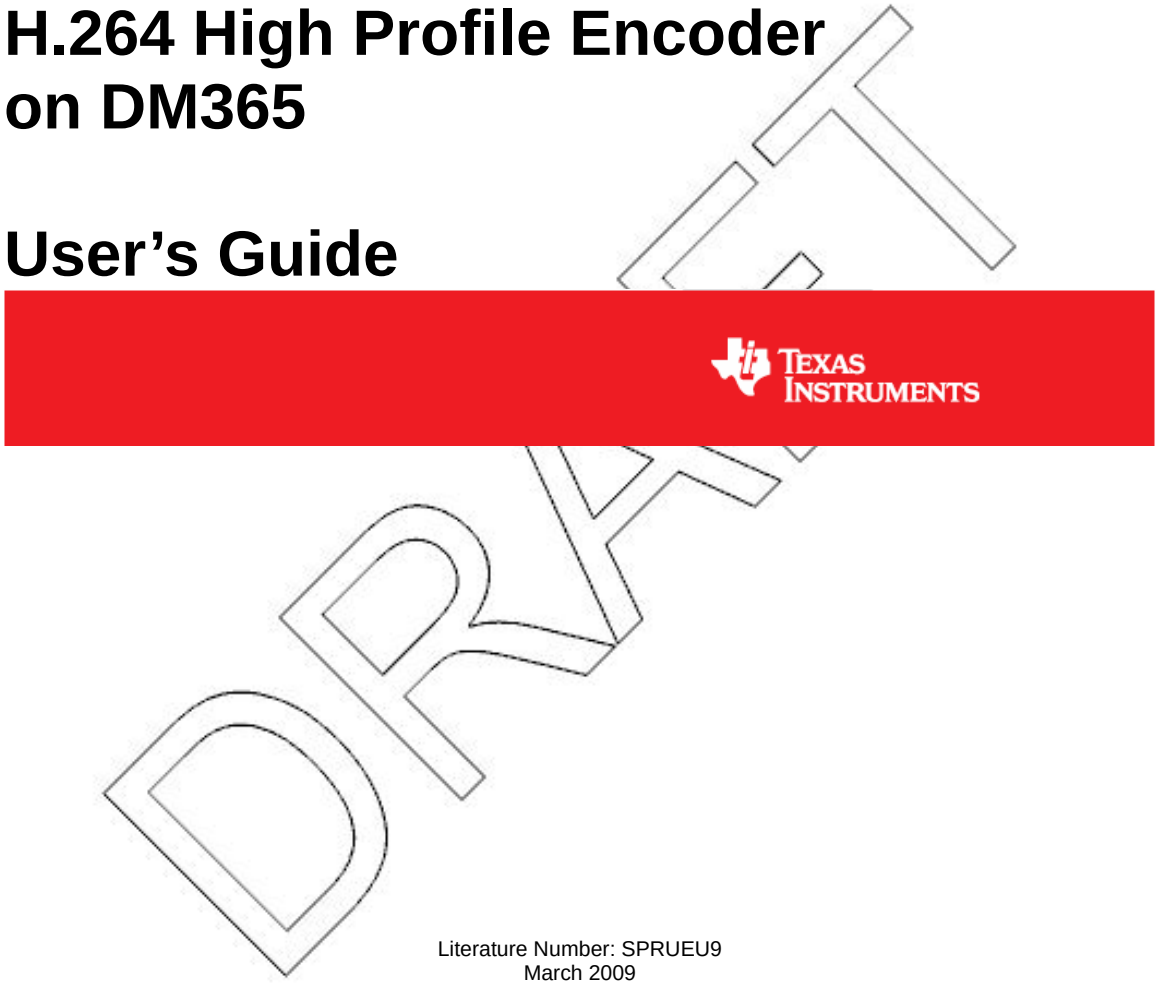
H.264 High Profile Encoder on DM365

User's Guide



Literature Number: SPRUEU9
March 2009

IMPORTANT NOTICE

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

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

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

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

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

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

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

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

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

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

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

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

Read This First

About This Manual

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

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

Intended Audience

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

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

How to Use This Manual

This document includes the following chapters:

- **Chapter 1 – Introduction**, provides a brief introduction to the XDAIS and XDM standards, Frame work Components, 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 – Time-Stamp Insertion**, describes insertion of frame time-stamp through the Supplemental Enhancement Information (SEI) Picture Timing message.

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* (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* (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, along 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 Recommendation*

Abbreviations

The following abbreviations are used in this document.

Table 1-1. List of Abbreviations

Abbreviation	Description
ASO	Arbitrary Slice Ordering
AVC	Advanced Video Coding
BIOS	TI's simple RTOS for DSPs
CAVLC	Context Adaptive Variable Length Coding
CABAC	Context Adaptive Binary Arithmetic Coding
D1	720x480 or 720x576 resolutions in progressive scan
DCT	Discrete Cosine Transform
DDR	Double Data Rate
DMA	Direct Memory Access
FC	Framework components

Abbreviation	Description
FMO	Flexible Macro-block Ordering
HD 720 or 720p	1280x720 resolution in progressive scan
HDTV	High Definition Television
HDVICP	High Definition Video and Imaging Co-processor sub-system
IDR	Instantaneous Decoding Refresh
ITU-T	International Telecommunication Union
JM	Joint Menu
JVT	Joint Video Team
MB	Macro Block
MBAFF	Macro Block Adaptive Field Frame
MJCP	MPEG JPEG Co-Processor
MPEG	Motion Pictures Expert Group
MV	Motion Vector
NAL	Network Adaptation 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
RTOS	Real Time Operating System
RMAN	Resource Manager
SEI	Supplemental Enhancement Information
SPS	Sequence Parameter Set
VGA	Video Graphics Array
VICP	Video and Imaging Co-Processor
XDAIS	eXpressDSP Algorithm Interface Standard

Abbreviation	Description
XDM	eXpressDSP Digital Media
YUV	Color space in luminance and chrominance form

Note:

MJCP and VICP refer to the same hardware co-processor blocks.

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 High Profile Encoder on DM365) and version number. The version number of the codec is included in the Title of the Release Notes that accompanies this codec.

Trademarks

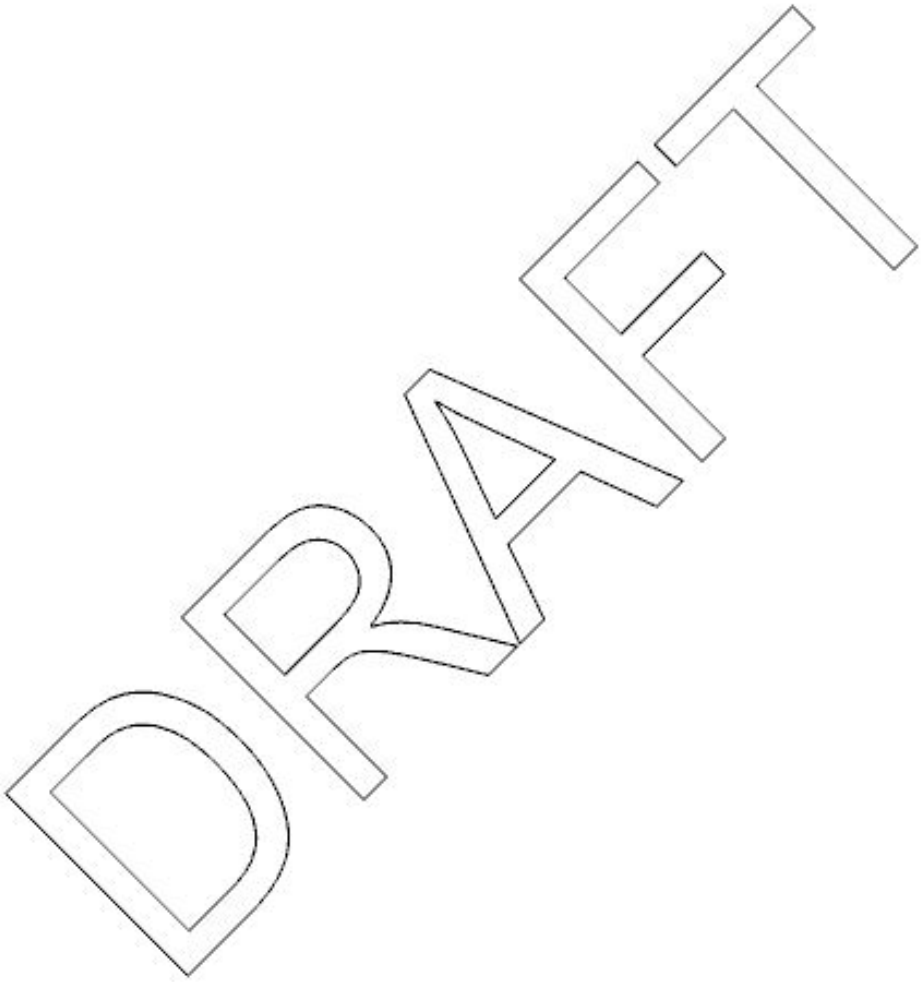
Code Composer Studio, DSP/BIOS, eXpressDSP, TMS320, TMS320C64x, TMS320C6000, TMS320DM644x, and TMS320C64x+ are trademarks of Texas Instruments.

All trademarks are the property of their respective owners.

Contents

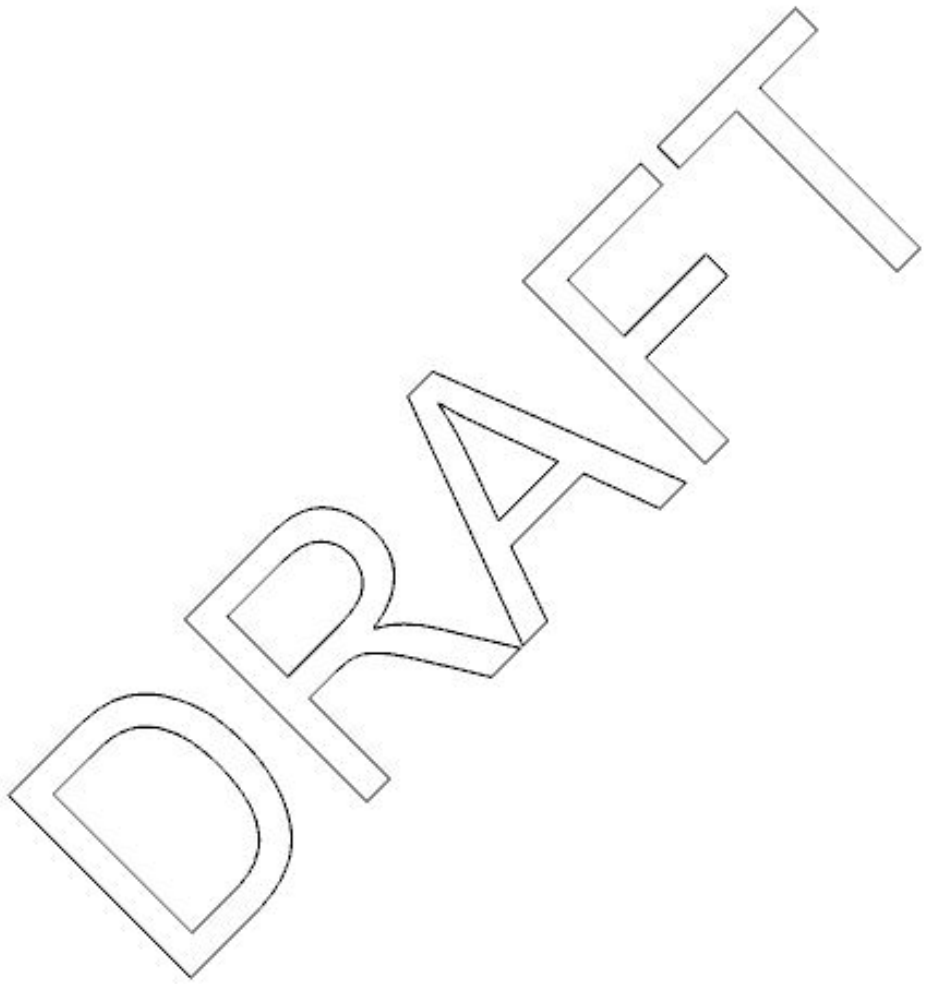
Figures

This page is intentionally left blank



Tables

This page is intentionally left blank



Chapter 1

Introduction

This chapter provides a brief introduction to XDAIS, XDM, and DM365 software architecture. It also provides an overview of TI's implementation of the H.264 High Profile Encoder on the DM365 platform and its supported features.

Topic **Page**

1.1 Software Architecture

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

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

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

Following diagram shows the software architecture.

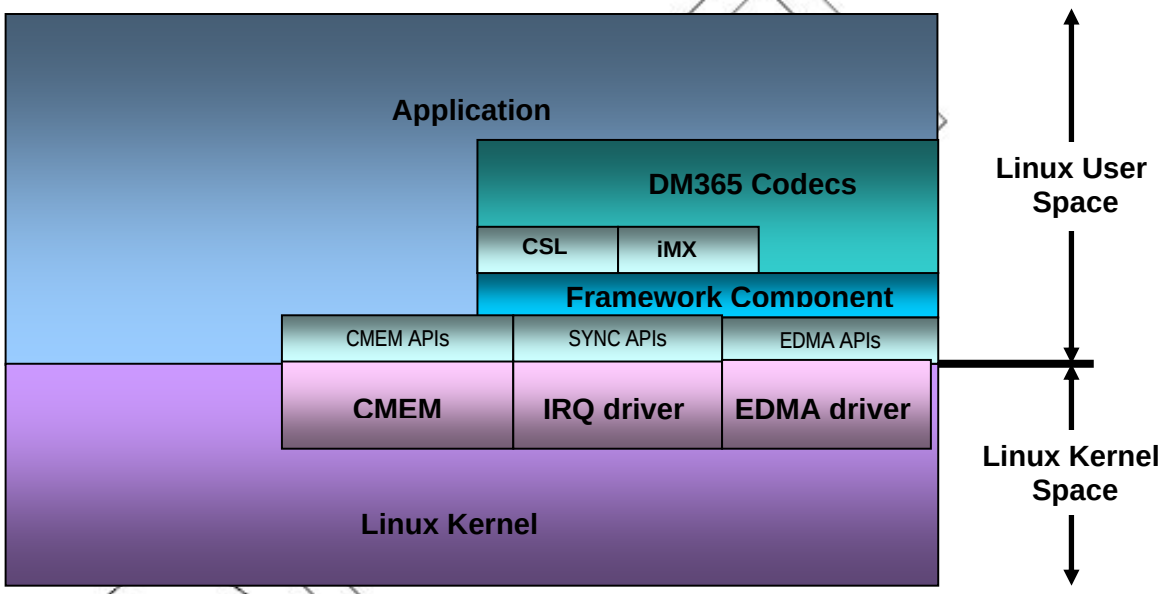


Figure 1-1. Software Architecture.

