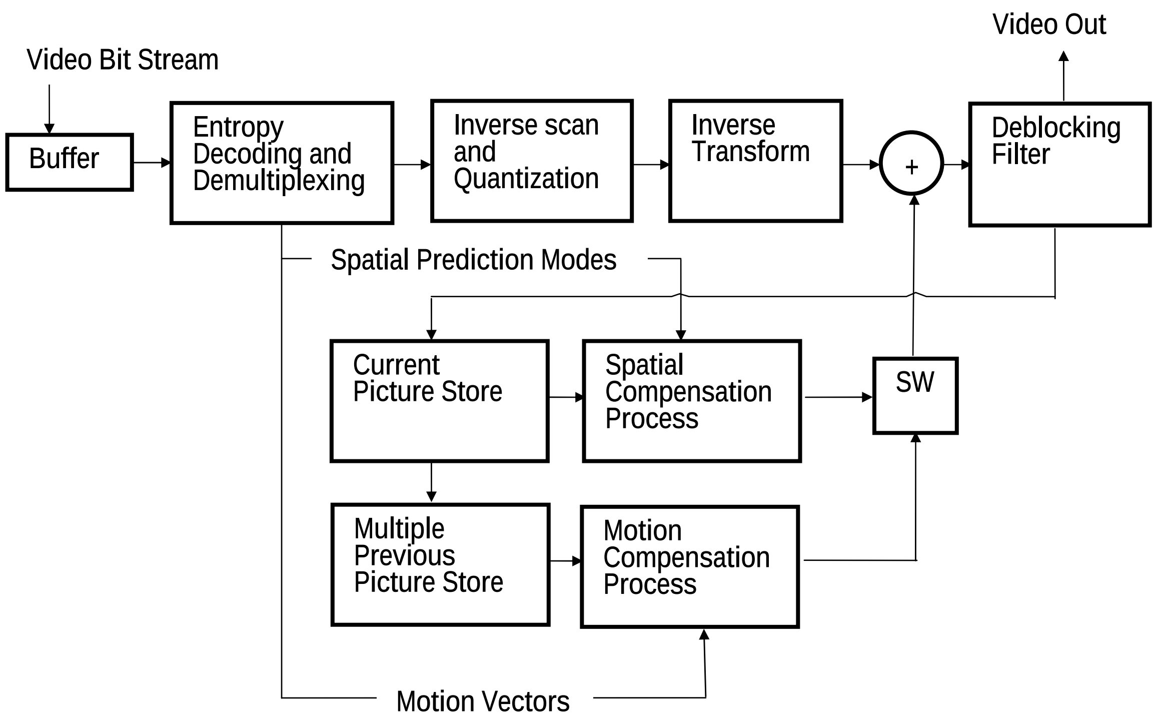

Video Bit Stream
Buffer
Entropy Decoding and Demultiplexing
Inverse scan and Quantization
Inverse Transform
+
Deblocking Filter
Video Out
Spatial Prediction Modes
Current Picture Store
Spatial Compensation Process
SW
Multiple Previous Picture Store
Motion Compensation Process
Motion Vectors

Figure 1-1. Block Diagram of H.264 Decoder

From this point onwards all references to H.264 Decoder means H.264 Baseline Profile Decoder Profile (BP) decoder only.

1.3 Supported Services and Features

This user guide accompanies TI's implementation of H.264 Decoder on the C64x+ platform.

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

- eXpressDSP Digital Media (XDM 1.0 IVIDDEC2) compliant
- Supports up to Level 3.0 features of the Baseline Profile (BP)
- Supports progressive frame 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 size being non-multiple of 16 through frame cropping
- Supports frame width of the range of 32 to 720 pixels

- Supports byte-stream syntax and NAL unit format for the input bit stream
- Supports long term reference frames
- Supports gaps in `frame_num`
- Supports decoding of streams with IPCM coded macro blocks
- Supports skipping of non-reference pictures
- Supports configurable delay for display of frames
- Supports error resiliency
- Supports error concealment
- Outputs are available in YUV 420 planar and 422 interleaved little endian formats
- Tested for compliance with JM version 11.0 reference decoder
- Supports ASO and FMO error-concealment features
- Supports redundant slices
- Supports parsing of Supplemental Enhancement Information (SEI) and Video Usability Information (VUI)
- Adaptive reference picture marking
- Reference picture list reordering
- Supports all resolutions up to D1 (PAL and NTSC) including CIF and QCIF

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.

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 DM644x EVM with XDS560 USB.

This codec can also be used on any of TI's C64x+ based platforms such as DM644x, DM648, DM643x, OMAP35xx and their derivatives.

2.1.2 Software

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

- **Development Environment:** This project is developed using Code Composer Studio version 3.3.24.1.
- **Code Generation Tools:** This project is compiled, assembled, archived, and linked using the code generation tools version 6.0.8.

2.2 Installing the Component

The codec component is released as a compressed archive. To install the codec, extract the contents of the zip file onto your local hard disk. The zip file extraction creates a top-level directory called 100_V_H264AVC_D_2_00, under which another directory named DM644x_BP_001 is created.

Figure 2-1 shows the sub-directories created in the DM644x_BP_001 directory.

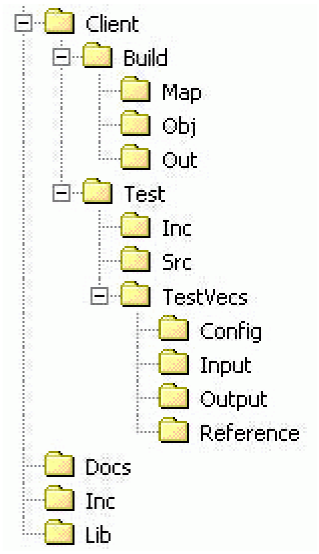


Figure 2-1. Component Directory Structure

Note:

If you are installing an evaluation version of this codec, the directory name will be 100E_V_H264AVC_D_2_00.

Table 2-1 provides a description of the sub-directories created in the DM644x_BP_001 directory.

Table 2-1. Component Directories

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

2.3 Before Building the Sample Test Application

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

This version of the codec has been validated with DSP/BIOS version 5.32.02 and Framework Component (FC) version 2.20.00.15.

2.3.1 Installing DSP/BIOS

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

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

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

<install directory>\CCStudio_v3.3

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 Framework Component (FC)

You can download FC from the TI external website:

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

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

<install directory>\CCStudio_v3.3

The test application uses the following DMAN3 files:

- Library file, dman3.a64P available in the <install directory>\CCStudio_v3.3\<fc_directory>\packages \ti\sdo\fc\dman3 directory.
- Header file, dman3.h available in the <install directory>\CCStudio_v3.3\<fc_directory>\packages \ti\sdo\fc\dman3 directory.
- Header file, idma3.h available in the <install directory>\CCStudio_v3.3\<fc_directory>\packages \ti\sdo\fc\acpy3 directory.

2.4 Building and Running the Sample Test Application

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

1) Verify that you have installed TI's Code Composer Studio version 3.3.24.1 and code generation tools version 6.0.8.

2) Verify that the codec object library h264vdec_ti.l64P exists in the \Lib sub-directory.

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

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

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

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

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

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

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

2.5 Configuration Files

This codec is shipped along with:

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

2.5.1 Generic Configuration File

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

The format of the Testvecs.cfg file is:

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

where:

- `X` may be set as:
  - 1 - for compliance checking, no output file is created
  - 0 - for writing the output to the output file

  The default setting of Testvecs.cfg file is for compliance checking.
- `Config` is the Decoder configuration file. For details, see section 2.5.2
- `Input` is the input file name (use complete path).
- `Output/Reference` is the output file name (if `X` is 0) or reference file name (if `X` is 1).

A sample Testvecs.cfg file is as shown.

```
1
..\..\Test\TestVecs\Config\Testparams.cfg
..\..\Test\TestVecs\Input\foreman_vga.264
..\..\Test\TestVecs\Reference\foreman_vga.yuv
0
..\..\Test\TestVecs\Config\Testparams.cfg
..\..\Test\TestVecs\Input\foreman_vga.264
..\..\Test\TestVecs\Output\foreman_vga.yuv
```

2.5.2 Decoder Configuration File

The decoder configuration file, Testparams.cfg contains the configuration parameters required for the decoder. The Testparams.cfg file is available in the \Client\Test\TestVecs\Config sub-directory.

A sample Testparams.cfg file is as shown.

```
# Input File Format is as follows
# <ParameterName> = <ParameterValue> # Comment
############################################################
Parameters
############################################################

ImageWidth = 720    # Image width in Pels, must be
                      multiples of 16
ImageHeight = 576   # Image height in Pels, must be
                      multiples of 16
ChromaFormat = 1    # 1 => XDM_YUV_420P,
                      4 => XDM_YUV_422ILE
FramesToDecode = 10 # Number of frames to be decoded
```

Any field in the `IVIDDEC2_Params` structure (see Section 4.2.1.8) can be set in the Testparams.cfg file using the syntax shown above.

2.6 Standards Conformance and User-Defined Inputs

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

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

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

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

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

4) Execute the sample test application. On successful completion, the application displays one of the following message for each frame:
   - o “Decoder compliance test passed/failed” (if `X` is 1)
   - o “Decoder output dump completed” (if `X` is 0)

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

2.7 Uninstalling the Component

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

2.8 Evaluation Version

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

Note:

Bit compliance test succeeds only for the example input file provided with the evaluation package, due to the presence of Texas Instruments logo in the decoded output. Hence, bit compliance should not be checked for other inputs.

Chapter 3

Sample Usage

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

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.

