H.264 Baseline Profile Encoder on DM648/TNETV2685

User's Guide

Literature Number: SPRUF68
July 2008

IMPORTANT NOTICE

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

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

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

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

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

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

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

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

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

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

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

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

Read This First

About This Manual

This document describes how to install and work with Texas Instruments' (TI) H.264 Baseline Profile Encoder implementation on the DM648/TNETV2685 platform. It also provides a detailed Application Programming Interface (API) reference and information on the sample application that accompanies this component.

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

Intended Audience

This document is intended for system engineers who want to integrate TI's codecs with other software to build a multimedia system based on the DM648/TNETV2685 platform.

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

How to Use This Manual

This document includes the following chapters:

- **Chapter 1 - Introduction**, provides a brief introduction to the XDAIS and XDM standards. It also provides an overview of the codec and lists its supported features.
- **Chapter 2 - Installation Overview**, describes how to install, build, and run the codec.
- **Chapter 3 - Sample Usage**, describes the sample usage of the codec.
- **Chapter 4 - API Reference**, describes the data structures and interface functions used in the codec.

Related Documentation From Texas Instruments

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

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

The following documents describe TMS320 devices and related support tools:

- *Design and Implementation of an eXpressDSP-Compliant DMA Manager for C6X1X* (literature number SPRA789) describes a C6x1x-optimized (C6211, C6711) ACPY2 library implementation and DMA Resource Manager.
- *TMS320c64x+ Megamodule* (literature number SPRAA68) describes the enhancements made to the internal memory and describes the new features which have been added to support the internal memory architecture's performance and protection.
- *TMS320C64x+ DSP Megamodule Reference Guide* (literature number SPRU871) describes the C64x+ megamodule peripherals.
- *TMS320C64x to TMS320C64x+ CPU Migration Guide* (literature number SPRAA84) describes migration from the Texas Instruments TMS320C64x™ digital signal processor (DSP) to the TMS320C64x+™ DSP.

- *TMS320C6000 Optimizing Compiler v 6.0 Beta User's Guide* (literature number SPRU187N) explains how to use compiler tools such as compiler, assembly optimizer, standalone simulator, library-build utility, and C++ name demangler.
- *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide* (literature number SPRU732) describes the CPU architecture, pipeline, instruction set, and interrupts of the C64x and C64x+ DSPs.
- *TMS320DM6443 Digital Media System-on-Chip* (literature number SPRS282)
- *TMS320DM6443 Digital Media System-on-Chip Errata (Silicon Revision 1.0)* (literature number SPRZ240) describes the known exceptions to the functional specifications for the TMS320DM6443 Digital Media System-on-Chip (DMSoC).
- *TMS320DM644x DMSoC DSP Subsystem Reference Guide* (literature number SPRUE15) describes the digital signal processor (DSP) subsystem in the TMS320DM644x Digital Media System-on-Chip (DMSoC).
- *TMS320DM644x DMSoC ARM Subsystem Reference Guide* (literature number SPRUE14) describes the ARM subsystem in the TMS320DM644x Digital Media System on a Chip (DMSoC).
- *DaVinci Technology - Digital Video Innovation Product Bulletin (Rev. A)* (literature number SPRT378A)
- *The DaVinci Effect: Achieving Digital Video Without Complexity White Paper* (literature number SPRY079)
- *DaVinci Benchmarks Product Bulletin* (literature number SPRT379)
- *DaVinci Technology for Digital Video White Paper* (literature number SPRY067)
- *The Future of Digital Video White Paper* (literature number SPRY066)

Related Documentation

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

- *ISO/IEC 11172-2 Information Technology -- Coding of moving pictures and associated audio for digital storage media at up to about 1.5Mbits/s -- Part 2: Video* (MPEG-1 video standard)
- *ITU-T Rec. H.264 | ISO/IEC 14496-10 AVC - Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification*

Abbreviations

The following abbreviations are used in this document.

Table 1-1. List of Abbreviations

Abbreviation	Description
API	Application Programming Interface
AVC	Advanced Video Coding
BP	Base Profile
CAVLC	Context Adaptive Variable Length Coding
CIF	Common Intermediate Format
COFF	Common Object File Format
DMA	Direct Memory Access
DMAN3	DMA Manager
DSP	Digital Signal Processing
EVM	Evaluation Module
GOP	Group Of Pictures
HEC	Header Extension Code
HPI	Half Pixel Interpolation
IDR	Instantaneous Decoding Refresh
MIR	Mandatory Intra Fresh
QCIF	Quarter Common Intermediate Format
QP	Quantization Parameter

Abbreviation	Description
QPI	Quarter Pixel Interpolation
QVGA	Quarter Video Graphics Array
SQCIF	Sub Quarter Common Intermediate Format
VGA	Video Graphics Array
XDAIS	eXpressDSP Algorithm Interface Standard
XDM	eXpressDSP Digital Media

Text Conventions

The following conventions are used in this document:

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

Product Support

When contacting TI for support on this codec, quote the product name (H.264 Baseline Profile Encoder on DM648/TNETV2685) and version number. The version number of the codec is included in the Title of the Release Notes that accompanies this codec.

