H.264 Base/Main/High Profile Decoder on DM365/DM368

User's Guide

Literature Number: SPRUEV0C
September 2010

IMPORTANT NOTICE

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

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

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

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

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

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

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

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

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

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Communications and Telecom	www.ti.com/communications
DSP	dsp.ti.com	Computers and Peripherals	www.ti.com/computers
Clocks and Timers	www.ti.com/clocks	Consumer Electronics	www.ti.com/consumer-apps
Interface	interface.ti.com	Energy	www.ti.com/energy
Logic	logic.ti.com	Industrial	www.ti.com/industrial
Power Mgmt	power.ti.com	Medical	www.ti.com/medical
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Space, Avionics & Defense	www.ti.com/space-avionics-defense
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video and Imaging	www.ti.com/video
Wireless	www.ti.com/wireless-apps		

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

Read This First

About This Manual

This document describes how to install and work with Texas Instruments' (TI) H.264 Base/Main/High Profile Decoder implementation on the DM365/DM368 platform. It also provides a detailed Application Programming Interface (API) reference and information on the sample application that accompanies this component.

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

Intended Audience

This document is intended for system engineers who want to integrate TI's codecs with other software to build a multimedia system based on the DM365/DM368 platform.

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

How to Use This Manual

This document includes the following chapters:

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

- **Appendix B- ARM926 TCM Buffer Usage By Codec**, describes the buffer usage for this codec
- **Appendix C - Recommended Application Behavior for Error streams**, describes the application behavior in case of an Error Stream
- **Appendix D - Revision History**, highlights the changes made to SPRUEV0B codec specific datasheet to make it SPRUEV0C

Related Documentation From Texas Instruments

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

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

Related Documentation

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

- *ISO/IEC 14496-10:2005 (E) Rec. H.264 (E) ITU-T Recommandation*

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
CPB	Coded Picture Buffer

Abbreviation	Description
CMEM	Generic memory manager in Linux
CSL	Chip Support library
D1	720x480 or 720x576 resolutions in progressive scan
DCT	Discrete Cosine Transform
DMA	Direct Memory Access
DPB	Decoded Picture Buffer
EDMA	Enhanced Direct Memory Access
FMO	Flexible Macro-block Ordering
Full HD	1920x1088 or 1920x1080 resolution
HDTV	High Definition Television
HDVICP	High Definition Video and Imaging Coprocessor sub-system
IDR	Instantaneous Decoding Refresh
ITU-T	International Telecommunication Union
IMX	Imaging Multimedia Extension
JM	Joint Menu
JVT	Joint Video Team
MB	Macro Block
MBAFF	Macro Block Adaptive Field Frame
MPEG	Motion Pictures Expert Group
MV	Motion Vector
NAL	Network Abstraction Layer
NTSC	National Television Standards Committee
PDM	Parallel Debug Manager
PicAFF	Picture Adaptive Field Frame
PMP	Portable Media Player
PPS	Picture Parameter Set
PRC	Perceptual Rate Control
RMAN	Resource Manager

Abbreviation	Description
RTOS	Real Time Operating System
SEI	Supplemental Enhancement Information
SPS	Sequence Parameter Set
SW	Switch
UUID	Universal Unique Identifier
VGA	Video Graphics Array
VUI	Video Usability Information
XDAIS	eXpressDSP Algorithm Interface Standard
XDM	eXpressDSP Digital Media
YUV	Color space in luminance and chrominance form

Text Conventions

The following conventions are used in this document:

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

Product Support

When contacting TI for support on this codec, quote the product name (H.264 BP/MP/HP Decoder on DM365/DM368) and version number. The version number of the codec is included in the Title of the Release Notes that accompanies this codec.

Trademarks

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

All trademarks are the property of their respective owners.

Contents

Read This First ..... **iii**
About This Manual iii
Intended Audience iii
How to Use This Manual iii
Related Documentation From Texas Instruments iv
Related Documentation iv
Abbreviations iv
Text Conventions vi
Product Support vi
Trademarks vi
Contents ..... **vii**
Figures ..... **ix**
Tables ..... **xi**
Introduction **1-1**
1.1 Software Architecture 1-2
1.2 Overview of XDAIS, XDM, and Framework Component Tools 1-2
1.2.1 XDAIS Overview 1-2
1.2.2 XDM Overview 1-3
1.2.3 Framework Component 1-4
1.3 Overview of H.264 BP/MP/HP Decoder 1-7
1.4 Supported Services and Features 1-9
Installation Overview **2-1**
2.1 System Requirements for Linux 2-2
2.1.1 Hardware 2-2
2.1.2 Software 2-2
2.2 Installing the Component for Linux 2-2
2.3 Building and Running the Sample Test Application on LINUX 2-5
2.4 Configuration Files 2-6
2.4.1 Generic Configuration File 2-6
2.4.2 Decoder Configuration File 2-7
2.4.3 Decoder Sample Base Param Setting 2-7
2.5 Uninstalling the Component 2-8
Sample Usage **3-1**
3.1 Overview of the Test Application 3-2
3.1.1 Parameter Setup 3-3
3.1.2 Algorithm Instance Creation and Initialization 3-3
3.1.3 Process Call 3-4
3.1.4 Algorithm Instance Deletion 3-5
3.2 Frame Buffer Management by Application 3-6
3.2.1 Frame Buffer Input and Output 3-6
3.2.2 Frame Buffer Memory Optimizations 3-8
3.2.3 Frame Buffer Management by Application 3-9
3.3 Handshaking Between Application and Algorithm 3-10
3.3.1 Resource Level Interaction 3-10
3.3.2 Handshaking Between Application and Algorithms 3-11

3.4 Cache Management by Application 3-13
3.4.1 Cache Usage by Codec Algorithm 3-13
3.4.2 Cache and Memory Related Call Back Functions for Linux 3-14
3.5 Sample Test Application 3-14
3.6 Error Reporting and Inconsistencies Within Error Codes 3-16
API Reference 4-1
4.1 Symbolic Constants and Enumerated Data Types 4-2
4.1.1 Common XDM Constants and Enumerated Data Types 4-2
4.2 Data Structures 4-8
4.2.1 Common XDM Data Structures 4-8
4.2.2 H.264 Decoder Data Structures 4-19
4.3 H.264 Decoder Low latency specific Data Structures and Enumerations 4-39
4.3.1 Structures 4-39
4.3.2 Constant 4-40
4.3.3 Typedef 4-40
4.3.4 Enum 4-41
4.4 Interface Functions 4-43
4.4.1 Creation APIs 4-44
4.4.2 Initialization API 4-46
4.4.3 Control API 4-47
4.4.4 Data Processing API 4-49
4.4.5 Termination API 4-53

Figures

Figure 1-1. Software Architecture.....1-2
Figure 1-2. Framework Component Interfacing Structure. 1-5
Figure 1-3. IRES Interface Definition and Function-calling Sequence.....1-6
Figure 1-4. Block Diagram of H.264 Decoder 1-9
Figure 2-1. Component Directory Structure for Linux.....2-3
Figure 3-1. Test Application Sample Implementation.....3-2
Figure 3-2. Process Call with Host Release.....3-4
Figure 3-3. Frame Buffer Pointer Implementation.....3-7
Figure 3-4. Interaction of Frame Buffers between Application and Framework 3-9
Figure 3-5. Process Call with Host Release.....3-11
Figure 3-6. Interaction Between Application and Codec.....3-11
Figure 3-7. Interrupt Between Codec and Application. 3-13
Figure A-1. VICP Buffers Managed By FC. 4-2

This page is intentionally left blank

Tables

Table 1-1. List of Abbreviations ..... **iv**
Table 2-2. Component Directories for Linux. **2-4**
Table 3-1. Process() Implementation **3-14**
Table 3-2 List of Codec Specific Error Codes. **3-16**
Table 4-1. List of Enumerated Data Types **4-2**

This page is intentionally left blank

Chapter 1

Introduction

This chapter provides a brief introduction to XDAIS, XDM, and DM365/DM368 software architecture. It also provides an overview of TI's implementation of the H.264 BP/MP/HP Decoder on the DM365/DM368 platform and its supported features.

Topic **Page**

1.1 Software Architecture	**1-2**
1.2 Overview of XDAIS, XDM, and Framework Component Tools	**1-2**
1.3 Overview of H.264 BP/MP/HP Decoder	**1-7**
1.4 Supported Services and Features	**1-9**

1.1 Software Architecture

DM365/DM368 codec provides XDM compliant API to the application for easy integration and management. The details of the interface are provided in the subsequent sections.

DM365/DM368 is a digital multi-media system on-chip primarily used for video security, video conferencing, PMP and other related application.

DM365/DM368 codec are OS agonistic and interacts with kernel through the Framework Component (FC) APIs. FC acts as a software interface between OS and the codec. FC manages resources and memory by interacting with kernel through predefined APIs.

Following diagram shows the software architecture.

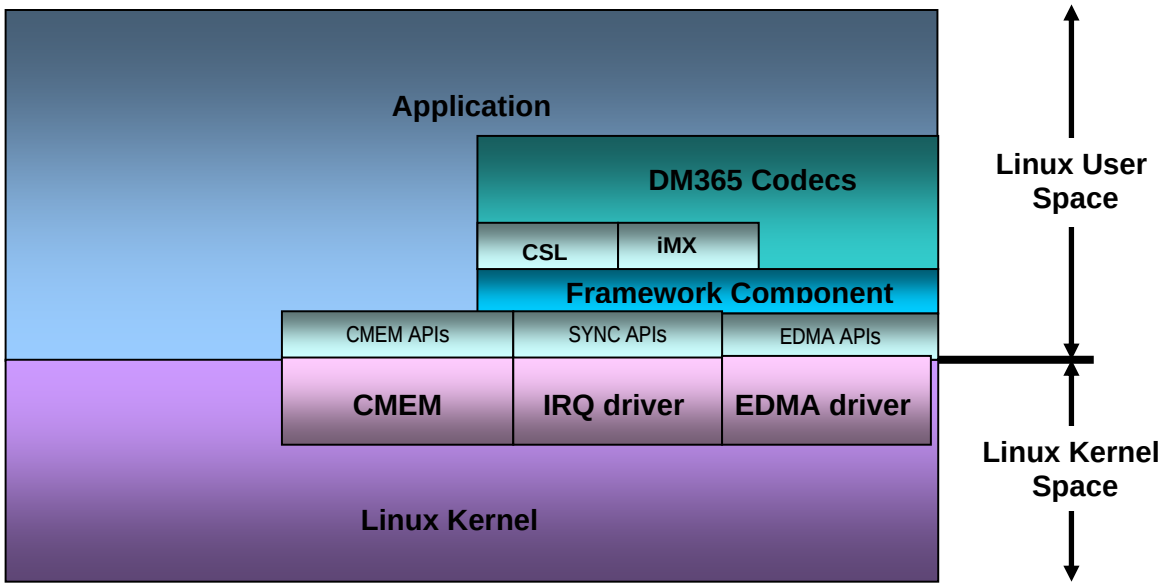


Figure 1-1. Software Architecture.

1.2 Overview of XDAIS, XDM, and Framework Component Tools

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

1.2.1 XDAIS Overview

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

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

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

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

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

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

1.2.2 XDM Overview

In the multimedia application space, you have the choice of integrating any codec into your multimedia system. For example, if you are building a video decoder system, you can use any of the available video decoders (such as MPEG4, H.263, 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:

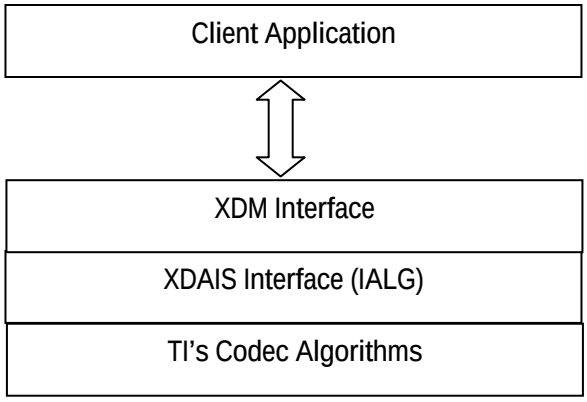
- `control()`
- `process()`

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. This API represents a blocking call for the encoder and the decoder, that is, with the usage of this API, the control is returned to the calling application only after encode or decode of one unit (frame) is completed. Since in case of DM365/DM368, the main encode or decode is carried out by the hardware accelerators, the host processor

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

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

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



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

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

1.2.3 Framework Component

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

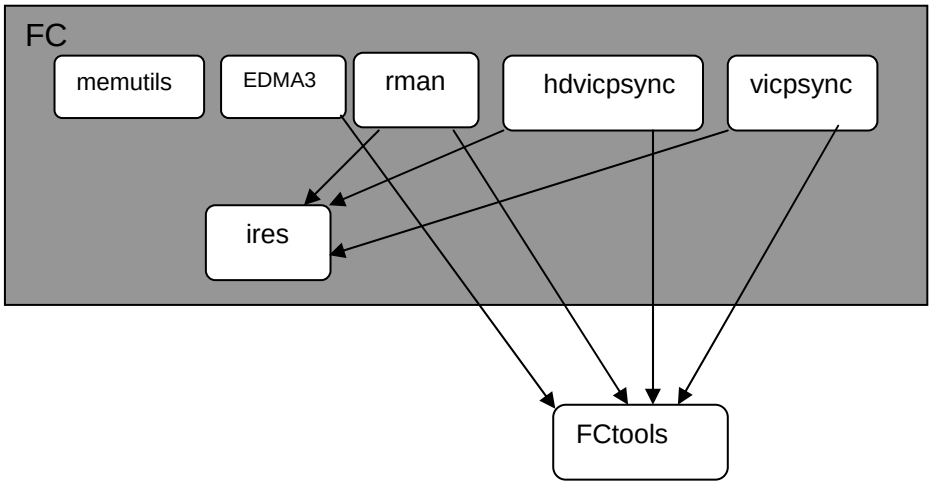


Figure 1-2. Framework Component Interfacing Structure.

Each component is explained in detail in the following sections.