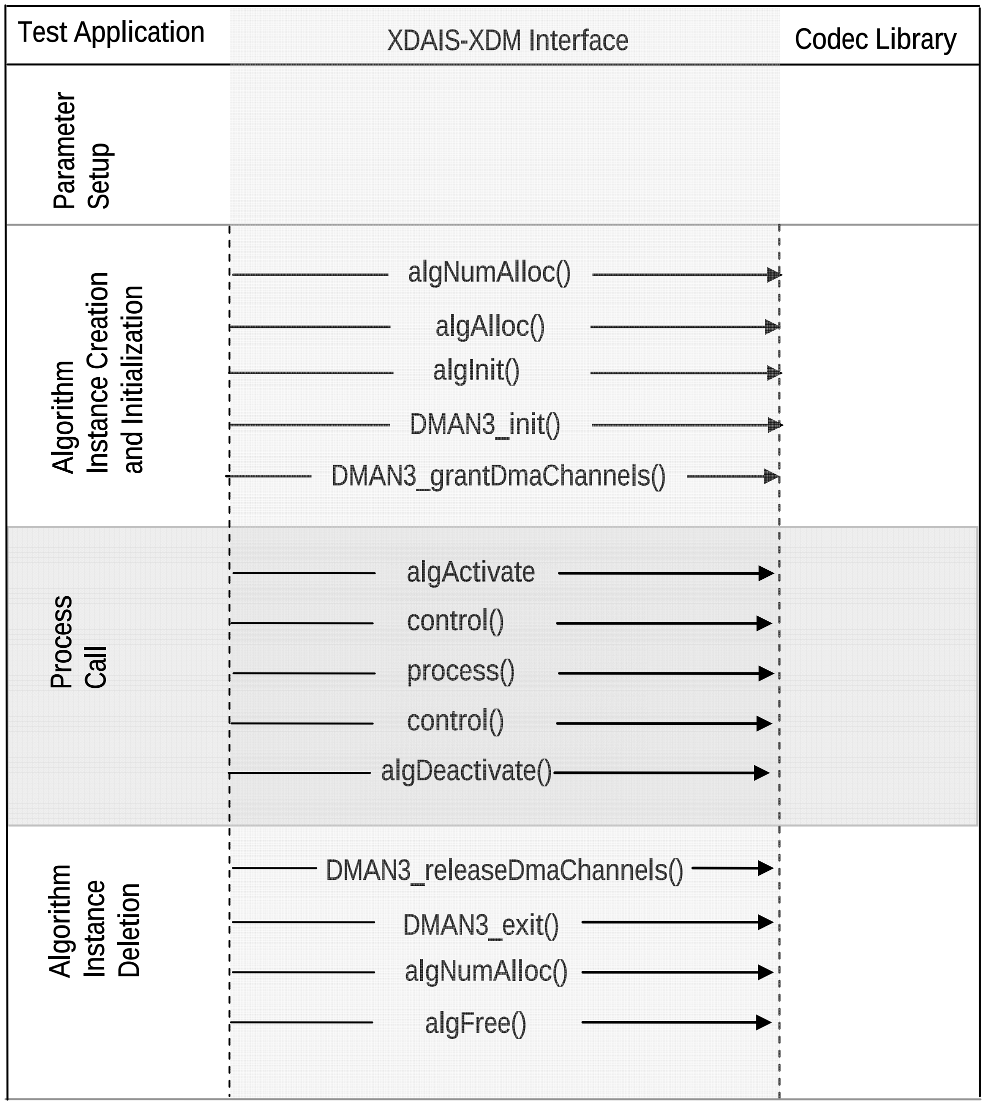


Figure 3-1. Test Application Sample Implementation

The test application is divided into four logical blocks:

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

3.1.1 Parameter Setup

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

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

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

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

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

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

4) Initializes the various DMAN3 parameters.

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

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

3.1.2 Algorithm Instance Creation and Initialization

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

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

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

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

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

After successful creation of the algorithm instance, the test application does DMA resource allocation for the algorithm. This requires initialization of DMA Manager Module and grant of DMA resources. This is implemented by calling DMAN3 interface functions in the following sequence:

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

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

Note:

DMAN3 function implementations are provided in dman3.a64P library.

3.1.3 Process Call

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

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

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

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

There could be any ordering of `control()` and `process()` functions. The following APIs are called in sequence:

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

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

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

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

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

3.1.4 Algorithm Instance Deletion

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

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

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

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

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

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

3.2 Frame Buffer Management by Application

3.2.1 Frame Buffer Input and Output

With the new XDM 1.0, decoder does not ask for frame buffer at the time of `alg_create()`. It uses buffer from `XDM1_BufDesc *outBufs`, which 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.

```
ALG_create();
ividDecFxns->control(XDM_GETBUFINFO); /* Returns default
WVGA size */
do{
ividDecFxns->control(XDM_SETPARAMS)
ividDecFxns->process(); //call the decode API
ividDecFxns->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 control call returns buffer that is of 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.

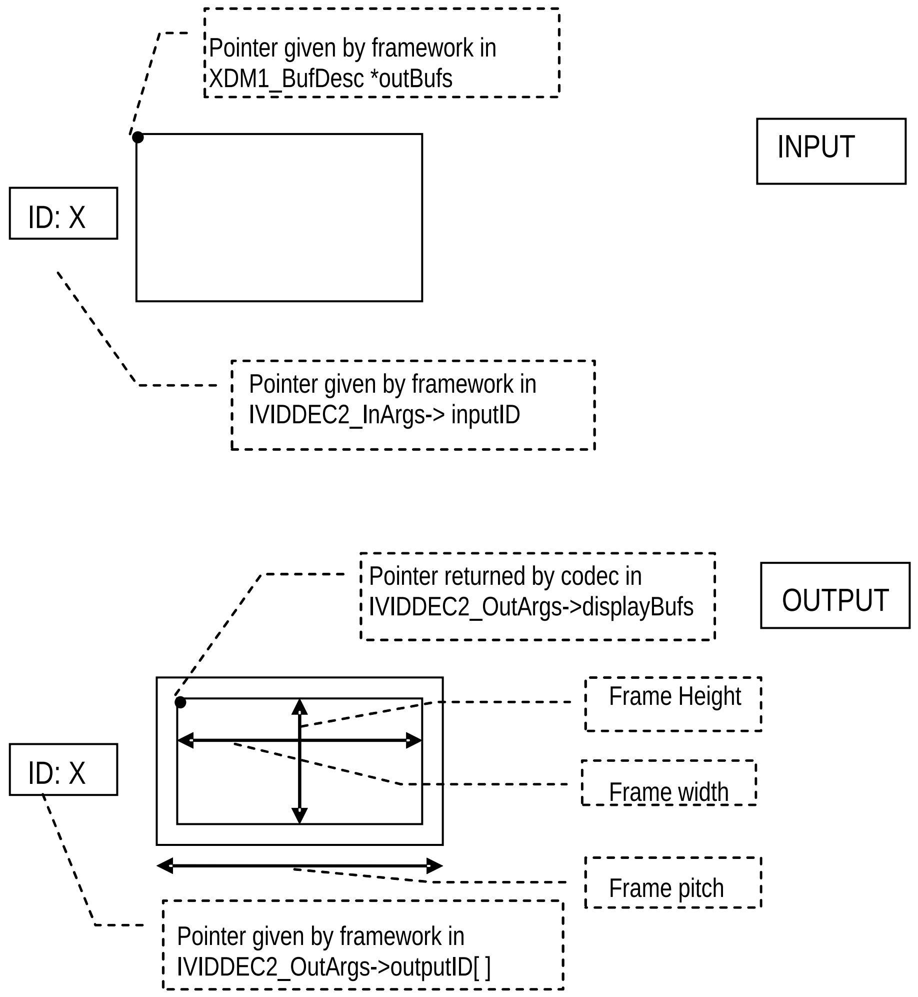

Pointer given by framework in XDM1_BufDesc *outBufs
INPUT
ID: X
Pointer given by framework in IVIDDEC2_InArgs-> inputID
Pointer returned by codec in IVIDDEC2_OutArgs->displayBufs
OUTPUT
Frame Height
ID: X
Frame width
Frame pitch
Pointer given by framework in IVIDDEC2_OutArgs->outputID[ ]

Figure 3-2. 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 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.

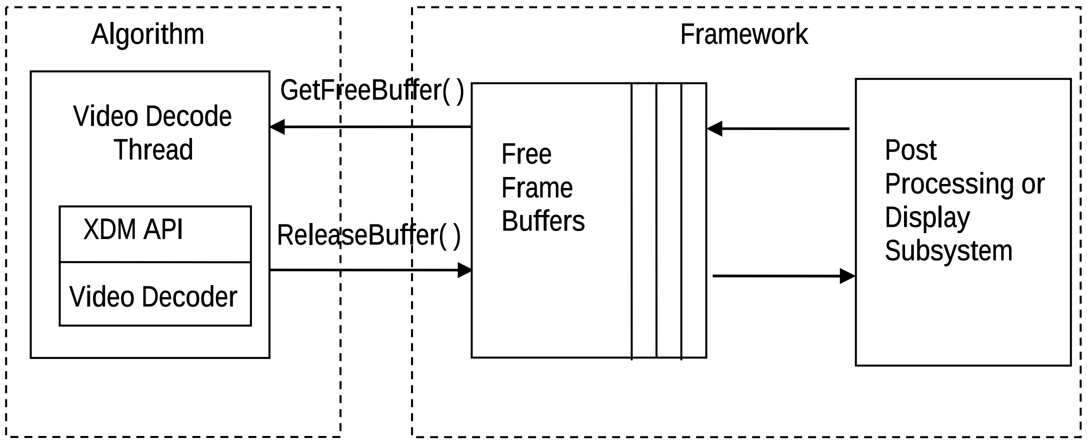


Figure 3-3. Interaction of Frame Buffers Between Application and Framework

The sample application also provides a prototype for managing frame buffers. It implements the following functions, which is 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 is 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 is 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 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*/

/*------------------Set Set params-----------------*/
TestApp_SetInitParams(&params.viddecParams);

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

/*--------------- Set Dynamic params ----------------*/
TestApp_SetDynamicParams(&dynamicParams.viddecDynamicParams);

/*-------------- Get Buffer information -------------*/
ividDecFxns->control(handle, XDM_GETBUFINFO);

BUFFMGR_Init();


/* 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() */

      ividDecFxns->control(handle, XDM_SETPARAMS);

/*-------------------------------------------------------*/
/* Start the process : To start decoding a frame      */
/* This will always follow a H264VDEC_decode_end call */
/*-------------------------------------------------------*/


 retVal = ividDecFxns->process(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 :      */
       ividDecFxns->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.viddecOutArgs.freeBufID);

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

BUFFMGR_ReleaseAllBuffers();

BUFFMGR_DeInit();

ALG_delete (handle);
```

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.