Trademarks

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

All trademarks are the property of their respective owners.

This page is intentionally left blank

Contents

Figures

This page is intentionally left blank

Tables

This page is intentionally left blank

Chapter 1

Introduction

This chapter provides a brief introduction to XDAIS and XDM. It also provides an overview of TI's implementation of the H.264 Baseline Profile Encoder on the DM648/TNETV2685 platform and its supported features.

Topic **Page**

1.1 Overview of XDAIS and XDM

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

1.1.1 XDAIS Overview

An eXpressDSP-compliant algorithm is a module that implements the abstract interface IALG. The IALG API takes the memory management function away from the algorithm and places it in the hosting framework. Thus, an interaction occurs between the algorithm and the framework. This interaction allows the client application to allocate memory for the algorithm and also share memory between algorithms. It also allows the memory to be moved around while an algorithm is operating in the system. In order to facilitate these functionalities, the IALG interface defines the following APIs:

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

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

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

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

1.1.2 XDM Overview

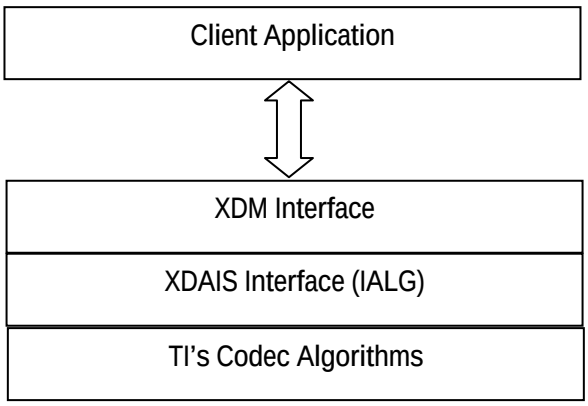
In the multimedia application space, you have the choice of integrating any codec into your multimedia system. For example, if you are building a video decoder system, you can use any of the available video decoders (such as MPEG4, H.263, or H.264) in your system. To enable easy integration with the client application, it is important that all codecs with similar functionality use similar APIs. XDM was primarily defined as an extension to XDAIS to ensure uniformity across different classes of codecs (for example audio, video, image, and speech). The XDM standard defines the following two APIs:

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

The `control()` API provides a standard way to control an algorithm instance and receive status information from the algorithm in real-time. The `control()` API replaces the `algControl()` API defined as part of the IALG interface. The `process()` API does the basic processing (encode/decode) of data.

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

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



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

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

1.2 Overview of H.264 Baseline Profile Encoder

H.264 is a video compression standard from ITU-T Video Coding Experts Group and ISO/IEC Moving Picture Experts Group. H.264 provides greater compression ratios at low bit-rate. The new advancements and greater compression ratios that are available at a low bit-rate has made devices ranging from mobile and consumer electronics to set-top boxes and digital terrestrial broadcasting to use the H.264 standard.

Figure 1-1 depicts the working of the H.264 Encoder algorithm.