IRES and RMAN Overview

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

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

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

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

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

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

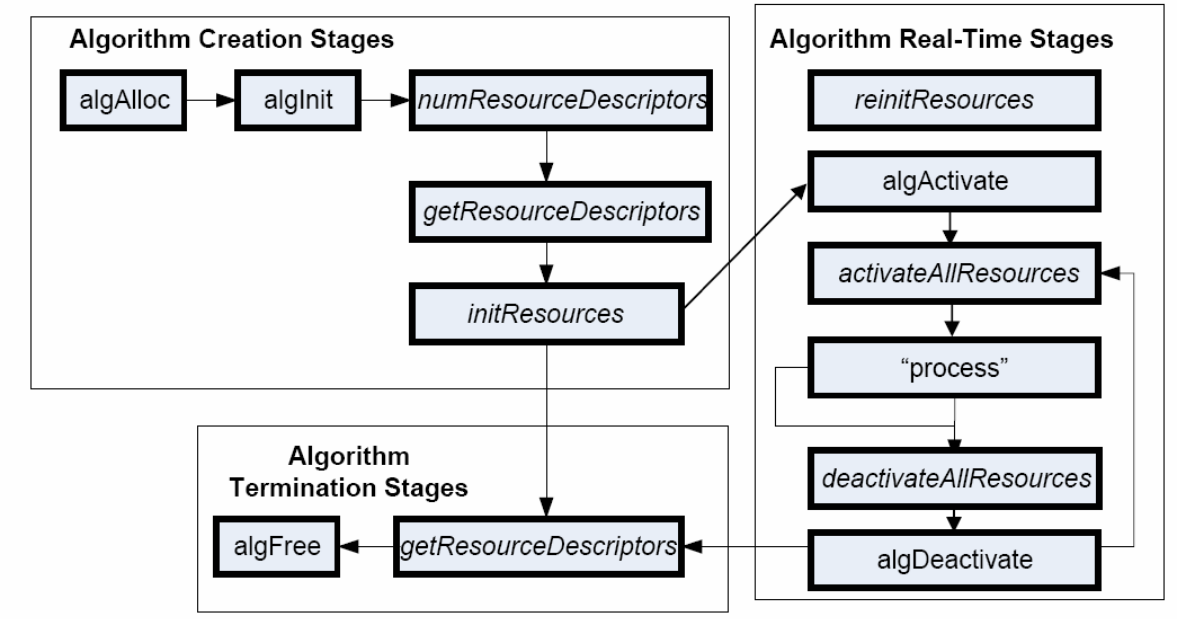


Figure 1-3. IRES Interface Definition and Function-calling Sequence.

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

HDVICP

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

EDMA3

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

Note:

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

VICP

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

HDVICP Sync

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

Memutils

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

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

1.3 Overview of H.264 BP/MP/HP 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 all decoders conforming to that profile. Each level specifies limits on the values that may be taken by the syntax elements in the profile.

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

- Baseline Profile:
  - Only I and P type slices are present
  - Only frame mode (progressive) picture types are present
  - Only CAVLC is supported
- Main Profile:
  - B type slices are present
  - Field, PicAFF and MBAFF type frames are present

- CABAC mode of entropy decoding
- Weighted prediction for motion compensation for both P and B slices.

- High Profile:
  - Scaling lists and quantization weight matrices
  - Support for transform 8x8 mode

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, and 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 is later quantized.
- Inter coded data: Motion information and residual error data (differential data between two frames), which is subjected to DCT and is 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 with the chroma components interleaved in little endian.

The following figure depicts the working of the decoder.

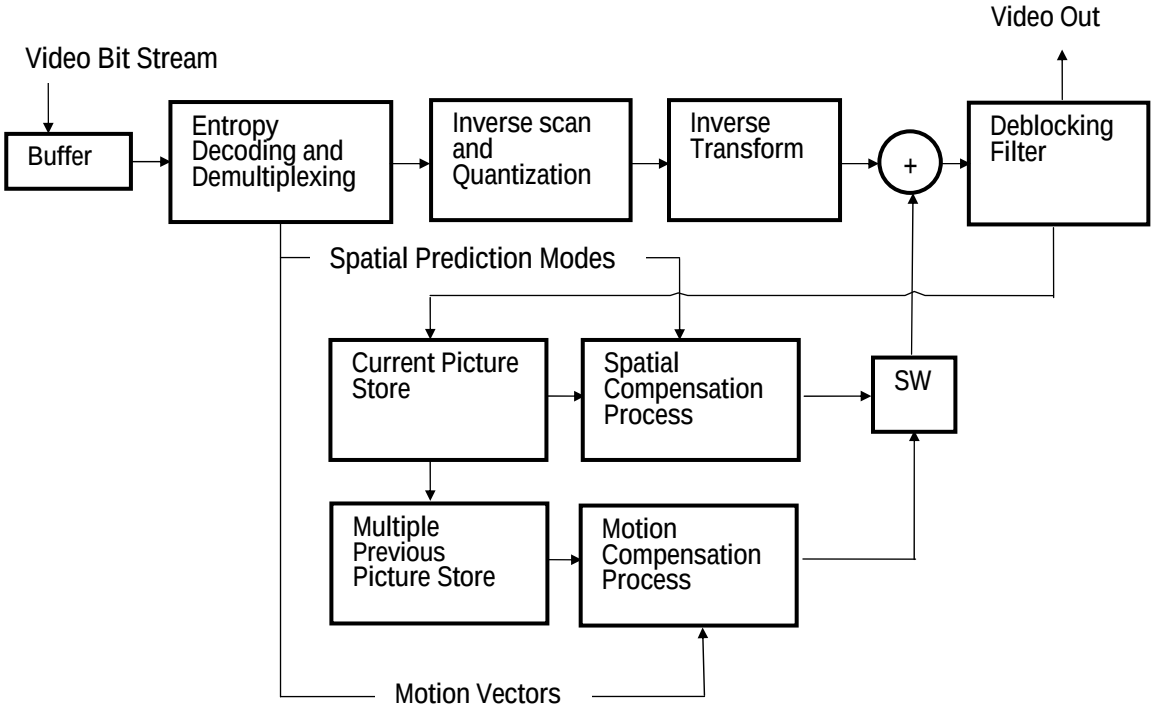


Figure 1-4. Block Diagram of H.264 Decoder

From this point onwards, all references to H.264 decoder mean H.264 Base/Main/High Profile (HP) decoder only.

1.4 Supported Services and Features

This user guide accompanies TI's implementation of H.264 Decoder on the DM365/DM368 platform.

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

- eXpressDSP Digital Media (XDM 1.2 IVIDDEC2) compliant
- Supports up to Level 4.2 features of the Base/Main/High Profile (limited by the Full HD resolution 1920x1088)
- Supports low latency feature – NAL layer interface with application
- Supports both byte stream and NAL stream mode of data input
- Supports progressive frame, field, PicAff and MBAFF types of picture decoding
- Supports multiple slices and multiple reference frames
- Supports CAVLC and CABAC decoding

- Supports Main Profile features like B-Slice decoding and CABAC
- Supports weighted prediction for motion compensation in both P and B-slices
- Supports transform 8x8 mode, interspersed with transform 4x4 MBs
- Supports parsing and decoding with scaling lists present both in SPS and PPS NAL units
- Supports second chroma qp index offset value present in PPS.
- Supports all intra-prediction and inter-prediction modes
- Supports up to 16 MV per MB
- Both adaptive and sliding window DPB management supported
- Output order conformance using frame bumping process supported
- Supports frame based decoding with frame size being multiples of 2
  - Field based decoding with frame height being multiples of 4 supported
- Outputs are available in YUV420 interleaved little endian format
- Supplemental Enhancement Information (SEI) and Video Usability Information (VUI)
- Uses configurable frame display delay for out of order display
- Performs basic error concealment on erroneous frames and reports the type of error occurred.
- Supports resolution up to 1080p(1920 x 1088)

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

- Error concealment features such as ASO/FMO and redundant slices
- Dynamic change in resolution
- Raw NAL unit format for input bit-stream
- Monochrome format

Note:

Support of non-multiple of 16 picture decode is using frame cropping information. Actual cropping is not done inside the library, only the relevant information is conveyed using the interface parameter. The application should do the actual crop before display.

Chapter 2

Installation Overview

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

Topic **Page**

2.1 System Requirements for Linux	**2-2**
2.2 Installing the Component for Linux	**2-2**
2.3 Building and Running the Sample Test Application on LINUX	**2-5**
2.4 Configuration Files	**2-6**
2.5 Uninstalling the Component	**2-8**

2.1 System Requirements for Linux

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

2.1.1 Hardware

- DM365/DM368 EVM (Set the bits 2 and 3 of switch SW4 to low(0) position and Set the bits 4 and 5 of switch SW5 to high(1) position)
- RS232 cable and network cable.

2.1.2 Software

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

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

2.2 Installing the Component for Linux

The codec component is released as a compressed archive. To install the codec, extract the contents of the tar file onto your local hard disk. The tar file extraction creates a directory called dm365_h264dec_xx_xx_xx _xx_production. Figure 2-1 shows the subdirectoriescreated in this directory.

Note:

xx_xx_xx_xx in the directory name is the version of the codec. For example, If the version of the codec is 02.00.01.00, then the directory created on extraction of tar file is dm365_h264dec _02_00_01_00_production

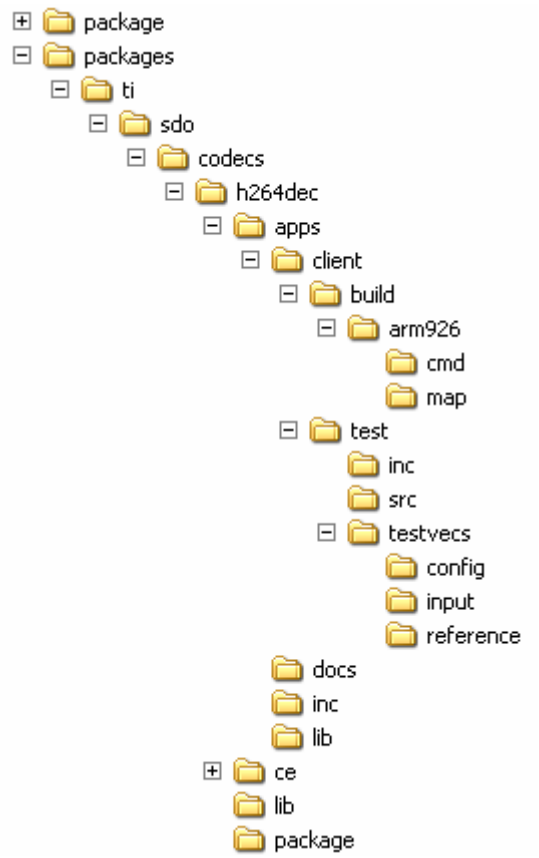


Figure 2-1. Component Directory Structure for Linux.

Table 2-2 provides a description of the sub-directories created in the dm365_h264dec_xx_xx_xx_xx_production directory.

Table 2-2. Component Directories for Linux.

Sub-Directory	Description
\package	Contains files related while building the package
\packages\ti\sdo\codecs\h264dec\lib	Contains the codec library files on host
\packages\ti\sdo\codecs\h264dec\docs	Contains user guide and release notes
\packages\ti\sdo\codecs\h264dec\apps\client\build\arm926	Contains the make file to built sample test application
\packages\ti\sdo\codecs\h264dec\apps\client\build\arm926\cmd	Contains a template (.xdt) file to used to generate linker command file
\packages\ti\sdo\codecs\h264dec\apps\client\build\arm926\map	Contains the memory map generated on compilation of the code
\packages\ti\sdo\codecs\h264dec\apps\client\test\src	Contains application C files
\packages\ti\sdo\codecs\h264dec\apps\client\test\inc	Contains header files needed for the application code
\packages\ti\sdo\codecs\h264dec\apps\client\test\testvecs\input	Contains input test vectors
\packages\ti\sdo\codecs\h264dec\apps\client\test\testvecs\output	Contains output generated by the codec
\packages\ti\sdo\codecs\h264dec\apps\client\test\testvecs\reference	Contains read-only reference output to be used for verifying against codec output
\packages\ti\sdo\codecs\h264dec\apps\client\test\testvecs\config	Contains configuration parameter files

2.3 Building and Running the Sample Test Application on LINUX

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

1) Verify that dma library, dma_ti_dm365.a, exists in the packages\ti\sdo\codecs\h264dec\lib.

2) Verify that codec object library, h264vdec_ti_arm926.a, exists in the \packages\ti\sdo\codecs\h264dec\lib.

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

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

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

6) In the folder \packages\ti\sdo\codecs\h264dec\client\build\arm926, change the paths in the file rules.make according to your setup.

7) Open the command prompt at the sub-directory \packages\ti\sdo\codecs\h264dec\client\build\arm926 and type the command **make**. This generates an executable file h264vdec-r in the same directory.

8) To run the executable generated from the above steps, branch to the directory where the executable is present and type ./h264vdec-r in the command window.

2.4 Configuration Files

This codec is shipped along with:

- Generic configuration file (testvecs_linux.cfg) – specifies input and reference files for the sample test application.
- Decoder configuration file (testparams5.cfg and testparamsHD.cfg) – specifies the configuration parameters used by the test application to configure the Decoder.

2.4.1 Generic Configuration File

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

The format of the testvecs_linux.cfg file is:

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

where:

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

A sample testvecs_linux.cfg file is as shown.

```
1
../../test/testvecs/config/testparams5.cfg
../../test/testvecs/input/foreman_part_qcif.264
../../test/testvecs/reference/foreman_part_qcif.crc
1
../../test/testvecs/config/testparams5.cfg
../../test/testvecs/input/foreman_qcif_IPB.264
../../test/testvecs/reference/foreman_qcif_IPB.crc
1
../../test/testvecs/config/testparams5.cfg
../../test/testvecs/input/foreman_qcif_IPB_Cabac.264
../../test/testvecs/reference/foreman_qcif_IPB_Cabac.crc
1
../../test/testvecs/config/testparams5.cfg
../../test/testvecs/input/foreman_qcif_IPB_T8_ScLst.264
../../test/testvecs/reference/foreman_qcif_IPB_T8_ScLst.crc
```

```
1
../../test/testvecs/config/testparams5.cfg
../../test/testvecs/input/foreman_qcif_OnlyT8.264
../../test/testvecs/reference/foreman_qcif_OnlyT8.crc
1
../../test/testvecs/config/testparams5.cfg
../../test/testvecs/input/foreman_qcif_Qmat.264
../../test/testvecs/reference/foreman_qcif_Qmat.crc
1
../../test/testvecs/config/testparams5.cfg
../../test/testvecs/input/foreman_qcif_T8.264
../../test/testvecs/reference/foreman_qcif_T8.crc
```

2.4.2 Decoder Configuration File

The decoder configuration file, testparams5.cfg contains the configuration parameters required for the decoder. The testparams5.cfg file is available in the \client\test\testvecs\config sub-directory.