4.1 Symbolic Constants and Enumerated Data Types

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

Table 4-1. List of Enumerated Data Types

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
`IVIDEO_FrameType`		
	`IVIDEO_NA_FRAME`	Frame type not available
	`IVIDEO_I_FRAME`	Intra coded frame. Default value
	`IVIDEO_P_FRAME`	Forward inter coded frame.
	`IVIDEO_B_FRAME`	Bi-directional inter coded frame. Not supported in this version of H.264 decoder.
	`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

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	`IVIDEO_MBAFF_B_FRAME`	Bi-directional inter coded MBAFF frame
	`IVIDEO_MBAFF_IDR_FRAME`	Intra coded MBAFF frame that can be used for refreshing video content.
	`IVIDEO_FRAMETYPE_DEFAULT`	Default set to `IVIDEO_I_FRAME`
`IVIDEO_ContentType`	`IVIDEO_CONTENTTYPE_NA`	Content type is not applicable
	`IVIDEO_PROGRESSIVE`	Progressive video content
	`IVIDEO_PROGRESSIVE_FRAME`	Progressive video content
	`IVIDEO_INTERLACED`	Interlaced video content. Not supported in this version of H.264 decoder.
	`IVIDEO_INTERLACED_FRAME`	Interlaced video content Not supported in this version of H264 Decoder.
	`IVIDEO_INTERLACED_TOPFIELD`	Interlaced picture, top field Not supported in this version of H264 Decoder.
	`IVIDEO_INTERLACED_BOTTOMFIELD`	Interlaced picture, bottom field Not supported in this version of H264 Decoder.
	`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 H.264 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 H.264 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.

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	`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.
	`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 applicable for H.264 decoder
	`XDM_LE_32`	32-bit little endian stream. Not applicable for H.264 decoder
	`XDM_LE_64`	64 bit little endian stream. Not applicable for H.264 decoder
	`XDM_BE_16`	16 bit big endian stream. Not applicable for H.264 decoder
	`XDM_BE_32`	32 bit big endian stream. Not applicable for H.264 decoder
	`XDM_BE_64`	64 bit big endian stream. Not applicable for H.264 decoder
`XDM_DecMode`	`XDM_DECODE_AU`	Decode entire access unit, including all the headers.
	`XDM_PARSE_HEADER`	Decode only header. `XDM_PARSE_HEADER` should be set for non-VLD units like SPS, PPS, SEI and so on, which occurs before start of a new frame and end of previous frame.
`XDM ChromaFormat`	`XDM_CHROMA_NA`	Chroma format not applicable

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_YUV_420P	YUV 4:2:0 planar
	XDM_YUV_422P	YUV 4:2:2 planar. Not applicable for H.264 decoder
	XDM_YUV_422IBE	YUV 4:2:2 interleaved (big endian). Not applicable for H.264 decoder
	XDM_YUV_422ILE	YUV 4:2:2 interleaved (little endian)
	XDM_YUV_444P	YUV 4:4:4 planar. Not applicable for H.264 decoder
	XDM_YUV_411P	YUV 4:1:1 planar. Not applicable for H.264 decoder
	XDM_GRAY	Gray format. Not applicable for H.264 decoder
	XDM_RGB	RGB color format. Not applicable for H.264 decoder
	XDM_CHROMAFORMAT_DEFAULT	Default set to XDM_YUV_422ILE
XDM_CmdId	XDM_GETSTATUS	Query algorithm instance to fill Status structure
	XDM_SETPARAMS	Set run-time dynamic parameters via the DynamicParams structure
	XDM_RESET	Reset the algorithm
	XDM_SETDEFAULT	Initialize all fields in DynamicParams structure to default values specified in the library
	XDM_FLUSH	Handle end of stream conditions. This command forces algorithm instance to output data without additional input.
	XDM_GETBUFINFO	Query algorithm instance regarding the properties of input and output buffers
	XDM_GETVERSION	Query the algorithms version. The result will be returned in the @c data field of the respective _Status structure.
	XDM_GETCONTEXTINFO	Query a split codec part for its context needs. Only split codecs are required to implement this command. Not supported in this version of H264 Decoder.

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
`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_PARAMSCHANGE`	Bit 8 ❑ 1 - Sequence Parameters Change. ❑ 0 - Ignore Not applicable for this version H264 decoder.
	`XDM_APPLIEDCONCEALMENT`	Bit 9 ❑ 1 - Applied concealment ❑ 0 - Ignore
	`XDM_INSUFFICIENTDATA`	Bit 10 ❑ 1 - Insufficient data ❑ 0 - Ignore
	`XDM_CORRUPTEDDATA`	Bit 11 ❑ 1 - Data problem/corruption ❑ 0 - Ignore
	`XDM_CORRUPTEDHEADER`	Bit 12 ❑ 1 - Header problem/corruption ❑ 0 - Ignore
	`XDM_UNSUPPORTEDINPUT`	Bit 13 ❑ 1 - Unsupported feature/parameter in input ❑ 0 - Ignore
	`XDM_UNSUPPORTEDPARAM`	Bit 14 ❑ 1 - Unsupported input parameter or configuration ❑ 0 - Ignore
	`XDM_FATALERROR`	Bit 15 ❑ 1 - Fatal error (stop decoding) ❑ 0 - Recoverable error

Note:

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

- Bit 16-32: Reserved
- Bit 8: Reserved
- Bit 0-7: Codec and implementation specific. The type of error encountered while decoding the bitstream is returned through `extendedError` field of `outputArgs`. Bits 8-15 are set as per XDM convention. Bits 0-7 are used to indicate errors specific to H.264

Decoder. The various error codes returned by the H.264 Decoder (in the lower 8-bits) and their values are given in Table 4-3. 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
`eH264VDEC_Profile`	`PROFILE_INVALID`	Used to indicate unsupported profile (-1)
	`BASELINE`	Baseline profile (0)
	`MAIN`	Main profile (1)
	`EXTENDED`	Extended profile (2)
	`MAX_PROFILES_TYPES`	Maximum Number of Profile (3)
`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
	`Level41`	12: Level 4.1
	`Level42`	13: Level 4.2
	`Level5`	14: Level 5
	`Level51`	15: Level 5.1
	`MAXLEVELID`	16: Maximum Level ID that is

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
		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)

The following table lists the detailed error codes and their values.

Table 4-3. Error Codes and Values.

Error codes	Description	Values
NAL Unit specific semantic Errors		
`H264D_ERR_SEM_NALU_EOSTRMREACHED`	Additional NALU is received after an end of stream NALU	`0x21`
`H264D_ERR_SEM_NALU_FORBIDDENBIT`	NALU syntax forbidden bit is not zero	`0x22`
`H264D_ERR_SEM_NALU_NALREFIDC`	The `nal_ref_idc` field has a value that violates constraints specified in the standard.	`0x23`
`H264D_ERR_SEM_NALU_NALUTYP`	Incorrect NALU type received. It may not be an illegal NALU type, but incorrect based on the type of previous NALU.	`0x25`
`H264D_ERR_SEM_NALU_EOSEQ`	End of sequence NALU is incorrectly received when a picture is partially decoded. This is not an error. It is displayed as a warning only.	`0x26`
`H264D_ERR_SEM_NALU_STARTCODEPREFIX`	Incorrect start code prefix or received less than 4 bytes at the end of the sequence. This is not an error. It is displayed as a warning only.	`0x27`
SPS specific semantic Errors		
`H264D_ERR_SEM_SPS_INVLD_PROFILE`	The profile specified in SPS is invalid or is unsupported by the decoder	`0x41`
`H264D_ERR_SEM_SPS_INVLD_LEVEL`	The level specified in SPS is invalid or is unsupported by the decoder	`0x42`
`H264D_ERR_SEM_SPS_POCTYPE`	The `pic_order_cnt_type` field decoded as part of SPS has an illegal value	`0x43`

Error codes	Description	Values
`H264D_ERR_SEM_SPS_MAXPOCLSB`	The `log2_max_pic_order_cnt_lsb_minus4` field decoded as part of SPS has an illegal value.	`0x44`
`H264D_ERR_SEM_SPS_NUMREFFRAMESINPOCCYCLE`	The `num_ref_frames_in_pic_order_cnt_cycle` field decoded as part of SPS has an illegal value	`0x45`
`H264D_ERR_SEM_SPS_DIRECT8X8FLAG`	The `direct_8x8_inference_flag` field decoded as part of SPS has an illegal value	`0x46`
`H264D_ERR_SEM_SPS_FRAMECROP`	The frame cropping parameters decoded as part of SPS have an illegal value	`0x47`
`H264D_ERR_SEM_SPS_ACTIVESPS_MISMATCH`	If in between access unit decoding, a SPS is received with same `seq_parm_set_id` as `active_sps_id`, then contents of `received_sps` should be equal to `active_sps`. Otherwise, this error code is set.	`0x48`
`H264D_ERR_SEM_SPS_SEQID`	The field `seq_parameter_set_id` of SPS has an illegal value	`0x49`
`H264D_ERR_SEM_SPS_UNSUPPORTEDPICWIDTH`	The width specified in SPS is not supported by the decoder	`0x4A`
`H264D_ERR_SEM_SPS_REF_FRAMES_BEYOND_LIMIT`	The number of reference frames specified in SPS is beyond the limit allowed by the standard	`0x4B`
`H264D_ERR_SEM_SPS_UNSUPPORTEDPICHEIGHT`	The height specified in SPS is not supported by the decoder	`0x4C`
PPS specific semantic Errors		
`H264D_ERR_SEM_PPS_PPSID`	The field `pic_parameter_set_id` part of PPS has an illegal value	`0x61`
`H264D_ERR_SEM_PPS_SEQID`	The `seq_parameter_set_id` field part of PPS has an illegal value	`0x62`
`H264D_ERR_SEM_PPS_SLCGRPMAPTYPE`	The `slice_group_map_type` field in PPS has an illegal or incorrect value	`0x63`
`H264D_ERR_SEM_PPS_TOPLEFT`	The field in PPS used for certain type of FMO has a wrong value	`0x64`
`H264D_ERR_SEM_PPS_BOTRIGHT`	The field in PPS used for certain type of FMO has a wrong value	`0x65`
`H264D_ERR_SEM_PPS_TOPBOTMOD`	The field in PPS used for certain type of FMO has a wrong value	`0x66`