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 share memory between algorithms. It also allows the memory to be moved around while an algorithm is operating in the system. In order to facilitate these functionalities, the IALG interface defines the following APIs:

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

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

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

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

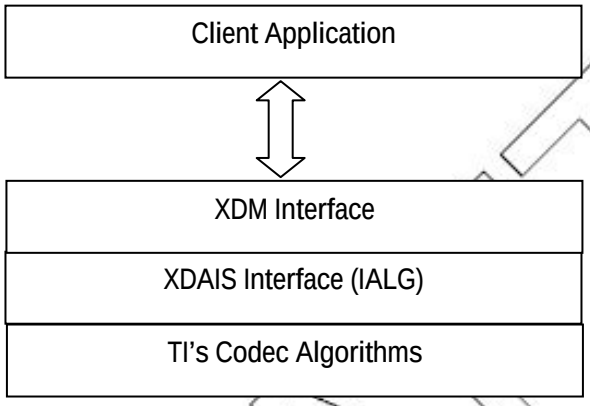
- `control()`
- `process()`

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, 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 encoder task when the frame level computation is happening on coprocessor.

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

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



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

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

1.2.3 Framework Component

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

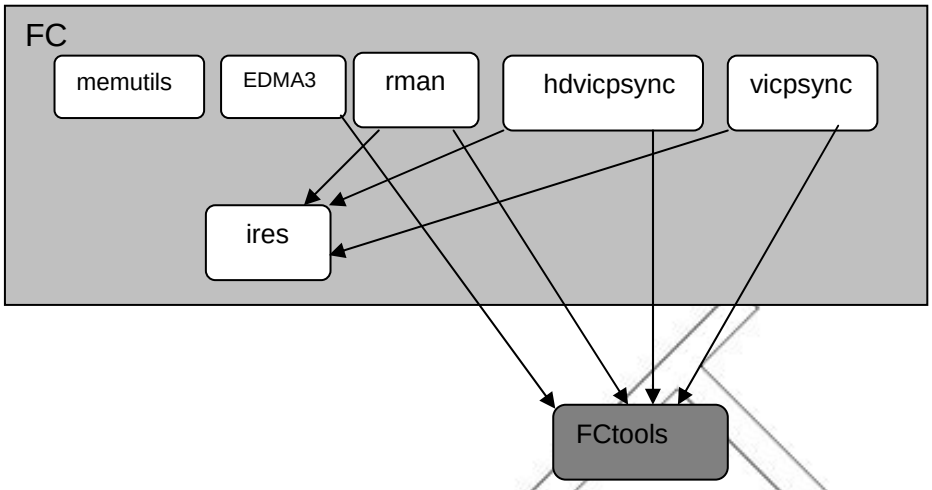


Figure 1-2. Framework Component Interfacing Structure.

Each component is explained in detail in the following sections.

1.2.3.1 IRES and RMAN

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 agrees on the concrete IRES resource types that are 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 include the following modules and interfaces to support algorithms requesting IRES-based resources:

- **IRES** - Standard interface allowing the client application to query and provide the algorithm with its requested IRES resources.
- **RMAN** - Generic IRES-based resource manager, which 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).

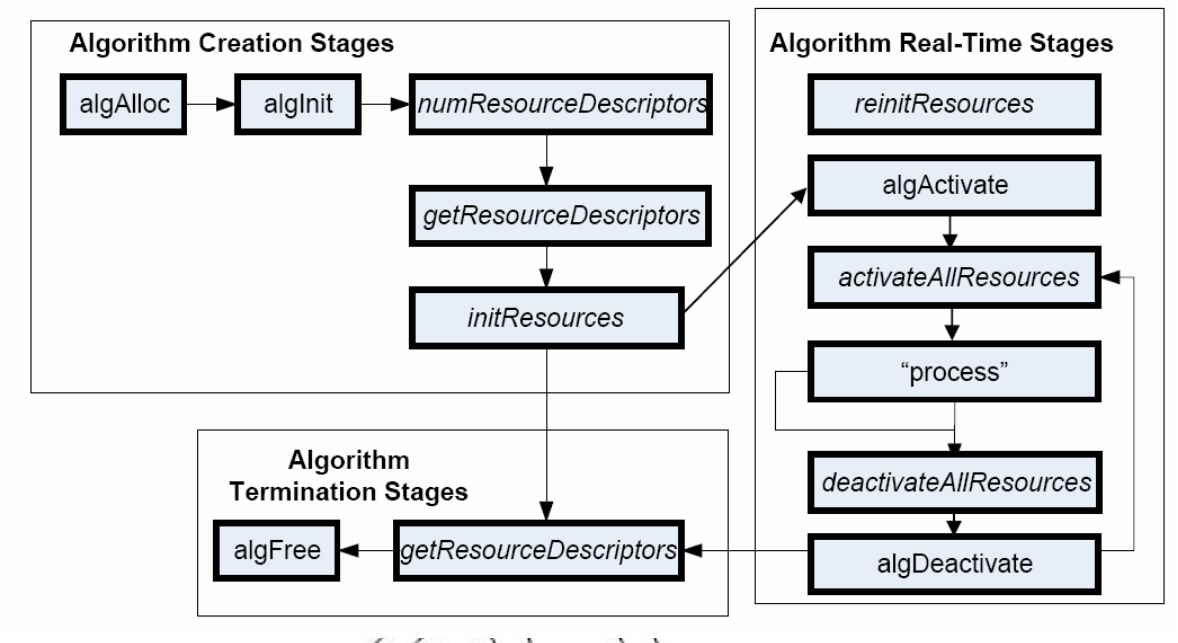


Figure 1-3. IRES Interface Definition and Function-calling Sequence.

1.2.3.2 HDVICP

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

1.2.3.3 EDMA3

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

Note:

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

1.2.3.4 VICP

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

1.2.3.5 HDVICP Sync

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

1.2.3.6 Memutils

This is 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 invalidate cache
- `getPhysicalAddr` – Obtains physical (hardware specific) address

1.3 Overview of H.264 High Profile Encoder

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

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

- Baseline Profile:
  - Only I and P type slices are present
  - Only frame mode (progressive) picture types are present
  - Only CAVLC is supported
  - ASO/FMO and redundant slices for error concealment is supported
- High Profile:
  - Only I, P, and B type slices are present

- Frame and field picture modes (in progressive and interlaced modes) picture types are present
- Both CAVLC and CABAC are supported
- ASO is not supported
- Transform 8x8 is supported
- Sequence scaling list is supported.

The input to the encoder is a YUV sequence, which can be of format 420 with the chroma components interleaved in little endian. The output of the encoder is an H.264 encoded bit-stream in the byte-stream syntax. The byte-stream consists of a sequence of byte-stream NAL unit syntax structures. Each byte-stream NAL unit syntax structure contains one start code prefix of size four bytes and value 0x00000001, followed by one NAL unit syntax structure. The encoded frame data is a group of slices, each is encapsulated in NAL units. The slice consists of the following:

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

The first frame is called Instantaneous Decode Refresh (IDR) picture frame. The decoder at the receiving end 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.

In motion estimation, the encoder searches for the best match in the available reference frame(s). After quantization, contents of some blocks become zero. H.264 Encoder tracks this information and passes the information of coded 4x4 blocks to inverse transform so that it can skip computation for those blocks that contain all zero co-efficients and are not coded.

H.264 Encoder defines in-loop filtering to avoid blocks across the 4x4 block boundaries. It is the second most computational task of H.264 encoding process after motion estimation. In-loop filtering is applied on all 4x4 edges as a post-process and the operations depend upon the edge strength of the particular edge.

H.264 Encoder applies entropy coding methods to use context based adaptivity, which in turn improves the coding performance. All the macro blocks, which belong to a slice, must be encoded in a raster scan order. Baseline profile uses the Context Adaptive Variable Length Coding (CAVLC). CAVLC is the stage where transformed and quantized coefficients are entropy coded using context adaptive table switching across different symbols. The syntax defined by the H.264 Encoder stores the information at 4x4 block level.

The following figure depicts the working of the encoder.

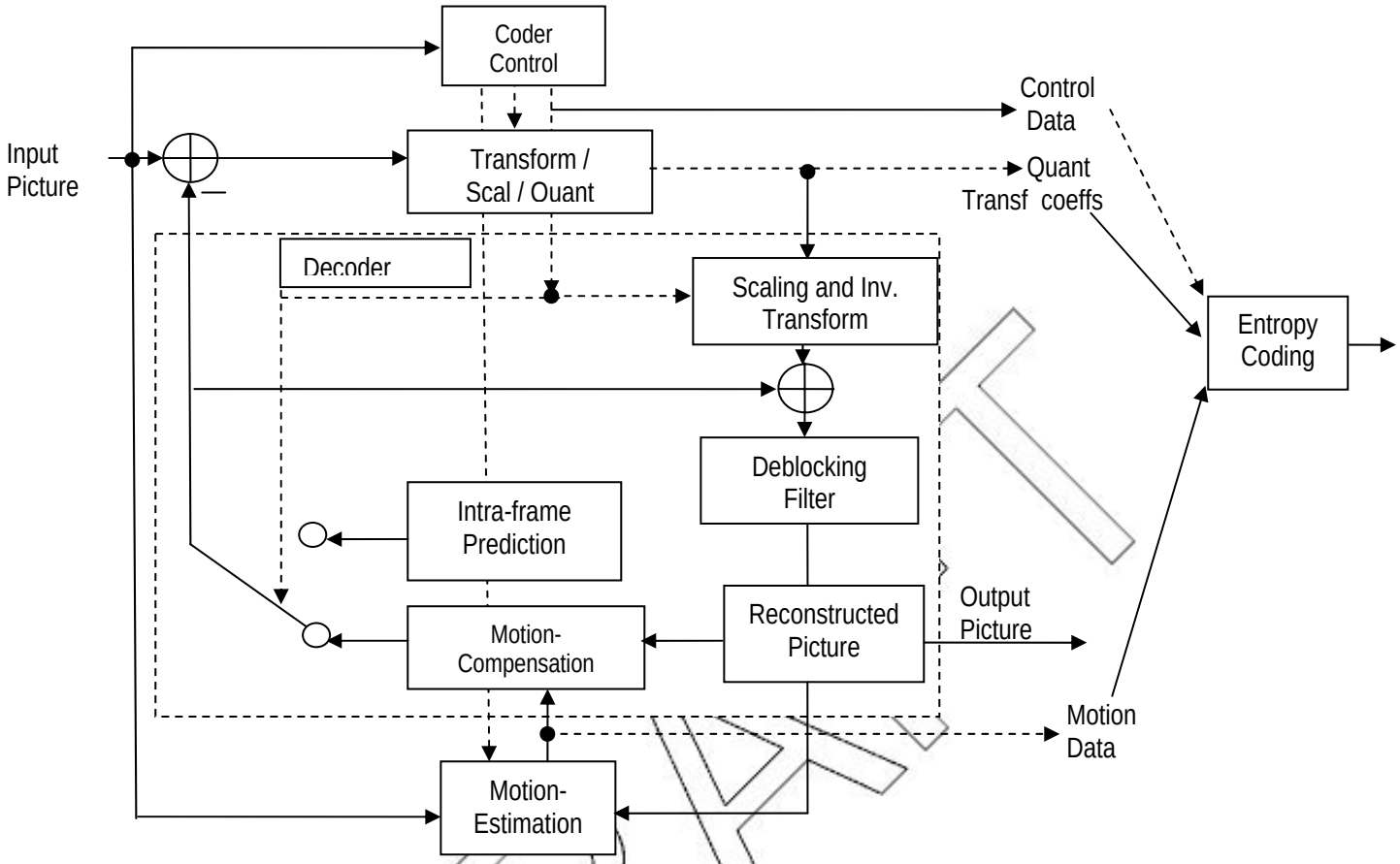


Figure 1-4. Block Diagram of H.264 Encoder.

From this point onwards, all references to H.264 Encoder mean H.264 High Profile (HP) Encoder only.

1.4 Supported Services and Features

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

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

- eXpressDSP Digital Media (XDM1.0 IVIDENC1) interface compliant
- Compliant with H.264 High Profile up to level 3.1
- Supports resolutions up to 720p(1280 x 720)
- Supports YUV420 semi planer input format for the frames
- Supports progressive and interlaced encoding
- Generates bit-stream compliant with H.264 standard

- Supports CAVLC and CABAC encoding
- Supports 16x16, 8x16, 16x8 and 8x8 MB partitions
- Supports sequence scaling matrix
- Supports transform 8x8 and transform 4x4
- Supports frame based encoding with frame size being multiples of 2
- Supports rate control (CBR and VBR)
- Supports Insertion of Buffering Period and Picture Timing Supplemental Enhancement Information (SEI) and Video Usability Information (VUI)
- Supports Unrestricted Motion Vectors (UMV)
- Supports Half Pel and Quarter Pel Interpolation for motion estimation

DM365 H.264 encoder can be configured in two modes:

- Standard quality, standard feature, which gives performance of 720P@30fps
- High quality, full feature, which is for D1@30fps.

Supported features in High quality mode:

- Supports TI's proprietary motion estimation supported (2 types of search algorithms
- Supports all 16x16, 8x8 and 4x4 intra-prediction modes
- Supports multiple slice encoding upto 720p resolution (for CAVLC only)
- Supports 4-motion vector per macroblock till 720p resolution
- Supports Adaptive Intra Refresh (AIR)

Supported features in standard quality mode:

- Supports TI's proprietary motion estimation supported (Low power ME)
- Supports all 16x16, 8x8 and 4x4 intra-prediction modes supported in I-Frame and INTRA16x16 DC is supported in P-frames
- Supports only single slice per frame
- Supports only single motion vector per macroblock

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

- Error Resilience features such as ASO/FMO and redundant slices
- Adaptive Reference Picture Marking
- Reference Picture List Reordering

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 NO-OS Standalone

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

2.1.1 Hardware

- DM365 EVM (Set the bits 2 and 3 of switch SW4 to high(1) position; Set the bits 4 and 5 of SW5 to high(1) position)
- XDS560R JTAG

2.1.2 Software

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

- **Development Environment:** This project is developed using Code Composer Studio version 3.3.81.6 (Service Release-11)
- **Code Generation Tools:** This project is compiled, assembled, archived, and linked using the TI ARM code generation tools
- DM365 functional simulator

2.2 System Requirements for Linux

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

2.2.1 Hardware

- DM365 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

2.2.2 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 tool chain.
- **ARM Tool Chain**: This project is compiled and linked using MVL ARM tool chain.

2.3 Installing the Component for NO-OS Standalone

The codec component is released as a compressed archive. To install the codec, extract the contents of the zip file onto your local hard disk. The zip file extraction creates a directory called 210_V_H264AVC_E_01_00. Figure 2-1. shows the sub-directories created in this directory.