A sample testparams5.cfg file for 720p stream is as shown.

```
# New Input File Format is as follows
# <ParameterName> = <ParameterValue> # Comment
############################################################
# Parameters
############################################################
ImageWidth = 1280 # Image width in Pixels, must be multiple of 16
ImageHeight = 720 # Image height in Pixels, must be multiple of 16
FramesToDecode = 8000 # Number of frames to be coded
Closedloopflag = 0 # 0 universal 1 closedloop
```

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

2.4.3 Decoder Sample Base Param Setting

The decoder can be run in `IVIDDEC2` base class setting. The following list provides the typical values of `IVIDDEC2` base class variables.

```
typedef struct IVIDDEC2_Params {
XDAS_Int32 size;
XDAS_Int32 maxHeigh = 1080;
XDAS_Int32 maxWidth = 1920;
XDAS_Int32 maxBitRate = 4194304;
XDAS_Int32 dataEndianness = XDM_BYTE;
XDAS_Int32 forceChromaFormat = XDM_YUV_420SP;
} IVIDDEC2_Params;
typedef struct IH264VDEC_Params {
IVIDDEC2_Params viddecParams;
XDAS_Int32  displayDelay = DISPLAY_DELAY;
void    *hdvicpHandle = (void*) NULL;
XDAS_Int32 levelLimit = LEVEL_4_2;
XDAS_Int32 frame_closedloop_flag = 1;
XDAS_Int32      inputDataMode = IH264VDEC_TI_ENTIREFRAME;
XDAS_Int32      sliceFormat = IH264VDEC_TI_BYTESTREAM;
```

```
} IH264VDEC_Params;

typedef struct IVIDDEC2_DynamicParams {
XDAS_Int32 size;
XDAS_Int32 decodeHeader = XDM_DECODE_AU;
XDAS_Int32 displayWidth = DEFAULT_DISPLAY_WIDTH;
XDAS_Int32 frameSkipMode = IVIDEO_NO_SKIP;
XDAS_Int32 frameOrder = IVIDDEC2_DISPLAY_ORDER
XDAS_Int32 newFrameFlag;
XDAS_Int32 mbDataFlag;
} IVIDDEC2_DynamicParams;
typedef struct IH264VDEC_DynamicParams {
IVIDDEC2_DynamicParams viddecDynamicParams;
IH264VDEC_TI_DataSyncGetFxn   getDataFxn = dataSyncPutData;
IH264VDEC_TI_DataSyncHandle   dataSyncHandle = NULL;
XDAS_Int8 resetHDVICPeveryFrame = 1;
} IH264VDEC_DynamicParams;
typedef struct IVIDDEC2_InArgs {
XDAS_Int32 size;
XDAS_Int32 numBytes = validBytes;
XDAS_Int32 inputID = buffEle->bufId;
} IVIDDEC2_InArgs;
typedef struct IH264VDEC_InArgs {
 IVIDDEC2_InArgs   viddecInArgs;
XDAS_Int32       seiVuiParseFlag = 1;
sSeiVuiParams_t *seiVuiBufferPtr = &(SeiVui_obj);
XDAS_Int32    numBytesNextFrame;
XDAS_Int32
pNumBytesInDesc[IH264VDEC_TI_MAXNUMBLOCKS];
}IH264VDEC_InArgs;
```

Note:

Set the value of `resetHDVICPeveryFrame` in `IH264VDEC_Dynamic Params` to 1.

2.5 Uninstalling the Component

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

Chapter 3

Sample Usage

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

Topic **Page**

3.1 Overview of the Test Application	**3-2**
3.2 Frame Buffer Management by Application	**3-6**
3.3 Handshaking Between Application and Algorithm	**3-10**
3.4 Cache Management by Application	**3-13**
3.5 Sample Test Application	**3-14**
3.6 Error Reporting and Inconsistencies Within Error Codes	**3-16**

3.1 Overview of the Test Application

The test application exercises the extended class (extended over `IVIDDEC2`) structure `IH264VDEC_Obj` of the H.264 Decoder library. The main test application files are TestAppDecoder_ih264_arm926.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.

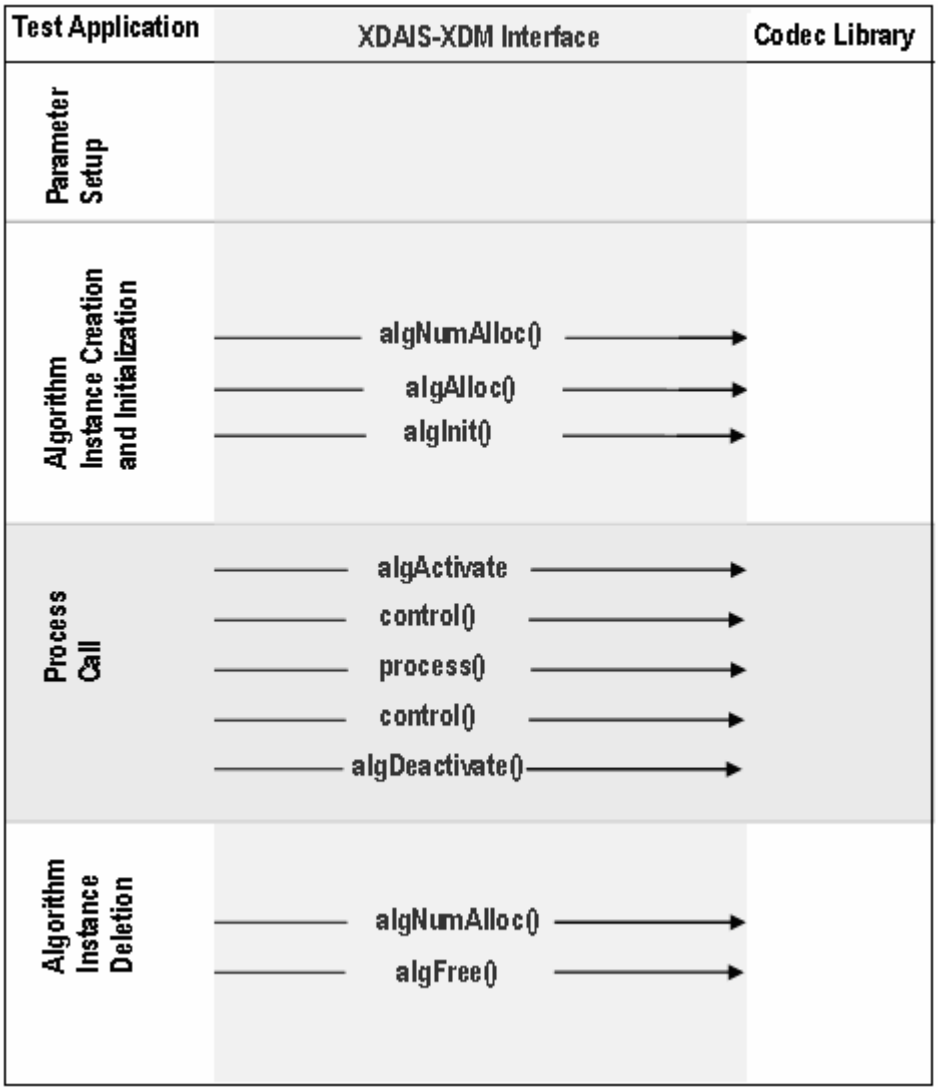


Figure 3-1. Test Application Sample Implementation

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_linux.cfg and reads the compliance checking parameter, Decoder configuration file name (testparams5.cfg), input file name, and output/reference file name.

2) Opens the Decoder configuration file, (testparams5.cfg) and reads the various configuration parameters required for the algorithm.

3) Sets the `IVIDDEC2_Params` structure based on the values it reads from the testparams5.cfg file.

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

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

3.1.2 Algorithm Instance Creation and Initialization

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

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

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

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

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

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

Note:

DMAN3 function and IDMA3 interface is not implemented in DM365/DM368 codecs. Instead, it uses a DMA resource header file that gives the framework the flexibility to change DMA resource to codec.

3.1.3 Process Call

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

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

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

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

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

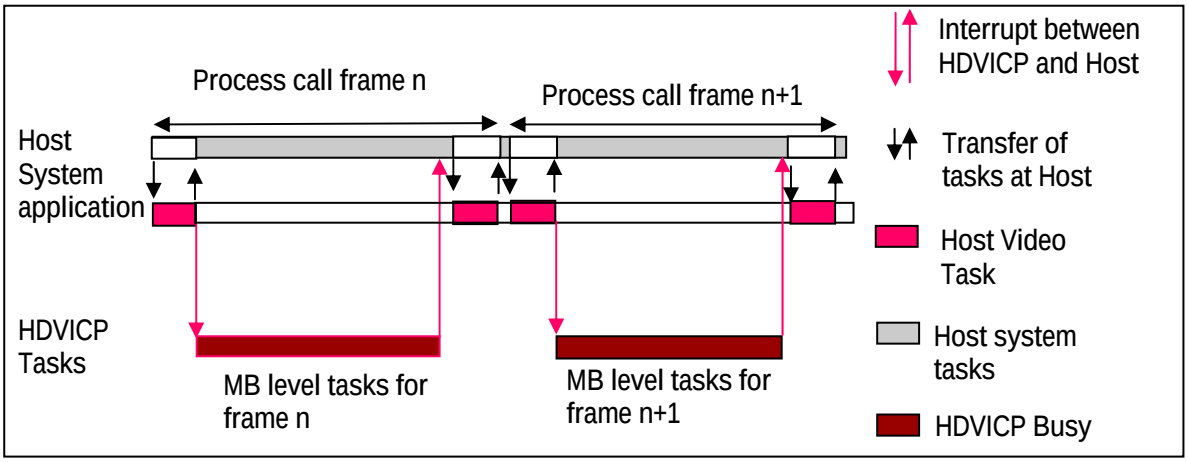


Figure 3-2. Process Call with Host Release

Note:

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

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

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

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

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

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

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

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

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

3.1.4 Algorithm Instance Deletion

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

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

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

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

3.2 Frame Buffer Management by Application

3.2.1 Frame Buffer Input and Output

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

```
H264VDEC_create();
H264VDEC_control(XDM_GETBUFINFO); /* Returns maximum number of
                                     buffers with size based on
                                     resolution configured */
do
{
     H264VDEC_decode();           /* Call the decode API */
}
while(all_frames)
```

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

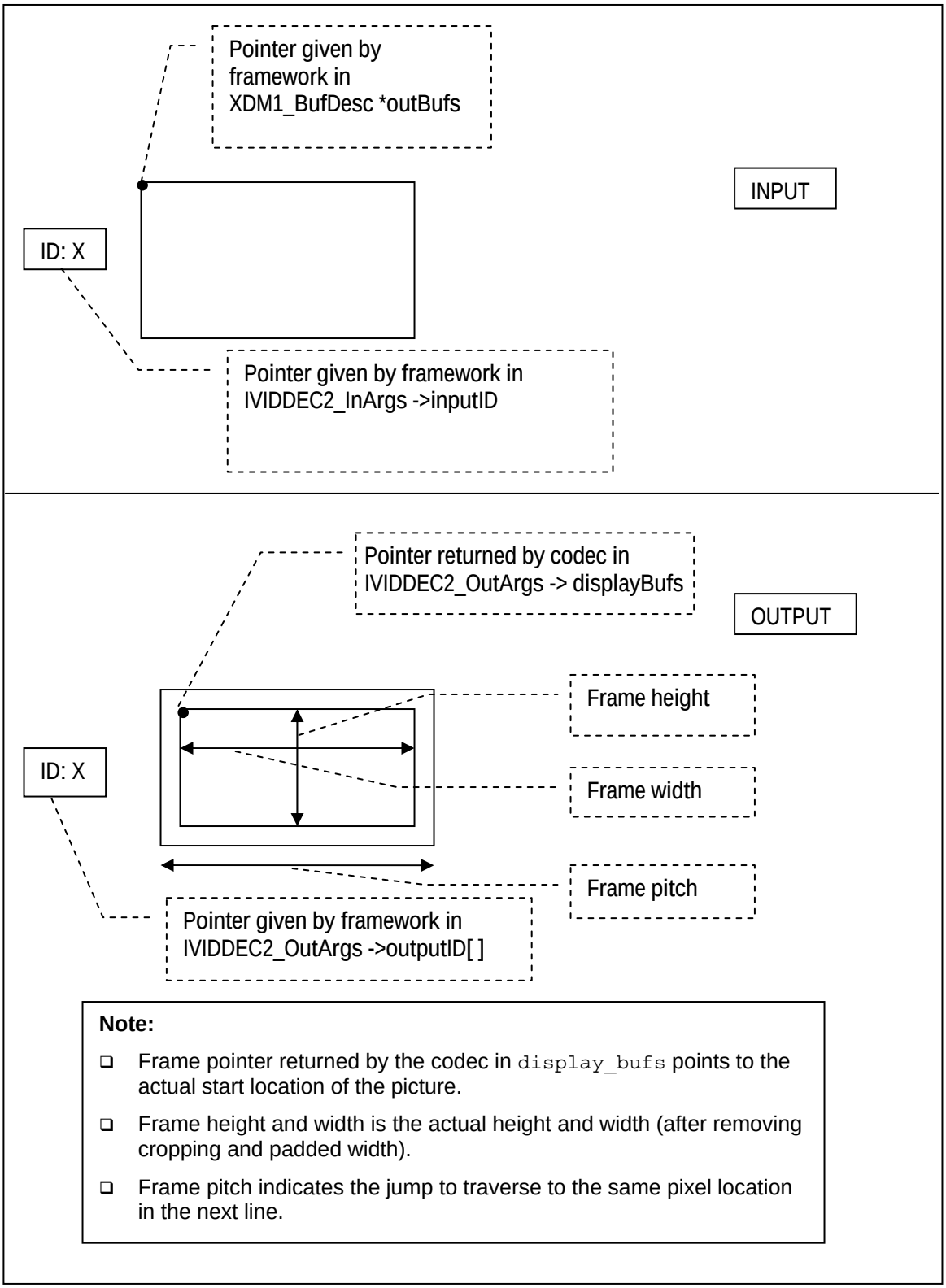


Figure 3-3. Frame Buffer Pointer Implementation

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

Note:

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

3.2.2 Frame Buffer Memory Optimizations

The `control()` API function (with `XDM_GETBUFINFO` command) requests the default maximum number of buffers of the size, based on the configured resolution. However, the actual memory required may be less compared to the one requested through `XDM_GETBUFINFO` for a given stream to be decoded. This is because stream properties are unknown to the decoder at that moment. After first successful frame decode, `XDM_GETBUFINFO` can return the correct memory requirements. Hence, to save on memory requirements, two `XDM_GETBUFINFO control()` API calls can be made. After first call, application may choose to allocate only one buffer of the requested resolution. Rest of the buffers can be allocated for exact resolution after second call.

```
H264VDEC_create();

H264VDEC_control(XDM_GETBUFINFO); /* Returns maximum number of
buffers with size based on resolution configured */

/* Allocate memory for 1 frame buffer */
frame_num = 0;
isReinitDone = 0;

do
{
    /* Get a free frame buffer */
    BUFFMGR_GetFreeBuffer();

    /* Process Call decode API */
    H264VDEC_decode();

    /* Release the frame buffer, if any */
    BUFFMGR_ReleaseBuffer();

    if(isReinitDone == 0)
    {
        if(NULL == BUFFMGR_QueryFreeBuffer())
        {
            /* Returns exact size buffers */
            H264VDEC_control(XDM_GETBUFINFO);

            isReinitDone = 1;

            /* Allocate memory for rest of frame buffers */
         }
```

```
    }
}
while(all_frames)
```

Before the allocation of rest of the buffers, as long as the first frame buffer is not locked by the decoder, there is no need to call second `XDM_GETBUFINFO control()` API. Once the buffer is locked by the decoder, the application may choose to call second `XDM_GETBUFINFO control()` API.