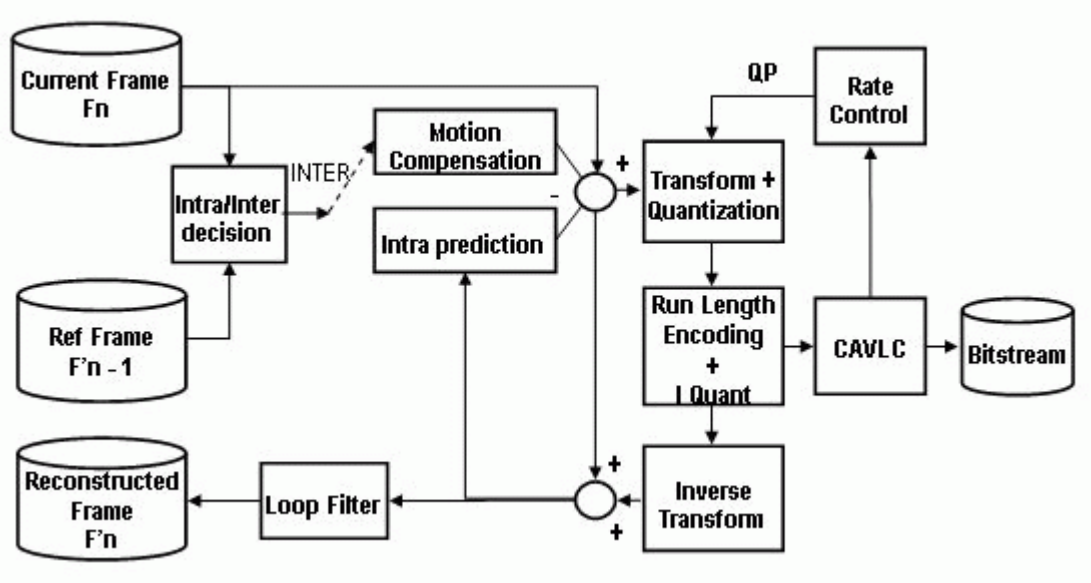


Figure 1-1. Working of H.264 Video Encoder

H.264 Encoder performs operations on a set of N macro blocks. The selection of macro blocks depends on the availability of internal memory. The operations such as Motion Compensation, Transform and Quantization, Run Length Encoding, Inverse Quantization, and Inverse Transform Blocks are called, for all the inter macro blocks.

The encoder is designed such that, it always tries to maximize the throughput of each unit by allowing it to perform on maximum possible number of macro blocks.

Motion Estimation is the step where encoder searches for the best match in the available reference frame(s). After quantization, contents of some blocks changes to 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 contains all zero coefficients 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 depends on the edge strength of the particular edge.

H.264 Encoder applies entropy coding methods to use context based adaptivity, which improves the coding performance. All macro blocks that

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 co-efficients 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.

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

1.3 Supported Services and Features

This user guide accompanies TI's implementation of H.264 Encoder on the DM648/TNETV2685 platform.

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

- Supports H.264 baseline profile up to level 3
- Supports quarter-pel interpolation for motion estimation
- Supports in-loop filtering which can be switched off for whole picture as well for slice boundaries
- Supports user controllable multiple slices per picture
- Supports error-robustness features like intra slice insertion in inter frames, adaptive intra refresh, constrained intra prediction and forcefully encoding any frame as Instantaneous Decoding Refresh (IDR)
- Supports user controllable quantization parameter range
- Supports unrestricted motion vector search that allows motion vectors to be outside the frame boundary
- Supports image width and height which are non-multiple of 16
- Controls the balance between encoder speed and quality by using the user definable motion estimation settings

The other explicit features that TI's H.264 Encoder provides are:

- Supports TI proprietary rate control algorithms
- Supports arbitrary resolutions up to PAL D1 (720x576), including standard image sizes such as SQCIF, QCIF, CIF, QVGA, and VGA
- Supports user configurable Group of Pictures (GOP) length

- Supports user configurable parameters like pic_order_cnt_type, log2_max_frame_num_minus4, and chroma_qp_index_offset
- Supports YUV422 interleaved and YUV420 planar color sub-sampling formats
- eXpressDSP Digital Media (XDM 0.9 IVIDENC) compliant

This version of the codec does not support the following features of the standard:

- No constraint kept to encode a macro block within 3200 bits as per the standard.

Chapter 2

Installation Overview

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

Topic **Page**

2.1 System Requirements

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

2.1.1 Hardware

This codec has been built and tested on the DM648/TNETV2685 EVM with XDS560 JTAG emulator.

2.1.2 Software

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

- **Development Environment:** This project is developed using Code Composer Studio version 3.3.24.1.
- **Code Generation Tools:** This project is compiled, assembled, archived, and linked using the code generation tools version 6.0.7.

2.2 Installing the Component

The codec component is released as a compressed archive. To install the codec, extract the contents of the zip file onto your local hard disk. The zip file extraction creates a top-level directory called 100_V_H264AVC_E_1_14, under which another directory named DM648_TNETV2685_BP_001 is created.

Figure 2-1 shows the sub-directories created in the DM648_TNETV2685_BP_001 directory.

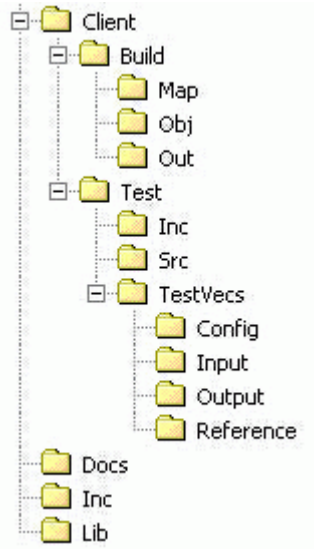


Figure 2-1. Component Directory Structure

Note:

If you are installing an evaluation version of this codec, the directory name will be 100E_V_H264AVC_E_1_14.

Table 2-1 provides a description of the sub-directories created in the DM648_TNETV2685_BP_001 directory.

Table 2-1. Component Directories

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

2.3 Before Building the Sample Test Application

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

This version of the codec has been validated with DSP/BIOS version 5.31 and Framework Component (FC) version 1.10.01.

2.3.1 Installing DSP/BIOS

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

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

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

<install directory>\CCStudio_v3.3

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

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

2.3.2 Installing Framework Component (FC)

You can download FC from the TI external website:

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

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

<install directory>\CCStudio_v3.3

The test application uses the following DMAN3 files:

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

2.4 Building and Running the Sample Test Application

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

1) Verify that you have installed TI's Code Composer Studio version 3.3.24.1 and code generation tools version 6.0.7.