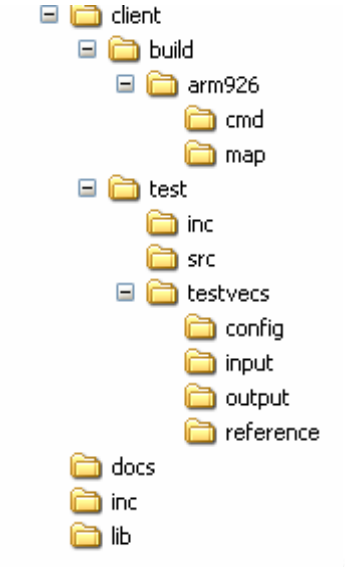


Figure 2-1. Component Directory Structure for Standalone

Table 2-1 provides a description of the sub-directories created in the 210_V_H264AVC_E_01_00 directory.

Table 2-1. Component Directories for Standalone

Sub-Directory	Description
\inc	Contains XDM related header files which allow interface to the codec library
\lib	Contains the codec library file on host
\docs	Contains user guide and release notes
\client\build\arm926	Contains the sample test application project (.pjt) file to be used on host (ARM926) side
\client\build\arm926\cmd	Contains command file for compilation of the code on host side
\client\build\arm926\map	Contains the memory map generated on compilation of the code
\client\test\src	Contains application C files
\client\test\inc	Contains header files needed for the application code
\client\test\testvecs\input	Contains input test vectors

Sub-Directory	Description
\client\test\testvecs\output	Contains output generated by the codec
\client\test\testvecs\reference	Contains read-only reference output to be used for verifying against codec output
\client\test\testvecs\config	Contains configuration parameter files

2.4 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_h264enc_01_00_00_production. Figure 2-2 shows the sub-directories created in this directory.

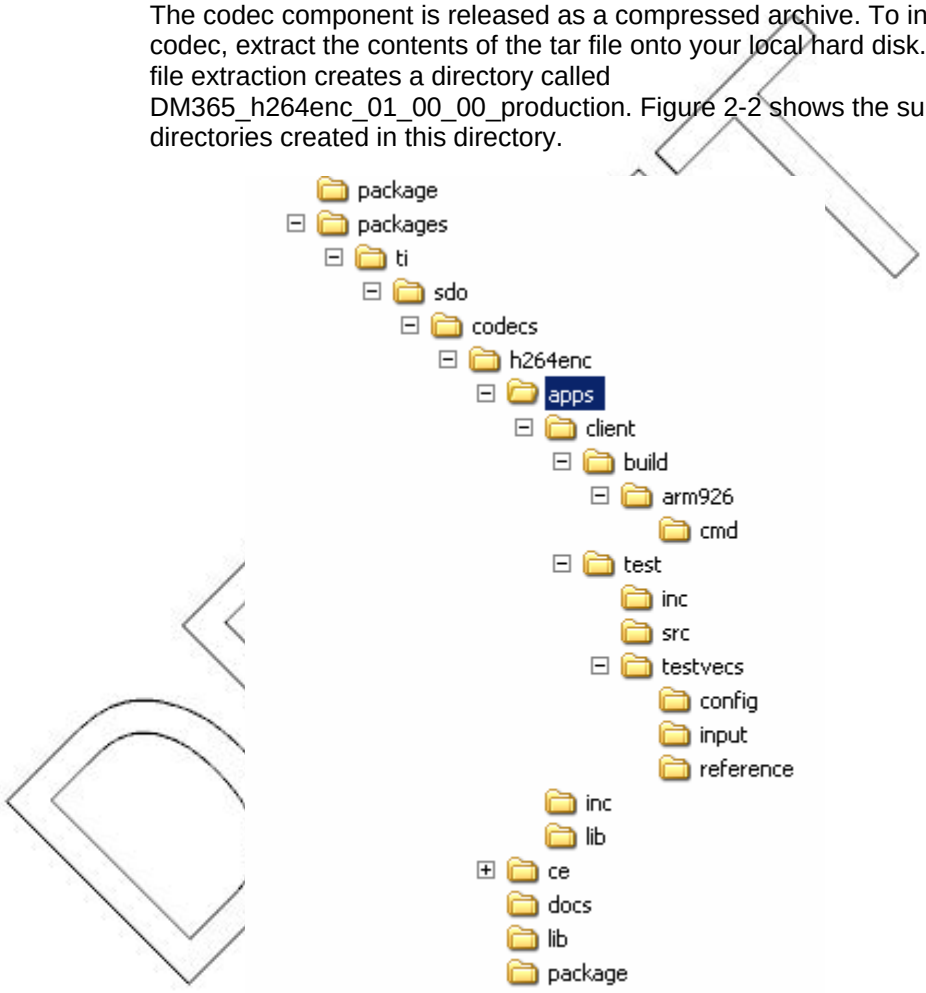


Figure 2-2. Component Directory Structure for Linux.

Table 2-1 provides a description of the sub-directories created in the DM365_h264enc_01_00_00_production directory.

Table 2-2. Component Directories for Linux.

Sub-Directory	Description
\package	Contains files related while building the package
\packages\ti\sdo\codecs\h264enc\lib	Contains the codec library files on host
\packages\ti\sdo\codecs\h264enc\docs	Contains user guide and release notes
\packages\ti\sdo\codecs\h264enc\apps\client\build\arm926	Contains the makefile to built sample test application
\packages\ti\sdo\codecs\h264enc\apps\client\build\arm926\cmd	Contains a template (.xdt) file to used to generate linker command file
\packages\ti\sdo\codecs\h264enc\apps\client\build\arm926\map	Contains the memory map generated on compilation of the code
\packages\ti\sdo\codecs\h264enc\apps\client\test\src	Contains application C files
\packages\ti\sdo\codecs\h264enc\apps\client\test\inc	Contains header files needed for the application code
\packages\ti\sdo\codecs\h264enc\apps\client\test\testvecs\input	Contains input test vectors
\packages\ti\sdo\codecs\h264enc\apps\client\test\testvecs\output	Contains output generated by the codec
\packages\ti\sdo\codecs\h264enc\apps\client\test\testvecs\reference	Contains read-only reference output to be used for verifying against codec output
\packages\ti\sdo\codecs\h264enc\apps\client\test\testvecs\config	Contains configuration parameter files

2.5 Building the Sample Test Application for EVM Standalone

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

Note:

Sample test application can be built either on QT Code Composer Studio configuration ,on DM365 EVM standalone or DM365 EVM LINUX. The build steps are same for both. The steps specified in this section are explained assuming simulator configuration.

1) Verify that you have an installation of TI's Code Composer Studio version 3.3.81.6 (Service Release-11) and code generation tools as provided in the Release Note.

2) Verify the SDXDS560R JTAG driver installation version 30329A

3) Check SW4 and SW5 switch positions of the DM365 EVM. Bit 2 and 3 of SW4 should be set to 1 and remaining should be set to 0.

4) Open Setup Code Composer Studio v3.3.

5) Select **File > Import** and browse for the .ccs file and add it.

6) Save the configuration and exit from setup Code Composer Studio. **PDM** opens and displays both ARM926 and ARM968 processors.

7) Right click on ARM926 and connect.

8) Double-click ARM926 to launch Code Composer Studio IDE for the host processor.

9) Verify if the codec object library h264venc_ti_arm926.lib exists in the \lib sub-directory.

10) Verify if the codec DMA object library dma_ti_dm365.lib exists in the \lib sub-directory.

11) Ensure that you have installed the XDC and Framework components releases with version numbers that are mentioned in the release notes.

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

13) Ensure that the installed XDC directory is in the general search PATH.

14) Open the MS-DOS command prompt at the directory \client\build\arm926 sub-directory of the release folder.

15) Type the command **gmake** at the prompt and this generates an executable file, h264venc_ti_arm926testapp.out in the \client\build\arm926\out sub-directory.

2.6 Running the Sample Test Application on EVM Standalone

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

1) Verify that you have an installation of TI's Code Composer Studio version 3.3.81.6 with Service Release 11 and code generation tools as provided in the Release Note.

2) Verify the SDXDS560R JTAG driver installation version 30329A.

3) Check SW4 and SW5 switch positions of the DM365 EVM. Bit 2 and 3 of SW4 should be set to 1 and remaining should be set to 0. All bits should be set to 0 for SW5.

4) Open Setup Code Composer Studio version 3.3.

5) Select **File > Import**, browse for the .ccs file, and add it.

6) Save the configuration and exit from setup Code Composer Studio. **PDM** opens and displays both ARM926 and ARM968 processors.

7) Right click on ARM926 and connect.

8) Double-click ARM926 to launch Code Composer Studio IDE for the host processor.

9) Add the GEL file and initialize it properly

10) Select **File > Load Program** in Host Code Composer Studio, browse to the \client\build\arm926\out\ sub-directory, select the codec executable created in step 14 in Section 2.5, and load it into Code Composer Studio in preparation for execution.

11) Select **Debug > Run** in Host Code Composer Studio to execute encoder on host side.

12) Double-click **ARM968** in PDM to launch Code Composer Studio IDE for the local processor. Wait until the message “Control is given to ARM968” is displayed on stdout.

13) Select **Debug > Run** on the coprocessor Code Composer Studio. In case a warning dialog box appears, ignore the warning message and click **OK**.

    The sample test application takes the input files stored in the \client\test\testvecs\input sub-directory, runs the codec, and stores the output in \client\test\testvecs\output sub-directory.

    For each encoded frame, the application displays a message indicating the frame number and the bytes generated.

    After the encoding is complete, the application displays a summary of total number of frames encoded.

14) Halt the coprocessor from Code Composer Studio IDE.

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

6) 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.

To run the executable generated from the above steps:

1) Load the kernel modules by typing the command ./loadmodules.sh which initializes the CMEM pools.

2) Now branch to the directory where the executable is present and type ./h264venc-r in the command window to run.

2.8 Configuration Files

This codec is shipped along with:

- Generic configuration file (testvecs.cfg) – list of configuration files for running the codec on sample test application.
- Encoder configuration file (testparams.cfg) – specifies the configuration parameters used by the test application to configure the Encoder.

2.8.1 Generic Configuration File

The sample test application shipped along with the codec uses the configuration file, Testvecs.cfg for determining the input and reference files for running the codec and checking for compliance. The testvecs.cfg file is available in the \packages\ti\sdo\codecs\h264enc\apps\client\test\testvecs\config sub-directory.

The format of the testvecs.cfg file is:

```
X
config
input
output/reference
recon
```

where:

- `X` may be set as:
  - 1 - for compliance checking, no output file is created
  - 0 - for writing the output to the output file
- `config` is the Encoder configuration file. For details, see Section 2.8.2.
- `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) (use complete path).
- `recon` is reconstructed YUV output file name (use complete path).

A sample testvecs.cfg file is as shown:

For output dump mode:

```
0
..\..\..\test\testvecs\config\testparams.cfg
..\..\..\test\testvecs\input\input.yuv
..\..\..\test\testvecs\output\output.264
..\..\..\test\testvecs\output\recon.yuv
```

For reference bit-stream compliance test mode:

```
1
..\..\..\test\testvecs\config\testparams.cfg
..\..\..\test\testvecs\input\input.yuv
..\..\..\test\testvecs\reference\reference.264
..\..\..\test\testvecs\output\recon.yuv
```

2.8.2 Encoder Configuration File

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

A sample Testparams.cfg file is as shown:

```
# Config File Format is as follows

# <ParameterName> = <ParameterValue> # Comment

########################################################

Parameters

########################################################

ImageWidth = 1280   # Image width in Pels, must
                      be multiple of 16

ImageHeight = 720    # Image height in Pels, must
                      be multiple of 16

FrameRate   = 30000  # Frame Rate per second*1000
                      (1-100)

BitRate    = 4000000  # Bitrate(bps)  #if ZERO=>> RC
                      is OFF

ChromaFormat  = 9  # 9 => XDM_YUV_420P

InterlacedVideo = 0  # 0: Progressive, 1 :Interlaced
TimerScale = 60.     # Timer Resolution for Picture
                      Timing

NumUnitsInTicks = 1  # Number of Timer units per
Tick

AspectRatioWidth = 1   # Aspect Ratio Width Scale

AspectRatioHeight = 1  # Aspect Ratio Height Scale
```

```
PixelRange      = 1    # 1 =>Y- 0 to 255, Cb/Cr-0 to
                           255
                         0 => Y-16 to 235, Cb/Cr-16
                           to 240

EnableVUIParam  = 1    # 1 => Enable VUI parameters,
                         0 => Disable VUI Parameters

EnableBufSEI    = 1 # 1 => Enable Buffering Period
                           SEI Message,
                       0 => Disable

ME_Type         = 0           # ME search algorithm
                                0 => Normal,
                                1 => Low Power


RC_PRESET       = 5           # 1 => Low Delay,
                                2 => Storage,
                                3 => 2 Pass,
                                4 => None,
                                5 => user defined


ENC_PRESET      = 3           # 3 => User Defined
Parameters

#######################################################

                  # Encoder Control

#######################################################

ProfileIDC      = 66    # Profile IDC (66=baseline,
77=main, 88=extended, 100=highprofile)

LevelIDC     = 30  # Level IDC (e.g. 20 = level 2.0)

IntraPeriod     = 30          # Period of I-Frames

IDRFramePeriod  = 0           # Period of IDR Frames

FramesToEncode  = 10          # Number of frames to
                              be coded

SliceSizeInBits  = 2000       # Size of each slice
                              in bits

EnMeMultiPart   = 1           # 1 => Enable MB
                              Partitions,
                              0=> Single MV for each
                              MB

RateControl     = 0           # 0 => CBR,
                               1 => VBR,

                               2 = Fixed QP

MaxDelay        = 1000        # Delay Parameter for
                              Rate Control in
```

```
                                Milliseconds

QPInit          = 28            # Initial QP for RC
                                (0-51)

QPISlice        = 48            # Quant. param for I
                                Slices (0-51)

QPSlice         = 48            # Quant. param for non
                                   - I slices (0-51)

MaxQP           = 51        # Maximum value for QP
                  (0-51)

MinQP           = 0         # Minimum value for QP
                  (0-51)

IntraThrQF      = 5         # Intra Thresholding QF
                  (0-5)

AirRate         = 20        # Number of Forced Intra
MBs

UnRestrictedMV  = 1         #1: Enable 0:Disable

EntropyCodingMode = 1       # Entropy Coding Mode
(0 = CAVLC, 1 = CABAC)

Transform8x8FlagIntra = 1  # 0 = Disable, 1 = Enable

Transform8x8FlagInter = 1  # 0 = Disable, 1 = Enable

SequenceScalingFlag = 0    # 0 = Disable, 1 = Enable

PerceptualRC    = 1         # 1 => Enable Perceptual
QP modulation, 0 => Disable

EncoderQuality  = 1         # 0 => Standard Quality,
1 => High Quality

#########################################################
               Loop filter parameters

#########################################################

LoopFilterDisable = 0   # Disable loop filter in
slice header(0=Filter,1=No Filter, 2 = Disable
across Slice Boundaries)
```

To check the functionality of the codec for the inputs other than those provided with the release, change the configuration file accordingly, and follow the steps as described in Section 2.4.