Note:

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

The number of buffers requested may be same in both the control calls. If the application has the prior knowledge of the level of the streams to be decoded, it can configure an appropriate value for `levelLimit` during `H264VDEC_create()`as described in 4.2.2.1. The decoder in that case requests the maximum number of buffers required for the decoded resolution and level configured.

3.2.3 Frame Buffer Management by Application

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

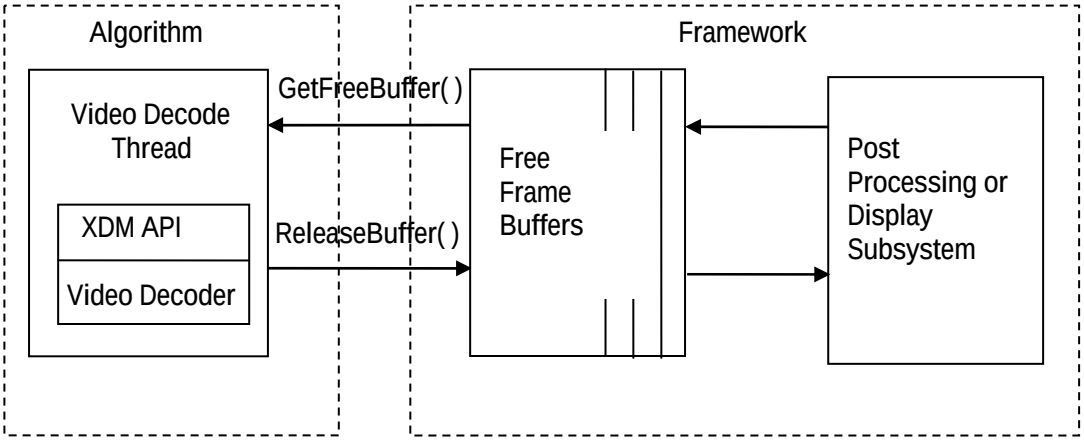


Figure 3-4. Interaction of Frame Buffers between Application and Framework

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

1) `BUFFMGR_Init()`

   The `BUFFMGR_Init` function is called by the test application to initialize the global buffer element array to default and to allocate the required

number of memory data for reference and output buffers. The maximum required DPB size is defined by the supported profile and level.

2) `BUFFMGR_ReInit()`

The `BUFFMGR_ReInit` function allocates global luma and chroma buffers and allocates entire space to the first element. This element will be used in the first frame decode. After the picture height and width and its luma and chroma buffer requirements are obtained, the global luma and chroma buffers are re-initialized to other elements in the buffer array.

3) `BUFFMGR_GetFreeBuffer()`

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

4) `BUFFMGR_QueryFreeBuffer()`

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

5) `BUFFMGR_ReleaseBuffer()`

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

6) `BUFFMGR_DeInit()`

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

3.3 Handshaking Between Application and Algorithm

3.3.1 Resource Level Interaction

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

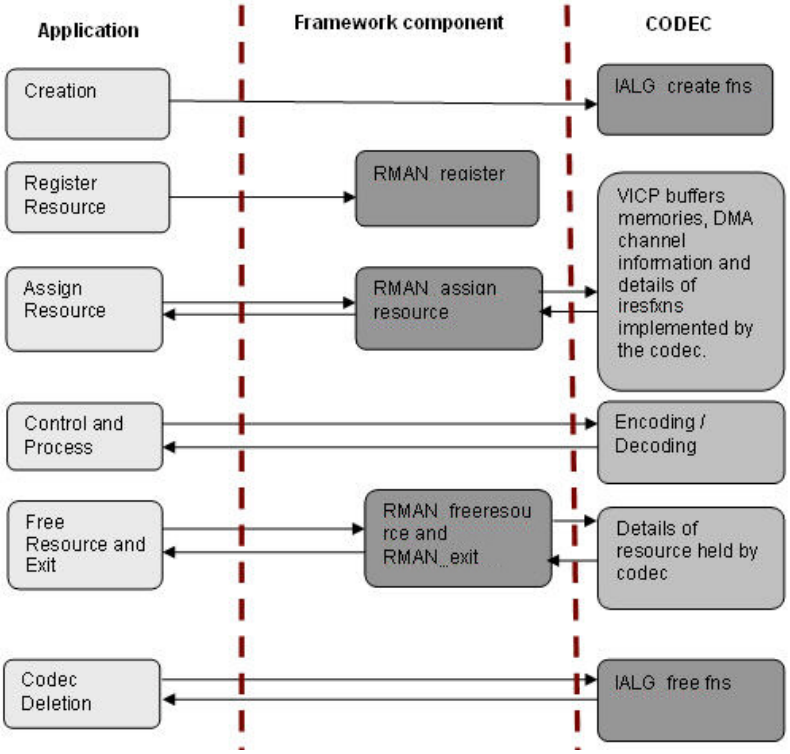


Figure 3-5. Process Call with Host Release.

3.3.2 Handshaking Between Application and Algorithms

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

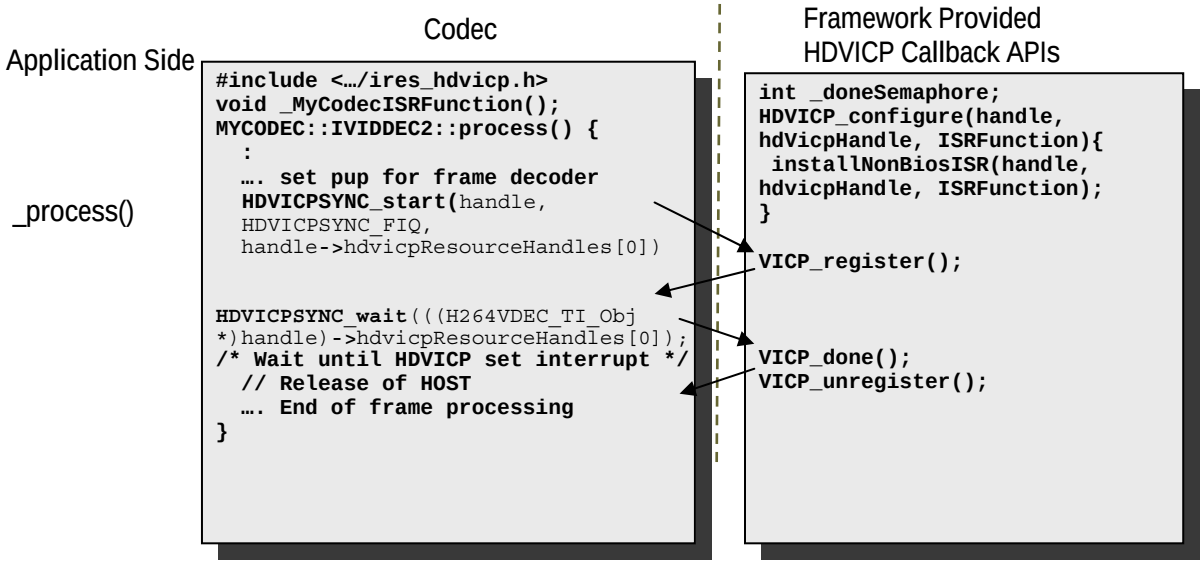


Figure 3-6. Interaction Between Application and Codec.

Note:

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

The functions to be implemented by the application are:

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

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

2) `HDVICPSYNC_wait (IRES_HDVICP_Handle hdvicpHandle)`

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

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

   Interrupts from HDVICP to Host ARM926 is used to inform when the frame processing is done. HDVICP sends interrupt which maps to `INT No 10 (KALINT9 Video IMCOP)` of ARM926 INTC. After receiving this interrupt, the semaphore on which the codec task was waiting gets released and the execution resumes after the `HDVICPSYNC_wait()` function.

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

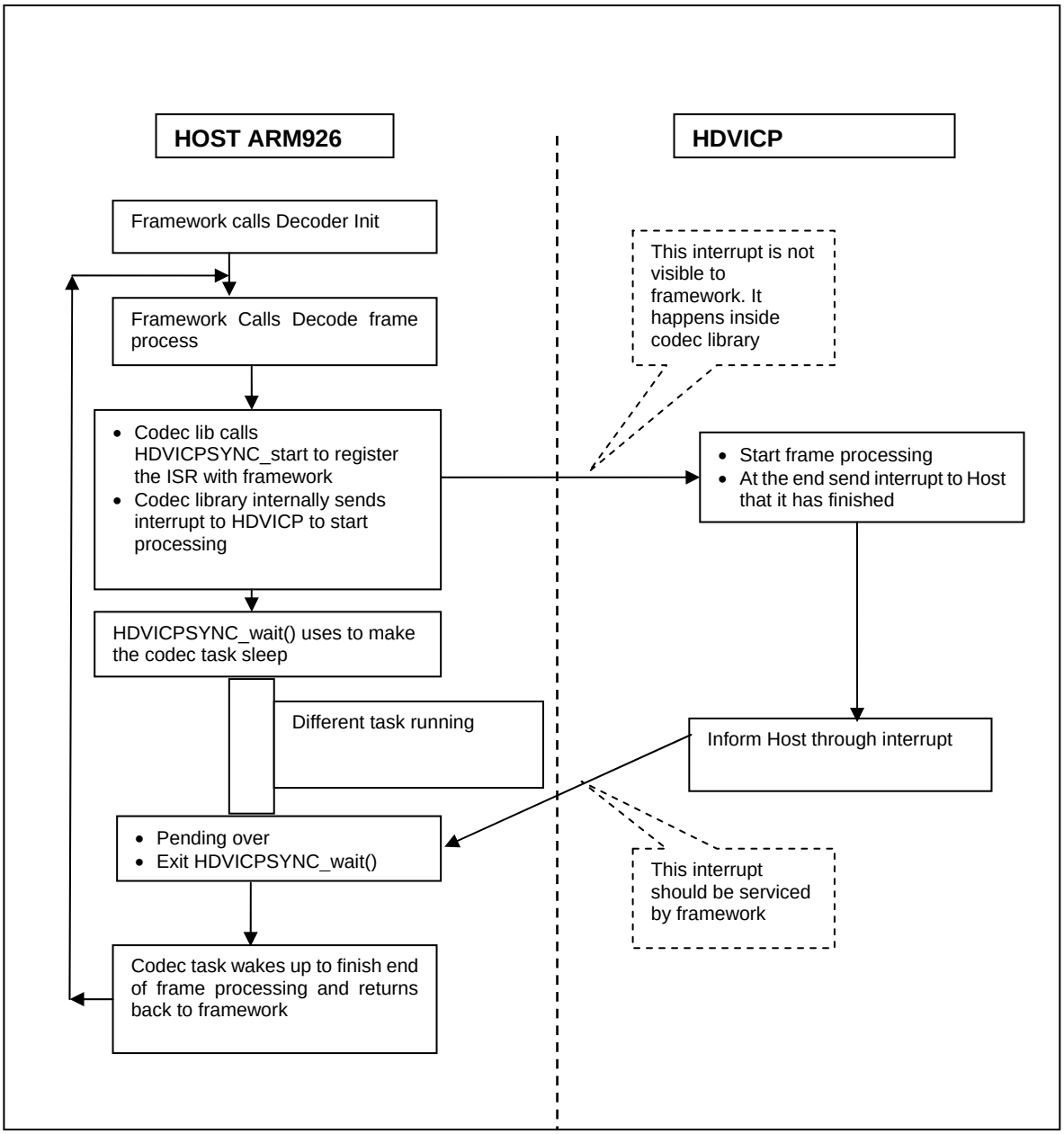


Figure 3-7. Interrupt Between Codec and Application.

3.4 Cache Management by Application

3.4.1 Cache Usage by Codec Algorithm

The codec source code and data, which runs on Host ARM926, can be placed in DDR (Double Data Rate). The host of DM365/DM368 has MMU and cache that the application can enable for better performance. Since

the codec also uses DMA, there can be inherent cache coherency problems when application turns on the cache.

3.4.2 Cache and Memory Related Call Back Functions for Linux

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

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

cacheInvalidate

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

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

cacheWb

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

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

cacheWbInv

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

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

getPhysicalAddr

This API obtains the physical address.

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

3.5 Sample Test Application

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

Table 3-1. Process() Implementation

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

```
iresStatus = (IRES_Status) RMAN_register(&IRESMAN_VICP2,
(IRESMAN_Params *)&configParams);
iresStatus = (IRES_Status) RMAN_register(&IRESMAN_HDVICP,
(IRESMAN_Params *)&configParams);

/*---------------- Decoder creation -----------------*/
handle = H264VDEC_create(&fxns, &params)

/*Getting instance of algorithms that implements IALG and
IRES functions*/
iErrorFlag = RMAN_assignResources((IALG_Handle)handle,
                            &H264VDEC_TI_IRES, /*
IRES_Fxns* */
                              1 /* scratchId */);
/* Get Buffer information  */
iErrorFlag = H264VDEC_control(
            handle, // Instance Handle
            XDM_GETSTATUS, // Command
            NULL, // Pointer to Dynamicparam structure
            &status// Pointer to the status structure
        );

/* do while Loop for decode Call  for the given stream */
/* Read the bitstream in the Application Input Buffer */
TestApp_ReadByteStream(inFile);
/*-----------------------------------------------------*/
/* Start the process : To start Decoding a frame      */
/* This will always follow a H264VDEC_decode_end call */
/*-----------------------------------------------------*/

 iErrorFlag = H264VDEC_decode (
                handle,   // Instance Handle    - Input
                &inobj,   // Input Buffers      - Input
                &outobj,  // Output Buffers     - Output
                &inargs,  // Input Parameters   - Input
                &outargs  // Output Parameters  - Output
            );
/* Get the statatus of the Decoder using control */
H264VDEC_control(
               handle,         // Instance Handle
               XDM_GETSTATUS,  // Command - GET STATUS
               NULL, // Input
               &status         // Output
                );
     }
/* end of Do-While loop - which Decodes frames  */
/* Free Input and output buffers */
FreeH264IOBuffers(
            &inobj, // Pointer to Input Buffer Descriptor
&outobj // Pointer to Output Buffer Descriptor  );
/* Free assigned resources */
RMAN_freeResources((IALG_Handle)(handle),
                    &H264VDEC_TI_IRES, /* IRES_Fxns* */
                   );
/* Delete the decoder Object handle*/
H264VDEC_delete(handle);
/* Unregister protocal*/
RMAN_unregister(&IRESMAN_EDMA3CHAN);
RMAN_exit();
```

Note:

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

3.6 Error Reporting and Inconsistencies Within Error Codes

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

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

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

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

Currently error in VUI is ignored and not reported as it may complicate the application specially in decode header mode as explained in 4.2.1.9 `IVIDDEC2_DynamicParams`.From LSB side first 8 bits(bit 0 to bit 7) in `extendedError` are codec specific error codes set by the decoder. Here bit 7, also referred as `H264VDEC_EOSTR` in Table 4-1 is set to flag the end of stream. For the remaining 7 bits (bit 0 to bit 7) Table 3-2 can be used to interpret the Error code. In case of multiple errors encountered during a picture decode, only one codec specific error code is reported as there is no provision to update multiple codec specific error codes.