Error codes	**Description**	**Values**
`H264D_ERR_SEM_PPS_RUNLENGTH`	The field in PPS used for certain type of FMO has a wrong value	`0x67`
`H264D_ERR_SEM_PPS_SLCGRPCHNGRATE`	The field in PPS used for certain type of FMO has a wrong value	`0x68`
`H264D_ERR_SEM_PPS_PICSIZEMAPUNITS`	The field `pic_size_in_map_units_minus1` in PPS has an incorrect value	`0x69`
`H264D_ERR_SEM_PPS_NUMREFIDXACTIVEL0`	The field `num_ref_idx_l0_active_minus1` in PPS has an illegal value	`0x6A`
`H264D_ERR_SEM_PPS_NUMREFIDXACTIVEL1`	The field `num_ref_idx_l0_active_minus1` in PPS has an illegal value	`0x6B`
`H264D_ERR_SEM_PPS_INITDQP`	The field `pic_init_qp_minus26` in PPS has a value out of bounds with what is specified by standard	`0x6C`
`H264D_ERR_SEM_PPS_INITDQS`	The field `pic_init_qs_minus26` in PPS has a value out of bounds with what is specified by standard	`0x6D`
`H264D_ERR_SEM_PPS_QPINDEXOFFSET`	The field `chroma_qp_index_offset` in PPS has a value out of bounds with what is specified by standard	`0x6E`
`H264D_ERR_SEM_PPS_ACTIVEPPS_MISMATCH`	If in between access unit decoding, a PPS is received with same `pic_parm_set_id as active_pps_id`, then contents of `received_pps` should be equal to `active_pps`. Otherwise, this error code is set.	`0x6F`
`H264D_ERR_SEM_PPS_NUMSLCGRP`	The num_slice_groups_minus1 field in PPS has an illegal value	`0x70`
`H264D_ERR_SEM_PPS_SLCGRPID`	The field `slice_group_id` in PPS has an incorrect value (based on `num_slice_groups_minus1` field)	`0x71`
`H264D_ERR_SEM_PPS_BIPREDIDC_INVALID`	The `weighted_bipred_idc` field in PPS has an illegal value	`0x72`
Slice Header semantic Errors		
`H264D_ERR_SEM_SLCHDR_DELTAPICCNTBOT`	The `delta_pic_order_cnt_bottom` field in slice header has an incorrect value	`0x81`
`H264D_ERR_SEM_SLCHDR_PICPARAMSETID`	The `pic_parameter_set_id` field in slice header has an illegal value	`0x82`
`H264D_ERR_SEM_SLCHDR_SLCTYP`	Incorrect or unsupported slice type detected	`0x83`

Error codes	Description	Values
H264D_ERR_SEM_SLCHDR_FIRSTMBINSLC	The `first_mb_in_slice` field is greater than `PicSizeInMbs`	0x84
H264D_ERR_SEM_SLCHDR_IDRPICID	The `idr_pic_id` field in slice header has an illegal value	0x85
H264D_ERR_SEM_SLCHDR_REDUNDANTPICCNT	The field `redundant_pic_cnt` in slice header has an illegal value	0x86
H264D_ERR_SEM_SLCHDR_NUMREFIDXACTIVEL0	The `num_ref_idx_l0_active_minus1` decoded in slice header or obtained from PPS (based on `num_ref_idx_active_override_flag`) has an illegal value	0x87
H264D_ERR_SEM_SLCHDR_NUMREFIDXACTIVEL1	The `num_ref_idx_l1_active_minus1` decoded in slice header or obtained from PPS (based on `num_ref_idx_active_override_flag`) has an illegal value	0x88
H264D_ERR_SEM_SLCHDR_CABACINITIDC	The field `cabac_init_idc` in slice header has an illegal value	0x89
H264D_ERR_SEM_SLCHDR_SLCQSDELTA	The value of `slice_qs_delta+ pic_init_qs_minus` is out of bounds with what is specified by standard	0x8B
H264D_ERR_SEM_SLCHDR_DISABLEDEBLOCKFILTERIDC	The `disable_deblocking_filter_idc` field parsed in slice header has an illegal value	0x8C
H264D_ERR_SEM_SLCHDR_PICINVAR	This is set if any of the conditions governing syntax elements in slice headers (for multiple slices per picture) is not satisfied.	0x8D
H264D_ERR_SEM_SLCHDR_SLCALPHAC0OFFSET	The field `slice_alpha_c0_offset_div2` has a value out of bounds	0x8E
H264D_ERR_SEM_SLCHDR_SLCBETAOFFSET	The field `slice_beta_offest_div2` has a value out of bounds	0x8F
H264D_ERR_SEM_SLCHDR_NON_ZERO_FRAME_NUM_IN_IDR	The `frame_num` field has non-zero value in an IDR slice	0x90
H264D_ERR_SEM_SLCHDR_ILLEGAL_PRED_WEIGHT	Any of the variables associated with the computation of prediction weights has an illegal value	0x91
H264D_ERR_SEM_SLCHDR_UNSUPPORTED_LEVEL	Level specified in bitstream is greater than the supported level	0x93
H264D_ERR_SEM_SLCHDR_SPS_CHANGE_IN_NONIDR	Change of IDR detected in a non-IDR picture	0x94

Error codes	**Description**	**Values**
`H264D_ERR_SEM_SLCHDR_WAIT_SYNC_POINT`	Decoding is skipping NAL units till a valid sync point is found	`0x95`
`H264D_ERR_SEM_SLCHDR_SLC_GRP_CHNG_CYCLE`	Slice_group_change_cycle value parsed in the slice header is illegal. This is not an error. It is displayed as a warning only.	`0x96`
CAVLC semantic Errors		
`H264D_ERR_SEM_CAVLC_LEVEL_DECODE`	Error in CAVLD level decoding	`0xA1`
`H264D_ERR_SEM_CAVLC_CTOKEN_YY_AC`	Error in CTOKEN for luma AC coefficients	`0xA2`
`H264D_ERR_SEM_CAVLC_CTOKEN_YY_DC`	Error in CTOKEN for luma DC coefficients	`0xA3`
`H264D_ERR_SEM_CAVLC_CTOKEN_UV_AC`	Error in CTOKEN for chroma AC coefficients	`0xA4`
`H264D_ERR_SEM_CAVLC_CTOKEN_UV_DC`	Error in CTOKEN for chroma DC coefficients	`0xA5`
`H264D_ERR_SEM_CAVLC_LEVEL_YY_AC`	Error in level of luma AC coefficients	`0xA6`
`H264D_ERR_SEM_CAVLC_LEVEL_YY_DC`	Error in level of luma DC coefficients	`0xA7`
`H264D_ERR_SEM_CAVLC_LEVEL_UV_AC`	Error in level of chroma AC coefficients	`0xA8`
`H264D_ERR_SEM_CAVLC_LEVEL_UV_DC`	Error in level of chroma DC coefficients	`0xA9`
`H264D_ERR_SEM_CAVLC_TOTZERO_YY_AC`	Error in total zero value for luma AC coefficients	`0xAA`
`H264D_ERR_SEM_CAVLC_TOTZERO_YY_DC`	Error in total zero value for luma DC coefficients	`0xAB`
`H264D_ERR_SEM_CAVLC_TOTZERO_UV_AC`	Error in total zero value for chroma AC coefficients	`0xAC`
`H264D_ERR_SEM_CAVLC_TOTZERO_UV_DC`	Error in total zero value for chroma DC coefficients	`0xAD`
`H264D_ERR_SEM_CAVLC_RUNBEF_YY_AC`	Error in run before value for luma AC coefficients	`0xAE`
`H264D_ERR_SEM_CAVLC_RUNBEF_YY_DC`	Error in run before value for luma DC coefficients	`0xAF`
`H264D_ERR_SEM_CAVLC_RUNBEF_UV_AC`	Error in run before value for chroma AC coefficients	`0xB0`
`H264D_ERR_SEM_CAVLC_RUNBEF_UV_DC`	Error in run before value for chroma DC coefficients	`0xB1`
These error codes reports that the specific feature is not available in the decoder, or implementation specific errors.		

Error codes	Description	Values
H264D_ERR_IMPL_PPSUNAVAIL	The PPS referred to in the slice header is unavailable	0xC1
H264D_ERR_IMPL_SPSUNAVAIL	The SPS referred by the PPS id specified in slice header is unavailable	0xC2
H264D_ERR_IMPL_NOMEMORY	Memory insufficient to buffer MMCO commands or HRD CPB count is greater than available memory	0xC3
H264D_ERR_IMPL_CORRUPTED_BITSTREAM	Corruption in bit-stream	0xC5
H264D_ERR_IMPL_NOTSUPPORTED_REDUNTANT_PICTURE	Redundant picture not supported for this profile	0xCA
H264D_ERR_IMPL_NOTSUPPORTED_GAPSINFRAMENUM	Gaps in frame number or wrong frame number coded.	0xCD
H264D_ERR_IMPL_NOTSUPPORTED_ASOFMO	ASO/FMO not supported for this profile	0xCE
H264D_ERR_IMPL_INSUFFICIENT_DATA	Data insufficient to decode a picture	0xD1
H264D_ERR_XDM_API_BADPARAMETERS	NULL pointers or incorrect sizes of IVIDDEC2 structures	0xD2
H264D_ERR_XDM_API_BADNALU_PARSE_HEADER	Indicates decoder header is set to XDM_PARSE_HEADER for VCL NAL units (IDR and slice type). XDM_PARSE_HEADER should be set for non-VLD units like SPS, PPS, SEI, and so on, which occurs before start of a new frame and end of previous frame.	0xD3
Annex B and other semantic errors		
H264D_ERR_SEM_MBPRED_REFIDXL0	Decoded reference index exceeds the maximum ref_idx	0xE1
H264D_ERR_SEM_RPLR	Reference picture list reordering is executed more than the bound	0xE6
H264D_ERR_SEM_RPLR_PICNUMSIDC	Value of reordering_of_pic_nums_idc is out of bounds	0xE7
H264D_ERR_SEM_RPLR_ABSDIFFPICNUMMINUS1	Value of abs_diff_pic_num_minus1 is out of bounds	0xE8
H264D_ERR_SEM_MBLAYER_QPDELTA	Decoded MB_QP_Delta is out of bounds	0xEB
H264D_ERR_SEM_MBLAYER_MBTYPE	Decoding of MB_type had an error	0xEC
H264D_ERR_SEM_MBLAYER_CBP	Decoding of CBP had an error	0xED
H264D_ERR_SEM_SLCDATA_MBSKIPRUN	Value of mb_skip_run is out of bounds	0xEE

Error codes	Description	Values
`H264D_ERR_SEM_NOT_FRAME_MBS_ONLY`	Non-frame MBs are not supported at this level of the standard	`0xF1`
`H264D_ERR_SEM_ILLEGAL_INTRA_PRED_MODE`	Decoded value of the chroma intra prediction mode is out of bounds	`0xF4`
`H264D_ERR_SEM_ILLEGAL_VALUE_OCCURED_TERMINATE`	Indicates that `mb_mode` is illegal for the ref-idx decoding	`0xF5`
`H264D_TI_MB_NO_ERR`	Successful macro block decoding with out errors	`0x00`
`H264D_TI_MB_ERR_I`	Error in decoding a particular macro block within a I slice	`0x01`
`H264D_TI_MB_ERR_P`	Error in decoding a particular macro block within a P slice	`0x02`
`H264D_TI_MB_ERR_I_DP`	Error in decoding a particular macro block within a data partitioned I slice	`0x03`
`H264D_TI_MB_ERR_P_DP`	Error in decoding a particular macro block within a data partitioned P slice	`0x04`

Note:

Error code is not captured for illegal MV difference values parsed in the H.264 Decoder. The illegal MV difference, example, MV difference values, which are out of min/max bounds for a particular input stream resolution is taken care (clipped) during bounding box calculation for the reference region.