2.8.3 Encoder Sample Base Param Setting

The encoder can be run in IVIDENC1 base class setting. The extended parameter variables of encoder will then assume default values. The following list provides the typical values of IVIDENC1 base class variables.

```
typedef struct IVIDENC1_Params {

XDAS_Int32 size;

XDAS_Int32 encodingPreset = XDM_HIGH_SPEED; // Value
= 2

XDAS_Int32 rateControlPreset = IVIDEO_STORAGE;
//value = 2

XDAS_Int32 maxHeight =  720;

XDAS_Int32 maxWidth =  1280;

XDAS_Int32 maxFrameRate = 30000;

XDAS_Int32 maxBitRate = 10000000;

XDAS_Int32 dataEndianness = XDM_BYTE;

XDAS_Int32 maxInterFrameInterval = 1;

XDAS_Int32 inputChromaFormat = XDM_YUV_420SP;
//value = 9

XDAS_Int32 inputContentType = IVIDEO_PROGRESSIVE;

XDAS_Int32 reconChromaFormat  XDM_YUV_420SP;
//value = 9;

} IVIDENC1_Params;


typedef struct IVIDENC1_DynamicParams {

XDAS_Int32 size;            /**< @sizeField */

XDAS_Int32 inputHeight;     /**< Input frame height.
*/

XDAS_Int32 inputWidth;      /**< Input frame width.
*/

XDAS_Int32 refFrameRate = 30000;

XDAS_Int32 targetFrameRate  = 3000;

XDAS_Int32 targetBitRate;  < 10000000 /**< Target
bit rate in bits per second. */

XDAS_Int32 intraFrameInterval = 29;

XDAS_Int32 generateHeader = 0;
```

```
XDAS_Int32 captureWidth; // for demo, same as
inputWith

XDAS_Int32 forceFrame;  = IVIDEO_NA_FRAME

XDAS_Int32 interFrameInterval = 1;

XDAS_Int32 mbDataFlag = 0;

} IVIDENC1_DynamicParams;


typedef struct IVIDENC1_InArgs {

XDAS_Int32 size;            /**< @sizeField */

XDAS_Int32 inputID;  /* as per application*/

XDAS_Int32 topFieldFirstFlag = 0;

} IVIDENC1_InArgs;
```

2.9 Standards Conformance and User-Defined Inputs

To check the reference bit-stream conformance of the codec for the default input file shipped along with the codec, follow the steps as described in Section 2.6 or 2.7.

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

1) Copy the input files to the \client\test\testvecs\input sub-directory.

2) Copy the reference files to the \client\test\testvecs\reference sub-directory.

3) Edit the configuration file, Testvecs.cfg available in the \client\test\testvecs\config sub-directory. For details on the format of the testvecs.cfg file, see section 2.8.

   For each encoded frame, the application displays the message indicating the frame number. In reference bit-stream compliance check mode, the application additionally displays FAIL message, if the bit-stream does not match with reference bit-stream.

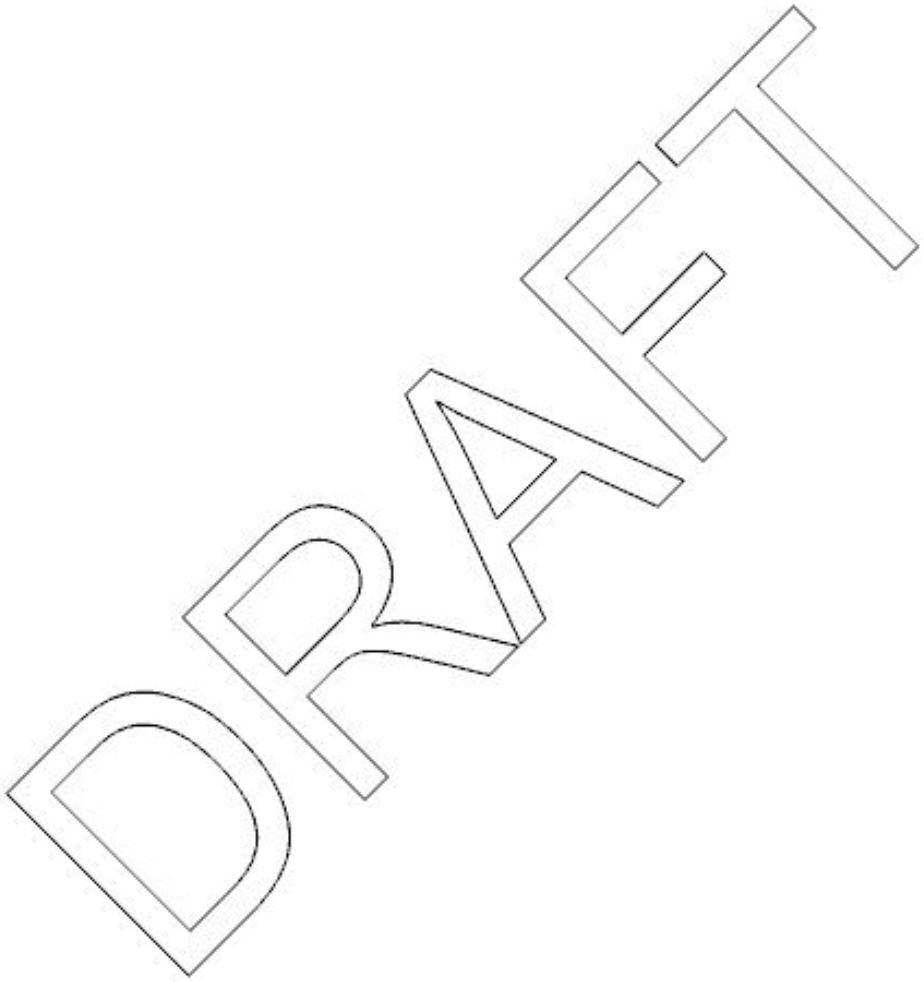
   After the encoding is complete, the application displays a summary of total number of frames encoded. In reference bit-stream compliance check mode, the application additionally displays PASS message, if the bit-stream matches with the reference bit-stream.

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

2.10 Uninstalling the Component

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

This page is intentionally left blank



Chapter 3

Sample Usage

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

3.1 Overview of the Test Application

The test application exercises the `IVIDENC1` base class of the H.264 Encoder library. The main test application files are h264encoderapp.c and h264encoderapp.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.

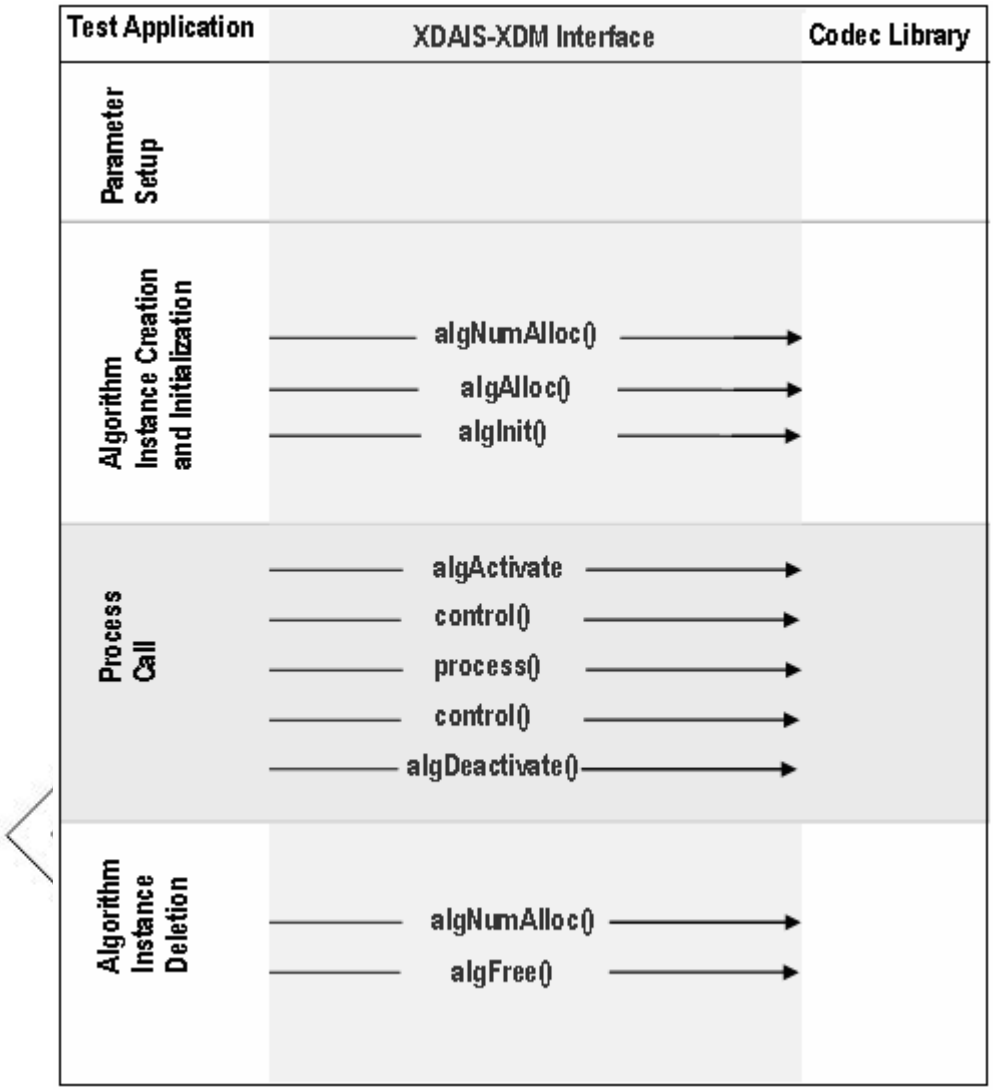


Figure 3-1. Test Application Sample Implementation

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 Encoder configuration files.

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

1) Opens the generic configuration file, testvecs.cfg and reads the list of Encoder configuration file name (testparams.cfg).

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

   For more details on the configuration files, see Section 2.8.

3) Sets the `IVIDENC1_Params` structure based on the values it reads from the Testparams.cfg file.

4) Sets the extended parameters of the `IH264VENC_Params` structure based on the values it reads from the testparams.cfg file.

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

3.1.2 Algorithm Instance Creation and Initialization

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

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

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

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

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

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

Note:

DMAN3 function and IDMA3 interface is not implemented in DM365 codecs. Instead, it uses a DMA resource header file, which 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 4.2.1.10). The inputs to the `process()` functions are input and output buffer descriptors, pointer to the `IVIDENC1_InArgs` and `IVIDENC1_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, which performs book-keeping operations and updates the output parameters structure -`IVIDENC1_OutArgs`.

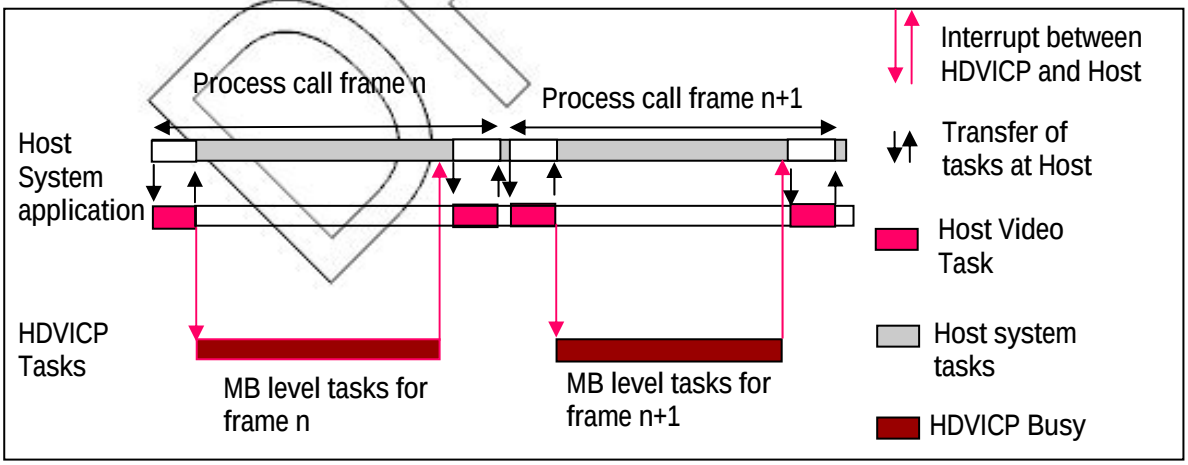


Figure 3-2. Process Call with Host Release

Note:

- The process call returns control to the application after the initial setup related tasks are completed.
- Application can schedule a different task to use the Host resource released free.
- All service requests from HDVICP are handled through interrupts.
- Application resumes the suspended process call after handling the last service request for HDVICP.
- 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. The `algActivate()` and `algDeactivate()` XDAIS functions activate and deactivate the algorithm instance respectively. Once the algorithm is activated, the `control()` and `process()` functions can be of any order. The following APIs are called in a sequence:

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

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

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

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

The for loop encapsulates frame level `process()` call and updates the input buffer and the output buffer pointer every time before the next call. The for loop runs for the designated number of frames and breaks-off whenever an error condition occurs.

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 Handshaking Between Application and Algorithm

3.2.1 Resource Level Interaction

Following diagram explains 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.

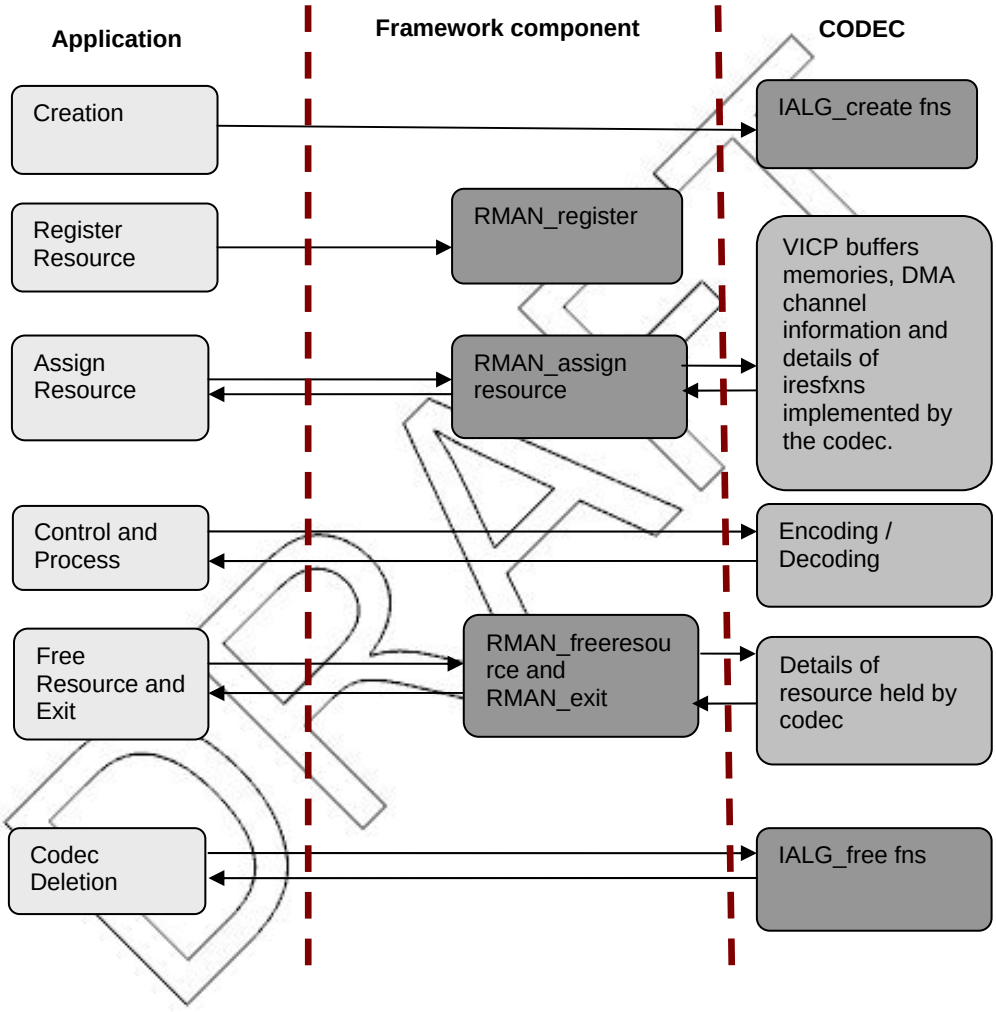


Figure 3-3. Resource Level Interaction.