Table 3-2 List of Codec Specific Error Codes.

Error Code	Error Name	Error Description
0x1	IH264VDEC_EX_ERR_DANGLING_FLD	Expecting second field but got a new frame number
0x2	IH264VDEC_EX_ERR_INVALID_SPS_CNT	SPS ID exceeded maximum allowed

Error Code	Error Name	Error Description
0x3	IH264VDEC_EX_ERR_INVALID_LEVEL	Invalid stream level
0x4	IH264VDEC_EX_ERR_INVALID_CONSTRAINT_SET3_FLAG	Profile related invalid constraint flag
0x5	IH264VDEC_EX_ERR_UNSUPP_PROFILE_IDC	Stream profile not supported
0x6	IH264VDEC_EX_ERR_UNSUPP_CHROMA_FORMAT_IDC	Chroma format not supported
0x7	IH264VDEC_EX_ERR_SCAL_LIST_NOT_SUPPORTED	Scaling list not supported
0x8	IH264VDEC_EX_ERR_INV_MAX_FRM_NUM	Invalid maximum frame number
0x9	IH264VDEC_EX_ERR_WRONG_POC	Invalid Picture Order Count
0xA	IH264VDEC_EX_ERR_MAX_POC_CYCLE_EXCEEDED	Error with Picture Order Count Cycle
0xB	IH264VDEC_EX_ERR_INV_POC_TYPE	Invalid Picture Order Count type
0xC	IH264VDEC_EX_ERR_NO_OF_REF_FRMS_EXCEEDED	Invalid reference frames count
0xD	IH264VDEC_EX_ERR_UNSUPP_RESOLUTION	Stream resolution not supported
0xE	IH264VDEC_EX_ERR_MAXCPBCNT_EXCEEDED	Error CPB Count
0xF	IH264VDEC_EX_ERR_LEVEL_SEQ_PAR_SETID_EXCEEDED	Invalid SPS ID
0x10	IH264VDEC_EX_ERR_CABAC_NOT_SUPPORTED	CABAC not supported (for Baseline Profile)
0x11	IH264VDEC_EX_ERR_WEIGHTED_PRED_NOT_SUPPORTED	Weighted Prediction not supported (for Baseline Profile)
0x12	IH264VDEC_EX_ERR_INV_SLICE_QP	Invalid slice QP
0x13	IH264VDEC_EX_ERR_INV_CHROMA_QP_OFFSET	Invalid Chroma QP Offset
0x14	IH264VDEC_EX_ERR_INV_INFERENCE_FLAG	Invalid inference flag
0x15	IH264VDEC_EX_ERR_CHANGE_IN_RESO	Dynamic change in resolution not supported

Error Code	Error Name	Error Description
0x16	IH264VDEC_EX_ERR_FIRST_MB_IN_SLICE	Unexpected value of First MB number in slice. This could happen either due to bit-stream corruption or ASO feature of baseline profile. Hence, for baseline profile stream, in addition to this error, XDM_UNSUPPORTEDINPUT of XDM_ErrorBit is also set
0x17	IH264VDEC_EX_ERR_FMO_NOT_SUPPORTED	FMO not supported
0x18	IH264VDEC_EX_ERR_MAX_PPS_CNT_EXCEEDED	PPS ID exceeded maximum allowed
0x19	IH264VDEC_EX_ERR_INV_SLICE_TYPE	Invalid Slice type
0x1A	IH264VDEC_EX_ERR_INV_PPS_ID	Invalid PPS ID
0x1B	IH264VDEC_EX_ERR_INV_DPB_LIMIT	Error in DPB Limit
0x1C	IH264VDEC_EX_ERR_NALU_TYPE_NOT_SUPPORTED	NAL not supported
0x1D	IH264VDEC_EX_ERR_INV_ASPECT_RATIO_IDC	Invalid Aspect ratio
0x1E	IH264VDEC_EX_ERR_LEVEL_LIMIT_VIOLATED	Invalid combination of resolution and level
0x1F	IH264VDEC_EX_ERR_INV_MAX_REFIDX	Non existing reference index referred
0x20	IH264VDEC_EX_ERR_FRAME_NUM_ERROR	Error in frame number
0x21	IH264VDEC_EX_ERR_UNSUPPORTED_SLICE_TYPE	Slice type not supported
0x22	IH264VDEC_EX_ERR_GARBAGE_DATA_IN_BITSTREAM	Non decodeable data also present
0x23	IH264VDEC_EX_ERR_MAX_NO_DATA_IN_NALU	Input stream size not sufficient enough
0x24	IH264VDEC_EX_ERR_BITSTREAM_CORRUPTION	Bitstream corruption at block level
0x25	IH264VDEC_EX_ERR_NO_NAL_SYNC_MARKER	Error in finding start code
0x26	IH264VDEC_EX_ERR_MISSING_MBS	Not all MBs data present in stream
0x27	IH264VDEC_EX_ERR_CORRU_IN_MB_DATA	Bitstream corruption at MB level

Error Code	Error Name	Error Description
0x28	IH264VDEC_EX_ERR_RS_NOT_SUPP	Redundant Slices not supported
0x29	IH264VDEC_EX_ERR_INV_CROPPING	Invalid cropping parameters
0x2A	IH264VDEC_EX_ERR_SLICE_ALFA_C0_OFFSET	Invalid alpha offset
0x2B	IH264VDEC_EX_ERR_SLICE_BETA_OFFSET	Invalid beta offset
0x2C	IH264VDEC_EX_ERR_INV_MAX_WEIGHT_BIPRED_IDC	Error in Bi-prediction weights
0x2D	IH264VDEC_EX_ERR_INV_IDR_PIC_ID	Invalid ID for IDR picture
0x2E	IH264VDEC_EX_ERR_INV_PIC_ORDER_CNT_LSB	Error in Picture Order Count LSB
0x2F	IH264VDEC_EX_ERR_INV_DEBLOCKING_FILT_IDC	Invalid De-blocking filter IDC
0x30	IH264VDEC_EX_ERR_INV_REFIDX	Decoding P or B picture without any reference picture
0x31	IH264VDEC_EX_ERR_INV_FIRST_MB_IN_SLICE	Error in first MB number in slice
0x32	IH264VDEC_EX_ERR_INV_CABAC_IDC	Invalid CABAC IDC
0x33	IH264VDEC_EX_ERR_INSUFF_BUFSIZE	XDM_GETVERSION buffer size not sufficient enough
0x34	IH264VDEC_EX_ERR_INV_ARG_IN_PROCESS_CALL	Invalid parameters in process call arguments
0x35	IH264VDEC_EX_ERR_INV_IMPLEMENTATION_ID	Implementation ID not as expected
0x36	IH264VDEC_EX_ERR_INV_CODEC_ID	Codec ID not as expected
0x37	IH264VDEC_EX_ERR_INV_ARG_IN_CONTROL_CALL	Invalid parameters in control call arguments
0x38	IH264VDEC_EX_ERR_ALGO_NOT_ACTIVATED	Resources not activated
0x39	IH264VDEC_EX_ERR_RESOURCE_NOT_INITIALIZED	Resources not initialized
0x3A	IH264VDEC_EX_ERR_INVALID_LEVEL_SETTING_TO_4_2	Invalid level configured, setting to default 42

Error Code	Error Name	Error Description
0x3B	IH264VDEC_EX_ERR_INAPPROPRIATE_LEVEL_FOR_GIVEN_RESOLN	Resolution not supported for configured level
0x3C	IH264VDEC_EX_ERR_INVALID_DYNAMIC_PARAMS_SIZE	Incorrect dynamic params size
0x3D	IH264VDEC_EX_ERR_DISPLAY_WIDTH_DYNAMIC_CHANGE_INVALID	Change in `displayWidth` not allowed at run-time
0x3E	IH264VDEC_EX_ERR_INVALID_DISPLAY_WIDTH	`displayWidth` configured is too high, negative or insufficient for the stream to be decoded
0x3F	IH264VDEC_EX_ERR_DISPLAY_WIDTH_NOT_MULTIPLE_OF_32	`displayWidth` configured not multiple of 32 bytes
0x40	IH264VDEC_EX_ERR_RESET_HDVICP_VALUE_NOT_SUPPORTED	Invalid value specified for Reset HDVICP flag
0x41	IH264VDEC_EX_ERR_FRAME_ORDER_DYNAMIC_CHANGE_INVALID	Change in `frameOrder` not allowed at run-time
0x42	IH264VDEC_EX_ERR_FRAME_ORDER_INVALID	Invalid `frameOrder` value
0x43	IH264VDEC_EX_ERR_DECODE_HEADER_VALUE_INVALID	Invalid `decodeHeader` value
0x44	IH264VDEC_EX_ERR_UNALIGNED_PIC_BUFFER	Picture buffers not word aligned
0x45	IH264VDEC_EX_ERR_HEADER_NOT_FOUND	Header (SPS and PPS) not found in stream buffer
0x46	IH264VDEC_EX_ERR_INVALID_HEADER_FOUND	Invalid header decoded
0x47	IH264VDEC_EX_ERR_VERSION_BUFFER_INVALID	Invalid pointer to XDM_GETVERSION buffer
0x48	IH264VDEC_EX_ERR_SEI_VUI_BUFFER_INVALID	Invalid pointer to SEI VUI buffer
0x49	IH264VDEC_EX_ERR_FRAME_SKIP_NOT_IMPLEMENTED	Unsupported value of dynamic param `frameSkipMode`
0x4A	IH264VDEC_EX_ERR_NEW_FRAME_LOGIC_NOT_IMPLEMENTED	Unsupported value of dynamic param `newFrameFlag`
0x4B	IH264VDEC_EX_ERR_MB_DATA_NOT_SUPPORTED	Unsupported value of dynamic param `mbDataFlag`
0x4C	IH264VDEC_EX_ERR_WRONG_INPUT_ID	Unexpected input ID when `outBufsInUseFlag` set to 1 in previous process call

Error Code	Error Name	Error Description
0x4D	IH264VDEC_EX_ERR_SEI_NAL	Error in SEI message
0x4E	IH264VDEC_EX_ERR_INVALID_RESERVED_BITS	Invalid value of reserved bits in SPS
0x4F	IH264VDEC_EX_ERR_REF_FRAME_IN_ERROR	If the current picture refers to an already decoded error frame.
0x50	IH264VDEC_EX_ERR_GET_DATA_FXN_NULL	getDataFxn pointer required for low latency interface callback is NULL
0x51	IH264VDEC_EX_ERR_CALLBK	Callback returned with either XDM_ERROR or invalid numBlocks provided or one of the baseAddrs provided is NULL pointer
0x52	IH264VDEC_EX_ERR_UNINTENSIONAL_FRAME_LOST	Frame loss occurs, previous frame might be lost.

This page is intentionally left blank

Chapter 4

API Reference

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

Topic **Page**

4.1 Symbolic Constants and Enumerated Data Types	**4-2**
4.2 Data Structures	**4-8**
4.3 H.264 Decoder Low latency specific Data Structures and Enumerations	**4-39**
4.4 Interface Functions	**4-43**

4.1 Symbolic Constants and Enumerated Data Types

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

4.1.1 Common XDM Constants and Enumerated Data Types

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

Table 4-1. List of Enumerated Data Types

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	IVIDDEC2_MAX_IO_BUFFERS	20 – used to assign maximum number of buffers used in IVIDDEC2_OutArgs and IVIDDEC2_Status structures.
IVIDEO_FrameType	IVIDEO_I_FRAME	Intra coded progressive frame. This also represents the frame type, when returning after decode of intra coded first field of an interlaced frame
	IVIDEO_P_FRAME	Forward inter coded frame. This also represents the frame type, when returning after decode of forward inter coded first field of an interlaced frame
	IVIDEO_B_FRAME	Bi-directional inter coded frame. This also represents the frame type, when returning after decode of bi-directional inter coded first field of an interlaced frame
	IVIDEO_IDR_FRAME	Intra coded frame that can be used for refreshing video content. This also represents the frame type, when returning after decode of IDR coded first field of an interlaced frame
	IVIDEO_II_FRAME	Interlaced frame, top field is an I or IDR, bottom field is again I or IDR frame
	IVIDEO_IP_FRAME	Interlaced frame, top field is an I or IDR frame, bottom field is a P frame.
	IVIDEO_IB_FRAME	Interlaced frame, top field is an I or IDR frame, bottom field is a B frame.

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

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	IVIDEO_INTERLACED	Interlaced video content.
IVIDEO_FrameSkip	IVIDEO_NO_SKIP	Do not skip the current frame. This is the default value.
	IVIDEO_SKIP_P	Skip forward inter coded frame. Not supported in this version of H264 Decoder.
	IVIDEO_SKIP_B	Skip bi-directional inter coded frame. Not supported in this version of H264 Decoder.
	IVIDEO_SKIP_I	Skip intra coded frame. Not supported in this version of H264 Decoder.
	IVIDEO_SKIP_IP	Skip I and P frame/field(s). Not supported in this version of H264 Decoder.
	IVIDEO_SKIP_IB	Skip I and B frame/field(s). Not supported in this version of H264 Decoder.
	IVIDEO_SKIP_PB	Skip P and B frame/field(s). Not supported in this version of H264 Decoder.
	IVIDEO_SKIP_IPB	Skip I/P/B/BI frames. Not supported in this version of H264 Decoder.
	IVIDEO_SKIP_IDR	Skip IDR Frame. Not supported in this version of H264 Decoder.
XDM_DataFormat	XDM_BYTE	Big endian stream
	XDM_LE_16	16-bit little endian stream. Not supported in this version of H264 Decoder.
	XDM_LE_32	32-bit little endian stream. Not supported in this version of H264 Decoder.
XDM_ChromaFormat	XDM_YUV_420P	YUV 4:2:0 planar. Not supported in this version of H264 Decoder.
	XDM_YUV_422P	YUV 4:2:2 planar. Not supported in this version of H264 Decoder.