4.2 Data Structures

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

4.2.1 Common XDM Data Structures

This section includes the following common XDM data structures:

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

4.2.1.1 XDM_BufDesc

‖ Description

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

‖ Fields

Field	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 (example, it was filled by DMA or other hardware accelerator that does not write through the algorithm CPU), then bits in this mask should not be set.

4.2.1.5 XDM_AlgBufInfo

|| Description

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

|| Fields

Field	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 H.264 Baseline Profile Decoder, the buffer details are:

- Number of input buffer required is 1.
- Number of output buffer required is 1 for YUV 422ILE and 3 for YUV420P.
- 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 625SD format are:
  - For YUV 420:
    Y buffer = 720 * 576
    U buffer = 360 * 288
    V buffer = 360 * 288
  - For YUV 422ILE:
    Buffer = 720 * 576 * 2

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

4.2.1.6 IVIDEO1_BufDesc

|| Description

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

|| Fields

Field	Datatype	Input/ Output	Description
`numBufs`	`XDAS_Int32`	Output	Number of buffers
`frameWidth`	`XDAS_Int32`	Output	Width of the video frame
`frameHeight`	`XDAS_Int32`	Output	Height of the video frame
`framePitch`	`XDAS_Int32`	Output	Frame pitch used to store the frame
`bufDesc[IVIDEO_MAX_YUV_BUFFERS]`	`XDM1_SingleBufDesc`	Output	Pointer to the vector containing buffer addresses
`extendedError`	`XDAS_Int32`	Output	Extended Error Field
`frameType`	`XDAS_Int32`	Output	Video frame types. See `IVIDEO_FrameType` enumeration for details.
`topFieldFirstFlag`	`XDAS_Int32`	Output	Flag to indicate when the application should display the top field first.
`repeatFirstFieldFlag`	`XDAS_Int32`	Output	Flag to indicate when the first field should be repeated. Default value is 0. Not supported in this version of H264 Decoder.
`frameStatus`	`XDAS_Int32`	Output	Video output buffer status. See `IVIDEO_OutputFrameStatus` enumeration for details.
`repeatFrame`	`XDAS_Int32`	Output	Number of times the display process needs to repeat the displayed progressive frame. Default value is 0. Not supported in this version of H264 Decoder.
`contentType`	`XDAS_Int32`	Output	Content type of the buffer. See `IVIDEO_ContentType` enumeration for details.
`chromaFormat`	`XDAS_Int32`	Output	Chroma formats. See `XDM_ChromaFormat` for details.

Note:

`IVIDEO_MAX_YUV_BUFFERS:`

- ❑ Maximum YUV buffers - one each for Y, U, and V.

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. Decoder uses internal default values if the data structure is set to NULL.

|| Fields

Field	Datatype	Input/ Output	Description
`size`	`XDAS_Int32`	Input	Size of the basic or extended (if being used) data structure in bytes. Default value: size of `(IH264VDEC_Params)`
`maxHeight`	`XDAS_Int32`	Input	Maximum video height to be supported in pixels Default value: 576
`maxWidth`	`XDAS_Int32`	Input	Maximum video width to be supported in pixels Default value: 720
`maxFrameRate`	`XDAS_Int32`	Input	Maximum frame rate in fps * 1000 to be supported. Default value: 0

Field	**Datatype**	**Input/ Output**	**Description**
`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 value: 0
`dataEndianness`	`XDAS_Int32`	Input	Endianness of input data. See `XDM_DataFormat` enumeration for details. Default value: `XDM_BYTE`
`forceChromaFormat`	`XDAS_Int32`	Input	Sets the output to the specified format. For example, if the output should be in YUV 4:2:2 interleaved (little endian) format, set this field to `XDM_YUV_422ILE`. See `XDM_ChromaFormat` enumeration for details. Default value: `XDM_YUV_420P`

Note:

- H.264 Decoder does not use the `maxFrameRate` and `maxBitRate` fields for creating the algorithm instance.
- Supports frame width of the range of 32 to 720 pixels
- Supports all resolutions up to D1 (PAL and NTSC), CIF, QCIF and so on.
- `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. Call the control API and use `IH264VDEC_SETDEFAULT` command if you are not sure of the values to be specified for these parameters. The decoder uses internal default values, if the data structure is set to NULL.

|| Fields

Field	**Datatype**	**Input/ Output**	**Description**
`size`	`XDAS_Int32`	Input	Size of the basic or extended (if being used) data structure in bytes. Default value: size of `(IH264VDEC_DynamicParams)`
`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 ❑ Defualt value: `XDM_DECODE_AU`

Field	Datatype	Input/ Output	Description
`displayWidth`	`XDAS_Int32`	Input	If the field is set to: ❑ 0 - Uses decoded image width as pitch ❑ If any other value greater than the decoded image width is given, then this value in pixels is used as pitch. ❑ Default value: 0
`frameSkipMode`	`XDAS_Int32`	Input	Frame skip mode. See `IVIDEO_FrameSkip` enumeration for details. Default value: `IVIDEO_NO_SKIP`
`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 in 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:

- ❑ Application can retrieve decoded information of the non VCL (Video Coded Layer) NAL units, appearing before the start of new frame and at the end of previous frames (like SPS, PPS, SEI, and so on), by setting dynamic param `decodeHeader` to `XDM_PARSE_HEADER` before making process call. The `decodeHeader` set to `XDM_PARSE_HEADER` for VCL NAL units is considered as bad input parameter by decoder. This restriction is because decoder operates in frame base mode only and decodes entire frame once slice type NAL is encountered.
- ❑ If the application requires the decoder to skip decoding of non-reference frames, then the `frameSkipMode` field has to be set to `IVIDEO_SKIP_B`. (see Section 4.1 for details.)
- ❑ If `displayWidth` is non-zero, then it has to be an even number.
- ❑ If the specified `displayWidth` is less than the `imagewidth`, it is still considered and image is written at a resolution equal to display width.
- ❑ If the `displayWidth` is set to 0 and frame cropping parameters are present in the bit-stream, then the cropped image width is taken as the pitch.
- ❑ `frameOrder` is a set to the enum value of `IVIDDEC2_DISPLAY_ORDER`.
- ❑ H264 Decoder does not support `newFrameFlag` and `mbDataFlag` in

this version. Their values are set as zero.

- ❑ MbDataFlag, is not supported in this version of H264 Decoder and is set to default value 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:

- ❑ NumBytes should be greater then or equal to 4 bytes else decoder will return error type as H264D_ERR_IMPL_INSUFFICIENT_DATA.
- ❑ H.264 Decoder copies the inputID value to the outputID value of IVIDDEC2_OutArgs structure, when maxDisplayDelay of IH264VDEC_Params extended input params structure is set to 0.

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.

Field	Datatype	Input/ Output	Description
data	XDM_SingleBufDesc	Output	Buffer information structure for information passing buffer.
maxNumDisplayBufs	XDAS_Int32	Output	The maximum number of buffers required by the 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:

- If cropping of pixels is specified in the bit stream, then the `outputHeight` and `outputWidth` returned is equal to the cropped image size. `outputWidth` returned is independent of the `display width`, given in the `DynamicParams`.
- Algorithm sets the `frameRate` and `bitRate` fields to zero.
- `contentType` is always returned as `IVIDEO_PROGRESSIVE`.

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

Field	**Datatype**	**Input/ Output**	**Description**
			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 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 chooses 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:

- ❑ With frame reordering, the display order is independent of decode order. When the algorithm is ready for display it copies the `inputID` value of a given decoded frame to the `outputID` value of `IVIDDEC2_OutArgs` structure. The algorithm sets `displayBufs` pointers accordingly.
- ❑ When there is no frame ready to be displayed after a given decode call, the first pointer of `displayBufs` structure is set to NULL.
- ❑ To support frame-reordering, delay is present between decoding of a frame and its display. This delay amount is configurable depending on the application scenario. The delay needs to be specified in

maxDisplayDelay (element of IH264VDEC_InArgs).

- ❑ The first frame to be displayed is returned after first N+1 frames are decoded by the decoder (N is the configured delay). Hence N buffers are locked within the decoder. However if the maxDisplayDelay specified by the client is more than what is actually required for decoding of that stream (this is calculated by the decoder looking at the level and frame resolution), the decoder will lock only the required number of frames within.
- ❑ Due to reordering of frames allowed in H264 standard, the delay requirement can be in the range 5 -16 (depending on the resolution of the image).
- ❑ Based on the application scenario, this delay should be configured. However, for most of the used case scenarios, a delay of 0 frame suffices.

4.2.2 H.264 Decoder Data Structures

This section includes the following H.264 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 the H.264 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.
InputStreamFormat	XDAS_Int32	Input	Indicates the type of bit-stream the application gives to the algorithm. ❑ 0 - ByteStreamFormat, ❑ 1 - NAL unit format ❑ Default value is 0.