3.2.2 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.

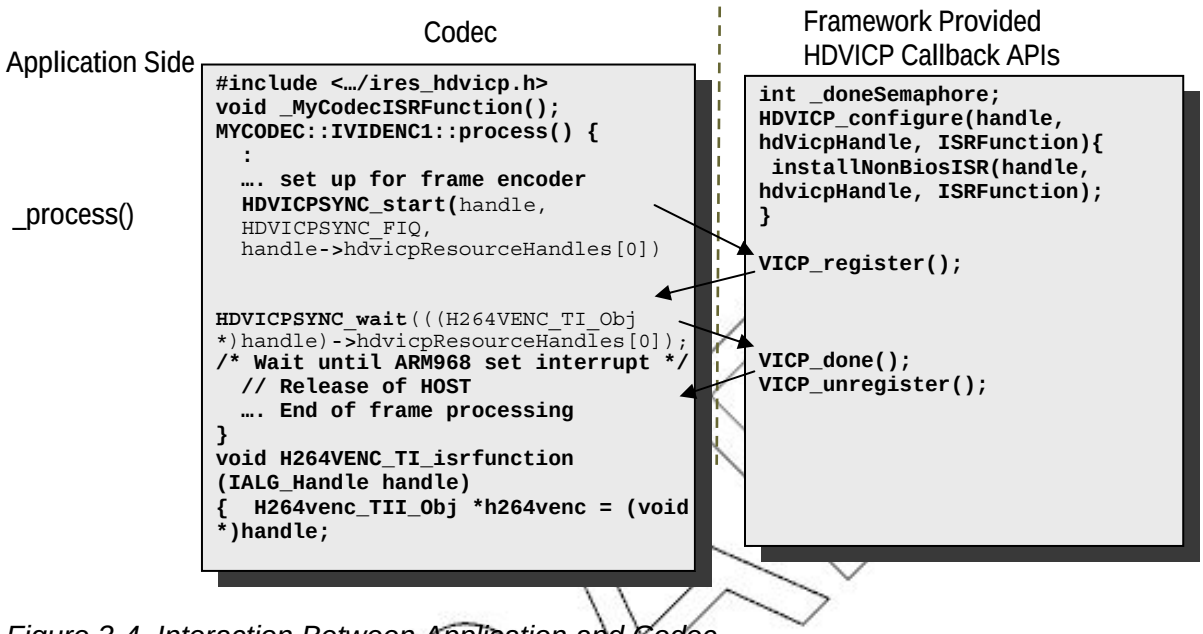


Figure 3-4. Interaction Between Application and Codec.

Note:

- Process call architecture shares Host resource among multiple threads.
- ISR ownership is with the FC resource 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 ARM968 to Host ARM926 is used to inform when the frame processing is done. HDVICP sends interrupt which maps to `INT No 10 (KALINT9 Video MJCP)` 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.

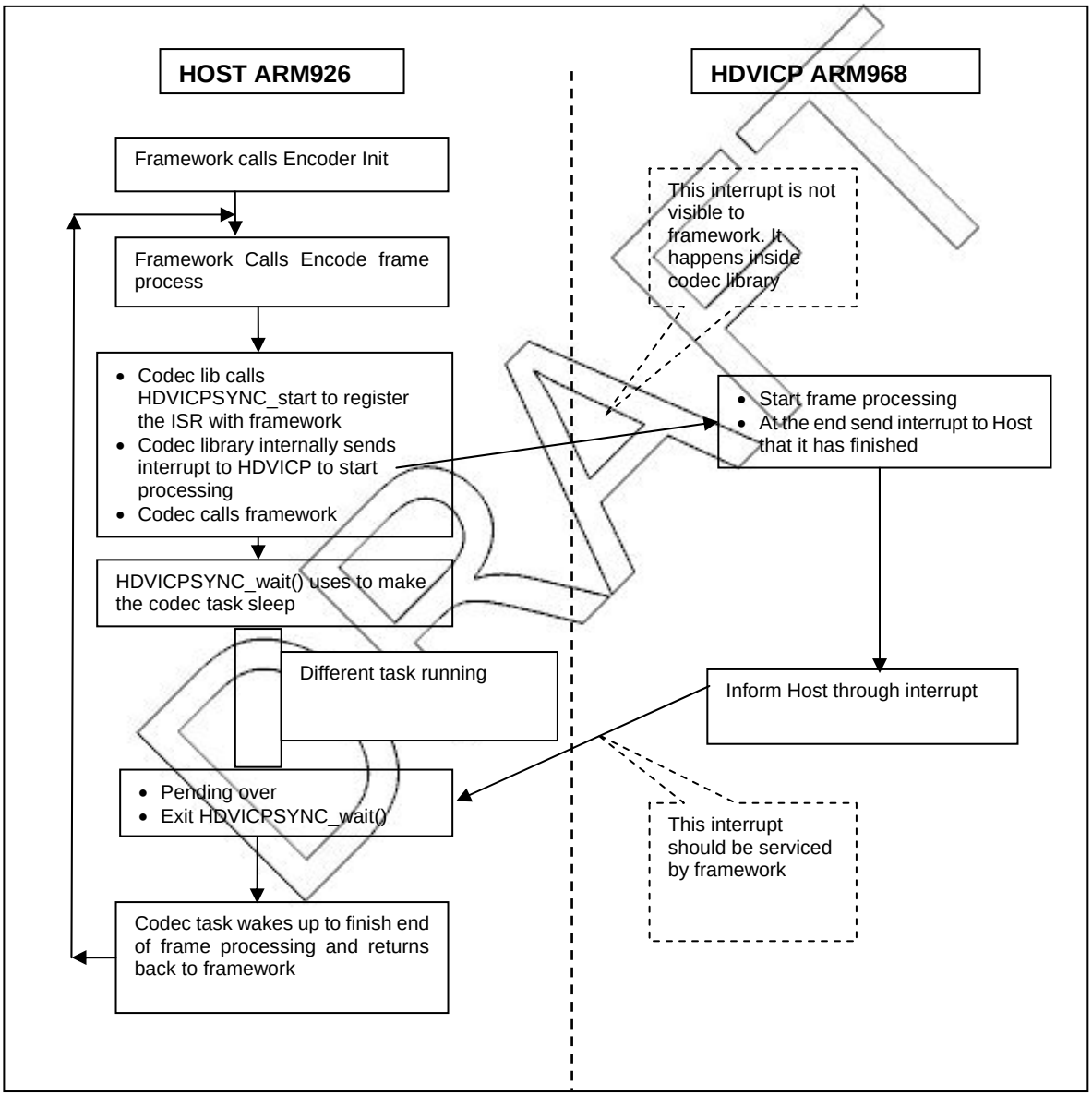


Figure 3-5. Interrupt Between Codec and Application.

3.3 Cache Management by Application

3.3.1 Cache Usage By Codec Algorithm

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

3.3.2 Cache Related Call Back Functions for Standalone

To resolve the cache coherency issues, codec library uses cache clean/flush call back functions. These functions are provided by the application and the codec uses it through a call back function wrapper. In case the application operates without cache, the application can make the call back functions dummy. DM365 codec software extends the HDVICP call back functions for this use.

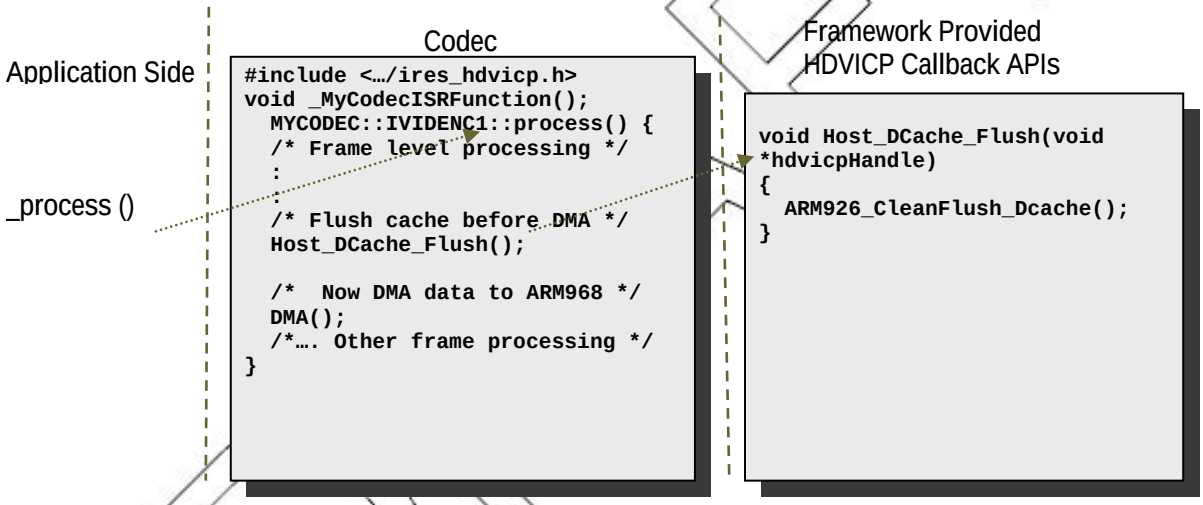


Figure 3-6. Cache Interaction Between Codec and Application.

3.3.3 Cache and Memory Related Call Back Functions for Linux

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

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

3.3.3.1 cacheInvalidate

In cache invalidation process, the entries of the cache are deleted. This API invalidates a range of cache.

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

3.3.3.2 cacheWb

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

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

3.3.3.3 cacheWbInv

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

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

3.3.3.4 getPhysicalAddr

This API obtains the physical address.

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

3.4 Sample Test Application

The test application exercises the `IVIDENC1` base class of the H.264 Encoder.

Table 3-1. process () Implementation

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

/*---------------- Encoder creation -----------------*/
handle = H264VENC_create(&fxns, &params)

/*Getting instance of algorithms that implements IALG and
IRES functions*/
iErrorFlag = RMAN_assignResources((IALG_Handle)handle,
                            &H264VENC_TI_IRES, /* IRES_Fxns* */
                              1 /* scratchId */);
/* Get Buffer information  */
iErrorFlag = H264VENC_control(
            handle,         // Instance Handle
            XDM_GETSTATUS,  // Command
            &dynamicparams, // Pointer to Dynamicparam structure
            &status         // Pointer to the status structure
        );
/*SET BASIC INPUT PARAMETERS */
iErrorFlag = H264VENC_control(
            handle,         // Instance Handle
            XDM_GETSTATUS,  // Command
            &dynamicparams, // Pointer to Dynamicparam structure
            &status         // Pointer to the status structure
        );
/* Based on the Num of buffers requested by the algorithm,
   the application will allocate for the same here
  */
AllocateH264IOBuffers(
            status, // status structure
            &inobj, // Pointer to Input Buffer Descriptor
&outobj) // Pointer to Output Buffer Descriptor
        );
/*Set Dynamic input parameters */
iErrorFlag = H264VENC_control(
            handle,         // Instance Handle
            XDM_GETSTATUS,  // Command
            &dynamicparams, // Pointer to Dynamicparam structure
            &status         // Pointer to the status structure
        );

/* for Loop for encode Call  for a given no of frames */
For(;;)
/* Read the input frame in the Application Input Buffer */
ReadInputData (inFile);
/*-----------------------------------------------------*/
/* Start the process : To start Encoding a frame      */
/* This will always follow a H264VENC_encode_end call */
```

```
/*-----------------------------------------------------*/

 iErrorFlag = H264VENC_encode (
                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 Encoder using control */
H264VENC_control(
                    handle,         // Instance Handle
                    XDM_GETSTATUS,  // Command - GET STATUS
                    &dynamicparams, // Input
                    &status         // Output
                );
     }
/* end of Do-While loop - which Encodes 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),
                    &H264VENC_TI_IRES, /* IRES_Fxns* */
                   );
/* Delete the encoder Object handle*/
H264VENC_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 shows the basic flow of the code.

Chapter 4

API Reference

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

4.1 Symbolic Constants and Enumerated Data Types

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

4.1.1 Common XDM Symbolic Constants and Enumerated Data Types

Table 4-1. List of Enumerated Data Types

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
IVIDEO_FrameType	IVIDEO_I_FRAME	Intra coded frame
	IVIDEO_P_FRAME	Forward inter coded frame
	IVIDEO_B_FRAME	Bi-directional inter coded frame. Not supported in this version of H.264 Encoder.
	IVIDEO_IDR_FRAME	Intra coded frame that can be used for refreshing video content
	IVIDEO_II_FRAME	Interlaced frame, both fields are I frames..
	IVIDEO_IP_FRAME	Interlaced frame, first field is an I frame, second field is a P frame.
	IVIDEO_IB_FRAME	Interlaced frame, first field is an I frame, second field is a B frame. Not supported in this version of H.264 Encoder.
	IVIDEO_PI_FRAME	Interlaced frame, first field is a P frame, second field is an I frame. Not supported in this version of H.264 Encoder.
	IVIDEO_PP_FRAME	Interlaced frame, both fields are P frames.
	IVIDEO_PB_FRAME	Interlaced frame, first field is a P frame, second field is a B frame. Not supported in this version of H.264 Encoder.
	IVIDEO_BI_FRAME	Interlaced frame, first field is a B frame, second field is an I frame. Not supported in this version of H.264 Encoder.
	IVIDEO_BP_FRAME	Interlaced frame, first field is a B frame, second field is a P frame. Not supported in this version of H.264 Encoder.