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_YUV_422IBE	YUV 4:2:2 interleaved (big endian). Not supported in this version of H264 Decoder.
	XDM_YUV_422ILE	YUV 4:2:2 interleaved (little endian). Not supported in this version of H264 Decoder.
	XDM_YUV_444P	YUV 4:4:4 planar. Not supported in this version of H264 Decoder.
	XDM_YUV_411P	YUV 4:1:1 planar. Not supported in this version of H264 Decoder.
	XDM_GRAY	Gray format. Not supported in this version of H264 Decoder.
	XDM_RGB	RGB color format. Not supported in this version of H264 Decoder.
	XDM_YUV_420SP	YUV 420 semiplaner (Luma 1st plane, * CbCr interleaved 2nd plane)
	XDM_ARGB8888	Alpha plane Not supported in this version of H264 Decoder
	XDM_RGB555	RGB 555 color format Not supported in this version of H264 Decoder
	XDM_RGB565	RGB 556 color format Not supported in this version of H264 Decoder
	XDM_YUV_444ILE	YUV 4:4:4 interleaved (little endian) Not supported in this version of H264 Decoder
XDM_CmdId	XDM_GETSTATUS	Query algorithm instance to fill Status structure.
	XDM_SETPARAMS	Set dynamic parameters via the DynamicParams structure. Most of the parameters in the structure are allowed to change only before the first process call in this version of H.264 Decoder.

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_RESET	Reset the algorithm. All fields in the internal data structures are reset and all internal buffers are flushed.
	XDM_SETDEFAULT	Initialize all fields in Params structure to default values specified in the library.
	XDM_FLUSH	Handle end of stream conditions. This command forces algorithm instance to output data without additional input.
	XDM_GETBUFINFO	Query algorithm instance regarding the properties of input and output buffers
	XDM_GETVERSION	Query the algorithm's version.
XDM_AccessMode	XDM_ACCESSMODE_READ	The algorithm reads from the buffer using the CPU.
	XDM_ACCESSMODE_WRITE	The algorithm writes from the buffer using the CPU.
XDM_ErrorBit	H264VDEC_EOSTR	Bit 7 ❑ 1 - End of Sequence or End of Stream Nal unit is followed by a VCL NAL ❑ 0 - Ignore
	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

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_FATALERROR	Bit 15 ❑ 1 - Fatal error (stop decoding) ❑ 0 - Recoverable error

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

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

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

4.2 Data Structures

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

4.2.1 Common XDM Data Structures

This section includes the following common XDM data structures:

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

4.2.1.1 XDM_BufDesc

‖ Description

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

‖ Fields

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

4.2.1.2 XDM1_BufDesc

Description

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

Fields

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

4.2.1.3 XDM_SingleBufDesc

Description

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

Fields

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

4.2.1.4 XDM1_SingleBufDesc

Description

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

Fields

Field	Data type	Input/ Output	Description
*bufs	XDAS_Int8	Input	Pointer to the buffer
bufSize	XDAS_Int32	Input	Size of the buffer in bytes
accessMask	XDAS_Int32	Output	If the buffer was not accessed by the algorithm processor (For example, it was filled through DMA or other hardware accelerator that does not write through the algorithm CPU), then no bits in this mask should be set. Note: This field is not updated by H264 decoder library.

4.2.1.5 XDM_AlgBufInfo

Description

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

Fields

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

Note:

For H.264 Base/Main/High Profile Decoder, the buffer details are:

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

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

4.2.1.6 IVIDEO1_BufDesc

Description

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

Fields

Field	Data type	Input/ Output	Description
`numBufs`	`XDAS_Int32`	Output	Number of buffers

Field	Data type	Input/ Output	Description
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 information
frameType	XDAS_Int32	Output	Type of the video frame. This takes one of the values from enumerated datatype IVIDEO_FrameType as described in Table 4-1.
topFieldFirstFlag	XDAS_Int32	Output	Flag to indicate when the application should display the top field first
repeatFirstFieldFlag	XDAS_Int32	Output	Flag to indicate when the first field should be repeated
frameStatus	XDAS_Int32	Output	Flag to indicate the status of the output frame. This takes one of the values from enumerated datatype IVIDEO_OutputFrameStatus as described in Table 4-1.
repeatFrame	XDAS_Int32	Output	Number of times the display process needs to repeat the displayed progressive frame
contentType	XDAS_Int32	Output	Content type of the buffer IVIDEO_ContentType
chromaFormat	XDAS_Int32	Output	Only supported value is XDM_YUV_420SP

4.2.1.7 IVIDDEC2_Fxns

Description

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

Fields

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

4.2.1.8 IVIDDEC2_Params

Description

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

Fields

Field	Data type	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes. Default value = 196
maxHeight	XDAS_Int32	Input	Maximum video height to be supported in pixels Default value = 1088 The minimum height supported by this implementation is 96 pixels (for luma).
maxWidth	XDAS_Int32	Input	Maximum video width to be supported in pixels Default value = 1920 The minimum width of the picture supported in this implementation is 112 pixels (for luma).
maxFrameRate	XDAS_Int32	Input	Maximum frame rate in fps * 1000 to be supported.
maxBitRate	XDAS_Int32	Input	Maximum bit-rate to be supported in bits per

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

Note:

- H.264 Decoder does not use the maxFrameRate and maxBitRate fields for creating the algorithm instance.
- Maximum video width supported is 1920 pixels (for 1080p format).
- The picture size constraint is on width (max 1920 pixels) and number of luma MBs (max 8160). This implies that even rotated pictures like 1088x1920 are also supported, along with 1920x1088 pictures.
- Maximum video height supported is 1920 pixels, provided it adheres to the maximum MB criteria of 8160. This additional requirement is imposed to ensure sufficiency of DPB buffers including padding requirements in the reference frames.
- dataEndianness field should be set to XDM_BYTE.
- Minimum supported width is 112 and height is 96.

4.2.1.9 IVIDDEC2_DynamicParams

‖ Description

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

‖ Fields

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

Field	Data type	Input/ Output	Description
decodeHeader	XDAS_Int32	Input	Number of access units to decode: ❑ 0 (XDM_DECODE_AU) - Decode entire frame including all the header (SPS and PPS) and other NALs that occur before the frame. If it gets SPS (or PPS), then it continues consuming all data till it gets PPS (or SPS). In case header is not decoded after the instance creation, decoder will continue to ignore all the data, unless it finds the header. If decoder cannot find the header in the stream, it returns error. Once the header is successfully decoded, the decoder expects one full frame of data. ❑ 1 (XDM_PARSE_HEADER) - Decode only header (SPS and PPS, Partial Slice header). Decoder looks for the header. Once header is decoded, it returns XDM_EOK. If it gets SPS (or PPS or Slice Header), then it continues consuming all the data till it gets PPS (or SPS). In case decoder cannot find both SPS , PPS and Slice header in the single stream buffer provided during the process () API call, it returns XDM_EFAIL. After the successful return of XDM_PARSE_HEADER, Application can set decodeHeader to XDM_PARSE_HEADER at any time, to get frame information. decodeHeader should be changed to XDM_DECODE_AU to resume normal decoding process. In case stream contains multiple headers of varying resolutions, it is recommended to use XDM_DECODE_AU or else correct information during control() API SETPARAMS is not guaranteed..
displayWidth	XDAS_Int32	Input	Display buffer pitch: ❑ 0 – Default Display buffer pitch. Configuring with the default value before first frame decode means, the decoder should derive the minimum possible display buffer pitch based on the decoded image width. ❑ Any non-zero value width be considered as display buffer pitch. However, after decoding the frame, if the configured displayWidth is found to be insufficient for the given stream, the process call is returned with fatal error. This can be used to configure only before the first frame decode. Run time change in displayWidth is not allowed.When calling control() API with SETPARAMS command after first frame decode, it must have the same value as the one used before first frame decode or must be set to default 0 value. This needs to be an integral multiple of 32 pixels and must be greater than the sum of padded area and the width of decoded image. The combined total padded area is 48 pixels.
displayHeight	XDAS_Int32	Input	This field is reserved. No use case assigned to it.

Field	Data type	Input/ Output	Description
frameSkipMode	XDAS_Int32	Input	Frame skip mode. See IVIDEO_FrameSkip enumeration for details. This is not supported by the current version of decoder and it hence expects frameSkipMode to be set to IVIDEO_NO_SKIP.
frameOrder	XDAS_Int32	Input	Frame display order. ❑ 0 (IVIDDEC2_DISPLAY_ORDER) – Decoder provides decoded output in actual order of display. When IVIDDEC2_DISPLAY_ORDER is configured, the display delay set through displayDelay as explained in section 4.2.2.1. IH264VDEC_Params is honoured. Hence, in case displayDelay is set less than the default value of 16, the correct display order in not guaranteed. ❑ 1 (IVIDDEC2_DECODE_ORDER) – Decoder provides decoded output in order of decoding. This sets the display delay to 0, that is, the frame buffer is given back for display as soon as it is decoded. The display delay set through display delay as explained in section 4.2.2.1 IH264VDEC_Params gets overwriten internally. This can be used to configure only before the first frame decode. Run time change in frame order is not allowed. When calling control() API with SETPARAMS command after first frame decode, it must have the same value as the one used before first frame decode.
newFrameFlag	XDAS_Int32	Input	Flag to indicate that, the algorithm should start a new frame. Valid values are XDAS_TRUE and XDAS_FALSE. This is useful for error recovery, For example when the end of frame cannot be detected by the codec but is known to the application. This logic is not implemented in the current version of decoder and it thus expects newFrameFlag to be set to XDAS_FALSE.
mbDataFlag	XDAS_Int32	Input	Flag to indicate that the algorithm should generate MB Data in addition to decoding the data. Valid values are XDAS_TRUE and XDAS_FALSE. This is not supported in the current version of decoder and it thus expects mbDataFlag to be set to XDAS_FALSE.

4.2.1.10 IVIDDEC2_InArgs

Description

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

Fields

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

Note:

- H.264 Decoder returns a failure if inputID is 0, otherwise it copies the inputID value to the outputID value of IVIDDEC2_OutArgs structure.
- H.264 decoder always expects numBytes to be equal to 1 frame of data.if numBytes is less than 4, a failure is returned.

4.2.1.11 IVIDDEC2_Status

Description

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

Fields

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

Field	Data type	Input/ Output	Description
extendedError	XDAS_Int32	Output	Extended error code. See XDM_ErrorBit enumeration for details.
data	XDM_SingleBufDesc	Input/Output	Buffer information structure for information passing buffer.
maxNumDisplayBufs	XDAS_Int32	Output	The maximum number of buffers that is required by the codec. The maximum number of buffers can be IVIDDEC2_MAX_IO_BUFFERS. In this decoder implementation it is always set to 17, if levelLimit field of IVIDDEC2_Params is not set. However, in case levelLimit is set to an appropriate valid value, maxNumDisplayBufs is derived based on levelLimit, maxWidth and maxHeight fields of IVIDDEC2_Params.
outputHeight	XDAS_Int32	Output	Output height in pixels
outputWidth	XDAS_Int32	Output	Output width in pixels
frameRate	XDAS_Int32	Output	Average frame rate in fps * 1000. The average frame rate for all video decoders is 30 fps.
bitRate	XDAS_Int32	Output	Average bit-rate in bits per second
contentType	XDAS_Int32	Output	Video content. See IVIDEO_ContentType enumeration for details.
outputChromaFormat	XDAS_Int32	Output	Only supported value is XDM_YUV_420SP.
bufInfo	XDM_AlgBufInfo	Output	Input and output buffer information. See XDM_AlgBufInfo data structure for details.

Note:

- Algorithm sets the frameRate and bitRate fields to zero.
- H.264 Decoder does not use the buffer descriptor meant for passing additional information between the application and the decoder.
- Fields which are not updated may return an uninitialized value when read.

4.2.1.12 IVIDDEC2_OutArgs

Description

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

Fields

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

Note:

- H.264 Decoder copies the inputID value to the outputID value of IVIDDEC2_OutArgs structure.
- decodedBufs, OutputMbDataID and mbDataBuf is not given as output in this version of the decoder. These fields may return an uninitialized value when read.
- After decoding first field of an interlaced frame, outBufsInUseFlag is set to 1.

4.2.2 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	Data type	Input/ Output	Description
viddecParams	IVIDDEC2_Params	Input	See IVIDDEC2_Params data structure for details.
displayDelay	XDAS_Int32	Input	Display delay before which the decoder starts to output frames for display. ❑ Default value: 16 (when base class is used). ❑ Valid range: [0, 16] The minimum display delay that gurantees the order of display in H.264 encoded stream is 16. Delay in the display can also be controlled by frameOrder. See section 4.2.1.9 IVIDDEC2_DynamicParams
hdvicpHandle	Void *	Input	HDVICP related handle object. Default value: NULL (when base class is used)
disableHDVICPeveryFrame	XDAS_Int8	Input	Flag to indicate if the co-processor needs to be disable and enabled at end of every frame. This is for power optimization ❑ Default value: NULL (when base class is used) ❑ Valid values: 0 – is taken as reset-disabled,

Field	Data type	Input/ Output	Description
levelLimit	XDAS_Int32	Input	Maximum level, as specified by H.264 standard, to be supported. Level is primarily used to restrict the amount of memory requested during the creation of the codec and XDM_GETBUFINFO control command. ❑ Default value: 0 (Library default level of 42 is configured) ❑ Valid range: 10 (corresponding to Level 1.0) to 42 (corresponding to level of 4.2), in addition to default value 0. However, all integer values between 10 and 42 are not valid. For more details, See *ISO/IEC 14496-10:2005 (E) Rec. H.264 (E) ITU-T Recommandation*.
frame_closedloop_flag	XDAS_Int32	Input	Flag for specifying Universal decoder mode or closed loop decoder mode ❑ Default value = 0 Universal decoder mode. ❑ Value = 1 Closed loop decoder mode.
inputDataMode	XDAS_Int32	Input	Mode for specifying low latency interface ❑ Default value = 1 . Low latency feature turned off. Minimum 1 frame worth of encoded is required to pass to decoder ❑ Value = 0 Low latency feature turned on. Decoder process call can be made with as minimum as 1 NAL worth of data in input buffer.
sliceFormat	XDAS_Int32	Input	Input Nal unit format ❑ Default value = 1 Input data in Byte stream format ❑ Value = 0 Input data in NAL stream format

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	Data type	Input/ Output	Description
viddecDynamicParams	IVIDDEC2_DynamicParams	Input	See IVIDDEC2_DynamicParams data structure for details.
getDataFxn	IH264VDEC_TI_DataSyncGetFxn	Input	Pointer to callback module for low latency feature
dataSyncHandle	IH264VDEC_TI_DataSyncHandle	Input	Handle to DataSync descriptor
resetHDVICPeveryFrame	XDAS_Int8	Input	Flag to reset HDVICP at the start of every frame being decoded. This is useful for multi-channel and multi-format encoding/decoding. ❑ Default value: 1 – ON (when base class is used). ❑ 0 – OFF Default value - 1 H264 decoder has three libraries, which dynamically gets loaded into HDVICP depending on the picture type. These three libraries correspond to different picture types namely ❑ Frame ❑ Field ❑ MBAff. If this flag is set, then H.264 decoder assumes that the memory of HDVICP was overwritten by some other codec or by other instance of same codec with different picture type. For example, Application will set this flag to 1 if running another instance of different codec like H264 encoder or if running another H264 decoder instance with different picture type. However, the application can set this flag to 0 for better performance if it runs multiple instances of H264 decoder with same picture type.