2) Verify that the codec object library, h264venc_ti.l64P exists in the \Lib sub-directory.

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

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

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

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

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

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

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

2.5 Configuration Files

This codec is shipped along with:

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

2.5.1 Generic Configuration File

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

The format of the Testvecs.cfg file is:

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

where:

- `X` may be set as:
  - 1 - for compliance checking, no output file is created
  - 0 - for writing the output to the output file
- `Config` is the Encoder configuration file. For details, see Section 2.5.2.
- `Input` is the input file name (use complete path).
- `Output/Reference` is the output file name (if `X` is 0) or reference file name (if `X` is 1) with complete path.

A sample Testvecs.cfg file is as shown:

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

2.5.2 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:

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

ImageWidth     = 640       # Image width in Pels
ImageHeight    = 480       # Image height in Pels
FrameRate      = 30000     # Frame Rate per second*1000 (1-
                             100)
Bitrate        = 2048000   # Bitrate(bps)  #if ZERO=>> RC
                             is OFF
ChromaFormat   = 1         # 1 => XDM_YUV_420P,
                             3 => XDM_YUV_422IBE,
                             4 => XDM_YUV_422ILE
IntraPeriod    = 30        # Period of I-Frames
FramesToEncode = 5          # Number of frames to be coded
```

Any field in the `IVIDENC_Params` structure (see Section 4.2.1.5) can be set in the Testparams.cfg file using the syntax shown above. If you specify additional fields in the Testparams.cfg file, ensure to modify the test application appropriately to handle these fields.

2.6 Standards Conformance and User-Defined Inputs

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

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

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

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

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

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

   - "Encoder compliance test passed/failed" (if `X` is 1)
   - "Encoder output dump completed" (if `X` is 0)

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

2.7 Uninstalling the Component

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

2.8 Evaluation Version

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

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 `IVIDENC` base class of the H.264 Encoder library. The main test application files are TestAppEncoder.c and TestAppEncoder.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.

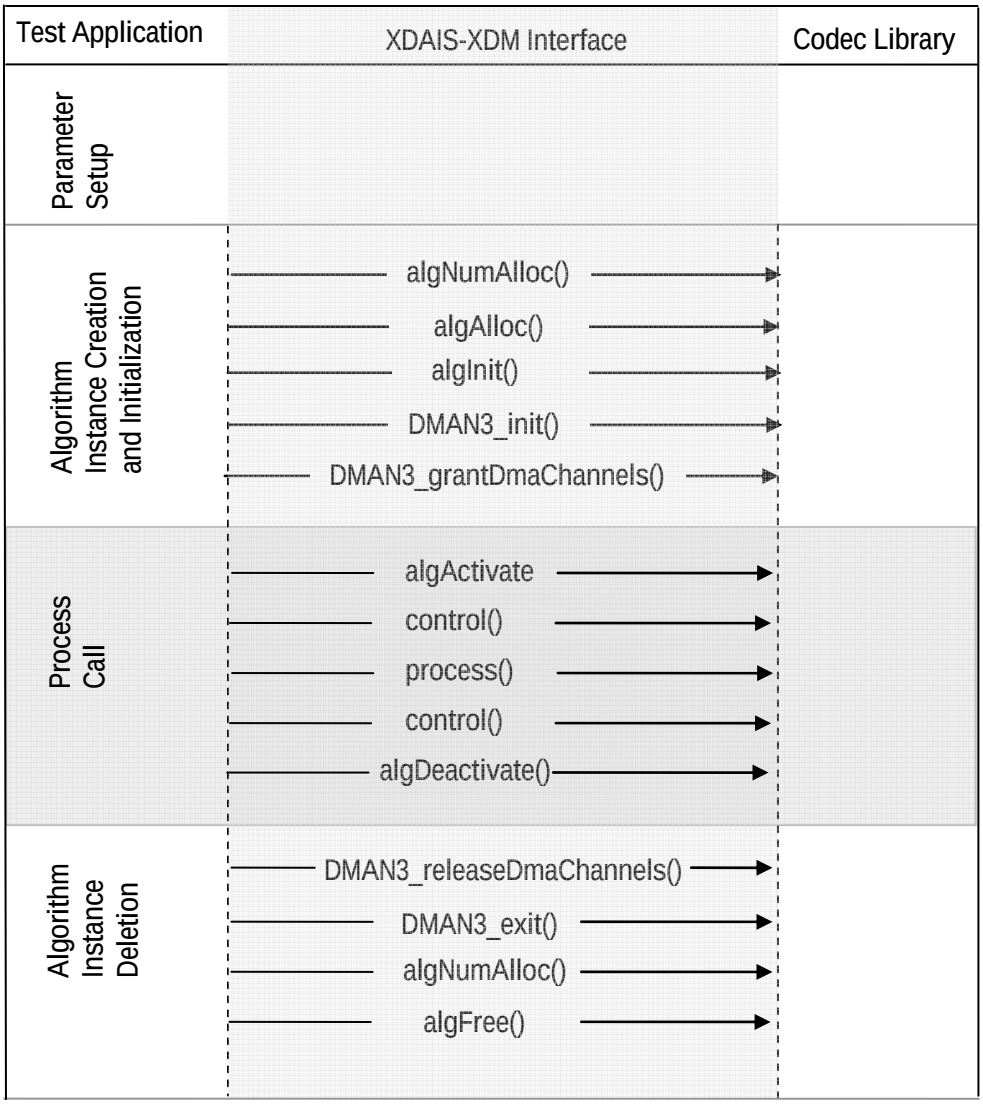


Figure 3-1. Test Application Sample Implementation

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 compliance checking parameter, Encoder configuration file name (Testparams.cfg), input file name, and output/reference file name.

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 2.5.

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

4) Initializes the various DMAN3 parameters.

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

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

3.1.2 Algorithm Instance Creation and Initialization

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

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

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

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

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

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

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

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

Note:

DMAN3 function implementations are provided in dman3.a64P library.

3.1.3 Process Call

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

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