Field	Datatype	Input/ Output	Description
`maxDisplayDelay`	`XDAS_Int32`	Input	This is the maximum number of frames processed by the decoder without displaying the output. It is required when there is frame re-ordering. The application should allocate sufficient output buffers (equal to `maxDisplayDelay`+1), if this feature is used. For bit-streams that do not use frame delay, a value of 0 is sufficient (minimum memory allocation by client). See section 4.2.1.12 (`IVIDDEC2_OutArgs`) for details. Default value: 0

4.2.2.2 IH264VDEC_DynamicParams

‖ Description

This structure defines the run-time parameters and any other implementation specific parameters for the H.264 Decoder 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.
`mbErrorBufFlag`	`XDAS_Int32`	Input	If the application needs the error status for each MB, then this flag should be set to 1. Otherwise, it should be set to 0.
`mbErrorBufSize`	`XDAS_UInt32`	Input	Size of the buffer where MB error status will be stored by the decoder.
`Sei_Vui_parse_flag`	`XDAS_Int32`	Input	If the application is interested in SEI or VUI information, then this needs to be set to 1. Otherwise, this needs to be set to 0.
`numNALunits`	`XDAS_Int32`	Input	It specifies the number of NAL units the application gives to the algorithm if the `InputStreamFormat` is NAL unit Stream.

Note:

- If the application wants to get the decoded MB error status, the application should turn ON the `mbErrorBufFlag` and pass the buffer pointer to the decoder algorithm through `outputBufDesc.bufs[3]` when the output chroma format is YUV420 (0, 1 and 2 locations are used for YUV buffers) and through `outputBufDesc.bufs[1]` when the output chroma format is YUV422. This is mentioned in the sample application for reference. Lib will dump the required data. Access mask setting is not possible. Hence, the application needs to take care the cache clean up for every call.
- The pointer to an array that specifies the length of each NAL unit for the NAL stream and is passed to the decoder algorithm through `inputBufDesc.descs[1].buf`, the buffer is of type `XDAS_Int32`. The algorithm must decode during a single call. For the byte stream format, `numNALunits` is set (default value) zero.
- To get SEI and VUI information, application should set the flag `Sei_Vui_parse_flag and` pass the structure pointer to the decoder algorithm through `outputBufDesc.bufs[4]` when the output chroma format is YUV420 and through `outputBufDesc.bufs[2]` when the output chroma format is YUV422. This is mentioned in the sample application. Access mask setting is not possible. Hence, application needs to take care the cache clean up for every call.
- All these are implemented in the sample test application.
- For a particular MB, if the value returned by the decoder in `outputBufDesc.bufs[3]` or in `outputBufDesc.bufs[1]` buffer based on the chroma format is equal to that defined in `H264D_TI_MB_NO_ERR`, then the MB has no error. However, if any other value is observed, then the MB is errorneous.

4.2.2.3 IH264VDEC_InArgs

|| Description

This structure defines the run-time input arguments for the H.264 Decoder instance object.

|| Fields

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

4.2.2.3.1 sSeiVuiParams_t

‖ ‖ Description

This structure defines Supplemental Enhancement Information (SEI) messages and parameters that describe the values of various Video Usability parameters (VUI).

‖ ‖ Fields

Field	Datatype	Input/ Output	Description
`parsed_flag`	`unsigned int`	Output	❑ 1 - Indicates that in the current process call, contents of the structure is updated ❑ 0 - Indicates contents of the structure is not updated
`vui_params`	`sVSP_t`	Output	Video Usability Information
`sei_messages`	`sSeiMessages_t`	Output	Supplemental Enhancement Information

Note:

A brief description of SEI and VUI contents are given below. For details see H.264 standard (*ISO/IEC 14496-10:2005 (E) Rec.- Information technology – Coding of audio-visual objects – H.264 (E) ITU-T Recommendation.)*

*4.2.2.3.2 sSeiMessages*_t

‖ Description

Structure containing Supplemental Enhancement Information messages.

‖ Fields

Field	Datatype	Input/ Output	Description
`parsed_flag`	`unsigned int`	Output	❑ 1 - Indicates that in the current process call, contents of the structure is updated ❑ 0 - Indicates contents of the structure is not updated
`frame_freeze_repetition`	`sFullFrameFreezeRepetition_t`	Output	Specifies the persistence of the full-frame freeze SEI message, and may specify a picture order count interval within which another full-frame freeze SEI message or a full-frame freeze release SEI or the end of the coded video sequence is present in the bit stream.

Field	Datatype	Input/ Output	Description
`frame_freeze_release`	`sFullFrameFreezeRelease_t`	Output	Cancels the effect of any full-frame freeze SEI message sent with pictures that precede the current picture in output order.
`prog_refine_start`	`sProgRefineStart_t`	Output	Specifies the beginning of a set of consecutive coded pictures that is labeled as the current picture followed by a sequence of one or more pictures of refinement of the quality of the current picture, rather than as a representation of a continually moving scene.
`prog_refine_end`	`sProgRefineEnd_t`	Output	Specifies end of progressive refinement.
`recovery_pt_info`	`sRecoveryPointInfo_t`	Output	The recovery point SEI message assists a decoder in determining when the decoding process will produce acceptable pictures for display after the decoder initiates random access or after the encoder indicates a broken link in the sequence.
`pic_timing`	`sPictureTiming_t`	Output	Specifies timing information regarding CPB delays, DPB output delay and so on.

4.2.2.3.3 sFullFrameFreezeRepetition_t

|| Description

Structure contains information regarding frame freeze.

|| Fields

Field	Datatype	Input/ Output	Description
`parsed_flag`	`unsigned int`	Output	❑ 1 - Indicates that in the current process call, contents of the structure is updated ❑ 0 - Indicates contents of the structure is not updated
`full_frame_freeze_repetition_period`	`unsigned int`	Output	Specifies the persistence of the full-frame freeze SEI message

4.2.2.3.4 sFullFrameFreezeRelease_t

‖ Description

Structure contains information regarding frame freeze.

‖ Fields

Field	Datatype	Input/ Output	Description
`parsed_flag`	`unsigned int`	Output	❑ 1 - Indicates that in the current process call, contents of the structure is updated ❑ 0 - Indicates contents of the structure is not updated
`full_frame_freeze_ release_flag`	`unsigned char`	Output	Cancels the effect of any full-frame freeze SEI message sent with pictures that precede the current picture in output order.

4.2.2.3.5 sProgRefineStart_t

‖ Description

Structure contains information regarding progressive refinement.

‖ Fields

Field	Datatype	Input/ Output	Description
`parsed_flag`	`unsigned int`	Output	❑ 1 - Indicates that in the current process call, contents of the structure is updated ❑ 0 - Indicates contents of the structure is not updated
`progressive_refinem ent_id`	`unsigned int`	Output	Specifies an identification number for the progressive refinement operation.
`num_refinement_step s_minus1`	`unsigned int`	Output	Specifies the number of reference frames in the tagged set of consecutive coded pictures

4.2.2.3.6 sProgRefineEnd_t

‖ Description

Structure contains information regarding progressive refinement.

‖ Fields

Field	Datatype	Input/ Output	Description
`parsed_flag`	`unsigned int`	Output	❑ 1 - Indicates that in the current process call, contents of the structure is updated ❑ 0 - Indicates contents of the structure is not updated
`progressive_ refinement_id`	`unsigned int`	Output	Specifies an identification number for the progressive refinement operation.

4.2.2.3.7 sRecoveryPointInfo_t

‖ Description

Structure contains information regarding recovery points.

‖ Fields

Field	Datatype	Input/ Output	Description
`parsed_flag`	`unsigned int`	Output	❑ 1 - Indicates that in the current process call, contents of the structure is updated ❑ 0 - Indicates contents of the structure is not updated
`recovery_frame_cnt`	`unsigned int`	Output	Specifies the recovery point of output pictures in output order.
`exact_match_flag`	`unsigned char`	Output	Indicates whether decoded pictures subsequent to the specified recovery point in output order. Derived by starting the decoding process at the access unit associated with the recovery point . SEI message will be an exact match to the pictures that would be produced by starting the decoding process at the location of a previous IDR access unit in the NAL unit stream.
`broken_link_flag`	`unsigned char`	Output	Indicates broken link in the NAL unit stream
`changing_slice_grou p_idc`	`unsigned char`	Output	Indicates if decoded pictures are correct or approximately correct in content. Subsequent to the recovery point in output order when all macroblocks of the primary coded pictures are decoded within the changing slice group period.

4.2.2.3.8 sPictureTiming_t

|| Description

Structure contains timing information such as DPB delay and CPD delay.

|| Fields

Field	**Datatype**	**Input/ Output**	**Description**
`parsed_flag`	`unsigned int`	Output	❑ 1 - Indicates that in the current process call, contents of the structure is updated ❑ 0 - Indicates contents of the structure is not updated
`cpb_removal_delay`	`unsigned int`	Output	Specifies how many clock ticks to wait after removal from the CPB of the access unit associated with the most recent buffering period SEI message before removing from the buffer the access unit data associated with the picture timing SEI message.
`dpb_output_delay`	`unsigned int`	Output	Used to compute the DPB output time of the picture.
`pic_struct`	`unsigned int`	Output	Indicates whether a picture should be displayed as a frame or field
`clock_timestamp_flag`	`unsigned int`	Output	❑ 1 - Indicates number of clock timestamp syntax elements present and follow immediately ❑ 0 - Indicates associated clock timestamp syntax elements not present
`ct_type`	`unsigned int`	Output	Indicates the scan type(interlaced or progressive) of the source material
`nuit_field_based_flag`	`unsigned int`	Output	Used to calculate the `clockTimestamp`
`counting_type`	`unsigned int`	Output	Specifies the method of dropping values of n_frames
`full_timestamp_flag`	`unsigned int`	Output	❑ 1 - Specifies that the n_frames syntax element is followed by seconds_value, minutes_value, and hours_value. ❑ 0 - Specifies that the n_frames syntax element is followed by seconds_flag
`discontinuity_flag`	`unsigned int`	Output	Indicates whether the difference between the current value of `clockTimestamp` and the value of `clockTimestamp` computed from the previous `clockTimestamp` in output order can be interpreted as the time difference between the times of origin or capture of the associated frames or fields.
`cnt_dropped_f`	`unsigned int`	Output	Specifies the skipping of one or more values