4.2.2.3 IH264VDEC_InArgs

‖ Description

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

‖ Fields

Field	Data type	Input/ Output	Description
viddecInArgs	IVIDDEC2_InArgs	Input	See IVIDDEC2_InArgs data structure for details.
seiVuiParseFlag	XDAS_Int32	Input	Flag to enable SEI and VUI NAL units parsing in the decoder. ❑ Default value: 0 [SEI parsing is disabled] ❑ Valid values: 1 to enable any other value disables parsing.
seiVuiBufferPtr	sSeiVuiParams_t *	Input	Object pointer of SEI and VUI structure. This pointer should have a valid memory allocation when Sei_Vui_parse_flag is set. Default value: NULL Even if Sei_Vui_parse_flag is 1, but SeiVui_buffer_ptr = NULL, then Sei_Vui_parse_flag is considered to be 0 and SEI/VUI parsing is disabled.
numBytesNextFrame	XDAS_Int32	Input	Size of the input data for next frame provided to algorithm,. This field is currently not used.
pNumBytesInDesc[IH264VDEC_TI_MAXNUMBLOCKS]	XDAS_Int32	Input	Array to specify valid bytes in input buffers provided at the time of decoder process call. Applicable only when low latency feature is turned ON and more than one buffer for input bitstream is provided at the time of decoder process call

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

Field	Data type	Input/ Output	Description
frameType	XDAS_Int32	Output	When decodHeader = XDM_PARSE_HEADER, This field will have current frame type returned by codec, when frame have multiple slice this field will have frametype present first slice header.
firstMBinSlc	XDAS_Int32	Output	When decodHeader = XDM_PARSE_HEADER, This field will have first MB number in a slice, when frame have multiple slice this field will have first MB number of first slice.
frmNuminSlcHdr	XDAS_Int32	Output	When decodHeader = XDM_PARSE_HEADER, This field will have frame number present in a slice header

4.2.2.5 IH264VDEC_OutArgs

‖ Description

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

‖ Fields

Field	Data type	Input/ Output	Description
viddecOutArgs	IVIDDEC2_OutArgs	Output	See IVIDDEC2_OutArgs data structure for details.
pixelRange	XDAS_UInt8	Output	❑ 0 - Range of Y from 16 to 235, Cb and Cr from 16 to 240 ❑ 1 - Range of Y from 0 to 255, Cb and Cr from 0 to 255.
parWidth	XDAS_UInt16	Output	Pixel aspect ratio width
parHeight	XDAS_UInt16	Output	Pixel aspect ratio height
numErrMbs	XDAS_UInt16	Output	Number of erroneous MBs in the output. **Note**: Due to the nature of VLD processing, even unintended bit-combinations may be a valid code resulting in successful decoding. This affects reporting the exact position of error in the input stream.
seiVui_buffer_ptr	sSeiVuiParams t *	Output	Object pointer of SEI and VUI structure. This pointer is same as that given in InArgs, if Sei_Vui_parse_flag is set. It contains updated fields after parsing VUI and SEI from bit-stream.

Field	Data type	Input/ Output	Description
bytesConsumedNext Frame	XDAD_Int32	output	Size of the bytes consumed for next frame preprocessing by the codec,. This field is currently not used.
numBlocksConsumed	XDAS_UInt8	output	Number of NALs consumed from last set of NALs provided through through getDataFxn callback for that process call

4.2.2.5.1 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	Data type	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.)*

4.2.2.5.2 sSeiMessages_t

Description

This structure contains supplementary enhancement information messages.

Fields

Field	Data type	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 shall be present in the bit-stream.
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. This is followed by a sequence of one or more pictures of refinement with the quality of the current picture, rather than as a representation of a continually moving scene.
prog_refine_end	sProgRefineEnd_t	Output	Specifies the 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 delay, DPB output

Field	Data type	Input/ Output	Description
			delay, and so on.
user_data_registered	sUserDataRegisteredITUT_t	Output	This message contains user data registered as specified by ITU-T Recommendation T.35.
user_data_unregistered	sUserDataUnregistered_t	Output	This message contains unregistered user data identified by its universal unique identifier (UUID).
buffering_period_info	sBufferingPeriod_t	Output	This structure contains the buffering period SEI message elements.

4.2.2.5.3 sUserDataRegisteredITUT_t

Description

Structure contains user data registered as specified by ITU-T Recommendation T.35.

Fields

Field	Data type	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
buffer_size	unsigned int	Input	The size of buffer allocated for copying the user data
user_data_size	unsigned int	Output	The size of user data copied in the buffer provided
buffer_size_insufficient	unsigned int	Output	❑ 0 - Indicates that the buffer provided is sufficient to hold the complete user data ❑ 1 - Indicates that the buffer provided is insufficient to hold the complete user data. Also, when buffer pointer is NULL, this field is set to 1.
itu_t_t35_country_code	unsigned int	Output	Byte that has a value specified as a country code by ITU-T Recommendation T.35 Annex A.
itu_t_t35_country_code_extension_byte	Unsigned int	Output	Byte that has a value specified as a country code by ITU-T Recommendation T.35 Annex B.
itu_t_t35_payload_byte	Unsigned char*	Input	Byte that contains data registered as specified by ITU-T Recommendation T.35. If this byte parsing is not intended and no memory is allocated, then this byte pointer is set to NULL by the application.

4.2.2.5.4 sUserDataUnregistered_t

‖ Description

Structure contains unregistered user data identified by a universal unique identifier (UUID).

‖ Fields

Field	Data type	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
buffer_size	unsigned int	Input	The size of buffer allocated for copying the user data
user_data_size	unsigned int	Output	The size of user data copied in the buffer provided
buffer_size_insufficient	unsigned int	Output	❑ 0 - Indicates that the buffer provided is sufficient to hold the complete user data 1 - Indicates that the buffer provided is insufficient to hold the complete user data. Also, when buffer pointer is NULL, this field is set to 1.
user_data_payload_byte	unsigned char*	Input	Byte that contains data having syntax and semantics as specified by the UUID generator. If this byte parsing is not intended, then this byte pointer is set to NULL by the application.

4.2.2.5.5 sBufferingPeriod_t

‖ Description

This structure contains the buffering period SEI message elements.

‖ Fields

Field	Data type	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
seq_parameter_set_id	unsigned int	Output	Specifies the sequence parameter set that contains the sequence HRD attributes. The value of seq_parameter_set_id will be equal to the value of seq_parameter_set_id in the picture parameter set, referenced by the primary coded picture associated with the buffering period SEI message. The value of seq_parameter_set_id will be in the range of 0 to 31,

Field	Data type	Input/ Output	Description
			inclusive of both 0 and 31.
initial_cpb_removal_delay[MAXCPBCNT]	unsigned int	Output	Specifies the delay for the SchedSelIdx-th CPB between: ❑ Time of arrival in the CPB of the first bit of the coded data associated with the access unit, which is associated with the buffering period SEI message. ❑ Time of removal from the CPB of the coded data associated with the same access unit, for the first buffering period after HRD initialization. The syntax element has a length in bits given by initial_cpb_removal_delay_length_minus1 + 1. It is in units of a 90 kHz clock. The value of initial_cpb_removal_delay[SchedSelIdx] should not be equal to 0 and should not exceed 90000 * (CpbSize[SchedSelIdx] ÷ BitRate[SchedSelIdx]), the time-equivalent of the CPB size in 90 kHz clock units.
initial_cpb_removal_delay_offset [MAXCPBCNT]	unsigned int	Output	Specifies the initial delivery time of coded access units to the CPB, for the SchedSelIdx-th CPB in combination with the cpb_removal_delay. The value of initial_cpb_removal_delay_offset[SchedSelIdx] is in units of a 90 kHz clock. The initial_cpb_removal_delay_offset[SchedSelIdx] syntax element is a fixed length code. The length in bits is given by initial_cpb_removal_delay_length_minus1 + 1. This syntax element is not used by decoders and is needed only for the delivery scheduler (HSS) specified in Annex C.

4.2.2.5.6 sFullFrameFreezeRepetition_t

Description

Structure contains information regarding frame freeze.

Fields

Field	Data type	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.

4.2.2.5.7 sFullFrameFreezeRelease_t

Description

Structure contains information regarding frame freeze.

Fields

Field	Data type	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 the output order.

4.2.2.5.8 sProgRefineStart_t

Description

Structure contains information regarding progressive refinement.

Fields

Field	Data type	Input/ Output	Description
parsed_flag	unsigned int	Output	❑ 1 - Indicates that in the current process call, contents of the structure is updated

Field	Data type	Input/ Output	Description
			❑ 0 - Indicates contents of the structure is not updated
progressive_refinement_id	unsigned int	Output	Specifies an identification number for the progressive refinement operation.
num_refinement_steps_minus1	unsigned int	Output	Specifies the number of reference frames in the tagged set of consecutive coded pictures.

4.2.2.5.9 sProgRefineEnd_t

‖ Description

Structure contains information regarding progressive refinement.

‖ Fields

Field	Data type	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.

4.2.2.5.9.1 sRecoveryPointInfo_t

‖ Description

Structure contains information regarding recovery points.

‖ Fields

Field	Data type	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 the output order.
exact_match_flag	unsigned char	Output	Indicates whether decoded pictures in the output order and subsequent to the specified recovery point in the output order is derived

Field	Data type	Input/ Output	Description
			by starting the decoding process at the access unit associated with the recovery point. SEI message must exactly match the pictures that will be produced by starting the decoding process at the location of a previous IDR access unit, in the NAL unit stream. The value 0 indicates that the match need not be exact and the value 1 indicates that the match must be exact. When decoding starts from the location of the recovery point of SEI message, all references to the not available reference pictures must be inferred as: ❑ references to pictures containing only macro-blocks coded using Intra macr-oblock prediction modes ❑ having sample values given by Y samples equal to 128, Cb samples equal to 128, and Cr samples equal to 128 (mid-level grey) for determining the conformance of the value of `exact_match_flag`
`broken_link_flag`	`unsigned char`	Output	Indicates the presence or absence of a broken link in the NAL unit stream.
`changing_slice_group_idc`	`unsigned char`	Output	Indicates whether decoded pictures are correct or approximately correct in content and subsequent to the recovery point in the output order when all macro-blocks of the primary coded pictures are decoded within the changing slice group period.

4.2.2.5.9.2 sPictureTiming_t

‖ Description

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

‖ Fields

Field	Data type	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

Field	Data type	Input/ Output	Description
			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 time of origin or capture of the associated frames or fields.
cnt_dropped_flag	unsigned int	Output	Specifies the skipping of one or more values 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.

Field	Data type	Input/ Output	Description
hours_value	unsigned int	Output	Specifies the value of hH used to compute clockTimestamp.
time_offset	unsigned int	Output	Specifies the value of tOffset used to compute clockTimestamp.

4.2.2.5.9.3 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	Data type	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	Equal to 1 specifies that the overscan_appropriate_flag is present. When overscan_info_present_flag is equal to 0 or is not present, the preferred display method for the video signal is unspecified.
overscan_appropriate_flag	unsigned int	Output	Indicates whether cropped decoded pictures are suitable for display or not.
video_signal_type_present_flag	unsigned int	Output	This flag indicates whether video_format, video_full_range_flag and colour_description_present_flag are present or not.
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.

Field	Data type	Input/ Output	Description
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 co-efficients for deriving Luma and chroma data from RGB components.
chroma_location_info_present_flag	unsigned int	Output	This flag indicates 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	It 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	It tells 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.
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

Field	Data type	Input/ Output	Description
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

4.2.2.5.9.4 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	Data type	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 bit-rate for the i^{th} CPB.
cpb_size_scale	unsigned int	Output	Together with cpb_size_value[i], specifies the maximum CPB size for the i^{th} CPB.
bit_rate_value[i]	unsigned int	Output	Maximum input bit-rate for the i^{th} CPB
cpb_size_value[i]	unsigned int	Output	Maximum CPB size for the i^{th} CPB

Field	Data type	Input/ Output	Description
vbr_cbr_flag[i]	unsigned int	Output	Specifies the i^{th} 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 1 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.
- Flag `parsed_flag` is present as the first element of structure of every SEI message, VUI structure, and the `SEI_VUI` structure. If this flag is set to 1 by decoder, the contents of the structure gets updated and the buffer pointer is copied to `IH264VDEC_OutArgs`. This flag is updated every process call, depending on whether SEI or VUI is parsed
- In case of multiple headers (SPS) parsed in a single process call, VUI buffer may get overwriten by new VUI params, and thus the final VUI params may not be the one, actually intended for use.

The decoder supports parsing of the following SEI messages:

- 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
- User data unregistered SEI message
- User data registered SEI message

The decoder skips all other types of SEI messages.

4.3 H.264 Decoder Low latency specific Data Structures and Enumerations

This section includes the following H.264 Decoder Low Latency specific structures, constant, typedefs and enumerations:

- IH264VDEC_TI_DataSyncDesc.
- IH264VDEC_TI_MAXNUMBLOCKS
- IH264VDEC_TI_DataSyncHandle
- IH264VDEC_TI_DataSyncPutGetFxn
- IH264VDEC_TI_DataMode
- IH264VDEC_TI_SliceFormat enumeration.