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

3) Calls the `process()` function to encode/decode a single frame of data. The behavior of the algorithm can be controlled using various dynamic parameters (see Section 4.2.1.6). The inputs to the process function are input and output buffer descriptors, pointer to the `IVIDENC_InArgs` and `IVIDENC_OutArgs` structures.

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

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

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

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

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

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

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

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

3.1.4 Algorithm Instance Deletion

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

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

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

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

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

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

This page is intentionally left blank

Chapter 4

API Reference

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

Topic **Page**

4.1 Symbolic Constants and Enumerated Data Types

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

Table 4-1. List of Enumerated Data Types

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
IVIDEO_FrameType	IVIDEO_I_FRAME	Intra coded frame
	IVIDEO_P_FRAME	Forward inter coded frame
	IVIDEO_B_FRAME	Bi-directional inter coded frame
	IVIDEO_IDR_FRAME	Intra coded frame that can be used for refreshing video content.
IVIDEO_ContentType	IVIDEO_PROGRESSIVE	Progressive video content (default value).
	IVIDEO_INTERLACED	Interlaced video content. Not supported in this version of H264 Encoder.
IVIDEO_RateControlPreset	IVIDEO_NONE	No rate control is used
	IVIDEO_LOW_DELAY	Constant Bit-rate (CBR) control for video conferencing (default value).
	IVIDEO_STORAGE	Variable Bit-rate (VBR) control for local storage (DVD) recording.
	IVIDEO_TWOPASS	Two pass rate control for non-real time applications. Not supported in this version of H264 Encoder.
	IVIDEO_USER_DEFINED	User defined configuration using advanced parameters.
IVIDEO_SkipMode	IVIDEO_FRAME_ENCODED	Input content encoded.
	IVIDEO_FRAME_SKIPPED	Input content skipped, that is, not encoded.
XDM_DataFormat	XDM_BYTE	Big endian stream (default value)
	XDM_LE_16	16-bit little endian stream. Not supported in this version of H264 Encoder.
	XDM_LE_32	32-bit little endian stream. Not supported in this version of H264 Encoder.
XDM ChromaFormat	XDM_YUV_420P	YUV 4:2:0 planar

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_YUV_422P	YUV 4:2:2 planar. Not supported in this version of H264 Encoder.
	XDM_YUV_422IBE	YUV 4:2:2 interleaved (big endian)
	XDM_YUV_422ILE	YUV 4:2:2 interleaved (little endian) (default value)
	XDM_YUV_444P	YUV 4:4:4 planar. Not supported in this version of H264 Encoder.
	XDM_YUV_411P	YUV 4:1:1 planar. Not supported in this version of H264 Encoder.
	XDM_GRAY	Gray format. Not supported in this version of H264 Encoder.
	XDM_RGB	RGB color format. Not supported in this version of H264 Encoder.
XDM_CmdId	XDM_GETSTATUS	Query algorithm instance to fill Status structure.
	XDM_SETPARAMS	Set run-time dynamic parameters through the DynamicParams structure.
	XDM_RESET	Reset the algorithm.
	XDM_SETDEFAULT	Initialize all fields in Params structure to default values specified in the library.
	XDM_FLUSH	Handle end of stream conditions. This command forces algorithm instance to output data without additional input.
	XDM_GETBUFINFO	Query algorithm instance regarding the properties of input and output buffers.
XDM_EncodingPreset	XDM_DEFAULT	Default setting of the algorithm specific creation time parameters.
	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.

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_USER_DEFINED	User defined configuration using advanced parameters.
XDM_EncMode	XDM_ENCODE_AU	Encode entire access unit (default value).
	XDM_GENERATE_HEADER	Encode only header.
IH264VENC_LoopFilterParams	FILTER_ALL_EDGES	Enable filtering of all the edges.
	DISABLE_FILTER_ALL_EDGES	Disable filtering of all the edges.
	DISABLE_FILTER_SLICE_EDGES	Disable filtering of slice edges.