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	IVIDEO_BB_FRAME	Interlaced frame, both fields are B frames. Not supported in this version of H.264 Encoder.
	IVIDEO_MBAFF_I_FRAME	Intra coded MBAFF frame. Not supported in this version of H.264 Encoder.
	IVIDEO_MBAFF_P_FRAME	Forward inter coded MBAFF frame. Not supported in this version of H.264 Encoder.
	IVIDEO_MBAFF_B_FRAME	Bi-directional inter coded MBAFF frame. Not supported in this version of H.264 Encoder.
	IVIDEO_MBAFF_IDR_FRAME	Intra coded MBAFF frame that can be used for refreshing video content. Not supported in this version of H.264 Encoder.
	IVIDEO_FRAMETYPE_DEFAULT	The default value is set to IVIDEO_I_FRAME.
IVIDEO_ContentType	IVIDEO_CONTENTTYPE_NA	Content type is not applicable. Encoder assumes IVIDEO_PROGRESSIVE.
	IVIDEO_PROGRESSIVE	Progressive video content. This is the default value.
	IVIDEO_INTERLACED	Interlaced video content.
IVIDEO_RateControlPreset	IVIDEO_NONE	No rate control is used
	IVIDEO_LOW_DELAY	Constant Bit-Rate (CBR) control for video conferencing. This is the default value.
	IVIDEO_STORAGE	Variable Bit-Rate (VBR) control for local storage and recording.
	IVIDEO_USER_DEFINED	User defined configuration using advanced parameters (extended parameters).
	IVIDEO_TWOPASS	Two pass rate control for non real time applications. Not supported in this version of H.264 Encoder.
	IVIDEO_RATECONTROLPRESET_DEFAULT	Set to IVIDEO_LOW_DELAY
IVIDEO_SkipMode	IVIDEO_FRAME_ENCODED	Input content encoded

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	IVIDEO_FRAME_SKIPPED	Input content skipped, that is, not encoded
	IVIDEO_SKIPMODE_DEFAULT	Default value is set to IVIDEO_FRAME_ENCODE
XDM_DataFormat	XDM_BYTE	Big endian stream. This is the default value.
	XDM_LE_16	16-bit little endian stream. Not supported in this version of H.264 Encoder.
	XDM_LE_32	32-bit little endian stream. Not supported in this version of H.264 Encoder.
XDM_ChromaFormat	XDM_CHROMA_NA	Chroma format not applicable. Encoder assumes IH264VENC_YUV_420IUV
	XDM_YUV_420P	YUV 4:2:0 planar. Not supported in this version of H.264 Encoder.
	XDM_YUV_422P	YUV 4:2:2 planar. Not supported in this version of H.264 Encoder.
	XDM_YUV_422IBE	YUV 4:2:2 interleaved (big endian). Not supported in this version of H.264 Encoder.
	XDM_YUV_422ILE	YUV 4:2:2 interleaved (little endian). Not supported in this version of H.264 Encoder.
	XDM_YUV_444P	YUV 4:4:4 planar. Not supported in this version of H.264 Encoder.
	XDM_YUV_411P	YUV 4:1:1 planar. Not supported in this version of H.264 Encoder.
	XDM_GRAY	Gray format. Not supported in this version of H.264 Encoder.
	XDM_RGB	RGB color format. Not supported in this version of H.264 Encoder.
	XDM_YUV_420SP	YUV 420 semiplaner (Luma 1st plane, * CbCr interleaved 2nd plane)

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_ARGB8888	Alpha plane
	XDM_RGB555	RGB 555 color format
	XDM_RGB565	RGB 556 color format
	XDM_YUV_444ILE	YUV 4:4:4 interleaved (little endian)
XDM_CmdId	XDM_GETSTATUS	Query algorithm instance to fill Status structure
	XDM_SETPARAMS	Set run-time dynamic parameters through the DynamicParams structure
	XDM_RESET	Reset the algorithm
	XDM_SETDEFAULT	Initialize all fields in DynamicParams structure to default values specified in the library
	XDM_FLUSH	Handle end of stream conditions. This command forces algorithm instance to output data without additional input. Not supported in this version of H.264 Encoder.
	XDM_GETVERSION	Query the algorithm version. Not supported in this version of H.264 Encoder.
	XDM_GETBUFINFO	Query algorithm instance regarding the properties of input and output buffers.
XDM_EncodingPreset	XDM_DEFAULT	Default setting of the algorithm specific creation time parameters. This uses XDM_HIGH_QUALITY settings.
	XDM_HIGH_QUALITY	Set algorithm specific creation time parameters for high quality (default setting).
	XDM_HIGH_SPEED	Set algorithm specific creation time parameters for high speed.
	XDM_USER_DEFINED	User defined configuration using advanced parameters.
XDM_EncMode	XDM_ENCODE_AU	Encode entire access unit. This is the default value.
	XDM_GENERATE_HEADER	Encode only header.

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
XDM_ErrorBit	XDM_APPLIEDCONCEALMENT	Bit 9 ❑ 1 – Applied concealment ❑ 0 – Ignore
	XDM_INSUFFICIENTDATA	Bit 10 ❑ 1 – Insufficient data ❑ 0 – Ignore
	XDM_CORRUPTEDDATA	Bit 11 ❑ 1 – Data problem/corruption ❑ 0 – Ignore
	XDM_CORRUPTEDHEADER	Bit 12 ❑ 1 – Header problem/corruption ❑ 0 – Ignore
	XDM_UNSUPPORTEDINPUT	Bit 13 ❑ 1 – Unsupported feature/parameter in input ❑ 0 – Ignore
	XDM_UNSUPPORTEDPARAM	Bit 14 ❑ 1 – Unsupported input parameter or configuration ❑ 0 – Ignore
	XDM_FATALERROR	Bit 15 ❑ 1 – Fatal error (stop encoding) ❑ 0 – Recoverable error

Note:

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

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

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

4.1.2 H264 Encoder Symbolic Constants and Enumerated Data Types

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
IH264VENC_Level	IH264VENC_LEVEL_10	Level 1.0 identifier for H.264 Encoder
	IH264VENC_LEVEL_1b	Level 1.b identifier for H.264 Encoder
	IH264VENC_LEVEL_11	Level 1.1 identifier for H.264 Encoder
	IH264VENC_LEVEL_12	Level 1.2 identifier for H.264 Encoder
	IH264VENC_LEVEL_13	Level 1.3 identifier for H.264 Encoder
	IH264VENC_LEVEL_20	Level 2.0 identifier for H.264 Encoder
	IH264VENC_LEVEL_21	Level 2.1 identifier for H.264 Encoder
	IH264VENC_LEVEL_22	Level 2.2 identifier for H.264 Encoder
	IH264VENC_LEVEL_30	Level 3.0 identifier for H.264 Encoder
	IH264VENC_LEVEL_31	Level 3.1 identifier for H.264 Encoder

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
- IVIDEO_BufDesc
- IVIDEO1_BufDescIn
- IVIDENC1_Fxns
- IVIDENC1_Params
- IVIDENC1_DynamicParams
- IVIDENC1_InArgs
- IVIDENC1_Status
- IVIDENC1_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 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 High Profile Encoder, the buffer details are:

- Number of input buffer required is 2 for YUV 420P with chroma interleaved.
- Number of output buffer required is 1.
- The input buffer sizes (in bytes) for worst case HD 720 format are:

  For YUV 420P:
  Y buffer = 1280 * 720
  UV buffer = 1280 * 360

The above input buffer size calculation is done assuming that the capture width is same as image width. For details on capture width, see Section 4.2.1.10.

For interlaced sequence, encoder ignores the input field buffers if they are stored in interleaved or non-interleaved format. But, it expects the start pointer of top or bottom field be given to it during the process call of the top or bottom fields, respectively. The pitch to move to the next line of the field is conveyed using `captureWidth` of `DynamicParams`.

- There is no restriction on output buffer size except that it should be enough to store one frame of encoded data.The output buffer size returned by the `XDM_GETBUFINFO` command assumes that the worst case output buffer size is (`frameHeight*frameWidth`)/2.

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

4.2.1.3 XDM1_BufDesc

Description

This structure defines the buffer descriptor for input and output buffers in XDM 1.0 IVIDENC1.

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.

4.2.1.4 XDM_SingleBufDesc

Description

This structure defines the single buffer descriptor for input and output buffers in XDM 1.0 IVIDENC1.

Fields

Field	Data type	Input/ Output	Description
`*buf`	`XDAS_Int8`	Input	Pointer to a buffer address
`bufSize`	`XDAS_Int32`	Input	Size of the buffer in bytes

4.2.1.5 XDM1_SingleBufDesc

Description

This structure defines the single buffer descriptor for input and output buffers in XDM 1.0 IVIDENC1.

Fields

Field	Data type	Input/ Output	Description
*buf	XDAS_Int8	Input	Pointer to a buffer address
bufSize	XDAS_Int32	Input	Size of buffer in bytes
accessMask	XDAS_Int32	Input	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 bits in this mask should not be set. **Note**: This feature is not supported in this version of H264 Encoder.

4.2.1.6 IVIDEO_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
width	XDAS_Int32	Input	Padded width of the video data
*bufs[XDM_MAX_IO_BUFFERS]	XDAS_Int8	Input	Pointer to the vector containing buffer addresses
bufSizes[XDM_MAX_IO_BUFFERS]	XDAS_Int32	Input	Size of each buffer in bytes
numBufs	XDAS_Int32	Input	Number of buffers

4.2.1.7 IVIDEO1_BufDescIn

Description

This structure defines the buffer descriptor for input video buffers.

Fields

Field	Data type	Input/ Output	Description
numBufs	XDAS_Int32	Input	Number of buffers in bufDesc[]
frameWidth	XDAS_Int32	Input	Width of the video frame
frameHeight	XDAS_Int32	Input	Height of the video frame
framePitch	XDAS_Int32	Input	Frame pitch used to store the frame. This field is not used by the encoder.
bufDesc[XDM_MAX_IO_BUFFERS]	XDM1_SingleBufDesc	Input	Picture buffers

4.2.1.8 IVIDENC1_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.

4.2.1.9 IVIDENC1_Params

Description

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

Fields

Field	Data type	Input/ Output	Description
`size`	`XDAS_Int32`	Input	Size of the basic or extended (if being used) data structure in bytes. Default size is size of `IVIDENC1_Params` structure.
`encodingPreset`	`XDAS_Int32`	Input	Encoding preset. See `XDM_EncodingPreset` enumeration for details. This version does not accept presetting of encoder parameters. It should be set to the default value. Default value = `XDM_USER_DEFINED`.
`rateControlPreset`	`XDAS_Int32`	Input	Rate control preset. See `IVIDEO_RateControlPreset` enumeration for details. Default value = `IVIDEO_STORAGE`.
`maxHeight`	`XDAS_Int32`	Input	Maximum video height to be supported in pixels. Default value = 720
`maxWidth`	`XDAS_Int32`	Input	Maximum video width to be supported in pixels. Default value = 1280.
`maxFrameRate`	`XDAS_Int32`	Input	Maximum frame rate in fps * 1000 to be supported. Default value = 30000.
`maxBitRate`	`XDAS_Int32`	Input	Maximum bit-rate to be supported in bits per second. Default value = 10000000.
`dataEndianness`	`XDAS_Int32`	Input	Endianness of input data. See `XDM_DataFormat` enumeration for details. Default value = `XDM_BYTE`.
`maxInterFrameInterval`	`XDAS_Int32`	Input	Distance from I-frame to P-frame: ❑ 1 - If no B-frames ❑ 2 - To insert one B-frame This parameter is not supported as B-frames are not supported. Set value = 1

Field	Data type	Input/ Output	Description
inputChromaFormat	XDAS_Int32	Input	Input chroma format. See XDM_ChromaFormat and IH264VENC_ChromaFormat enumeration for details. Set value as = XDM_YUV_420SP. Other values are not supported.
inputContentType	XDAS_Int32	Input	Input content type. See IVIDEO_ContentType enumeration for details. Default value = IVIDEO_PROGRESSIVE.
reconChromaFormat	XDAS_Int32	Input	Chroma formats for the reconstruction buffers. Set value as = XDM_YUV_420SP. Other values are not supported.

Note:

The maximum video height and width supported are 720 and 1280 pixels respectively.

For the supported maxBitRate values, see Annex A in *ISO/IEC 14496-10*.

The following fields of IVIDENC1_Params data structure are level dependent:

- maxHeight
- maxWidth
- maxFrameRate
- maxBitRate

To check the values supported for maxHeight and maxWidth use the following expression:

maxFrameSizeinMbs >= (maxHeight*maxWidth) / 256;

See Table A.1 – Level Limits in *ISO/IEC 14496-10* for the supported maxFrameSizeinMbs values.

For example, consider you have to check if the following values are supported for level 2.0:

- maxHeight = 480
- maxWidth = 720

The supported maxFrameSizeinMbs value for level 2.0 as per Table A.1 – Level Limits is 396.

Compute the expression as:

```
maxFrameSizeinMbs >= (480*720) / 256
```

The value of `maxFrameSizeinMbs` is 1350 and hence the condition is not true. Therefore, the above values of `maxHeight` and `maxWidth` are not supported for level 2.0.

The maximum value for `maxFrameRate` and `maxBitRate` is 30 (30000) and 10000000 respectively.

Use the following expression to check the supported `maxFrameRate` values for each level:

```
maxFrameRate <= maxMbsPerSecond / FrameSizeinMbs;
```

See Table A.1 – Level Limits in *ISO/IEC 14496-10* for the supported values of `maxMbsPerSecond`.

Use the following expression to calculate `FrameSizeinMbs`:

```
FrameSizeinMbs = (inputWidth * inputHeight) / 256;
```

See Table A.1 – Level Limits in *ISO/IEC 14496-10* for the supported values of Max Video bit-rate.

During creation time, these values are checked against the maximum values defined for the encoder. If the specified values exceed or do not match the limit supported by encoder, the encoder creation will fail. Since the actual height and width are specified later using control operation with dynamic parameters, the level based checking is done during the control operation.

4.2.1.10 IVIDENC1_DynamicParams

Description

This structure defines the run-time parameters for an algorithm instance object. Set this data structure to `NULL`, if you are not sure of the values to be specified for these parameters.

Fields

Field	Data type	Input/ Output	Description
`size`	`XDAS_Int32`	Input	Size of the basic or extended (if being used) data structure in bytes. Default value is size of `IVIDENC1_DynamicParams` structure.