Field	Datatype	Input/ Output	Description
lag			of n_frames using the counting method
n_frames	unsigned int	Output	Specifies the value of nFrames used to compute clockTimestamp.
seconds_value	unsigned int	Output	Specifies the value of sS used to compute clockTimestamp.
minutes_value	unsigned int	Output	Specifies the value of mM used to compute clockTimestamp.
hours_value	unsigned int	Output	Specifies the value of hH used to compute clockTimestamp.
time_offset	unsigned int	Output	Specifies the value of offset used to compute clockTimestamp

4.2.2.3.9 sVSP_t

|| Description

This structure defines parameters that describe the values of various video usability parameters that come as a part of Sequence Parameter Set in the bit-stream.

|| Fields

Field	Datatype	Input/ Output	Description
parsed_flag	unsigned int	Output	❑ 1 - Indicates that in the current process call, contents of the structure is updated ❑ 0 - Indicates contents of the structure is not updated
aspect_ratio_info_present_flag	unsigned int	Output	Indicates whether aspect ratio idc is present or not.
aspect_ratio_idc	unsigned int	Output	Aspect ratio of Luma samples
sar_width	unsigned int	Output	Horizontal size of sample aspect ratio
sar_height	unsigned int	Output	Vertical size of sample aspect ratio
overscan_info_present_flag	unsigned int	Output	Overscan appropriate flag
overscan_appropriate_flag	unsigned int	Output	Cropped decoded pictures are suitable for display or not.
video_signal_type_present_flag	unsigned int	Output	This flag tells whether video_format, video_full_range_flag and colour_description_present_flag are present or not

Field	Datatype	Input/ Output	Description
`video_format`	`unsigned int`	Output	Video format indexed by a table. For example, PAL/NTSC
`video_full_range_flag`	`unsigned int`	Output	Black level, luma and chroma ranges. It should be used for BT.601 compliance
`colour_description_present_flag`	`unsigned int`	Output	Indicates whether `colour_primaries`, `transfer_characteristics` and `matrix_coefficients` are present.
`colour_primaries`	`unsigned int`	Output	Chromaticity co-ordinates of source primaries
`transfer_characteristics`	`unsigned int`	Output	Opto-electronic transfer characteristics of the source picture
`matrix_coefficients`	`unsigned int`	Output	Matrix coefficients for deriving Luma and chroma data from RGB components.
`chroma_location_info_present_flag`	`unsigned int`	Output	This flag tells whether `chroma_sample_loc_type_top field` and `chroma_sample_loctype bottom_field` are present.
`chroma_sample_loc_type_top_field`	`unsigned int`	Output	Location of `chroma_sample` top field
`chroma_sample_loc_type_bottom_field`	`unsigned int`	Output	Location of `chroma_sample` bottom field
`timing_info_present_flag`	`unsigned int`	Output	Indicates whether `num_units_in_tick, time_scale, and fixed_frame_rate_flag` are present.
`num_units_in_tick`	`unsigned int`	Output	Number of units of a clock that corresponds to 1 increment of a clock tick counter
`time_scale`	`unsigned int`	Output	Indicates actual increase in time for 1 increment of a clock tick counter
`fixed_frame_rate_flag`	`unsigned int`	Output	Indicates how the temporal distance between HRD output times of any two output pictures is constrained
`nal_hrd_parameters_present_flag`	`unsigned int`	Output	Indicates whether `nal_hrd_parameters` are present
`nal_hrd_parameters`	`sHrdParm_t`	Output	See `sHrdParm_t datastructure` for details.
`vcl_hrd_parameters_present_flag`	`unsigned int`	Output	Indicates whether `vcl_hrd_parameters` are present
`vcl_hrd_parameters`	`sHrdParm_t`	Output	See `sHrdParm_t datastructure` for details.

Field	Datatype	Input/ Output	Description
`low_delay_hrd_flag`	`unsigned int`	Output	HRD operational mode as in Annex C of the standard
`pic_struct_present_flag`	`unsigned int`	Output	Indicates whether picture timing SEI messages are present
`bitstream_restriction_flag`	`unsigned int`	Output	Indicates if the bit-stream restriction parameters are present
`motion_vectors_over_pic_boundaries_flag`	`unsigned int`	Output	Specifies whether motion vectors can point to regions outside the picture boundaries.
`max_bytes_per_pic_denom`	`unsigned int`	Output	Maximum number of bytes not exceeded by the sum of sizes of all VCL NAL units of a single coded picture
`max_bits_per_mb_denom`	`unsigned int`	Output	Maximum number of bits taken by any coded MB
`log2_max_mv_length_vertical`	`unsigned int`	Output	Maximum value of any motion vector's vertical component
`log2_max_mv_length_horizontal`	`unsigned int`	Output	Maximum value of any motion vector's horizontal component
`num_reorder_frames`	`unsigned int`	Output	Maximum number of frames that need to be re-ordered
`max_dec_frame_buffering`	`unsigned int`	Output	Size of HRD decoded buffer (DPB) in terms of frame buffers.

4.2.2.3.10 sHrdParm_t

|| Description

This structure defines the HRD parameters that come in a H264 bit-stream as a part of video usability Information.

|| Fields

Field	Datatype	Input/ Output	Description
`cpb_cnt`	`unsigned int`	Output	Number of alternative CPB specifications in the bit stream
`bit_rate_scale`	`unsigned int`	Output	Together with `bit_rate_value[i]`, it specifies the maximum input bitrate for the ith CPB.
`cpb_size_scale`	`unsigned int`	Output	Together with `cpb_size_value[i]`, specifies the maximum CPB size for the ith CPB.

Field	Datatype	Input/ Output	Description
bit_rate_value[i]	unsigned int	Output	Maximum input bitrate for the ith CPB
cpb_size_value[i]	unsigned int	Output	Maximum CPB size for the ith CPB
vbr_cbr_flag[i]	unsigned int	Output	Specifies the ith CPB is operated in Constant Bit-rate mode or variable bit-rate mode
initial_cpb_removal_delay_length_minus1	unsigned int	Output	Length in bits of initial_cpb_removal_length syntax element
cpb_removal_delay_length_minus1	unsigned int	Output	Length in bits of cpb_removal_delay_length syntax element
dpb_output_delay_length_minus1	unsigned int	Output	Length in bits of dpb_output_delay_length syntax element
time_offset_length	unsigned int	Output	Length in bits of time_offset syntax element

Note:

SEI / VUI parsing is handled by the decoder as follows:

If the application is interested in SEI / VUI, then the Sei_Vui_parse_flag (element of IH264VDEC_InArgs) needs to be set to one and the buffer (structure) pointer needs to be passed in seiVui_buffer_ptr (element of IH264VDEC_InArgs). When the sei_Vui_parse_flag is set to 1, the decoder parses the SEI / VUI information and updates the buffer allotted by the application.
A flag, parsed_flag, is present as the first element of structure of every SEI message, VUI structure and the SEI_VUI structure. This flag when set to one by the decoder indicates that in the current process call, contents of this structure was updated. The pointer of the buffer is copied to the pointer in the IH264VDEC_OutArgs.

Currently, parsing of the following SEI messages are supported:

- Full-frame freeze SEI message
- Full-frame freeze release
- Progressive refinement segment start
- Progressive refinement segment end
- Recovery point SEI message
- Picture timing SEI message

Other types of SEI messages will be skipped by the decoder.

4.2.2.4 IH264VDEC_Status

|| Description

This structure defines parameters that describe the status of the H.264 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
`profile`	`eH264VDEC_Profile`	Output	Profile of the bit stream. The H.264 decoder supports only baseline profile.
`level`	`eLevelNum_t`	Output	Level number of the bit stream. The H.264 decoder supports only upto Level 3.
`Qp`	`XDAS_Int32`	Output	Frame quantization parameter
`last_decoded_mb_addr`	`XDAS_UInt32`	Output	This is the address in Raster scan order of the last macroblock decoded in the frame.
`slice_header_frame_num`	`XDAS_UInt32`	Output	This is the frame number derived from slice headers in a given frame.
`full_frame_decoded`	`XDAS_UInt32`	Output	The flag indicates whether the full frame is decoded without any errors.
`poc_num`	`XDAS_Int32`	Output	This is the POC number of a given frame.
`display_frame_skip_flag`	`XDAS_Int32`	Output	This flag, when set to one indicates that the frame returned in this call was skipped and hence nothing was written into this buffer.
`numNALdecoded`	`XDAS_Int32`	Output	Number of NAL units decoded by the algorithm during a single call of the algorithm. It is equal to the number of NAL units given by the application or the number of NAL units required to complete a frame decoding.

Note:

Following is the decoder behavior for supporting frame size being non-multiple of 16 through frame cropping:

- The decoder populates the output buffers at a resolution equal to the size of the cropped image. Also, it returns status parameters for picture resolution (`outputHeight` and `outputWidth`) as equal to the cropped values.
- If the `displayWidth` (element in `DynamicParams`) is lesser than the cropped image width, then the decoder writes at a width equal to the display width.

Following is the behavior of the decoder to handle skipping of non-reference frames.

- If the application needs the decoder to skip non-reference frames, then it has to set `frameSkipMode` (element in `dynamicParams`) equal to `IVIDEO_SKIP_B` and call the control API with `XDM_SETPARAMS` option.
- Decoder skips decoding only when `frameSkipMode` is set to `IVIDEO_SKIP_B` and the current frame is not referenced in the future. The buffer allotted during frame skip mode will be locked inside the decoder irrespective of whether the frame was actually skipped or not.
- When the buffer pointer of skipped frame is returned by the decoder, the `display_frame_skip_flag` (element in `IH264VDEC_OutArgs`) will be set to one indicating that nothing was written into this buffer.
- In order to come out of the frame skip mode, the application has to set `frameSkipMode` (element in `dynamicParams`) equal to `IVIDEO_NO_SKIP` and call the control API with `XDM_SETPARAMS` option.
- **Important:**
  In skip mode, decoder skips non-reference frames even if it is a P frame, as the H264 standard allows P frames to be non-reference frames.

4.2.2.5 IH264VDEC_OutArgs

‖ Description

This structure defines the run-time output arguments for the H.264 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 (APIs) used in the H.264 Decoder. The APIs are logically grouped into the following categories:

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

You must call these APIs in the following sequence:

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

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

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

4.3.1 Creation APIs

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

‖ Name

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

‖ Synopsis

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

‖ Arguments

```
Void
```

‖ Return Value

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

‖ Description

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

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

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

‖ See Also

`algAlloc()`

|| Name

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

|| Synopsis

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

|| Arguments

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

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

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

|| Return Value

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

|| Description

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

The first argument to `algAlloc()` is a pointer to a structure that defines the creation parameters. This pointer may be `NULL;` however, in this case, `algAlloc()` must assume default creation parameters and must not fail.