IH264VENC_Level	IH264_LEVEL_10	H.264 Level 1.0
	IH264_LEVEL_1b	H.264 Level 1.b
	IH264_LEVEL_11	H.264 Level 1.1
	IH264_LEVEL_12	H.264 Level 1.2
	IH264_LEVEL_13	H.264 Level 1.3
	IH264_LEVEL_20	H.264 Level 2.0
	IH264_LEVEL_21	H.264 Level 2.1
	IH264_LEVEL_22	H.264 Level 2.2
	IH264_LEVEL_30	H.264 Level 3.0
IH264VENC_PicOrderCountType	IH264_POC_TYPE_0	POC Type 0
	IH264_POC_TYPE_2	POC Type 2
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

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	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 (see Table 4-2)

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

The H.264 Encoder specific error status messages are listed in Table 4-2.

Table 4-2. H.264 Encoder Error Statuses

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
IH264VENC_ErrorBit	IH264VENC_SEQPARAMERR	Bit 0 ❑ 1 - Error during sequence parameter set generation ❑ 0 - Ignore
	IH264VENC_PICPARAMERR	Bit 1 ❑ 1 - Error during picture parameter set generation ❑ 0 - Ignore
	IH264VENC_COMPRESSEDSIZEOVERFLOW	Bit 2 ❑ 1 - Compressed data exceeds the maximum compressed size limit ❑ 0 - Ignore
	IH264VENC_INVALIDQPPARAMETER	Bit 3 ❑ 1 - Out of range initial quantization parameter ❑ 0 - Ignore
	IH264VENC_INVALIDPROFILELEVEL	Bit 4 ❑ 1 - Invalid profile or level ❑ 0 - Ignore
	IH264VENC_INVALIDRCALGO	Bit 5 ❑ 1 - Invalid rate control algorithm ❑ 0 - Ignore
	IH264VENC_SLICEEXCEEDSMAXBYTES	Bit 6 ❑ 1 - Slice exceeds the maximum allowed bytes ❑ 0 - Ignore
	IH264VENC_DEVICENOTREADY	Bit 7 ❑ 1 - Device is not ready ❑ 0 - Ignore

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`
- `XDM_AlgBufInfo`
- `IVIDEO_BufDesc`
- `IVIDENC_Fxns`
- `IVIDENC_Params`
- `IVIDENC_DynamicParams`
- `IVIDENC_InArgs`
- `IVIDENC_Status`
- `IVIDENC_OutArgs`

4.2.1.1 XDM_BufDesc

Description

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

Fields

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

4.2.1.2 XDM_AlgBufInfo

Description

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

Fields

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

Note:

For H.264 Encoder, the buffer details are:

- Number of input buffer required is 1 for YUV 422ILE and 3 for YUV420P
- Number of output buffer required is 1
- The input buffer sizes (in bytes) for worst case PAL-D1 format are:

For YUV 420P:
Y buffer = 720 * 576
U buffer = 360 * 288
V buffer = 360 * 288

For YUV 422ILE:
Buffer = 720 * 576 * 2

- There is no restriction on output buffer size except that it should contain atleast one frame of encoded data.

These are the maximum buffer sizes but you can reconfigure depending on the input format.

4.2.1.3 IVIDEO_BufDesc

Description

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

Fields

Field	Datatype	Input/ Output	Description
`numBufs`	`XDAS_Int32`	Input	Number of buffers
`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

4.2.1.4 IVIDENC_Fxns

Description

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

Fields

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

4.2.1.5 IVIDENC_Params

Description

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

Fields

Field	Datatype	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes.
encodingPreset	XDAS_Int32	Input	Encoding preset. See XDM_EncodingPreset enumeration for details.
rateControlPreset	XDAS_Int32	Input	Rate control preset: See IVIDEO_RateControlPreset enumeration for details.
maxHeight	XDAS_Int32	Input	Maximum video height to be supported in pixels.
maxWidth	XDAS_Int32	Input	Maximum video width to be supported in pixels.

Field	Datatype	Input/ Output	Description
maxFrameRate	XDAS_Int32	Input	Maximum frame rate in fps * 1000 to be supported.
maxBitRate	XDAS_Int32	Input	Maximum bit-rate to be supported in bits per second.
dataEndianness	XDAS_Int32	Input	Endianness of input data. See XDM_DataFormat enumeration for details.
maxInterFrameInterval	XDAS_Int32	Input	Distance from I-frame to P-frame: ❑ 1 - If no B-frames ❑ 2 - To insert one B-frame
inputChromaFormat	XDAS_Int32	Input	Input chroma format. See XDM_ChromaFormat enumeration for details.
inputContentType	XDAS_Int32	Input	Input content type. See IVIDEO_ContentType enumeration for details.

Note:

For the supported maxBitRate values, see Table A.1 – Level Limits in *ISO/IEC 14496-10*.

The following fields of IVIDENC_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.

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;
```