Field	Data type	Input/ Output	Description
inputHeight	XDAS_Int32	Input	Height of input frame in pixels. Input height can be changed before start of encoding within the limits of maximum height set in creation phase. inputHeight must be multiple of two. Minimum height supported is 96. Irrespective of interlaced or progressive content, input height should be given as frame height. **Note**: **Progressive**: When the input height is a non-multiple of 16, the encoder expects the application to pad the input frame to the nearest multiple of 16 at the bottom of the frame. In this case, the application should set input height to actual width but should provide the padded input YUV data buffer to encoder. The encoder then puts the difference of the actual height and padded height as crop information in the bit-stream. **Interlaced**: When the input height is a non-multiple of 32, the encoder expects the application to pad the input frame to the nearest multiple of 32 at the bottom of the frame. In this case, the application should set input height to actual width but should provide the padded input YUV data buffer to encoder. The encoder then puts the difference of the actual height and padded height as crop information in the bit-stream. Default value = 720.
inputWidth	XDAS_Int32	Input	Width of input frame in pixels. Input width can be changed before the start of encoding within the limits of maximum width set in creation phase. inputWidth must be multiples of two. Minimum width supported is 128. **Note**: When the input width is a non-multiple of 16, the encoder expects the application to pad the input frame to the nearest multiple of 16 to the right of the frame. In this case, application should set inputWidth to actual width but should provide the padded input YUV data buffer to encoder. The encoder then puts the difference of the actual width and padded width as crop information in the bit-stream. Default value = 1280
refFrameRate	XDAS_Int32	Input	Reference or input frame rate in fps * 1000. For example, if the frame rate is 30, set this field to 30000. This parameter is not supported. Should be set equal to targetFrameRate.

Field	Data type	Input/ Output	Description
targetFrameRate	XDAS_Int32	Input	Target frame rate in fps * 1000. For example, if the frame rate is 30, set this field to 30000. Default value = 25000. Frame rate should be in multiple of 0.5 fps
targetBitRate	XDAS_Int32	Input	Target bit-rate in bits per second. For example, if the bit-rate is 2 Mbps, set this field to 2000000. Default value = 10000000.
intraFrameInterval	XDAS_Int32	Input	Interval between two consecutive intra frames. ❑ 0: First frame will be intra coded ❑ 1: No inter frames, all intra frames ❑ 2: Consecutive IPIPIP ❑ 3: 1PPIPPIPP or IPBIPBIPB, and so on
generateHeader	XDAS_Int32	Input	Encode entire access unit or only header. See XDM_EncMode enumeration for details. Default value = XDM_ENCODE_AU.
captureWidth	XDAS_Int32	Input	Capture width parameter enables the application to provide input buffers with different line width (pitch) alignment than image width. For progressive content, if the parameter is set to: ❑ 0 - Encoded image width is used as pitch. ❑ < encoded image width - If capture width is set less than encoded image width, then capture width is ignored and encoded image width is used as pitch. ❑ >= encoded image width - capture width is used as pitch. For interlaced content, captureWidth should be equal to the pitch/stride value needed to move to the next row of pixel in the same field.
forceFrame	XDAS_Int32	Input	Force the current (immediate) frame to be encoded as a specific frame type. Only the following values are supported ❑ IVIDEO_NA_FRAME - No forcing of any specific frame type for the frame. ❑ IVIDEO_I_FRAME - Force the frame to be encoded as I frame. ❑ IVIDEO_IDR_FRAME - Force the frame to be encoded as an IDR frame. Default value = IVIDEO_NA_FRAME.
interFrameInterval	XDAS_Int32	Input	Number of B frames between two reference frames; that is, the number of B frames between two P frames or I/P frames. This parameter is not supported. It should be set to 0.

Field	Data type	Input/ Output	Description
mbDataFlag	XDAS_Int32	Input	Flag to indicate that the algorithm should use MB data supplied in additional buffer within inBufs. This parameter is not supported. It should be set to 0.

Note:

The following are the limitations on the parameters of IVIDENC1_DynamicParams data structure:

- inputHeight <= maxHeight
- inputWidth <= maxWidth
- refFrameRate <= maxFrameRate
- targetFrameRate <= maxFrameRate
- targetFrameRate should be multiple of 500
- The value of the refFrameRate and targetFrameRate should be the same.
- targetBitRate <= maxBitRate
- The inputHeight and inputWidth must be multiples of two.
- The inputHeight, inputWidth, and targetFrameRate should adhere to the standard defined level limits. For an incorrect level, the encoder tries to match the best level for the parameters provided. However, if it changes level 3.1, an error is reported. As per the requirement, level limit can be violated for targetBitRate.
- When inputHeight/inputWidth are non-multiples of 16, encoder expects the application to pad the input frame to the nearest multiple of 16 at the bottom/right of the frame. In this case, application sets the inputHeight/inputWidth to the actual height/width; however, it should provide the padded input YUV data buffer to the encoder.
- When inputWidth is non-multiple of 16, the encoder expects capture width as padded width(nearest multiple of 16). If the capture width is 0 or less than padded width, then the capture width is assumed to be the padded width. In all other cases, the capture width provided through input parameter is used for input frame processing.
- The encoder flag errors for incorrect values of inputWidth, inputHeight, targetBitRate, and targetFrameRate. For other parameters, the encoder will issue a warning and continue encoding with default parameters.
- intraFrameInterval is used to signal the I frame interval in H.264. There is one more field in extended dynamic params called idrFrameInterval, which specifies the IDR frame interval for H.264. With each IDR frame, SPS and PPS is sent. The first frame of the sequence is always an IDR frame

4.2.1.11 IVIDENC1_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.
`inputID`	`XDAS_Int32`	Input	Identifier to attach with the corresponding encoded bit stream frames. This is useful when frames require buffering (for example, B frames), and to support buffer management. When there is no re-ordering, `IVIDENC1_OutArgs::outputID` will be the same as this `inputID` field. Zero (0) is not a supported `inputID`. This value is reserved for cases when there is no output buffer provided.
`topFieldFirstFlag`	`XDAS_Int32`	Input	Flag to indicate the field order in interlaced content. Valid values are XDAS_TRUE and XDAS_FALSE. This field is only applicable to the input image buffer. This field is only applicable for interlaced content and not progressive. Currently, supported value is XDAS_TRUE.

4.2.1.12 IVIDENC1_Status

Description

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

Fields

Field	Data type	Input/ Output	Description
`size`	`XDAS_Int32`	Input	Size of the basic or extended (if being used) data structure in bytes.
`extendedError`	`XDAS_Int32`	Output	Extended error code. See `XDM_ErrorBit` enumeration for details.
`data`	`XDM1_SingleBufDesc`	Input	Buffer descriptor for data passing
`bufInfo`	`XDM_AlgBufInfo`	Output	Input and output buffer information. See `XDM_AlgBufInfo` data structure for details.

4.2.1.13 IVIDENC1_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.
extendedError	XDAS_Int32	Output	Extended error code. See XDM_ErrorBit enumeration for details.
bytesGenerated	XDAS_Int32	Output	The number of bytes generated.
encodedFrameType	XDAS_Int32	Output	Frame types for video. See IVIDEO_FrameType enumeration for details. Following values are only supported ❑ IVIDEO_I_FRAME ❑ IVIDEO_IDR_FRAME ❑ IVIDEO_P_FRAME ❑ IVIDEO_II_FRAME ❑ IVIDEO_PP_FRAME
inputFrameSkip	XDAS_Int32	Output	Frame skipping modes for video. See IVIDEO_SkipMode enumeration for details.
outputID	XDAS_Int32	Output	Output ID corresponding to the encoder buffer. This can also be used to free the corresponding image buffer for further use by the client application code. In this encoder, outputID is set to IVIDENC1_InArgs::inputID.
encodedBuf	XDM1_SingleBufDesc	Output	The encoder fills the buffer with the encoded bit-stream. In case of sequences with only I and P frames, these values are identical to outBufs passed in IVIDENC1_Fxns::process() The encodedBuf.bufSize field returned corresponds to the actual valid bytes available in the buffer. The bit-stream is in encoded order. The outputId and encodedBuf together provide information related to the corresponding encoded image buffer.
reconBufs	IVIDEO1_BufDesc	Output	Pointer to reconstruction buffer descriptor.

4.2.2 H.264 Encoder Data Structures

This section includes the following H.264 Encoder specific extended data structures:

- IH264VENC_Params
- IH264VENC_DynamicParams
- IH264VENC_InArgs
- IH264VENC_Status
- IH264VENC_OutArgs
- IH264VENC_Fxns

4.2.2.1 IH264VENC_Params

Description

This structure defines the creation parameters and any other implementation specific parameters for a H.264 Encoder instance object. The creation parameters are defined in the XDM data structure, IVIDENC1_Params.

Fields

Field	Data type	Input/ Output	Description
videncParams	IVIDENC1_Params	Input	See IVIDENC1_Params data structure for details. The size parameter in videncParams is set to size of IH264VENC_Params structure by default while using extended parameters.
profileIdc	XDAS_Int32	Input	Profile identification for the encoder. The current version supports High Profile. The value must be set to 66(Base line profile), 77(main profile), 100(high profile). Default value = 100.
levelIdc	XDAS_Int32	Input	Level identification for the encoder. See IH264VENC_Level enumeration for details. Default value = IH264VENC_LEVEL_31.
aspectRatioX	XDAS_Int32	Input	X scale for Aspect Ratio. The value should be greater than 0 and co-prime with AspectRatioY. Default value = 1
aspectRatioY	XDAS_Int32	Input	Y scale for Aspect Ratio The value should be greater than 0 and co-prime with AspectRatioX. Default value = 1.

Field	Data type	Input/ Output	Description
pixelRange	XDAS_Int32	Input	The range for the luma and chroma pixel values ❑ 0 – Restricted Range ❑ 1 – Full Range (0-255) Default value = 1
meAlgo	XDAS_Int32	Input	The type of Motion Estimation Search Algorithm ❑ 0 – Normal Search ❑ 1 – Low Power Search with vertical GMV Default value = 0 This feature is only present when encoder preset is XDM_HIGH_QUALITY or encQuality =1.
timeScale	XDAS_Int32	Input	Time resolution value for Picture Timing Information This should be greater than or equal to frame rate in fps. See Appendix A for more details. Default value = 150.
numUnitsInTicks	XDAS_Int32	Input	Units of Time Resolution constituting the single Tick See Appendix A for more details. Default value = 1.
enableVUIparams	XDAS_Int32	Input	Flag for Enable VUI Parameters ❑ 0 – Disable VUI Parameters ❑ 1 – Enable VUI Parameters Default value = 0.
entropyMode	XDAS_Int32	Input	Flag for Entropy Coding Mode ❑ 0 – CAVLC ❑ 1 – CABAC Default value = 1.
transform8x8FlagIntraFrame	XDAS_Int32	Input	Flag for 8x8 Transform for I frame ❑ 0 – Disable ❑ 1 – Enable Default value = 1.
transform8x8FlagInterFrame	XDAS_Int32	Input	Flag for 8x8 Transform for P frame ❑ 0 – Disable ❑ 1 – Enable Default value = 0.
SequenceScalingFlag	XDAS_Int32	Input	Flag for use of Sequence Scaling Matrix ❑ 0 – Disable ❑ 1 – Auto ❑ 2 – Low ❑ 3 – Moderate ❑ 4 – High Default value = 1.

Field	Data type	Input/ Output	Description
resetHDVICPeveryFrame	XDAS_Int32	Input	Flag to reset HDVICP at the start of every frame being encoded. This is useful for multi-channel and multi-format encoding. ❑ 1 – ON ❑ 0 – OFF Default value = 1.
disableHDVICPeveryFrame	XDAS_Int32	Input	Flag to disable HDVICP at the start of every frame being encoded. This is useful for power saving. ❑ 1 – ON ❑ 0 – OFF Default value = 0.
encQuality	XDAS_Int32	Input	Flag for High and low quality encoding ❑ 1 – High Quality, full feature ❑ 0 – Standard Quality, high speed Default value = 0
unrestrictedMV	XDAS_Int32	Input	Flag for enabling unrestricted MV ❑ 0 – Disable ❑ 1 – Enable Default value = 0 This feature is only present when encoder preset is XDM_HIGH_QUALITY or encQuality =1

Note:

- Default values of extended parameters are used when size fields are set to the size of base structure IVIDENC1_Params.
- aspectRatio and pixelRange information is included in the bit-stream only when enableVUIparams is set to 1.

If the level is not set appropriately, the encoder tries to fit a correct level. However, if it exceeds level 3.1, an error is reported.

4.2.2.2 IH264VENC_DynamicParams

Description

This structure defines the run-time parameters and any other implementation specific parameters for a H.264 Encoder instance object. The run-time parameters are defined in the XDM data structure, `IVIDENC1_DynamicParams`.

Fields

Field	Data type	Input/ Output	Description
`videncDynamicParams`	`IVIDENC1_DynamicParams`	Input	See `IVIDENC1_DynamicParams` data structure for details. The size parameter of `DynamicParams` is set to size of `IVIDENC1_DynamicParams` structure by default while using extended parameters.
`intraFrameQP`	`XDAS_Int32`	Input	Quantization Parameter (QP) of I-frames in fixed QP mode. Valid value is 0 to 51. It is useful only when: ❑ `rateControlPreset` of `IVIDENC1_Params` is equal to `IVIDEO_NONE`. ❑ `RcAlgo` = 2 (Fixed QP) ❑ `targetBitRate` = 0 Default value = 28
`interPFrameQP`	`XDAS_Int32`	Input	Quantization Parameter (QP) of P-frames in fixed QP mode. Valid value is 0 to 51. It is useful only when: ❑ `rateControlPreset` of `IVIDENC1_Params` is equal to `IVIDEO_NONE`. ❑ `RcAlgo` = 2 (Fixed QP) ❑ `targetBitRate` = 0 Default value = 28
`initQ`	`XDAS_Int32`	Input	Initial Quantization (QP) for the first frame. Valid value is 0 to 51. The parameter is applicable only when rate-control is enabled. Should be set based on the target bit-rate. Default value = 28
`rcQMax`	`XDAS_Int32`	Input	Maximum value of Quantization Parameter (QP) to be used while encoding. Valid value is 0 to 51. The value for `rcQMax` should not be less than `rcQMin`. The parameter is applicable only when rate-control is enabled. Default value = 45

Field	Data type	Input/ Output	Description
rcQMin	XDAS_Int32	Input	Minimum value of Quantization Parameter (QP) to be used while encoding. Valid value is 0 to 51. The value for rcQMin should not be greater than rcQMax. The parameter is applicable only when rate-control is enabled. Default value = 0.
airRate	XDAS_Int32	Input	Parameter for forced Intra MB insertion in P-frames. ❑ 0 – No forced Intra MBs ❑ n > 0 – number of forced Intra MB in each frame. Default value = 0. This feature is only present when encoder preset is XDM_HIGH_QUALITY or encQuality =1
sliceSize	XDAS_Int32	Input	Size of each slice in bits. ❑ 0 – Single Slice per Frame ❑ >0 – Multiple Slices with the size of each slice <= sliceSize Default value = 0 This feature is only present when encoder preset is XDM_HIGH_QUALITY or encQuality =1 Minimum slice size supported is 1024 bits.
lfDisableIdc	XDAS_Int32	Input	Option for Loop Filter Disable ❑ 0 – Loop Filter Enable ❑ 1 – Loop Filter Disable ❑ 2 – Disable Filter across slice boundaries Default value = 0
rcAlgo	XDAS_Int32	Input	Option for type of Rate Control Algorithm ❑ 0 – CBR ❑ 1 – VBR ❑ 2 – Fixed QP All the supported RC algorithms do not support quantization scale variation within the I-frames either at row level or at MB level. Default value = 1