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

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

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

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

|| See Also

`algNumAlloc(), algFree()`

4.3.2 Initialization API

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

‖ Name

`algInit()` – initialize an algorithm instance

‖ Synopsis

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

‖ Arguments

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

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

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

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

‖ Return Value

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

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

‖ Description

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

The first argument to `algInit()` is a handle to an algorithm instance. This value is initialized to the base field of `memTab[0]`.

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

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

The last argument is a pointer to a structure that defines the algorithm initialization parameters.

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

‖ See Also

`algAlloc(), algMoved()`

4.3.3 Control API

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

|| Name

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

|| Synopsis

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

|| Arguments

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

IVIDDEC2_Cmd id; /* algorithm specific control commands*/

IVIDDEC2_DynamicParams *params /* algorithm run time
parameters */

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

|| Return Value

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

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

|| Description

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

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

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

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

Note:

If you are using extended data structures, the third and fourth arguments must be pointers to the extended `DynamicParams` and `Status` data structures respectively. Also, ensure that the `size` field is set to the size of the extended data structure. Depending on the value set for the `size` field, the algorithm uses either basic or extended parameters.

‖ Preconditions

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

- `control()` can only be called after a successful return from `algInit()` and `algActivate()`.
- If algorithm uses DMA resources, `control()` can only be called after a successful return from `DMAN3_init()`.
- `handle` must be a valid handle for the algorithm's instance object.

‖ Postconditions

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

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

‖ Example

See test application file, TestAppDecoder.c available in the \Client\Test\Src sub-directory.

‖ See Also

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

4.3.4 Data Processing API

Data processing API is used for processing the input data.

‖ Name

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

‖ Synopsis

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

‖ Arguments

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

‖ Return Value

```
Void
```

‖ Description

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

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

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

‖ See Also

`algDeactivate()`

|| Name

`process()` – basic encoding/decoding call

|| Synopsis

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

|| Arguments

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

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

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

IVIDDEC2_InArgs *inargs /* algorithm runtime input
arguments */

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

|| Return Value

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

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

|| Description

This function does the basic encoding/decoding. The first argument to `process()` is a handle to an algorithm instance.

The second and third arguments are pointers to the input and output buffer descriptor data structures respectively (see `XDM_BufDesc` data structure for details).

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

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

Note:

If you are using extended data structures, the fourth and fifth arguments must be pointers to the extended `InArgs` and `OutArgs` data structures respectively. Also, ensure that the `size` field is set to the size of the extended data structure. Depending on the value set for the `size` field, the algorithm uses either basic or extended parameters.

|| Preconditions

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

- `process()` can only be called after a successful return from `algInit()` and `algActivate()`.

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

‖ Postconditions

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

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

‖ Example

See test application file, TestAppDecoder.c available in the \Client\Test\Src sub-directory.

‖ See Also

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

Note:

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

‖ Name

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

‖ Synopsis

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

‖ Arguments

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

‖ Return Value

```
Void
```

‖ Description

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

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

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

‖ See Also

`algActivate()`

4.3.5 Termination API

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

|| Name

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

|| Synopsis

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

|| Arguments

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

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

|| Return Value

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

|| Description

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

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

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

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

|| See Also

`algAlloc()`

4.4 Error Handling

This section describes the errors in the bit stream, the expected behavior of the decoder, and the recommended actions on the application side.

- ❑ When the decoder detects an error in the bit stream, the return value from the process call will be `IALG_EFAIL`.
- ❑ The type of the detected error will be indicated in the `extendedError` field of `OutArgs`. See `XDM_ErrorBit` enumeration for details.
- ❑ In any type of error scenario, there is no need for the application to reset the decoder.
- ❑ Ouput/display buffer handling in error scenarios:
  - o If the `maxDisplayDelay` is zero, then the decoder always returns back the same display buffer passed by the system in the current process call
  - o If the `maxDisplayDelay` is greater than zero, then the decoder returns display buffers in an order based on the display order logic specified by the standard. However, if the display order logic cannot be executed due to error `XDM_CORRUPTEDHEADER`, then decoder returns the display buffer given by the system in the current process call.
- ❑ When the error type is `XDM_CORRUPTEDHEADER`, the output height and width information present in the `OutArgs` or `Status` structure might not be reliable.

Note:

- ❑ In certain scenarios, the decoder returns a non-zero `extendedError`, with the process call returning `IALG_EOK`. This happens, if decoder detects errors in the bit stream, which do not obstruct the further decoding and reconstruction. For example, an error detected during parsing of a PPS, which is never referenced, will be reported in the `extendedError`, but the process call will still return `IALG_EOK`.
- ❑ The decoder attempts to return a non-`NULL` pointer for display in all scenarios, when the initial pipe-up for `maxDisplayDelay` is completed.

Appendix A

Revision History

This user guide revision history highlights the changes made to the SPRUEA1B codec specific user guide to make it SPRUEA1C.

Table A-1. Revision History for H264 Decoder on C64x+

Section	Additions/Modifications/Deletions
Global Changes	❑ Modified DSP/BIOS version to 5.32.02 ❑ Modified Code Generaion Tools version to 6.0.8 ❑ Modified Framework Component version to 2.20.00.15 ❑ Updated IVIDDEC to IVIDDEC2
Section 1.3	Supported Services and Features: ❑ Updated XDM version ❑ Added the following: o Supports Error resiliency o Supports all resolutions up to D1 (PAL and NTSC) including CIF and QCIF ❑ Deleted XDAIS compliant
Section 2.1.1	Hardware: ❑ Added supported platforms
Section 2.2	Installing the Component: ❑ Updated top-level directory name
Section 2.8	Evaluation Version: ❑ Added Note
Section 3.2	Added section Frame Buffer Management by Application
Section 3.3	Added section Sample Test Application
Section 4.1	Symbolic Constants and Enumerated Data Types: ❑ Modified Description of Symbolic Constants and Enumerated Data Types

Section	Additions/Modifications/Deletions
Table 4-1	List of Enumerated Data Types: ❑ Added the following Group or Enumeration Class: o `IVIDEO_OutputFrameStatus` o `XDM_DecMode` o `XDM_AccessMode` ❑ Updated the Symbolic Constant Names and description for the following Group or Enumeration Class o `IVIDEO_FrameType` o `IVIDEO_ContentType` o `IVIDEO_FrameSkip` o `XDM_DataFormat` o `XDM_ChromaFormat` o `XDM_CmdId` o `XDM_ErrorBit`
Table 4-2	Added the table H264 Decoder Enumerated Data Types
Table 4-3	Error Codes and Values: Added the following new error codes: ❑ `H264D_ERR_SEM_NALU_STARTCODEPREFIX` ❑ `H264D_ERR_SEM_SPS_INVLD_LEVEL` ❑ `H264D_ERR_SEM_SPS_UNSUPPORTEDPICHEIGHT` ❑ `H264D_ERR_SEM_SLCHDR_SLC_GRP_CHNG_CYCLE` ❑ `H264D_ERR_IMPL_NOTSUPPORTED_GAPSINFRAMENUM` ❑ `H264D_ERR_XDM_API_BADPARAMETERS` ❑ `H264D_ERR_XDM_API_BADNALU_PARSE_HEADER` ❑ `H264D_TI_MB_NO_ERR` ❑ `H264D_TI_MB_ERR_I` ❑ `H264D_TI_MB_ERR_P` ❑ `H264D_TI_MB_ERR_I_DP` ❑ `H264D_TI_MB_ERR_P_DP`
Section 4.2.1	Added the following Common XDM Data Structures: ❑ `XDM1_BufDesc` ❑ `XDM_SingleBufDesc` ❑ `XDM1_SingleBufDesc` ❑ `IVIDEO1_BufDesc`
Section 4.2.1.8	`IVIDDEC2_Params`: ❑ Added default values for each field ❑ Updated the note
Section 4.2.1.9	`IVIDDEC2_DynamicParams`: ❑ Updated description ❑ Added default values for each field ❑ Added the following fields: o `frameOrder` o `newFrameFlag` o `mbDataFlag` ❑ Updated the note
Section 4.2.1.10	`IVIDDEC2_InArgs`: ❑ Updated the note

Section	**Additions/Modifications/Deletions**
Section 4.2.1.11	`IVIDDEC2_Status`:
	Added the following fields:
	❑ `data`
	❑ `maxNumDisplayBufs`
	Updated the note
Section 4.2.1.12	`IVIDDEC2_OutArgs`:
	Added the following fields:
	❑ `outputID[XDM_MAX_IO_BUFFERS]`
	❑ `decodedBufs`
	❑ `displayBufs[XDM_MAX_IO_BUFFERS]`
	❑ `outputMbDataId`
	❑ `mbDataBuf`
	❑ `freeBufID[IVIDDEC2_FREE_BUFF_SIZE]`
	❑ `outBufsInUseFlag`
	Deleted the following fields:
	❑ `extendedError`
	❑ `decodedFrameType`
	❑ `outputID`
	❑ `displayBufs`
Section 4.2.2.1	`IH264VDEC_Params`:
	❑ Added default value for each field
	❑ Added `maxDisplayDelay` field
Section 4.2.2.2	`IH264VDEC_DynamicParams`:
	Added the following fields:
	❑ `mbErrorBufFlag`
	❑ `mbErrorBufSize`
	❑ `Sei_Vui_parse_flag`
	❑ `numNALunits`
	Added note
Section 4.2.2.3	`IH264VDEC_InArgs`:
	Deleted the following fields:
	❑ `numNALunits`
	❑ `*numBytesInNALarr`
	❑ `Sei_Vui_parse_flag`
	❑ `SeiVui_buffer_ptr`
	❑ `maxDisplayDelay`
Section 4.2.2.4	`IH264VDEC_Status`:
	Added the following fields:
	❑ `display_frame_skip_flag`
	❑ `numNALdecoded`
	Updated the note
Section 4.2.2.5	`IH264VDEC_OutArgs`:
	Deleted the following fields:
	❑ `display_frame_skip_flag`
	❑ `numNALdecoded`
	❑ `SeiVui_buffer_ptr`
	Deleted the note
Section 4.4	Added section on Error Handling