4.2.1.6 IVIDENC_DynamicParams

‖ Description

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

‖ Fields

Field	Datatype	Input/ Output	Description
`size`	`XDAS_Int32`	Input	Size of the basic or extended (if being used) data structure in bytes.
`inputHeight`	`XDAS_Int32`	Input	Height of input frame in pixels.
`inputWidth`	`XDAS_Int32`	Input	Width of input frame in pixels.
`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.
`targetFrameRate`	`XDAS_Int32`	Input	Target frame rate in fps * 1000. For example, if the frame rate is 30, set this field to 30000.
`targetBitRate`	`XDAS_Int32`	Input	Target bit-rate in bits per second. For example, if the bit-rate is 2 Mbps, set this field to 2097152.
`intraFrameInterval`	`XDAS_Int32`	Input	Interval between two consecutive intra frames. ❑ 0 - Only first frame to be intra coded ❑ 1 - No inter frames (all intra frames) ❑ 2 - Consecutive IP sequence (if no B frames)
`generateHeader`	`XDAS_Int32`	Input	Encode entire access unit or only header. See `XDM_EncMode` enumeration for details.

Field	Datatype	Input/ Output	Description
captureWidth	XDAS_Int32	Input	If the field is set to: ❑ 0 - Encoded image width is used as pitch. ❑ Any non-zero value, capture width is used as pitch (if capture width is greater than image width).
forceIFrame	XDAS_Int32	Input	Force current frame to be encoded as I-frame.

Note:

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

- inputHeight <= maxHeight
- inputWidth <= maxWidth
- refFrameRate <= maxFrameRate
- targetFrameRate <= maxFrameRate
- targetBitRate <= maxBitRate

The rate control used in H.264 Encoder can work for a target bit-rate of a minimum of 32 kbps and a maximum of 10 mbps up to level 3. However, the recommended range varies with the format.

For example, for NTSC D1, the recommended range is 1.5 mbps to 6.0 mbps.

4.2.1.7 IVIDENC_InArgs

Description

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

Fields

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

4.2.1.8 IVIDENC_Status

Description

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

Fields

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

4.2.1.9 IVIDENC_OutArgs

Description

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

Fields

Field	Datatype	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes.
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.
inputFrameSkip	XDAS_Int32	Output	Frame skipping modes for video. See IVIDEO_SkipMode enumeration for details.
reconBufs	IVIDEO_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`

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, `IVIDENC_Params`.

Fields

Field	Datatype	Input/ Output	Description
`videncParams`	`IVIDENC_Params`	Input	See `IVIDENC_Params` data structure for details.
`profileIdc`	`XDAS_Int32`	Input	Profile identification for the encoder. For Base Profile, the only allowed value is 66.
`levelIdc`	`XDAS_Int32`	Input	Level identification for the encoder. See `IH264VENC_Level` enumeration for details.
`rcAlgo`	`XDAS_Int32`	Input	Algorithm to be used by Rate Control Scheme. Valid values are 0 (DCES_TM5) and 1(PLR). It is useful only when `rateControlPreset` of `IVIDENC_Params` is equal to `IVIDEO_USER_DEFINED`.
`searchRange`	`XDAS_Int32`	Input	Integer pel search around 16x16 blocks. The center of search window is the predicted vector.

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, `IVIDENC_DynamicParams`.

‖ Fields

Field	Datatype	Input/ Output	Description
`videncDynamicParams`	`IVIDENC_DynamicParams`	Input	See `IVIDENC_DynamicParams` data structure for details.
`qpIntra`	`XDAS_Int32`	Input	Initial Quantization Parameter (QP) of I-frames. Valid value is 0 to 51. It is useful only when `rateControlPreset` of `IVIDENC_Params` is equal to `IVIDEO_NONE`.
`qpInter`	`XDAS_Int32`	Input	Initial Quantization Parameter (QP) of P-frames. Valid value is 0 to 51. It is useful only when `rateControlPreset` of `IVIDENC_Params` is equal to `IVIDEO_NONE`.
`qpMax`	`XDAS_Int32`	Input	Maximum Quantization Parameter (QP) to be used. Valid value is 0 to 51.
`qpMin`	`XDAS_Int32`	Input	Minimum Quantization Parameter (QP) to be used. Valid value is 0 to 51.
`lfDisableIdc`	`XDAS_Int32`	Input	See `IH264VENC_LoopFilterParams` enumeration for details.
`quartPelDisable`	`XDAS_Int32`	Input	❑ 1 - Disable quarter pel interpolation ❑ 0 - Enable quarter pel interpolation
`airMbPeriod`	`XDAS_Int32`	Input	Periodicity of intra macro block. Encoder should forcefully insert intra macro block at the period specified for airMbPeriod (any non-zero value).
`maxMBsPerSlice`	`XDAS_Int32`	Input	❑ 0-7 - No effect ❑ >7 - Maximum number of macro blocks in a slice