Field	Data type	Input/ Output	Description
maxDelay	XDAS_Int32	Input	Maximum acceptable delay in milliseconds for rate control. This value should be greater than 100ms. Currently, there is a maximum limit for this parameter but application can use up to 10000 ms. Typical value is 1000 ms. By default, this is set to 2000 ms at the time of encoder object creation.
intraSliceNum	XDAS_Int32	Input	Index of the slice to be coded as Fast Intra Update Slice. ❑ 0 – No Fast Intra Update Slice ❑ > 0 – Index of the slice to be coded as Intra Refresh Slice. (Slice numbering starting with 1) Default value = 0. If number of slice < IntraSliceNum, no forced intra slice occurs. This feature is only present when encoder preset is XDM_HIGH_QUALITY or encQuality =1
meMultiPart	XDAS_Int32	Input	Flag to enable multiple partitions of macro-blocks ❑ 0 – Single partition ❑ 1 – Multiple partitions Maximum of 8x8 partitions coded Default value = 0. This feature is only present when encoder preset is XDM_HIGH_QUALITY or encQuality =1
enableBufSEI	XDAS_Int32	Input	Flag for enabling Buffering Period SEI message ❑ 0 – Disable ❑ 1 – Enable Default value = 0
enablePicTimSEI	XDAS_Int32	Input	Flag for enabling Picture Timing SEI message ❑ 0 – Disable ❑ 1 – Enable This parameter is disabled if EnableBufSEI is disabled. Default value = 0

Field	Data type	Input/ Output	Description
intraThrQF	XDAS_Int32	Input	Quality factor for intra thresholding process. The encoder does the intra-prediction estimation process selectively for MBs in P-frame based on the threshold derived using the quality factor. ❑ Valid values : 0 – 5. ❑ 0 – Intra prediction estimation is avoided for most of the MBs in the P-frame. ❑ 5 – Intra prediction estimation is done for all MBs in the P-frame. Default value = 5 This feature is only present when encoder preset is XDM_HIGH_QUALITY or encQuality =1
perceptualRC	XDAS_Int32	Input	Flag for enabling Perceptual QP modulation of MBs ❑ 0 – Disable ❑ 1 – Enable Default value = 1 PRC is disable automatically for maxDelay<100 and rcAlgo = CBR
idrFrameInterval	XDAS_Int32	Input	Interval between two consecutive IDR frames. ❑ 0: first frame will be IDR coded ❑ 1: No inter frames, all IDR frames ❑ 2: Consecutive IDR P IDR P ❑ 3: IDR P P IDR P P IDR .. or IDR P B IDR P B IDR P B ….and so on Default value = 0.

Note:

- enablePicTimSEI values are used only when enableBufSEI is set to 1.
- rcAlgo values are used only when IVIDENC1_Params ->RateControlPreset = IVIDEO_USER_DEFINED.
- rcQMax, rcQMin, initQ, and maxDelay values are used only when the encoder does not run in fixed QP mode.
- Generally idrFrameInterval will be larger than intraFrameInterval. For example, idrFrameInterval = 300 and intraFrameInterval = 30. This means that at every 30th frame, there will be an I frame. But at every 300th frame, an IDR frame will be placed instead of I frame. IDR frame is used for syncronization.

- When CABAC is enabled, there are chances that the slice size may overshoot as slice gets terminated to the nearest row boundaries only.

4.2.2.3 IH264VENC_InArgs

Description

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

Fields

Field	Data type	Input/ Output	Description
videncInArgs	IVIDENC1_InArgs	Input	See IVIDENC1_InArgs data structure for details.
timeStamp	XDAS_Int32	Input	Time stamp value of the frame to be placed in bit stream. This should be integral multiple of TimerResolution/ (Frame rate in fps). Initial time stamp value (for first frame) should be 0. Default is calculated as Frame number * TimerResolution/ (Frame rate in fps). See Appendix A for more details.
insertUserData	XDAS_Int32	Input	This field is reserved
lenghtUserData	XDAS_Int32	Input	This field is reserved

Note:

TimeStamp is included only when IH264VENC_DynamicParams->EnablePicTimSEI is set to 1.

4.2.2.4 IH264VENC_Status

Description

This structure defines parameters that describe the status of the H.264 Encoder and any other implementation specific parameters. The status parameters are defined in the XDM data structure, `IVIDENC1_Status`.

Fields

Field	Data type	Input/ Output	Description
`videncStatus`	`IVIDENC1_Status`	Input/Output	See `IVIDENC1_Status` data structure for details.

4.2.2.5 IH264VENC_OutArgs

Description

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

Fields

Field	Data type	Input/ Output	Description
`videncOutArgs`	`IVIDENC1_OutArgs`	Output	See `IVIDENC1_OutArgs` data structure for details.
`numPackets`	`XDAS_Int32`	Output	Total number of packets/slices in the encoded frame
`packetSize`	`XDAS_Int32*`	Output	Pointer to buffer for writing individual packet size in bytes. Application should allocate the buffer with the size of (100 * 4) bytes and send the pointer to the encoder. 100 is the maximum number of packets supported. Only sizes of valid packets indicated by `numPackets` will be filled by the encoder and the remaining values in the buffer are invalid.
`offsetUserData`	`XDAS_Int32`	Output	This field is reserved

4.2.2.6 IH264VENC_Fxns

Description

This structure defines all of the operations for the H.264 Encoder instance object.

Fields

Field	Data type	Input/ Output	Description
ividenc	IVIDENC1_Fxns	Output	See IVIDENC1_Fxns data structure for details.

4.3 Interface Functions

This section describes the Application Programming Interfaces (APIs) used in the H.264 Encoder. The APIs are logically grouped into the following categories:

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

You must call these APIs in the following sequence:

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

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

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

4.3.1 Creation APIs

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

‖ Name

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

‖ Synopsis

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

‖ Arguments

```
Void
```

‖ Return Value

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

‖ Description

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

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

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

‖ See Also

`algAlloc()`

‖ Name

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

‖ Synopsis

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

‖ Arguments

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

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

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

‖ Return Value

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

‖ Description

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

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

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

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

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

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

‖ See Also

`algNumAlloc(), algFree()`

4.3.2 Initialization API

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

‖ Name

`algInit()` – initialize an algorithm instance

‖ Synopsis

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

‖ Arguments

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

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

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

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

‖ Return Value

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

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

‖ Description

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

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

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

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

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

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

‖ See Also

`algAlloc(), algMoved()`

4.3.3 Control API

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

‖ Name

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

‖ Synopsis

```
XDAS_Int32 (*control) (IVIDENC1_Handle handle,
IVIDENC1_Cmd id, IVIDENC1_DynamicParams *params,
IVIDENC1_Status *status);
```

‖ Arguments

```
IVIDENC1_Handle handle; /* algorithm instance handle */

IVIDENC1_Cmd id; /* algorithm specific control commands*/

IVIDENC1_DynamicParams *params /* algorithm run-time
parameters */

IVIDENC1_Status *status /* algorithm instance status
parameters */
```

‖ Return Value

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

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

‖ Description

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

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

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

The third and fourth arguments are pointers to the `IVIDENC1_DynamicParams` and `IVIDENC1_Status` data structures respectively.

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

‖ Preconditions

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

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

‖ Postconditions

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

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

‖ Example

See test application file, h264encoderapp.c available in the \client\test\src sub-directory.

‖ See Also

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

4.3.4 Data Processing API

Data processing API is used for processing the input data.

‖ Name

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

‖ Synopsis

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

‖ Arguments

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

‖ Return Value

```
Void
```

‖ Description

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

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

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

‖ See Also

`algDeactivate()`

‖ Name

`process()` – basic encoding/decoding call

‖ Synopsis

```
XDAS_Int32 (*process)(IVIDENC1_Handle handle,
IVIDEO1_BufDescIn *inBufs, XDM_BufDesc *outBufs,
IVIDENC1_InArgs *inargs, IVIDENC1_OutArgs *outargs);
```

‖ Arguments

```
IVIDENC1_Handle handle; /* algorithm instance handle */

IVIDEO1_BufDescIn *inBufs; /* algorithm input buffer
descriptor */

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

IVIDENC1_InArgs *inargs /* algorithm runtime input arguments
*/

IVIDENC1_OutArgs *outargs /* algorithm runtime output
arguments */
```

‖ Return Value

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

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

‖ Description

A call to function initiates the encoding/decoding process for the current frame.

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

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

The fourth argument is a pointer to the `IVIDENC1_InArgs` data structure that defines the run-time input arguments for an algorithm instance object.

The last argument is a pointer to the `IVIDENC1_OutArgs` data structure that defines the run-time output arguments for an algorithm instance object.

In case of interlaced content, process call has to be invoked for each field.

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

‖ Preconditions

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

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

‖ Postconditions

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

If the process operation is successful, the return value from this operation is equal to `IALG_EOK`; otherwise it is equal to either `IALG_EFAIL` or an algorithm specific return value.

‖ Example

See test application file, h264encoderapp.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. Pre-emption can happen only at frame boundaries and after `algDeactivate()` is called.
- The input data is either in 8-bit YUV 4:2:0. The encoder output is H.264 encoded bit stream.

‖ Name

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

‖ Synopsis

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

‖ Arguments

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

‖ Return Value

```
Void
```

‖ Description

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

The first (and only) argument to `algDeactivate()` is an algorithm instance handle. This handle is used by the algorithm to identify various buffers that must be saved prior to next cycle of `algActivate()` and processing.
For more details, see *TMS320 DSP Algorithm Standard API Reference* (literature number SPRU360).

‖ See Also

`algActivate()`

4.3.5 Termination API

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

‖ Name

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

‖ Synopsis

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

‖ Arguments

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

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

‖ Return Value

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

‖ Description

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

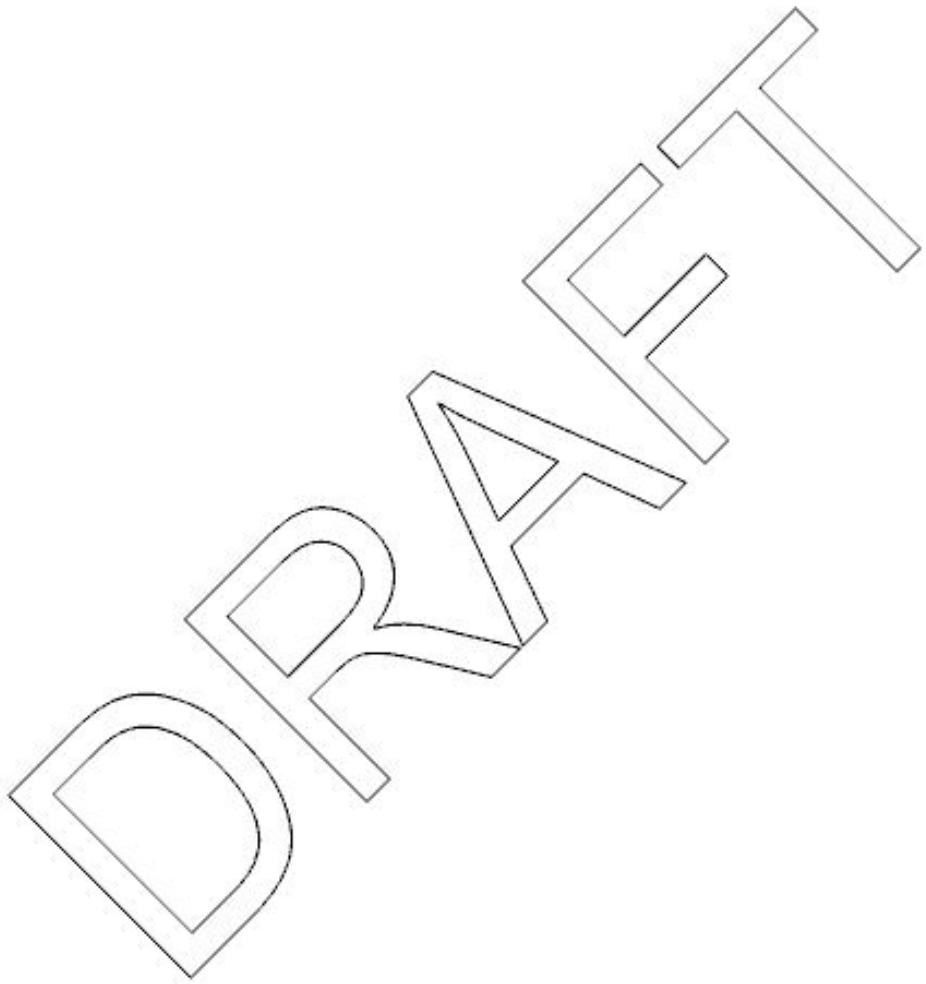
The second argument is a table of memory records that describe the base address, size, alignment, type, and memory space of all buffers previously allocated for the algorithm instance.
For more details, see *TMS320 DSP Algorithm Standard API Reference* (literature number SPRU360).

‖ See Also

`algAlloc()`

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

This page is intentionally left blank



Appendix A

Time-Stamp Insertion

A.1 Description

The DM365 H.264 Encoder supports insertion of frame time-stamp through the Supplemental Enhancement Information (SEI) Picture Timing message. The time-stamp is useful for audio-synchronization and determining the exact timing for display of frames. The parameters coded in the SEI Picture Timing Message are also useful for testing HRD compliance.

The application should take proper care while setting the parameters for time-stamp and the actual time-stamp for each frame. Ideally, the time-stamp can be set based on the frame-rate. This simplifies the process of generating time-stamps. However, the application is free to use any method of time-stamp generation.

Time-stamp based on frame-rate can be generated as follows.

Let f be the frame-rate of the sequence. Assuming a constant frame-rate sequence, set

```
TimeScale = k * f

NumUnitsinTicks = n

where k is an integer such that (k * f) and (k/n) are
integers

units_per_frame = k/n
```

For the first frame, set the `TimeStamp` parameter in `inArgs` structure to 0. For the subsequent frames, increment the `TimeStamp` by `units_per_frame`

Example 1.

```
f = 30.

Let k = 2

TimeScale = 2 * 30 = 60

NumUnitInTicks = 1

units_per_frame = 2

TimeStamp = 0, 2, 4, 6, 8…
```

Example 2.

```
f = 25
k = 2
TimeScale = 2 * 25 = 50
NumUnitsInTicks = 2
units_per_frame = 1
TimeStamp = 0, 1, 2, 3, 4…
```

Example 3.

```
f = 15
k = 1000
TimeScale = 1000 * 15 = 15000
NumUnitsInTicks = 1000
units_per_frame = 1
TimeStamp = 0, 1, 2, 3, 4…
```

Example 4.

```
f = 0.5
k = 200
TimeScale = 200 * 0.5 = 100
NumUnitsinTicks = 100
units_per_frame = 2
TimeStamp = 0, 2, 4, 6, 8
```