4.3.1 Structures

4.3.1.1 IH264VDEC_TI_DataSyncDesc

Description

This structure specifies the parameter used by application to provide input bit-stream buffer details to decoder

Fields

Field	Data type	Input/ Output	Description
Size	XDAS_Int32	Input	Size of this structure.
numBlocks	XDAS_Int32	Input	Number of blocks/NAL provided to decoder. Valid values are between 1 to IH264VDEC_TI_MAX_NUMBLOCKS
varBlockSizesFlag	XDAS_Int32	Input	Flag indicating whether any of the data blocks vary in size. Valid values are XDAS_TRUE and XDAS_FALSE. **Value of this flag is ignored at present.**
baseAddr	XDAS_Int32 *	Input	Array of pointers to the first byte of all (numBlocks) blocks provided for decoding NALs
blockSizes	XDAS_Int32*	Inout	This array contains the sizes of each valid blocks

4.3.2 Constant

4.3.2.1 IH264VDEC_TI_MAXNUMBLOCKS

Description

This MACRO defines max value of numBlocks accepted by accepted by decoder when operated in IH264VDEC_TI_SLICEMODE

4.3.3 Typedef

4.3.3.1 IH264VDEC_TI_DataSyncHandle

Description

Handle that identifies DataSync FIFO.

Field	Data type	Input/ Output	Description
IH264VDEC_TI_DataSyncHandle	Void *	Input	This handle is provided by the application to handle DataSync Fifo. Decoder passes this handle back to application when making request for more input data via callback

4.3.3.2 IH264VDEC_TI_DataSyncGetFxn

Pointer to callback module used by decoder to signal signal "data request" to application. Application need to define this API. Returns the success or failure status. Valid return value is `XDM_EOK` or `XDM_EFAIL`

Name

```
IH264VDEC_TI_DataSyncGetFxn.
```

Synopsis

```
typedef XDAS_Int32 (*IH264VDEC_TI_DataSyncGetFxn)(
IH264VDEC_TI_DataSyncHandle dataSyncHandle,
IH264VDEC_TI_DataSyncDesc dataSyncDesc);
```

Arguments

```
IH264VDEC_TI_DataSyncHandle dataSyncHandle /* Handle of
dataSync provided by application */
```

Arguments

```
IH264VDEC_TI_DataSyncDesc *dataSyncDesc /*
dataSyncDescriptor containing encoded bistream buffer
details */
```

Return Value

```
XDAS_Int32 /* Return Status – XDM_EOK/XDM_EFAIL */
```

4.3.4 Enum

4.3.4.1 IH264VDEC_TI_DataMode

Description

Describes the granularity of input data provided to decoder. This enumeration type is used by application to specify codec if input buffer carries bits sufficient to decoded one complete frame or partial frame (slices).

Fields

Enumeration Class	Symbolic Constant Name	Description
IH264VDEC_TI_DataMode	IH264VDEC_TI_SLICEMODE	provide bits sufficient to decoded slice of frame
	IH264VDEC_TI_ENTIREFRAME	provide bits to decode complete frame

4.3.4.2 H264VDEC_TI_SliceFormate

Description

This enumeration type is used by application to specify codec input slice format (NAL/Bytestream) type.

Fields

Enumeration Class	Symbolic Constant Name	Description
IH264VDEC_TI_DataMode	IH264VDEC_TI_NALSTREAM	Input data in NAL stream format
	IH264VDEC_TI_BYTESTREAM	Input data in BYTE stream format

Note:

Example Usage:

Configuring decoder

Assume multi slice scenario. Set decoder with below parameters:

```
IH264VDEC_Params->inputDataMode = 0
```

(slice mode)

```
IH264VDEC_Params->sliceFormat  = 1
```

(assuming byte stream encoding)

Application has to update the structure `IH264VDEC_TI_DataSyncDesc` specifying the number of NAL units to be processed, address of NAL units and size of each NAL unit. Decoder then processes and calls back Low latency API after decoding specified number of NAL units.

4.4 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* (SPRU360).

4.4.1 Creation APIs

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

‖ Name

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

‖ Synopsis

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

‖ Arguments

```
Void
```

‖ Return Value

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

‖ Description

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

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

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

‖ See Also

`algAlloc()`

‖ Name

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

‖ Synopsis

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

‖ Arguments

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

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

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

‖ Return Value

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

IALG_EFAIL /* Status indicating failure */
```

‖ Description

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

The first argument to `algAlloc()` is a pointer to a structure that defines the creation parameters. This pointer may be `NULL`; however, in this case, `algAlloc()` must assume default creation parameters and must not fail. If the size of this argument is not equal to the size of `IVIDDEC2_Params` or the size of `IH264VDEC_Params` then the default values of `IH264VDEC_Params` structure are used.

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

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

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

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

‖ See Also

`algNumAlloc(), algFree()`

4.4.2 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]`. This version of the decoder returns failure if this is `NULL`.

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

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

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

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

‖ See Also

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

4.4.3 Control API

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

‖ Name

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

‖ Synopsis

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

‖ Arguments

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

IVIDDEC2_Cmd id; /* algorithm specific control commands*/

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

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

‖ Return Value

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

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

‖ Description

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

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

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

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

Note:

- The number of input buffers is always equal to 1 and the number of output buffers is always equal to 2 (1 for Luma in Planar mode, other for chroma (CB and CR) in packed mode).
- IVIDDEC2_DynamicParams are used for setting run-time parameters. However, in this decoder implementation, all the parameters other than decodeHeader needs to be set before the first process call.

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

‖ Preconditions

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

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

‖ Postconditions

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

- If the control operation is successful, the return value from this operation is equal to `IALG_EOK`; otherwise it is equal to either `IALG_EFAIL` or an algorithm specific return value.
- If the control command is not supported, then an error bit in the extended error field of the `status` structure is set and `IALG_EOK` is returned.

Note:

The error bit `XDM_UNSUPPORTEDPARAM` is set only if the `status` structure is not NULL. Otherwise, this error is ignored and control API continues to return `IALG_EOK`. This deviation in control function implementation for unsupported commands is done to ease integration issues faced when multiple algorithms are integrated on the same platform.

- If the control command is not recognized (that is, not listed as any of the valid values for `XDM_CmdId` enumeration), then `IALG_EFAIL` is returned.

‖ Example

See test application file, TestAppDecoder_ih264_arm926.c available in the \client\test\src sub-directory.

‖ See Also

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

4.4.4 Data Processing API

Data processing API is used for processing the input data.

‖ Name

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

‖ Synopsis

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

‖ Arguments

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

‖ Return Value

```
Void
```

‖ Description

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

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

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

‖ See Also

`algDeactivate()`

‖ Name

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

‖ Synopsis

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

‖ Arguments

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

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

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

IVIDDEC2_InArgs *inargs /* algorithm runtime input
arguments */

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

‖ Return Value

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

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

‖ Description

This function does the basic encoding/decoding. This is a blocking call implementation and is based on two stage processing (`start()` and `end()`). The `process()` function calls the `start()` function to initiate the encode/decode call and then waits for the interrupt from `VIMCOP` on the `EVENT28`. On the receipt of the same, the `process_end()` is called to complete the encode/decode process. The pseudo implementation of the process API is shown in Table 3-1. Process() Implementation.

In a multi-tasking OS environment, where the `process()` call function is called in a separate thread/process, it is possible to free-up CPU cycles in-between the `start()` and end() stages of the `process()` call API. `HDVICP` set of global call-back functions are to be implemented by the system integrator and these are called by the algorithm library. In this set of functions OS-specific API functions can be called which can effectively put the `process()` API thread/process to sleep mode. vIMCOP issues an interrupt on completion of one full picture decode. These `HDVICP` set of functions need to re-activate the `process()` thread and resume with `end()` stage of `process()` API for logical completion of decode call. Refer to section 3.3.

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

The second and third arguments are pointers to the input and output buffer descriptor data structures respectively (see `XDM_BufDesc` data structure for details). When the flush mode is 0, the number of input buffers supported is 1 and the number of output buffers supported is 2. If it is less than these values, this version of the decoder returns failure. If it is more than these values, it takes only the first buffer for input and first two buffers for the output. In the flush mode (flush mode is 1) these fields are ignored.

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

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

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

‖ Preconditions

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

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

‖ Postconditions

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

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

‖ Example

See test application file, TestAppDecoder_ih264_arm926.c available in the \client\test\src sub-directory.

‖ See Also

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

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

‖ Name

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

‖ Synopsis

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

‖ Arguments

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

‖ Return Value

```
Void
```

‖ Description

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

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

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

‖ See Also

`algActivate()`

4.4.5 Termination API

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

‖ Name

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

‖ Synopsis

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

‖ Arguments

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

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

‖ Return Value

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

‖ Description

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

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

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

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

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

‖ See Also

`algAlloc()`

This page is intentionally left blank

Appendix A

VICP Buffer Usage By Codec

H.264 codec uses VICP buffers for its internal encode/decode operation. This buffer is accessed using EDMA. This section describes in brief how the buffers are used.

The Framework component (FC) manages the VCIP buffers using VCIP resource manager. In context of DM365/DM368, VICP buffers can be used by following algorithms:

- MPEG4 and JPEG running on MJCP
- H.264 codec running on HDVICP
- Preprocessing algorithms or noise filter running on IMX/NSF

Any of these algorithms can place its request to VICP buffers. FC services the VICP buffer request in a sequential manner.

	[Bytes]	Byte Address
iMX IMG Buf A	4096	0x00000 - 0x00FFF
iMX IMG Buf B	4096	0x01000 - 0x01FFF
iMX IMG Buf C	4096	0x02000 - 0x02FFF
iMX IMG Buf D	4096	0x03000 - 0x03FFF
iMX IMG Buf E	4096	0x04000 - 0x04FFF
iMX Coef 0	16384	0x05000 - 0x08FFF
iMX Coef 1	16384	0x09000 - 0x0CFFF
iMX Cmd 0	4096	0x0D000 - 0x0DFFF
iMX Cmd 1	4096	0x0E000 - 0x0EFFF
Sequencer PMEM	4096	0x0F000 - 0x0FFFF
		Register + Reserved
biMX cmd	1024	0x16000 - 0x163FF
biMX Org.	2048	0x16400 - 0x16BFF
biMX Ref.	8192	0x16C00 - 0x18BFF
QiQ Mem	1024	0x18C00 - 0x18FFF
DC/AC Pred Mem	256	0x19000 - 0x191FF
MV Mem.	512	0x91200 - 0x193FF
Huff Mem.	4096	0x19400 - 0x1A3FF
Seq Buf #1	4096	0x1A400 - 0x1B3FF
Seq Buf #2	4096	0x1B400 - 0x1C3FF
Seq Buf #3	2048	0x1C400 - 0x1CBFF
Seq Buf #4	2048	0x1CC00 - 0x1D3FF
Sequencer DMEM	4096	0x1F000 - 0x1F3FF

Figure A-1. VICP Buffers Managed By FC.

The above diagram shows the buffers of VICP managed by FC. The memories shaded in green are managed by FC. The memories in red are reserved for MJCP only.

FC gives the VICP memory to the algorithm from the start of the pool. Hence, it is the application's responsibility to instantiate the various algorithms in a way that an efficient usage of VICP buffers is achieved.

Appendix B

ARM926 TCM Buffer Usage By Codec

H.264 decoder uses ARM926 TCM buffers for its internal decode operation. This buffer is accessed using EDMA. This section describes the buffer usage.

ARM926 TCM buffer is used only in case of closed loop decode mode (`frame_closedloop_flag` = 1) and not in case of universal decode mode (`frame_closedloop_flag` = 0).

The ARM926 processor provides a complete high performance sub-system, which includes separate instruction, data, tightly-coupled memories (TCMs) and internal RAM interfaces. Instruction and data access is differentiated by accessing different memory map regions, with the instruction region from 0x0000 through 0x7FFF and data from 0x10000 through 0x17FFF.

In context of DM365/DM368, ARM926 DTCM can be used for the following:

- ARM926 for system level usage
- H.264 codec running on HDVICP

The ARM926 TCM buffer is used by H.264 codec for efficient DMA throughput to achieve noted performance in closed loop decode mode.

The ARM926 TCM buffers are managed by Framework component (FC) using ARM TCM resource manager.

This page is intentionally left blank

Appendix C

Recommended Application Behaviour for Error Streams

Decoder will return different type of errors such as: error in SPS, error in PPS, Slice header error, MB data error, and so on. To see the list of codec specific error codes see section 3.6 .

All the errors mainly divided into two types

- Fatal errors
- Non-fatal errors.

C.1 Behavior of Decoder in Case of Error Stream

- Decoder will update extended error based on `XDM_ErrorBit`.
- Decoder process call will return `XDM_FAIL` if any error occurs and `XDM_EOK` for success.
- If Decoder encountered MB data error, decoder will do error concealment and the concealed frame will be available for display.
- If decoder found serious error in Header, decoder will return from that point with `bytesConsumed` equal to the stream data consumed till that point. Decoder will also release the output buffer given in that process call (will set `freeBufID[0] = inputID`). `outputID[0]` will be set to 0 (nothing to display).
- If decoder found frame loss error (one or more complete frame was missing) while parsing the header, decoder will return from that point with bytesConsumed set as 0. Decoder will also release the output buffer given in that process call (will set `freeBufID[0] = inputID`). `outputID[0]` will be set to 0 (nothing to display).

C.2 Recommended behavior of application in case of error stream

Once decoder returns `XDM_FAIL` application has to check extended error for reason to fail and identify the category based on bit 7 to 15. Fatal and non-fatal category can be derived from bit 15.

- If decoder returns FATAL error the application should do `XDM_RESET`. After `XDM_RESET`, the input bit stream provided to the decoder is considered as a fresh stream with no relation to the already decoded stream.

- If decoder returns Non-FATAL error :
  - If the application is interested in continuing with the same stream, then Application should not do `XDM_RESET`.
  - If the application is interested in providing a fresh stream, then Application should do `XDM_RESET`.
- When error in Header occurs, It is recommended that application pass available bit stream, so that decoder will try to get next valid header.

It is expected that application handles the case where decoder is returning the input buffer back in the same process call (though `freeBufID`) without actually outputting any buffer for display. Application in such cases can continue displaying the last frame

Appendix D

Revision History

This user guide revision history highlights the changes made to SPRUEV0B codec specific datasheet to make it SPRUEV0C.

Table D-1.Revision History for H.264 High Profile Decoder on DM365/DM368

Section	Addition/Deletion/Modification
Global	❑ Added Appendix A - VICP Buffer Usage By Codec ❑ Added Appendix B- ARM926 TCM Buffer Usage By Codec ❑ Added Appendix C - Recommended Application Behavior for Error Streams
Section 3.6	Error Reporting and Inconsistencies Within Error Codes: ❑ Updated Table 3-2 List of Codec Specific Error Codes.