Field	Datatype	Input/ Output	Description
maxBytesPerSlice	XDAS_Int32	Input	❑ 0 - No effect ❑ >0 - Maximum number of bytes in a slice
sliceRefreshRowStartNumber	XDAS_Int32	Input	Row number from which the slice need to be intra coded. For example, 1 indicates first row.
sliceRefreshRowNumber	XDAS_Int32	Input	Number of rows to be coded as intra slice
filterOffsetA	XDAS_Int32	Input	Alpha offset for loop filter. Valid value is an even number between -12 and 12, both inclusive.
filterOffsetB	XDAS_Int32	Input	Beta offset for loop filter. Valid value is an even number between -12 and 12, both inclusive.
log2MaxFNumMinus4	XDAS_Int32	Input	Limits the maximum frame number in the bit-stream to 1<< (log2MaxFNumMinus4 + 4). Valid value is 0 to 12, both inclusive.
chromaQPIndexOffset	XDAS_Int32	Input	Specifies the offset that is added to luma QP for addressing the table of QPC values for the chroma components. Valid value is between -12 and 12, both inclusive
constrainedIntraPredEnable	XDAS_Int32	Input	Controls the intra macro block coding in P slices ❑ 1 - Inter pixels cannot be used for intra macro block prediction ❑ 0 - Inter pixels can be used for intra macro block prediction
picOrderCountType	XDAS_Int32	Input	See IH264VENC_PicOrderCountType enumeration for details..

Note:

Any fields from the IH264VENC_DynamicParams structure is useful only when the encodingPreset field of IVIDENC_Params data structure is equal to XDM_USER_DEFINED.

4.2.2.3 IH264VENC_InArgs

Description

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

Fields

Field	Datatype	Input/ Output	Description
videncInArgs	IVIDENC_InArgs	Input	See IVIDENC_InArgs data structure for details.

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, IVIDENC_Status.

Fields

Field	Datatype	Input/ Output	Description
videncStatus	IVIDENC_Status	Output	See IVIDENC_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	Datatype	Input/ Output	Description
videncOutArgs	IVIDENC_OutArgs	Output	See IVIDENC_OutArgs 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* (literature number SPRU360).

4.3.1 Creation APIs

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

‖ Name

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

‖ Synopsis

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

‖ Arguments

```
Void
```

‖ Return Value

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

‖ Description

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

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

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

‖ See Also

`algAlloc()`

‖ Name

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

‖ Synopsis

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

‖ Arguments

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

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

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

‖ Return Value

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

‖ Description

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

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

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

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

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

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

‖ See Also

```
algNumAlloc(), algFree()
```

4.3.2 Initialization API

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

‖ Name

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

‖ Synopsis

```
XDAS_Int32 (*control) (IVIDENC_Handle handle, IVIDENC_Cmd
id, IVIDENC_DynamicParams *params, IVIDENC_Status
*status);
```

‖ Arguments

```
IVIDENC_Handle handle; /* algorithm instance handle */

IVIDENC_Cmd id; /* algorithm specific control commands*/

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

IVIDENC_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 `IVIDENC_DynamicParams` and `IVIDENC_Status` data structures respectively.

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

‖ Preconditions

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

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

‖ Postconditions

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

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

‖ Example

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

‖ See Also

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

4.3.4 Data Processing API

Data processing API is used for processing the input data.

‖ Name

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

‖ Synopsis

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

‖ Arguments

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

‖ Return Value

```
Void
```

‖ Description

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

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

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

‖ See Also

`algDeactivate()`

‖ Name

`process()` – basic encoding/decoding call

‖ Synopsis

```
XDAS_Int32 (*process)(IVIDENC_Handle handle, XDM_BufDesc
*inBufs, XDM_BufDesc *outBufs, IVIDENC_InArgs *inargs,
IVIDENC_OutArgs *outargs);
```

‖ Arguments

```
IVIDENC_Handle handle; /* algorithm instance handle */

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

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

IVIDENC_InArgs *inargs /* algorithm runtime input
arguments */

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

‖ Return Value

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

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

‖ Description

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

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

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

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

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

‖ Preconditions

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

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

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

‖ Postconditions

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

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

‖ Example

See test application file, TestAppEncoder.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 an uncompressed video frame in one of the format defined by `inputChromaFormat` of `IVIDENC_Params` structure .The encoder outputs H.264 compressed bit stream in the little-endian format.

‖ 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()`